03099

UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

POSGRADO EN CIENCIA E INGENIERÍA DE LA COMPUTACIÓN

TRANSFORMACIÓN DE PROGRAMAS LÓGICOS: ALGORITMOS DE CASAMIENTO DE CADENAS

T E S I S

QUE PARA OBTENER EL GRADO DE:

DOCTOR EN CIENCIAS
(COMPUTACIÓN)

P R E S E N T A:

MANUEL HERNÁNDEZ GUTIERREZ

DIRECTOR DE LA TESIS: DR. DAVID A. ROSENBLUETH LAGUETTE

MÉXICO, D.F. **Enero de 2004**

Universidad Nacional
Autónoma de México

UNAM – Dirección General de Bibliotecas
Tesis Digitales
Restricciones de uso

DERECHOS RESERVADOS ©
PROHIBIDA SU REPRODUCCIÓN TOTAL O PARCIAL

Todo el material contenido en esta tesis esta protegido por la Ley Federal del Derecho de Autor (LFDA) de los Estados Unidos Mexicanos (México).

El uso de imágenes, fragmentos de videos, y demás material que sea objeto de protección de los derechos de autor, será exclusivamente para fines educativos e informativos y deberá citar la fuente donde la obtuvo mencionando el autor o autores. Cualquier uso distinto como el lucro, reproducción, edición o modificación, será perseguido y sancionado por el respectivo titular de los Derechos de Autor.

Resumen

La transformación de programas es un método que permite obtener programas eficientemente ejecutables desde especificaciones que resuelven claramente un problema. En este método dividimos la actividad de la programación en distintas etapas. En una primera etapa escribimos un programa inicial que claramente satisfaga una especificación, pero sin consideraciones relacionadas con la eficiencia en ejecución de este programa. En una segunda etapa, generamos una sucesión de programas por medio de la aplicación de algunas reglas de transformación; cada regla debe preservar el significado del programa inicial, y el programa final de esta sucesión debe ser eficientemente ejecutable. El tema de estudio de esta tesis es la construcción de programas lógicos por transformación, y hemos tomado como caso particular de estudio el problema del casamiento exacto de cadenas.

Los algoritmos de casamiento de cadenas son frecuentemente objeto de investigación para ejemplificar la viabilidad de diversas metodologías de programación. En particular, el algoritmo de casamiento de cadenas de Knuth, Morris y Pratt es uno de los preferidos como caso de estudio en la transformación de programas. En cambio, el algoritmo de Boyer y Moore, que también resuelve el problema de casamiento de cadenas, no ha recibido tanta atención, a pesar de ser este algoritmo, y no el de Knuth, Morris y Pratt, el que ha mostrado una superioridad en la práctica. En esta tesis estudiaremos el algoritmo de Boyer y Moore en detalle y como caso de estudio principal de la transformación de programas lógicos.

Podemos dividir las aportaciones de esta tesis a la transformación de programas lógicos en dos partes. La primera parte consiste en material que no hemos publicado, pero que creemos es novedoso y útil, y tiene implicaciones para un trabajo a futuro. De entro de estos resultados, destacan una descripción taxonómica de algunos algoritmos de casamiento de cadenas y la utilización sistemática de la introducción de ecuaciones e inecuaciones. La segunda parte de nuestras contribuciones consta de material que ya hemos publicado. Este material consiste, por un lado, de la derivación de la parte de búsqueda de algunas variantes del algoritmo de casamiento de cadenas de Boyer y Moore; esta derivación nos ha permitido establecer la interrelación existente entre las funciones que utilizan algunos algoritmos de casamiento de cadenas y el desdoblado determinístico. Por otro lado, nuestra contribución a la deducción parcial disyuntiva consiste en una generalización de una estrategia de Pettorossi, Proietti y Renault. Esta generalización nos ha permitido derivar tanto la parte de búsqueda del algoritmo de Knuth, Morris y Pratt como la parte de búsqueda de una variante del algoritmo de Boyer y Moore.

Abstract

Program transformation is a method that allows to obtain highly efficient executable programs from highly abstract, formal specifications. This method can be described as follows. First, we write an initial program which clearly satisfies a given specification, without any concerns about the efficiency in execution of this program. Second, we generate a succession of programs beginning from the initial program. We construct this succesion by applying some transformation rules; each rule must be semantics-preserving: the meaning of the final program of this succession must be the same that the meaning of the initial program. The final program must be efficiently executable. The main topic of this thesis is the study of some tools for deriving, by transformation and in a constructive way, logic programs, and we have taken as a particular case of application the exact string-matching problem.

String-matching algorithms are often used as a case of study to exemplify the viability of several methodologies of programming. In particular, the Knuth, Morris and Pratt string-matching algorithm is the preferred algorithm in the area of program transformation. In contrast, the Boyer and Moore string-matching algorithm has not received so much attention, although this algorithm is more efficient and used in practice than the Knuth, Morris and Pratt algorithm. In this thesis we analyze the Boyer and Moore algorithm in detail.

We divide the contributions of this thesis to the program transformation into two main parts. The first part consists of work that has not been published, but that we believe is useful and new. Within these results, we point out our taxonomic description of some string-matching algorithms, the systematic introduction of equations and inequations, and the use of the deterministic unfolding as a valuable tool in logic program transformation.

The second part of our contributions consists of work that we have already published, and this material includes the following. On the one hand, we derived the search stage of the Boyer and Moore string-matching algorithm, and this derivation allowed us to establish the relationship between the functions used by the Boyer and Moore algorithm and the deterministic unfolding. On the other hand, our contribution to the disjuctive partial deduction consists in a generalization of a strategy of Pettorossi, Proietti and Renault. This generalization allowed us to derive both the search stage of the Knuth, Morris and Pratt algorithm and the search stage of a variant of the Boyer and Moore algorithm.

Agradecimientos

Agradezco al Dr. David A. Rosenblueth Laguette el haber dirigido esta tesis y mis estudios doctorales con gran dedicación y pericia, aportando siempre el equilibrio entre la libertad de mi pensamiento y la disciplina requerida por un proyecto de doctorado. Además, le doy las gracias por enseñarme a valorar las dificultades y los placeres de la lectura y la escritura, y por sus otros bellísimos obsequios intelectuales y artísticos (de entre los que destacan *Romeo and Juliet* de Shakespeare, y nuestra estancia en *Firenze, Italia*). Me siento afortunado de contar con su gran amistad.

Agradezco al Dr. Robert Kowalski su enorme aportación intelectual a mi vida científica por medio de sus escritos y de la programación lógica. Fue un gran honor para mí presentar algunos de mis resultados ante él durante una de sus estancias en México.

Agradezco a algunos participantes del congreso *Principles and Practice of Declarative Programming* (PPDP'01, Florencia, Italia) sus comentarios y sugerencias a una parte (difundida por medio de un artículo) de esta tesis. Mis especiales agradecimientos, en particular, al Dr. Olivier Danvy, al Dr. Michael Leuschel, y a los doctores Maurizio Proietti y Alberto Pettorossi. Maurizio Proietti influyó en algunos de mis conceptos generales de lo que es una especificación y un programa lógico, y Alberto Pettorossi influyó en algunos de mis lineamientos generales de investigación respecto a la transformación de programas.

Agradezco a los doctores Ernesto Bribiesca Correa, Mauricio Javier Osorio Galindo, Alfonso San Miguel Aguirre, Francisco Hernández Quiroz, Zbigniew Oziewicz Kwass, y a la doctora Atocha Aliseda Llera, sus útiles recomendaciones y valiosas sugerencias para el mejoramiento de esta tesis. El Dr. Mauricio me proporcionó algunos útiles consejos en la presentación de esta tesis, mientras que la Dra. Atocha me aportó algunas interesantes ideas al nivel de la filosofía de la construcción de los sistemas de transformación de programas.

Agradezco la amistad y apoyo de la Dra. Alejandra Cortéz Silva y del Dr. Felipe Lara Rosano.

Agradezco el apoyo económico que el CONACYT me ha brindado a través de los años, y en particular, en lo referente a la beca de doctorado que me otorgó.

Agradezco al IIMAS y a la UNAM por las facilidades materiales y humanas que me otorgaron para llevar a cabo mi proyecto de doctorado. Gracias a todos mis compañeros del Departamento de Ciencias de la Computación del IIMAS por el ambiente cálido y amable que siempre me brindaron. Mis agradecimientos especiales para el Prof. Carlos Velarde por su siempre generosa y bien dispuesta ayuda.

Agradezco con todo mi corazón el apoyo y amor que mi familia siempre me ha dado: a mi mamá por su inagotable cariño, a mi papá por su siempre incondicional apoyo, y a mi hermana Delia por siempre haber creído en mí. Un agradecimiento especial para mis amigos Manuel, Alejandro, y Alfredo (además de amigo, primo), y para mis amigas Fabiola, Laura, Hua, y Wendy, y a la amistad de mis tías Estela y Judith.

Un recuerdo particularme grato y mis profundos agradecimientos a algunos de mis seres queridos cuya presencia en mi alma me motivaron para seguir adelante: mi abuelo Concepción, mi amigo Ezequiel, mi amiga Pilar, y mi hermana Mónica.

Manuel Hernández Gutiérrez
México, D.F.,
a 12 de enero de 2004.

Índice General

Índice General 1

Índice de Figuras 7

1 Introducción 11

1.1 El problema de la escritura de programas correctos . . . 11

1.1.1 La crisis del *software*. . . . 11

1.1.2 La programación estructurada . . . 12

1.1.3 La programación declarativa . . . 12

1.2 Transformación de programas . . . 13

1.2.1 Objetivos de la transformación de programas . . . 13

1.2.2 La transformación de programas como método . . . 13

1.2.3 Reglas y estrategias en la transformación de programas . . . 14

1.3 Programación lógica . . . 15

1.4 Taxonomías de familias de algoritmos . . . 15

1.4.1 Enfoque taxónomico de la derivación de algoritmos . . . 16

1.4.2 Estructuración de conocimiento y taxonomías . . . 16

1.4.3 Reuso de desarrollos y taxonomías . . . 17

1.5 Algoritmos de casamiento de cadenas . . . 17

1.6 La derivación automática de programas . . . 18

1.6.1 Derivación de la parte de búsqueda del algoritmo de Knuth, Morris y Pratt . . . 18

1.6.2 Derivaciones de la parte de búsqueda del algoritmo de Boyer y Moore 18

1.7 Justificación de la derivación de algoritmos ya existentes 19
1.8 Aportaciones de esta tesis 20
1.9 Resumen de esta tesis 20

2 Programación lógica 23
2.1 Sintaxis de los programas lógicos 23
2.2 Semántica de los programas lógicos 25
2.3 La regla de resolución 26
2.4 La teoría estándar de igualdad 30
2.4.1 Teoría axiomática de igualdad 30
2.4.2 La teoría de desigualdad de Clark 30
2.5 Listas y abreviaciones 31

3 Transformación de programas lógicos 33
3.1 Transformación de programas 33
3.2 Las reglas de desdoblado/doblado: introducción 34
3.3 Problemática actual de la transformación de programas por el método de desdoblado/doblado 36
3.4 El sistema de transformación por desdoblado-doblado en programación lógica 38
3.5 El método de desdoblado/doblado: detalles técnicos 38
3.5.1 Ejemplos aplicados al problema de ordenamiento 46
3.5.2 Especialización de programas y deducción parcial 51
3.5.3 El desdoblado determinístico 52
3.6 Estrategias de transformación de programas 53
3.6.1 La estrategia S de Gallagher 53
3.6.2 La estrategia D de Pettorossi, Proietti y Renault 56
3.6.3 División en casos 56
3.6.4 El desdoblado determinístico y el problema de la terminación 59
3.6.5 El doblado 59
3.7 Esquemas teóricos para la transformación de programas 60

4 Algoritmos de casamiento de cadenas 63

4.1 Introducción . . . 63
4.2 Algunos algoritmos de casamiento de cadenas . . . 64
4.2.1 El algoritmo de fuerza bruta . . . 65
4.2.2 El algoritmo de Boyer y Moore y algunas variantes . . . 67
4.3 Los algoritmos MP y KMP . . . 71
4.3.1 El algoritmo de Morris y Pratt . . . 71
4.3.2 El algoritmo de Knuth, Morris y Pratt . . . 73

5 Derivación de la parte de búsqueda de una variante del algoritmo BM 77
5.1 Derivación de la parte de búsqueda del algoritmo BM . . . 77
5.2 Mayores detalles de nuestra derivación . . . 93
5.2.1 Observaciones . . . 102
5.3 Derivación formal de la parte de búsqueda del algoritmo BM . . . 104
5.4 Conclusiones . . . 108

6 Reuso de desarrollos y una derivación de la parte de búsqueda del algoritmo KMP 111
6.1 Reutilización de desarrollos derivacionales . . . 111
6.2 Derivación de la parte de búsqueda del algoritmo KMP . . . 111
6.3 El desdoblado determinístico y el algoritmo KMP . . . 117
6.3.1 Caso MP . . . 117
6.3.2 Caso KMP . . . 119
6.4 Conclusiones . . . 120
6.5 Apéndice: El algoritmo de Knuth, Morris y Pratt según Pettorossi, Proietti y Renault . . . 121

7 Una taxonomía de algoritmos de casamiento de cadenas 127
7.1 Reuso de desarrollos y familias de algoritmos . . . 127
7.2 Variantes del algoritmo BM . . . 128
7.2.1 Una variante menor . . . 128
7.2.2 La variante de Partsch . . . 128
7.2.3 La variante de Horspool . . . 129

7.3 La variante de Apostolico, Galil, y Giancarlo . . . 129

7.3.1 Variantes del algoritmo KMP . . . 132

7.4 Acerca del diseño y desarrollo de una taxomía de algoritmos de casamiento de cadenas . . . 133

7.5 Conclusiones . . . 134

8 La estrategia D' 135

8.1 Introducción . . . 135

8.1.1 El algoritmo BM y la deducción parcial . . . 136

8.1.2 El caso de la derivación automática de la parte de búsqueda de variante del algoritmo BM . . . 136

8.1.3 Motivación de la estrategia D' . . . 137

8.2 Algoritmos de casamiento de cadenas . . . 138

8.3 Deducción parcial disyuntiva y la estrategia D . . . 139

8.3.1 La deducción parcial disyuntiva . . . 139

8.3.2 La estrategia D' . . . 141

8.4 Una derivación de la parte de búsqueda de una variante del algoritmo BM . 143

8.5 Conclusiones . . . 149

9 Trabajo relacionado 151

9.1 Derivaciones del algoritmo BM . . . 151

9.2 Derivaciones del algoritmo KMP . . . 154

10 Conclusiones 155

10.1 Aportaciones . . . 155

10.1.1 Programación declarativa . . . 155

10.1.2 Reuso de desarrollos . . . 155

10.1.3 Contribuciones a la transformación de programas lógicos . . . 156

10.1.4 Contribuciones a la deducción parcial de programas lógicos . . . 157

10.2 Actualidad de nuestro trabajo . . . 158

10.3 Trabajo a futuro . . . 158

Bibliografía **161**

Índice de Materias **174**

ÍNDICE GENERAL

Índice de Figuras

3.1 Una identificación de nodos que son comunes a un nivel. 42

3.2 Efecto del doblado sobre los nodos que son comunes a un nivel. 42

3.3 La idea principal en el desdoblado determinístico. 53

3.4 La idea principal en una derivación determinística. 54

3.5 La idea principal del criterio para detener una derivación determinística. . . 54

3.6 La estrategia S de especialización de Gallagher. 55

3.7 La estrategia D de Pettorossi, Proietti y Renault. 57

4.1 La dirección de los desplazamientos del patrón sobre el texto. 65

4.2 Configuración inicial entre el patrón y el texto. 65

4.3 Posibles comparaciones entre los símbolos de patrón y del texto. 65

4.4 Algunas comparaciones símbolo a símbolo en una configuración inicial entre p y t. 66

4.5 Próxima configuración del algoritmo de fuerza bruta. 66

4.6 Dirección del proceso de casamiento símbolo a símbolo según el algoritmo BM. 67

4.7 Inicio del casamiento símbolo a símbolo en el algoritmo BM. 67

4.8 Ejemplo del funcionamiento del algoritmo BM. 67

4.9 Desplazamiento dado por la función δ_1. 68

4.10 Desplazamiento dado por la función δ_2. 69

4.11 El algoritmo de casamiento de cadenas de Boyer y Moore. 70

4.12 Dirección del intento de casamiento entre un patrón y un texto en el algoritmo KMP. 71
4.13 Inicio del casamiento símbolo a símbolo en el algoritmo KMP. 71
4.14 El algoritmo de Knuth, Morris y Pratt. 75

5.1 Una representación de la ocurrencia del patrón *aab* dentro del texto T. . . 78
5.2 División en casos. 82
5.3 Posibilidades de la variable de más a la derecha. 82
5.4 La trayectoria correspondiente a un casamiento parcial. 83
5.5 La trayectoria correspondiente a un casamiento total. 83
5.6 Posibilidades de A_3 dado un mal casamiento en b. 85
5.7 Eliminación de $\in$ y $\notin$. 85
5.8 $\in$ y $\notin$ a todo nivel. 96
5.9 Eliminación de $\in$ y $\notin$. 96
5.10 Caso en que y es comparado con z. 102
5.11 Una alineación redundante (pues $y \neq x$). 102

6.1 División sistemática de casos. 113
6.2 Casamiento parcial del patrón *aab* (subpatrón *aa*) dentro de un texto. . . . 113
6.3 El hallazgo del patrón *aab* dentro de un texto. 113
6.4 Derivaciones para BM (izquierda) y KMP (derecha) 114

7.1 Una taxonomía transformacional de algunos algoritmos de casamiento de cadenas. 133
7.2 Una clasificación taxonómica de algunos algoritmos de casamiento de cadenas. 134

8.1 Una variante del algoritmo de Boyer y Moore. 140
8.2 La estrategia D. 141
8.3 La estrategia D' representada gráficamente. 142

8.4 La estrategia D'. 143

ÍNDICE DE FIGURAS

Capítulo 1

Introducción

El tema de estudio principal de esta tesis es la transformación de programas lógicos como un método para la construcción de programas, junto con la aplicación de las técnicas transformacionales propias de este método a la derivación de algunos algoritmos de casamientos de cadenas.

En este capítulo presentamos una introducción a la transformación de programas y a su relación con la programación lógica, así como a la relación existente entre la transformación de programas lógicos y la derivación de la parte de búsqueda de algunos algoritmos de casamiento de cadenas. Presentamos además una introducción a la interrelación existente entre una variante de un importante algoritmo de casamiento de cadenas y la deducción parcial. El capítulo concluye con una descripción de nuestras aportaciones, un panorama formal de esta tesis, y unas notas sobre nuestra terminología y presentación.

1.1 El problema de la escritura de programas correctos

1.1.1 La crisis del *software*.

La frase "crisis del *software*" [Gri79] fue acuñada a finales de la década de los sesenta en el siglo pasado, y su formulación reconocía el hecho de que la escritura de programas era una actividad altamente informal y, por lo tanto, particularmente proclive a errores. Fue entonces cuando F.L. Bauer [BB79, Bau79a, Bau79b], E.W. Dijsktra [Dij72, Dij76, Dij79a, Dij79b], y C.A.R. Hoare [Hoa78], entre otros prominentes programadores y científicos, se plantearon el problema de cómo escribir programas a un costo y en un tiempo razonable, y que realmente satisfacieran las especificaciones dadas.

1.1.2 La programación estructurada

Una de las primeras soluciones a la crisis de *software* fue el diseño e implementación de nuevos lenguajes de programación en los que la modularización y la organización interna de los programas reportaran tanto el beneficio de la claridad como el de la manejabilidad [Gri78]. El enfoque de la *programación estructurada*, de acuerdo con la frase con que se le nombró a este tipo de programación, estaba determinado en gran medida por la *forma* de resolver un problema y por la *eficiencia* de la solución. En la programación estructurada el uso la asignación destructiva jugaba un papel decisivo, pero conducía a dificultades en la demostración de la corrección de los programas, lo que a su vez condujo a la programación estructurada al siguiente *enfoque verificacional*: primero escribimos un programa y después *verificamos* que es correcto. Sin embargo, este enfoque planteaba algunas dificultades al demostrar la corrección de un programa en la práctica, pues el método sólo exigía, en el mejor del casos, la verificación de que un programa era correcto, pero no brindaba ninguna información acerca de cómo construirlo. Pero aún en el aspecto de la verificación de corrección, la dificultad intrínseca de manejar matemáticamente la asignación destructiva generaba mayores problemas de los que resolvía. Por tales motivos, desde entonces se buscaron nuevas herramientas que facilitaran la programación de computadoras, y a continuación describimos una de estas herramientas.

1.1.3 La programación declarativa

Habiendo ya señalado el problema de la corrección de programas, en uno de los siguientes pasos hacia la obtención de programas correctos la atención se centró en *qué calcular* en vez de *cómo calcularlo*. Pero para ser práctico, este enfoque requería de algunas herramientas de apoyo. Por tal motivo, los seguidores de este enfoque inventaron o reconsideraron los *paradigmas declarativos*, de los que los ejemplos más sobresalientes son la *programación funcional* y la *programación lógica* (para los que existen excelentes introducciones en [BW88] y en [Kow79b, Llo87], respectivamente). Los lenguajes de programación que implementan estos paradigmas tienen en común la ausencia de la asignación destructiva, y generalmente también poseen una semántica operacional simple, por lo que son preferibles con respecto de los lenguajes imperativos para propósitos de tratamiento formal de programas (tal como fue enfáticamente afirmado por Backus, en su recepción del Premio Turing, y en [Bac78]), logrando evitar o aminorar las dificultades de trabajar con programas procedurales y en su lugar intentar trabajar con objetos matemáticos que son mejor conocidos [Kur88, Mee89]. Consideramos, por consiguiente, que la programación declarativa, y la programación lógica en particular, son campos ideales para la *transformación de programas*, que es el tema que tratamos en la próxima sección.

1.2 Transformación de programas

En otro paso más en dirección de la programación declarativa, se propuso la escritura de programas declarativos para después *transformarlos* por medio de una sucesión de aplicaciones de reglas de transformación que preservan el significado de los programas originales (idea que se remonta a [McC63b]) de tal manera que las implementaciones finales tengan una eficiencia aceptable, generalmente teniendo un contexto de ejecución bien definido [Dev90, Par90, MDM94, BdM97]. Por lo tanto, consideramos que el método de la transformación de programas es una de las herramientas que apoyan la programación (independientemente de que ésta sea declarativa o no). Presentamos más detalles de la transformación de programas en la siguiente subsección.

1.2.1 Objetivos de la transformación de programas

El objetivo principal de la transformación de programas es la escritura de programas que sean correctos y eficientemente ejecutables. Dentro del lineamiento de programación por transformación, nos comprometemos con la siguiente tarea:

> *Dado un programa correcto, pero ineficiente de acuerdo con un modelo de ejecución, derivamos del mismo un programa correcto y eficientemente ejecutable* por deducción [McC63a, Dij76, BD77, BW88].

Esta tarea involucra el desarrollo de programas paralelamente junto con su demostración de corrección dentro de un sistema formal deductivo.

Habiendo descrito uno de los objetivos de la transformación de programas, mencionamos que tal objetivo es uno de entre otros más ambiciosos y de largo plazo: la transformación de programas pretende mejorar la confiabilidad, la productividad, el mantenimiento, y el análisis del *software* sin sacrificar su buen desempeño [Pai96].

1.2.2 La transformación de programas como método

La transformación de programas es un *método de construcción de programas*. Algunos sistemas de transformación de programas que ejemplifican la versatilidad y potencia de este método son los siguientes:

- En el sistema de Partsch [Par90], por ejemplo, se enfatiza la *derivación de programas* como un método para sintetizar programas además de sólo transformarlos, y aún en el aspecto de la transformación se tiene también presente la implementación final.

- En [Dev90], Deville sigue el mismo enfoque de Partsch, pero con la programación lógica como marco de referencia general, y con el lenguaje de programación Prolog como lenguaje básico para la implementación de los programas finales.

- Un sistema como el de Mili et al. [MDM94], muestra la versatilidad en la derivación y en la transformación de programas, al mantener las especificaciones y los programas a nivel relacional, pero las implementaciones finales a nivel procedural.

- En un espíritu de trabajo similar, la teoría de las categorías y alegorías [BdM97] hacen el papel de la teoría relaciones en el trabajo de Mili et al., mientras que las implementaciones finales están escritas en un lenguaje funcional.

- Finalmente, dentro de nuestro recuento de trabajos representativos, el sistema de Pepper [Pep93] nos da un caso particular de un sistema híbrido, en donde se propone la utilización del enfoque transformacional y algebraico en estrecha colaboración con el enfoque verificacional.

1.2.3 Reglas y estrategias en la transformación de programas

Uno de los objetivos de la transformación de programas es la mecanización de los procesos de transformación, y este objetivo se satisface actualmente en diversos grados, lo que ha originado dos tipos de sistemas de transformación, ambos compartiendo la necesidad de tener *reglas y estrategias* [PP96d] que preserven la corrección de un programa durante una sucesión de transformación.

En un primer tipo, tenemos una cantidad enorme de reglas de transformación y a su vez un enorme espacio de búsqueda. Sistemas como el de Partsch [Par90] y el de Pepper [Pep93, Pep87b] son ejemplos del primer tipo de sistemas. Generalmente estos sistemas son dirigidos por un usuario experto, aunque algunas partes sean automáticas. En el segundo tipo de sistema, tenemos una limitada cantidad de reglas de transformación, pero a cambio generalmente poseemos también algunas estrategias que guían la aplicación automática de estas reglas. Tales sistemas son, por ejemplo, la evaluación y la deducción parcial [LS91, Kom92, Jon96, PP96b, PP98b, Leu98b], y ciertas partes relacionadas con las técnicas de supercompilación [GS95, HSG98].

A continuación describimos de una manera introductoria una de las principales herramientas que utilizaremos para la transformación de programas, la programación lógica.

1.3 Programación lógica

En esta tesis utilizamos el formalismo de la programación lógica porque, además de las ventajas como paradigma representante del enfoque declarativo, la programación lógica incorpora en su formalismo el no-determinismo. Lo deseable del no-determinismo es que nos permite frecuentemente dar una solución directa a muchos problemas [GB65, Flo67]. La utilidad del no-determinismo también está reconocida en el campo imperativo [Dij75]. En efecto, el no-determinismo permite la formulación de la solución de problemas sin restricciones artificiales, mientras que además también permite posponer algunas decisiones de diseño [Bau79b]. Frecuentemente, sin embargo, los programas no-determinísticos son ineficientes — por ejemplo, Feather comenta en [Fea87, página 185] que la reducción o la eliminación del no-determinismo puede conducir a una mejora en la eficiencia de los programas. En particular, los programas lógicos no-determinísticos con frecuencia son ineficientes cuando son ejecutados en un intérprete Prolog usual, por lo que es deseable mantener el no-determinismo en la especificación pero eliminarlo en el nivel de la implementación. Dado un programa lógico P, la declaratividad de la programación lógica está relacionada con lo que Kowalski [Kow79a] llamó *el componente lógico de P*, mientras que la parte del no-determinismo está relacionada con lo que él llamo *el componente de control de P*, de acuerdo con la seudo-ecuación: algoritmo = lógica + control.

Guttag señala, alternativamente, en [Gut86, página 55], que una de las más valiosas recompensas de la abstracción en el problema especificación-implementación es la posibilidad de escribir especificaciones sin considerar los algoritmos que las implementan. En este sentido, la delimitación abstracta de la especificación y la implementación separa la parte relevante del "qué" de la parte irrelevante, o al menos aplazable en su tratamiento, del "cómo". Las especificaciones corresponden al componente lógico; las implementaciones corresponden al componente de control.

De identificar y tratar separadamente cada componente en un programa tendríamos, de acuerdo con Kowalski, la ventaja de modificar el programa mientras preservamos su semántica. Ya que el no-determinismo es una de las partes que conforman el componente de control de un programa lógico, uno de nuestros objetivos es la manipulación del no-determinismo aplicando transformaciones que preserven corrección.

1.4 Taxonomías de familias de algoritmos

Una vez descrito el contexto que utilizamos para transformar programas, a continuación presentamos uno de los resultados laterales de este proceso.

1.4.1 Enfoque taxónomico de la derivación de algoritmos

La propuesta de examinar un conjunto de programas para aprender los principios que son comunes a cada uno de estos programas tiene su formulación explícita en [Par78]. La idea esencial de esta propuesta es que concentrarse en una familia de algoritmos en vez de uno sólo hace que el costo de mejoras y mantenimiento se reduzca.

Las derivaciones que presentamos en esta tesis están inmersas en un sistema deductivo de transformación. Cada derivación que desarrollamos genera un programa específico que implementa un algoritmo en particular, a pesar de que cada derivación comienza esencialmente desde el mismo programa. Sin embargo, existen conexiones entre algunos pasos de distintas derivaciones. Esto nos ha llevado a formular un *enfoque taxonómico* de la transformación de programas, en donde la clasificación de los algoritmos se lleva a cabo basándonos en *sucesiones de transformaciones de programas.* Obtenemos estas sucesiones de transformaciones de programas al aplicar ciertas reglas de transformación a un programa inicial.

Las diferencias entre las sucesiones de transformación generan distintos programas, y cada diferencia se genera al tomarse una decisión específica que corresponde a una parte esencial de un algoritmo (como es ejemplificado en el caso de algoritmos de ordenamiento en [CD80, Dar78], de [RS83] para el caso de algoritmos de gráficas, de [Möl93a] para ambos temas, de [Möl93b] para algoritmos de gráficas y de apuntadores, y de [WZ96] para el caso de algoritmos de casamiento múltiple de cadenas). Conforme construimos una sucesión de transformación de un programa, cada paso de transformación justifica de manera formal la eficiencia (bajo un modelo de ejecución) que podemos obtener. Por este medio, además, es posible identificar cuáles podrían ser las partes que más se prestan a una posible mecanización y cuáles no, lo que nos provee de valiosa información para distinguir las partes más heurísticas en el diseño de un algoritmo de aquellas otras que son mecanizables y que además mejoran la eficiencia del algoritmo dado.

1.4.2 Estructuración de conocimiento y taxonomías

Nuestra taxonomía de algoritmos de casamiento de cadenas estructura una serie de conocimientos acerca de este tipo de algoritmos. Estos conocimientos, en las presentaciones tradicionales de estos algoritmos, generalmente se presentan de manera desligada y aislada [Aho90, HS91, Ste94, Gus99], aunque existen importantes intentos para relacionar e integrar diversos algoritmos. En el libro de Bird [BW88], por ejemplo, se habla de una *síntesis de algoritmos*, o bien de *un desarrollo formal de programas*, idea más ampliamente realizada en el libro de Bird y de de Moor [BdM97]. Aún en un nivel imperativo, el libro

de Crochemore y Rytter [CR94] intenta enlazar la presentación de un nuevo algoritmo con algún otro ya conocido. En particular, Bird y de Moor enfatizan que una *álgebra de programación*, en el sentido lato de un sistema de manipulación algebraica, debe seguir un enfoque similar (al menos en espíritu) a los desarrollos de algoritmos de ordenamiento dados por Darlington en [Dar78], de manera que este enfoque nos permita el desarrollo sistemático de programas.

1.4.3 Reuso de desarrollos y taxonomías

Hemos notado que algunas sucesiones de transformaciones de programas comparten algunas características, lo que nos ha llevado a la idea de *reuso de derivaciones*. El concepto de reuso es una constante en el campo de la transformación de programas [PV91]. La ventaja de encontrar componentes de reuso en la metodología transformacional consiste, principalmente, en que las implementaciones finales son obtenidas por re-derivación. En general, tal como se afirma en [Pai96], el costo del desarrollo y el mantenimiento de los programas disminuye si al menos parte de las especificaciones, las transformaciones, y las derivaciones pueden reutilizarse convenientemente.

1.5 Algoritmos de casamiento de cadenas

Habiendo descrito ya nuestras herramientas principales de trabajo, en la siguiente subsección planteamos nuestro caso de estudio principal: el de los algoritmos de casamiento de cadenas.

Los algoritmos de casamiento de cadenas tienen una gran importancia dentro de los campos tales como el procesamiento de textos y la genética. Además, dada su bien definida área de estudio, representan un campo ideal en donde podemos aplicar algunas técnicas y análisis propios de las ciencias de la computación. En esta tesis mostramos cómo el desarrollo formal de cierta clase de algoritmos de cadenas está relacionado con la creación de algoritmos que tienen implementaciones eficientemente ejecutables. Para ello, nos hemos abocado a considerar nuestro estudio dentro del área de la programación lógica, y en particular, dentro del punto de vista transformacional. Usaremos la transformación de programas lógicos para derivar algoritmos de casamiento de cadenas comenzando desde versiones de fuerza bruta de los mismos, para obtener versiones eficientes tales como las dadas por los algoritmos de casamiento de cadenas de Boyer y Moore, y de Knuth, Morris y Pratt.

1.6 La derivación automática de programas

Dado el problema de la eficiencia de un programa las optimizaciones automáticas tienen un enorme beneficio, particularmente en el ámbito declarativo, ya que frecuentemente los lenguajes declarativos tienen múltiples capas de interpretación, lo que puede elevar el costo de computación en varios órdenes de magnitud. En la deducción parcial, por ejemplo, el objetivo principal es la obtención *automática* de programas eficientes cuando un subconjunto propio de los parámetros que sirven de entrada a este programa ya es conocido previamente a la ejecución del mismo.

Nuestra contribución, en esta dirección, radica en haber enriquecido una estrategia, conocida como la *estrategia D*, basada esencialmente en el algoritmo de determinización de autómatas por subconjuntos, de tal manera que la nueva estrategia pueda tratar el problema de derivar la parte de búsqueda del algoritmo de casamiento de cadenas de Boyer y Moore, y además de pasar la prueba, ahora ya tradicional, basada en la derivación de la parte de búsqueda del algoritmo de Knuth, Morris y Pratt. Por lo tanto, la nueva estrategia que propusimos es una estrategia ajustada a pasar una prueba más difícil que la usual.

1.6.1 Derivación de la parte de búsqueda del algoritmo de Knuth, Morris y Pratt

Tradicionalmente, el algoritmo de Knuth, Morris y Pratt ha servido como la prueba de fuerza de los evaluadores o los deductores parciales [CD89, GB91, GS95, Kur88, GK93, HSG98, ADR02], pese al hecho de que este algoritmo es raramente utilizado en la práctica. En efecto, normalmente el algoritmo preferido en la práctica es el algoritmo de Boyer y Moore, o alguna de sus variantes. En la siguiente subsección hacemos algunos comentarios acerca del algoritmo Boyer y Moore.

1.6.2 Derivaciones de la parte de búsqueda del algoritmo de Boyer y Moore

En [PS90], Partsch y Stomp utilizan la derivación del algoritmo de Boyer y Moore como un ejemplo de que la transformación de programas puede tratar con algoritmos en donde la sencillez es sacrificada en aras de la eficiencia. Una segunda derivación aparece en [Pep91], con una notación más ágil que la de Partsch y Stomp al nombrar las cadenas involucradas. Un resultado comparable al nuestro [HR01] en cuanto al objetivo de trabajar con el algoritmo de Boyer y Moore vía métodos transformacionales apareció en [ACDM01], aunque

aquí los autores utilizan la programación funcional como el ambiente para la escritura de sus programas. Trabajo más reciente aparece en [FKG02], en donde se deriva la parte de búsqueda de una variante del algoritmo de Boyer y Moore, en un resultado paralelo al que apareció en nuestro artículo [HR03].

1.7 Justificación de la derivación de algoritmos ya existentes

Uno de los objetivos de esta tesis es la obtención de las partes esenciales que conforman a algunos algoritmos de casamiento de cadenas. Sin embargo, algunos de estos algoritmos tienen ya algunos años de existencia, y su diseño estuvo al margen de una metodología basada en la transformación de programas. Por lo tanto, surge naturalmente la pregunta: ¿cuál sería el interés de derivar algoritmos que existen desde hace algún tiempo?

El estudio de la metodología de desarrollo de programas es reciente: generalmente, se considera que nació en la fecha del establecimiento de la problemática o crisis del *software*, a finales de los sesenta, en el siglo pasado [You79]. Como ya hemos comentado, la situación en esa época urgía a considerar la escritura de programas como una actividad tecnológica con métodos precisos y no, como hasta entonces se estaba realizando, con una informalidad e imprecisión que hacía que los programas fueran especialmente susceptibles de fallas y errores.

La derivación de algoritmos por métodos transformacionales y la transformación de programas nos proveen de herramientas metodológicas para escribir programas correctos partiendo desde especificaciones que dan solución a un problema. Por lo tanto, en la transformación de programas no se trata sólo de la obtención de tal o cual algoritmo, sino también del *proceso* que se sigue para esa obtención. El algoritmo que se deduzca, si bien interesante e importante por sí mismo, es sólo un caso de estudio más en el que los métodos de transformación de programas se aplican. Cada algoritmo exigirá un contexto para su derivación y, en el caso ideal, podrá derivarse automáticamente al menos una variante significativa de este algoritmo.

En la transformación de programas, los casos de estudio proveen del material que permitirá, en un futuro, contar con herramientas confiables y poderosas para tener la certeza de que derivamos programas correctos y de que abarcamos, en la práctica, cualquier problema que la teoría previamente nos explique que es resoluble. A este respecto, la filosofía de la construcción de programas por transformación señala que debemos hallar los principios tecnológicos que subyacen en la derivación de algoritmos y programas, y que estos principios sólo pueden hallarse al tratar diversos ejemplos que sean representativos de su respectiva área [Par86].

En el estado actual de la transformación de programas nuestro estudio de casos individuales de algoritmos, que presumiblemente sean lo suficientemente importantes, nos permitirá en un futuro establecer toda una metodología que apoye la derivación (o la imposibilidad de una derivación) de algoritmos y programas correctos a partir de especificaciones cuya corrección es fácilmente justificable.

1.8 Aportaciones de esta tesis

El núcleo de aportaciones de esta tesis las podemos ubicar en los siguientes rubros:

- en el campo de la transformación de programas y la derivación de algoritmos, los Capítulos 5 y 6 forman una parte del artículo [HR01], y presentan el núcleo de nuestras investigaciones en lo referente a la derivación manual de las partes de búsqueda del algoritmo de Boyer y Moore, y del algoritmo de Knuth, Morris y Pratt, junto con la respectiva derivación de algunas variantes de estos algoritmos;
- en el campo que involucra la idea de *reuso* y *taxonomías*, en el Capítulo 7 presentamos algunos resultados que enlazan nuestras derivaciones previas, y que también forman parte del artículo [HR01]. El hecho de que algunas partes en las derivaciones sean comunes nos permitió establecer un nexo transformacional entre el algoritmo de Boyer y Moore, y el de Knuth, Morris y Pratt;
- en el campo de la deducción parcial, en el Capítulo 8 presentamos una derivación automática de la parte de búsqueda de una variante del algoritmo de Boyer y Moore, y esta derivación está basada en una modificación de una estrategia de Pettorossi, Proietti y Renault, quienes elaboran su estrategia de acuerdo con el algoritmo de determinización de autómatas. Los resultados que presentamos en este capítulo están aceptados para su publicación en [HR03].

En el Capítulo 9 mostraremos trabajo relacionado al nuestro, y en el Capítulo 10 damos más detalles de nuestras aportaciones, así como la actualidad de nuestras investigaciones y las probables vertientes de investigación para el futuro.

1.9 Resumen de esta tesis

A continuación resumimos formalmente cada capítulo de que consta esta tesis.

En el Capítulo 2 presentamos los fundamentos de la programación lógica que serán necesarios para los propósitos de esta tesis.

En el Capítulo 3 mostramos las herramientas en las que basaremos el desarrollo de nuestras sucesiones de transformación. En particular, resumimos los hechos relevantes de la transformación de programas lógicos.

En el Capítulo 4 presentamos los dos algoritmos más sobresalientes en el área de casamiento de cadenas, junto con algunas variantes menores. Uno de los algoritmos que aquí presentamos es el algoritmo de Boyer y Moore, y el otro es el de Knuth, Morris y Pratt. Estos algoritmos preprocesan una de sus dos entradas, mientras la otra sólo será conocida en tiempo de ejecución.

En el Capítulo 5 derivamos la parte de búsqueda de algunas variantes del algoritmo de casamiento de cadenas de Boyer y Moore. Además, presentamos algunos resultados adicionales que relacionan lo que llamaremos *derivaciones determinísticas* con la funciones auxiliares que utiliza el algoritmo de Boyer y Moore.

En el Capítulo 6 derivamos la parte de búsqueda del algoritmo de casamiento de cadenas de Knuth, Morris y Pratt. Nuevamente relacionamos las derivaciones determinísticas con otra función auxiliar, conocida como *la tabla next*, que el algoritmo de Knuth, Morris y Pratt utiliza. Un reuso de los desarrollos que nos permitieron derivar las variantes del algoritmo de Boyer y Moore nos permitirá, a su vez, obtener un pre-algoritmo que ya incorpora algunas de las optimizaciones propias del algoritmo de Knuth, Morris y Pratt.

En el Capítulo 7 mostraremos la interrelación entre, por un lado, las derivaciones de las variantes del algoritmo Boyer y Moore, y por otro, la derivación del algoritmo de Knuth, Morris. Estableceremos esta relación basándonos en un enfoque transformacional.

En el Capítulo 8 establecemos algunos resultados de la transformación automática de programas. Durante la derivación de algunas variantes del algoritmo de Boyer y Moore nos percatamos de que había ciertas partes de nuestro trabajo que tenían un carácter algorítmico. Decidimos, en consecuencia, explorar las posibilidades de mecanizar la derivación de una variante simplificada del algoritmo de Boyer y Moore, y extendimos una estrategia de transformación de programas lógicos que utiliza definiciones que constan de varias cláusulas.

El Capítulo 9 muestra trabajo relacionado a nuestro tema.

En el Capítulo 10 presentamos algunas conclusiones, y finalizamos.

Notas acerca de nuestra terminología

Debido a la continua evolución de las Ciencias de la Computación, mucha de la terminología que utilizaremos todavía no está completamente bien establecida, por lo que hemos decidido presentar unas notas al respecto.

"Implementar" es un verbo que la Real Academia Española (RAE) [RAE01] ya ha aceptado, aunque con uso restringido a las áreas de informática. "Computación", que por cierto funciona las más de las veces como sinónimo de "informática", es una palabra que todavía no es de uso generalizado entre los hispanohablantes, pero en México tiene una amplia aceptación. Aplicamos una nota similar a "computadora". En ocasiones utilizamos el verbo "indexar" y sus derivados; son palabras ya aceptadas por la RAE. Para señalar la calidad de *completo* la RAE ha aceptado la palabra "completitud", pero también la palabra "completez" y "compleción". Utilizaremos, por uniformidad, "completitud". Introducimos la palabra "subsumir" como un neologismo derivado de la palabra técnica inglesa "to subsume", y "subsumción" como una palabra derivada en castellano. "Cache" es una palabra técnica de uso común en el diseño de *hardware*. Utilizamos la palabra "cadena" como una traducción de la palabra inglesa "string". Traducimos "lookahead" por "previsión", y "lookbehind" por "provisión" (en el sentido del efecto de proveer: preparar, reunir las cosas necesarias para un fin [RAE]). La palabra "corrección" es la palabra aceptada por la RAE para señalar la calidad de *correcto* (aunque tal vez "correctez" sería una palabra más apropiada). "Programar" y "programación de computadoras" tienen las acepciones que uno esperaría en los terrenos de la informática, de acuerdo con la RAE. La palabra "hardware" (definido como el conjunto de los componentes que integran la parte material de una computadora) también ya está permitida por la RAE, así como "software" (conjunto de programas), aunque con la acotación de que son voces inglesas.

Algunos nombres de funciones y de predicados estarán en inglés por que así son más fácilmente identificables en la literatura.

Convenciones de presentación

En la presentación y desarrollo de mi tesis he adoptado las siguientes convenciones: a) he enumerado (al menos) el material matemático al que hago referencia en el texto, por lo que es posible que material relevante en otros contextos quede sin enumerar; y b) frecuentemente he utilizado el nominativo de la primera persona en plural y sus derivados, pues considero que la lectura requiere de una colaboración entre un(a) lector(a) y un(a) escritor(a).

Capítulo 2

Programación lógica

A continuación presentamos un resumen de la programación lógica. Los textos que seguimos son, en la parte de lógica, [Kle52, Sch67] y [Doe94], y en la parte de programación lógica, los de [Llo87] y [Apt97].

Nuestro objetivo en esta introducción a la programación lógica no es tanto ser exhaustivos, sino más bien acordar las convenciones notacionales y terminológicas que adoptaremos de ahora en adelante. En particular, nos basamos en la teoría presentada en [Llo87], capítulos 1, 2, y parte del 3.

Suponemos que una teoría de primer orden está definida como en [Sch67], por lo que entramos de lleno en los conceptos propios de la progamación lógica.

2.1 Sintaxis de los programas lógicos

Un *término* es o una variable o una expresión de la forma

$$ft_1t_2\dots t_n \qquad (n \geq 0)$$

en donde f es un *símbolo de función* y cada t_i es a su vez un término. La *aridad* de $ft_1t_2\dots t_n$ es n. Si $n = 0$, el término f es una *constante*, y denotamos esta constante por la letra c, posiblemente con subíndices. Siguiendo la tradición, si $n > 0$ escribiremos $ft_1t_2\dots t_n$ como $f(t_1, t_2, \dots, t_n)$.

Un *átomo* es un objeto sintáctico de la forma

$$pt_1t_2\dots t_n \qquad (n \geq 0)$$

en donde p es un *símbolo de predicado* y cada t_i es un término. El término t_i se halla en el *i-ésimo argumento* del átomo p. Si $n = 0$ entonces p carece de argumentos, y p es entonces una *proposición*. Similarmente al caso de un término, y siguiendo la tradición, si $n > 0$ entonces escribiremos $pt_1t_2 \ldots t_n$ como $p(t_1, t_2, \ldots, t_n)$.

Una *literal* es un átomo o la negación de un átomo. Una literal es *positiva* si es un átomo. Una literal es *negativa* si es la negación de un átomo.

Una cláusula es un conjunto de literales. Una *cláusula definida* es una cláusula con exactamente una literal positiva. Una *cláusula de Horn* es una cláusula con a lo más una literal positiva; en lo que sigue llamaremos a las cláusulas de Horn simplemente cláusulas. Suponemos que las variables involucradas en una cláusula están cuantificadas universalmente.

Escribimos una cláusula C en la forma de una implicación:

$$p \leftarrow q_1, \ \ldots, \ q_n \qquad (n \geq 0) \qquad (2.1)$$

A cada q_i la llamamos una *submeta*. Si $n = 0$ la cláusula de Horn correspondiente se llama un *hecho* o *cláusula unitaria*. Si omitimos p en la implicación (2.1) tenemos una *pregunta*. Un átomo aislado o la conjunción de algunos átomos es una *meta*. Si omitimos p y $n = 0$ en (2.1) tenemos la *cláusula vacía*, $\Box$, comprendida como una *contradicción* (en realidad, que se ha inferido una contradicción).

Supongamos que tenemos una cláusula definida C de la forma (2.1). Cada q_i, $i \in \{1, \ldots, n\}$ es llamada una *submeta* de la cláusula definida C. Nos referimos al átomo p como la *cabeza* de la cláusula C, mientras que a la sucesión $q_1, \ldots, q_n$ la llamamos el *cuerpo* de la cláusula C. Por medio de *head*(C) y *body*(C) denotamos la cabeza y el cuerpo de una cláusula C, respectivamente.

Un *programa lógico* es un conjunto de cláusulas definidas.

Si P es un programa lógico, la *definición* de un predicado p es el subconjunto de cláusulas de P que tienen al símbolo de predicado p en la cabeza.

Una *expresión* es o un término, o una meta, o una cláusula, o un programa. Sea e una expresión. Denotamos por *term*(e) el conjunto de todos los términos de e, y por *var*(e) al conjunto de todas las variables que aparecen en e. Una expresión e sin variables (*var*(e) $= \emptyset$) se llama un *expresión aterrizada*.

Una *sustitución* θ es un mapeo finito del conjunto de variables al conjunto de términos, el cual asigna a cada variable X en su dominio un término t diferente de X. Escribimos una sustitución como

$$\{X_1/t_1, \ldots, X_n/t_n\}$$

en donde $X_1, \dots, X_n$ son variables distintas, cada t_i es un término y para cada i en $\{1, \dots, n\}$, $X_i \neq t_i$. Cuando $n = 0$, hablamos de la *sustitución identidad* ϵ. La composición y la asociatividad de sustituciones son el resultado de considerar a una sustitución como un mapeo. Utilizaremos yuxtaposición para denotar la aplicación de una sustitución θ a una expresión e: $e\theta$. Una expresión e_1 es una *instancia* de la expresión e_2 si existe alguna sustitución θ tal que $e_1 = e_2\theta$. Una cláusula es una *variante* de una cláusula C si difiere de C en a lo más el nombre de sus variables. Ahora definimos la noción de *subsumción* [Mah88, páginas 634–5]. Sean $C_1 : p_1 \leftarrow q_1, \dots, q_n$ y $C_2 : p_2 \leftarrow r_1, \dots, r_m$ dos cláusulas. La cláusula C_2 *subsume* a la cláusula C_1 (o la cláusula C_1 *es subsumida* por la cláusula C_2) si existe una sustitución θ tal que $p_1 = p_2\theta$ y $\{r_1\theta, \dots, r_m\theta\} \subseteq \{q_1, \dots, q_n\}$.

Dados dos términos t_1 y t_2, una sustitución θ es un *unificador* de t_1 y t_2 si $t_1\theta = t_2\theta$. De existir un unificador θ para los dos términos t_1 y t_2 decimos que ellos son *unificables*. Ahora bien, sean θ y τ unificadores. Decimos que θ *es más general que* τ si existe una sustitución η tal que $\tau = \theta\eta$, en donde $\theta\eta$ es la operación de composición de sustituciones. Llamamos a θ un *unificador más general (umg) de* t_1 *y* t_2 si θ es más general que cualquier unificador de t_1 y t_2.

2.2 Semántica de los programas lógicos

Sea P un programa, formado por las cláusulas C_1, C_2, $\dots$, C_n. El *universo de Herbrand* de P, UH_P, es el conjunto todos los términos sin variables que aparecen en los predicados que conforman a P (exceptuamos a *true*, *false*, e $=$). La *base de Herbrand* de P, $\mathcal{B}_P$ es el conjunto de todos los átomos con símbolos de predicados definidos en P y que tienen en sus argumentos elementos de UH_P. Necesariamente la base de Herbrand se compone de átomos aterrizados. Una *interpretación de Herbrand* de P es un subconjunto de la base Herbrand de P: $I \subseteq \mathcal{B}_P$.

Definimos ahora la noción de *verdad* en un programa P con respecto a una interpretación de Herbrand $I \subseteq \mathcal{B}_P$.

Una fórmula cerrada de primer orden ϕ es una *fórmula* en I, o I es un *modelo* de ϕ, $I \models \phi$, si y sólo si:

1. ϕ es el átomo *true*;

2. ϕ es un átomo de la forma $t = t$ para cualquier término aterrizado t;

3. ϕ es un átomo y $\phi \in I$;

4. ϕ es de la forma $\neg p$, y $p \notin I$;

5. ϕ es una cláusula aterrizada

$$p \leftarrow p_1,\ p_2,\ \ldots,\ p_n \tag{2.2}$$

   y p es verdadera en I o, para al menos un p_i, $p_i \notin I$;

6. una cláusula C que contenga variables es *verdadera* en I si y sólo si, para toda sustitución θ de las variables en C por elementos de $\mathcal{B}$, $p\theta$ está en I;

7. el programa

$$P = \{C_1, C_2, \ldots, C_n\} \tag{2.3}$$

   es *verdadero* en I si y sólo si cada cláusula C_i es verdadera en I.

En las fórmulas (2.2) y (2.3) la coma, por lo tanto, es el operador de conjunción $\wedge$ de la lógica de primer orden. Para una fórmula ϕ, ϕ es *falsa* en I si no es el caso que ϕ sea verdadera en I. Así pues, una interpretación de Herbrand I es un modelo de Herbrand de un programa P si y sólo si $I \models \forall X_1, \ldots, \forall X_n.P$, en donde $var(P) = \{X_1, \ldots, X_n\}$.

Una interpretación de Herbrand I en la cual un programa P es verdadero es un *modelo de Herbrand* de P.

La intersección de todos los modelos de Herbrand de un programa lógico P es también un modelo de P [Llo87, página 36], y lo llamamos el *modelo mínimo de Herbrand* (es mínimo en el sentido de la inclusión de conjuntos). Tal modelo es comúnmente considerado como *el significado del programa lógico P*.

Una caracterización del modelo mínimo de Herbrand de un programa lógico P es

$$M(P) = \{r(t_1, \ldots, t_n) \mid r(t_1, \ldots, t_n) \text{ es un átomo aterrizado y } P \models r(t_1, \ldots, t_n)\}$$

Dos programas P_1 y P_2 son *equivalentes* si $M(P_1) = M(P_2)$.

La descripción de un programa lógico P de acuerdo con su modelo mínimo de Herbrand forma la *interpretación declarativa* de P.

2.3 La regla de resolución

Para calcular efectivamente el modelo mínimo de Herbrand de un programa lógico dado necesitamos las siguientes consideraciones algorítmicas.

Sea G_1, q, G_2 una meta, en donde G_1 y G_2 son conjuntos de literales y q es un átomo, y sea C una cláusula. Sea $p \leftarrow H$ una variante de C, H un conjunto de literales, con su conjunto de variables disjunto del conjunto de variables de G_1, q, G_2. Supongamos que p y q unifican, con el umg θ. Entonces $(G_1, H, G_2)\theta$ es llamado *un resolvente-SLD de G_1, q, G_2 y C con respecto de q.* Llamamos a q el *átomo seleccionado* de G_1, q, G_2. Para resumir este proceso, escribimos

$$G_1, q, G_2 \overset{\theta}{\underset{C}{\Longrightarrow}} (G_1, H, G_2)\theta$$

y llamamos a este proceso un *paso de resolución* o un *paso de derivación-SLD.* A $p \leftarrow H$ la llamamos la *cláusula de entrada.*

A una sucesión de pasos de resolución la llamamos una *derivación-SLD.*

Una sucesión maximal

$$G_0 \overset{\theta_1}{\underset{C_1}{\Longrightarrow}} G_1 \quad \cdots \quad G_n \overset{\theta_{n+1}}{\underset{C_{n+1}}{\Longrightarrow}} G_{n+1} \quad \cdots$$

de pasos de resolución es llamada una *derivación-SLD* de G_0 en P si

1. $G_0, \ldots, G_{n+1}, \ldots$ son metas, cada una o vacía o con un átomo seleccionado;

2. $\theta_1, \ldots, \theta_{n+1}, \ldots$ son sustituciones;

3. $C_1, \ldots, C_{n+1}, \ldots$ son cláusulas de P,

y para todo paso la siguiente condición (*estandarización aparte*) se cumple (ver [Apt97, página 49] para mayores detalles): la cláusula de entrada empleada es disjunta en sus variables de la pregunta inicial G_0 y de las sustituciones y las cláusulas de entrada utilizadas en los anteriores pasos.

La *longitud* de una derivación-SLD es el número de pasos de derivación-SLD utilizados en ella. Las derivaciones-SLD pueden ser *finitas* o *infinitas.* Las finitas tienen una importancia computacional particular.

Consideremos una derivación-SLD finita

$$\chi := \quad G_0 \overset{\theta_1}{\underset{C_1}{\Longrightarrow}} G_1 \overset{\theta_2}{\underset{C_2}{\Longrightarrow}} \cdots \quad \overset{\theta_n}{\underset{C_n}{\Longrightarrow}} G_n$$

de una meta $G := G_0$. Una *regla de selección* $\mathcal{R}$ es una función que permite seleccionar un átomo en G_i en cada paso de resolución. La derivación-SLD χ es llamada *exitosa* si $G_n = \Box$. La restricción $(\theta_1 \ldots \theta_n)|_{var(G)}$ de la composición $\theta_1 \ldots \theta_n$ es llamada una *respuesta* de G y $G\theta_1 \ldots \theta_n$ es llamada una *instancia calculada* de G. La derivación-SLD

χ es *fallida* si G_n es no vacía y ninguna cláusula de P es aplicable al átomo seleccionado de G_n.

Un *árbol-SLD* para G en P vía una regla de selección $\mathcal{R}$ es un árbol tal que

- sus ramas son derivaciones-SLD de G en P vía $\mathcal{R}$, y
- todo nodo G con átomo seleccionado p tiene exactamente un descendiente inmediato para toda cláusula C de P que es aplicable a p, y este descendiente es un resolvente de G y C con respecto de p.

La programación lógica tiene una *interpretación procedural*, y gracias a esta interpretación es posible utilizar la lógica clausal para calcular. En consencuencia, entran a colación algunos aspectos tales como la *eficiencia* de un programa lógico, pues la eficiencia de un programa lógico no es una noción inherente al mismo, sino que resulta de un mapeo del programa a un proceso sobre una máquina [Mee89]. El planteamiento de lo que llamamos la eficiencia de un programa lógico es como sigue.

En las pp. 274–277 de [Dev90], Deville presenta la noción de *complejidad* aplicada a programas lógicos. Las conclusiones que Deville obtiene al respecto es que la complejidad de un programa lógico bajo la regla de resolución-SLD está directamente relacionada con la regla de computación y la estrategia de búsqueda. En consecuencia, una medida de la complejidad de un problema lógico sería el número de ramas exitosas que son necesarias para satisfacer una meta. Sin embargo, en la práctica (es decir, en una implementación de Prolog), el número de los nodos visitados en un arbol de derivación-SLD influye de manera decisiva en el funcionamiento de un programa. Nuestro enfoque abarca ambos puntos de vista: por un lado, nos interesan las optimizaciones brindadas a nivel teórico utilizando una regla de computación justa y una estrategia de búsqueda apropiada (quizás implementada por medio de un meta-intérprete); por otro lado, consideraremos que un programa lógico de casamiento de cadenas P_1 es mejor que otro P_2 si P_1 realiza menos comparaciones que P_2 para una cadena dada (pues una medida usual para medir el desempeño de los algoritmos de procesamiento de cadenas es el número de comparaciones entre los caracteres de la cadena patrón y la cadena texto). Si tomamos como una submeta en lo específico a una ecuación del tipo $X = a$, nuestra restricción nos lleva a solicitar que X sea instanciada la menor cantidad de veces posible (pues en cada instancia hay una comparación explícita). Optimizaciones de más bajo nivel requieren un conocimiento de la implementación disponible de Prolog (para aprovechar, por ejemplo, el mecanismo de indexamiento en una sucesión de cláusulas).

La introducción de consideraciones procedurales nos lleva a complementar la parte teórica, y consideramos una ventaja la lectura *dual* de los programas lógicos, como fórmulas

y como procedimientos, por lo que es necesario siempre estar atento a esta dualidad. En efecto, una simple cláusula como la siguiente:

$$p \leftarrow q,\ q \tag{2.4}$$

difiere enormemente de esta otra

$$p \leftarrow q \tag{2.5}$$

en términos procedurales y en relación a la pregunta $\leftarrow p$, aun cuando ambas tengan la misma lectura declarativa. La razón es que, dado que podemos resolver a q en la cláusula (2.4) con respecto a una cláusula de entrada $q \leftarrow H$, tenemos que resolverla dos veces, mientras que en la cláusula (2.5) sólo debemos resolverla una vez. Independientemente de lo difícil que resulte resolver H (si ello es posible), la cláusula (2.4) tiene un costo de *ejecución* doble, por lo que hablaremos de la *eficiencia* de un programa lógico de acuerdo con la interpretación procedural del mismo, y por medio del manejo de las siguientes funciones de costo: la longitud de las derivaciones SLD y el tamaño de los árboles SLD o bien, en el caso de programas lógicos que casan cadenas, en el número de comparaciones, símbolo a símbolo, necesarias para encontrar tales cadenas; por ejemplo, el programa lógico P_1 es más eficiente que el programa P_2 en relación a una pregunta $\leftarrow q$ si q es resuelta con una derivación-SLD en P_1 más corta que una derivación-SLD en P_2.

Una *especificación lógica* es un conjunto de átomos aterrizados. Dada una especificación *Esp*, un programa lógico *satisface correctamente (o es correcto con respecto a)* la especificación *Esp* si

$$M(P) = Esp \tag{2.6}$$

El conjunto *Esp* representa el *significado deseado* de un programa lógico. Un programa lógico P es *parcialmente correcto* con respecto a una especificación *Esp* si $M(P) \subseteq Esp$. Un programa P es *completo* con respecto a una especificación *Esp* si $Esp \subseteq M(P)$. Decimos que un programa P es *totalmente correcto* con respecto a la especificación *Esp* si P es parcialmente correcto y completo con respecto a *Esp*; por consiguiente, en este caso, un programa lógico totalmente correcto P satisface los requerimientos del significado deseado *Esp*.

Observemos que, dado que P_1 y P_2 son dos programas lógicos que satisfacen una especificación *Esp*, entonces P_1 y P_2 son equivalentes.

Cada programa P_1 tiene un significado $M(P_1)$, y así, dado cualquier otro programa P_2, llamaremos a P_1 *parcialmente correcto* con respecto a P_2 si $M(P_1) \subseteq M(P_2)$. Decimos que P_1 es *completo* con respecto a P_2 si $M(P_2) \subseteq M(P_1)$. Decimos que un programa P_1 es *totalmente correcto* con respecto P_2 si P_1 y P_2 son equivalentes. Abreviaremos *totalmente correcto* a *correcto*.

2.4 La teoría estándar de igualdad

A continuación presentamos la teoría estándar de igualdad. Posteriormente veremos qué papel juega esta teoría en el desarrollo de las derivaciones de algunos algoritmos de casamiento de cadenas.

2.4.1 Teoría axiomática de igualdad

La *teoría estándar de igualdad* consta de los siguiente axiomas:

Axioma de reflexividad:

$$x = x \leftarrow \ .$$

Axioma de simetría:

$$x = y \leftarrow y = x \ .$$

Axioma de transitividad:

$$x = z \leftarrow x = y,\ y = z \ ,$$

más los siguientes *esquemas de axiomas*:

Axiomas de sustitutividad en funciones: Para cada símbolo de función f de aridad n $(n \geq 1)$:

$$f(x_1, \ldots, x_n) = f(y_1, \ldots, y_n) \leftarrow x_1 = y_1,\ \ldots,\ x_n = y_n$$

Axiomas de sustitutividad en predicados: Para todo símbolo de predicado p de aridad n $(n \geq 1)$:

$$p(x_1, \ldots, x_n) \leftarrow p(y_1, \ldots, y_n),\ x_1 = y_1,\ \ldots,\ x_n = y_n$$

Nuestro uso de esta teoría nos permitirá hacer explícitas algunas operaciones que de otra manera quedarían implícitas dados los mecanismos de unificación.

2.4.2 La teoría de desigualdad de Clark

Dado que posteriormente también tendremos que trabajar con desigualdades, es necesario establecer esquemas que axiomaticen la teoría de igualdad ampliada con algunos axiomas para la relación de desigualdad ([Cla78, página 304]).

1. $f(x_1, \ldots, x_m) \neq g(x_1, \ldots, x_m)$ en donde $m, n \geq 0$ y f, g son símbolos de función distintos (el caso de constantes distintas está incluido aquí);

2. $\tau(x) \neq x$, en donde $\tau(x)$ es cualquier término que contiene a la variable x, y x es una variable libre en este término.

Utilizando esta teoría es posible, para el caso de términos aterrizados, prescindir de la utilización explícita de la *negación por falla*, pues si garantizamos que a y b son distintos elementos de un alfabeto dado (alfabeto en el sentido de un conjunto distinguible de símbolos), a la inecuación $a \neq b$ la podemos resolver inmediatamente. Ya que generalmente reducimos nuestro uso de negación sólo a inecuaciones (en el sentido de la negación de ecuaciones), prescindiremos de la negación por falla en lo que resta de esta tesis.

2.5 Listas y abreviaciones

En nuestro trabajo acerca de algoritmos de casamientos de cadenas utilizaremos frecuentemente a las *listas*. Las listas, informalmente, son sucesiones finitas de objetos. Formalmente, una lista es un término de una forma específica, como se detalla a continuación.

Definimos una lista por medio de inducción estructural como sigue:

1. $[\,]$ es una lista;

2. $[H \mid T]$ es una lista si T es una lista.

El término H es la *cabeza* de la lista, en tanto que T es la *cola* de lista. A la lista $[\,]$ la llamamos la lista vacía.

Inductivamente, abreviamos $[H_0 \mid [H_1, \ldots, H_n \mid T]]$ por medio de $[H_0, H_1, \ldots, H_n \mid T]$ y a $[H_1, H_2, \ldots, H_n \mid [\,]]$ por medio de $[H_1, H_2, \ldots, \mid H_n]$.

Cuando no se origine ninguna confusión, *omitiremos las comas entre los elementos de una lista.* Por ejemplo, escribimos la lista $[a, a, b, c]$ como $[aabc]$. Hacemos esto tanto por concisión como por facilidad en la manipulación formal de nuestras cadenas. Otra abreviación que utilizaremos cuando nos sea conveniente es la siguiente: $o_i : o_j$ abreviará la sucesión de objetos $o_i, o_{i+1}, \ldots, o_j$ si $i < j$ y la sucesión de objetos $o_i, o_{i-1}, \ldots, o_j$ si $j < i$. Si $i = j$, $o_i : o_j = o_i$. Para los objetos o_i y o_{i+1}, en secuencia ascendente o descendente, omitimos el símbolo : entre ellos y los yuxtaponemos, si no hay posibilidad de confusión. Para el caso ascendente, por ejemplo, ponemos $o_i o_{i+1}$ en vez de $o_i : o_{i+1}$. Una abreviación más que utilizamos es la siguiente: sea p una cadena; con $o_i : o_j = p$

abreviamos la secuencia de ecuaciones $o_i = p_i, \ldots, o_j = p_j$, sujeta a la condición de que la secuencia sea ascendente o descendente.

Posteriormente (en el capítulo 5) estableceremos una correspondencia natural entre las listas y las cadenas (sucesiones finitas de símbolos tomadas de lo que posteriormente definiremos como un *alfabeto*). Otras formulaciones de listas son posibles, pero creemos que la abstracción que utilizamos es suficiente para nuestros propósitos.

Capítulo 3

Transformación de programas lógicos

Una vez establecidas las bases esenciales de nuestro tema, procedemos ahora a presentar un sistema estándar de transformación de programas lógicos que está basado en el uso de *reglas y estrategias de transformación.*

3.1 Transformación de programas

La idea de la transformación de programas como un método para la construcción de programas se remonta a MacCarthy [McC63b]. Posteriormente, Wegbreit en [Weg76], llegó a trabajar con tácticas transformacionales que ahora serían reconocidas como modernas. Sin embargo, el establecimiento explícito de un sistema de transformación basado en reglas fue dado por Burstall y Darlington en [DB76]. A continuación describimos en cierto detalle este sistema.

Burstall y Darlington idearon un sistema para transformar programas escritos como ecuaciones recursivas de primer orden. Es en el artículo [BD77] en donde explícitamente ellos establecen las bases para la transformación de programas, de acuerdo con las siguientes propuestas:

- transformar programas que sean *declarativos*;
- transformar estos programas por medio de las reglas de *desdoblado, de doblado, y de la introducción de definiciones*; y, finalmente,

- idear *estrategias* que nos guíen en la aplicación de estas reglas.

Las reglas de Burstall y Darlington aplicadas al método de transformación de programas por desdoblado/doblado tienen una versión apropiada para las ecuaciones de primer orden, y su justificación teórica fue dada por Kott [Kot85]. Siendo el sistema de Burstall y Darlington adecuado en cuanto a la preservación de una semántica dada, Kott advirtió, sin embargo, acerca de los peligros de la aplicación de la regla de doblado, que podía conducir a programas no-terminantes. De cualquier forma, el sistema de Burstall y Darlington resultó ser de una sorprendente sencillez y aplicabilidad, además de ser esencialmente transferible al caso de la programación lógica, y a continuación lo describiremos en términos de uno de sus componentes principales: *las reglas de transformación*.

3.2 Las reglas de desdoblado/doblado: introducción

Enunciamos las reglas del sistema de Rod M. Burstall y John Darlingon [BD77] a continuación.

1. La regla de introducción de definiciones: Introduzca una nueva ecuación de tal manera que su lado izquierdo no sea una instancia del lado izquierdo de ninguna otra ecuación ya existente.

2. La regla de instanciación: Deduzca una instanciación de una ecuación ya existente.

3. La regla de desdoblado: Si $E \Leftarrow E'$ y $F \Leftarrow F'$ son ecuaciones y existe alguna ocurrencia en F' de una instancia de E, reemplácela por la correspondiente instancia de E', obteniendo F''; entonces agregue la ecuación $F \Leftarrow F''$.

4. La regla del doblado: Si $E \Leftarrow E'$ y $F \Leftarrow F'$ son ecuaciones y existe alguna ocurrencia en F' de una instancia de E', reemplácela por la correspondiente instancia de E, obteniendo F''; entonces agregue la ecuación $F \Leftarrow F''$.

Otras reglas auxiliares son la regla de abstracción (basada en la regla **where**, y las leyes de las expresiones primitivas (por ejemplo, la asociatividad de $+$)).

Como ya mencionamos, una *estrategia* es una sucesión bien definida de aplicaciones de reglas de transformación. Las estrategias son tan importantes que Pettorossi y Proietti, en [PP02], formularon la seudo-ecuación:

La derivación de programas = reglas + estrategias.

Una estrategia dada por Burstall y Darlington para dirigir el proceso de transformación es la siguiente:

1. Escriba las definiciones necesarias.
2. Instancie.
3. Para cada instanciación, desdoble repetidamente. A cada estado de desdoblado
   (a) intente aplicar las leyes de las primitivas y realice una abstracción **where**,
   (b) doble repetidamente.

Intuitivamente, podemos describir las reglas de desdoblado, doblado, y de introducción de definiciones como sigue [MH88]: el desdoblado es la sustitución de un nombre por la estructura que este nombre denota. Más en el espíritu de la programación lógica, el desdoblado de un átomo en el cuerpo de una cláusula es la sustitución del átomo por una instancia de la definición del mismo. En este sentido, es parecido a la resolución de este átomo pero considerando al mismo tiempo varias cláusulas de entrada, por lo que generalmente el desdoblado sustituye una cláusula de un programa por varias.

En cuanto al doblado, éste resulta de reconocer que una configuración dada D es similar a una configuración previa E, por lo que podemos escribir una definición recursiva denotando la configuración D y hacemos que E dependa de D. Si las dos configuraciones no son idénticas, escribimos una definición generalizada que abarque ambas configuraciones. Notamos aquí también la estrecha relación existente entre la regla de doblado y la introducción de definiciones, pero en general la introducción de definiciones puede hacerse de manera independiente. En el contexto de la programación lógica, además, entendemos la regla del doblado como una sustitución de algunos átomos en el cuerpo de una o varias cláusulas por otro átomo, mientras que entendemos la introducción de definiciones como la añadidura de algunas cláusulas, que satisfacen algunas restricciones, al programa actual.

En el campo de la programación lógica la regla del doblado, a diferencia de la del desdoblado, ha estado sujeta a cierta problemática técnica tanto como regla lógica como por el manejo de las variables involucradas, y en efecto, algunas formulaciones de esta regla han tenido serios errores, pero ya han sido corregidos [GS91, She92]. Para nuestros propósitos, hemos tomado la versión de las reglas de transformación de programas lógicos de [PP98b].

Antes de abordar la parte técnica de las reglas de transformación en la programación lógica, a continuación estableceremos un conjunto de preguntas que forman parte de la problemática actual del tipo de sistemas de transformación que toman estas reglas como básicas.

3.3 Problemática actual de la transformación de programas por el método de desdoblado/doblado

En el trabajo de Burstall y Darlington afloraron algunas consideraciones relacionadas con la transformación de programas que a la fecha no tienen una respuesta definitiva en general:

Pregunta 1 *¿En qué orden aplicamos la reglas de transformación a un programa dado?*

La respuesta a esta pregunta tiene relación con las estrategias que guían a un proceso de transformación. Puede ser que una estrategia sea completamente automática, de manera que un usuario sólo intervenga al inicio de la transformación (al dar el programa inicial), o bien puede ser semi-automática, tomando algunas decisiones sobre la marcha, o bien, finalmente, puede ser manual, en el sentido de que la computadora auxilie a un usuario en las operaciones repetitivas o de difícil cálculo, pero las decisiones más relevantes deben tomarse siempre a un meta-nivel. Por lo tanto, otras preguntas relacionadas que derivan de la anterior son las siguientes: (a) ¿en qué medida es automatizable la transformación de programas?, y (b) ¿en qué medida, si la automatización no es posible, deseamos limitar la intervención del programador?

Pregunta 2 *¿Cómo caracterizamos el conjunto de programas transformables de acuerdo con cierta estrategia?*

Normalmente diseñamos una estrategia para derivar un algoritmo específico. En una próxima etapa, surge la pregunta de si nuestra estrategia tiene la capacidad de derivar un conjunto de algoritmos que incluya el algoritmo inicial y otros más. En consecuencia surge la cuestión de conocer específicamente la clase de algoritmos a los que una estrategia dada puede aplicarse. Normalmente, tal identificación conlleva el conocimiento de la "esencia" (las partes sustanciales) de los algoritmos que tal estrategia explota en las derivaciones en las que la aplicamos.

Pregunta 3 *¿Es nuestro conjunto de reglas* completo*?*

Como muchos procesos científicos, la transformación de programas es *abductivo*. Dado un algoritmo A, y una sistema de transformación E, surge la preguna si con las herramientas provistas por E es posible derivar a A. Si no es posible derivar a A, normalmente lo que deseamos es adaptar a E a su nuevo contexto, generando quizás un nuevo sistema E' del que E sea un subsistema. De esta manera, los sistemas de transformación se enriquecen,

volviéndose más poderosos. En contraste, E' puede no prestarse a una fácil mecanización, y obligar a que sea el (la) programador(a) quien provea de las decisiones más sutiles en el desarrollo de un algoritmo dado.

Pregunta 4 *¿Con respecto a qué semántica debemos preservar corrección?*

En el caso de la programación lógica, una semántica de manejo accessible y todavía lo suficientemente expresiva como para ser útil es la semántica del modelo mínimo de Herbrand. Existen, sin embargo, otro tipo de semánticas que, si bien no pueden no estar dentro de las descripciones iniciales dadas por una especificación, brindan sin embargo significados que pueden ser útiles para el análisis de programas en lo que respecta a la mejora de eficiencia (por ejemplo, la semántica de fallas), por lo que siempre es necesario especificar claramente la semántica que deseamos preservar en el proceso de transformación de programas lógicos.

Pregunta 5 *¿Bajo qué condiciones un programa mejora vía la transformación de programas?*

Esta pregunta es de las más difíciles de contestar en la tranformación de programas. Primero, exigimos que exista un conjunto de funciones de costo de acuerdo con una semántica operacional dada. (Si la semántica operacional es abstracta, sin embargo, todavía nos podemos involucrar en una problemática relacionada con la eficiencia en una implementación efectiva del algoritmo dado.) Segundo, aún cuando exista una semántica operacional dada, la mayoría de las estrategias en la literatura no garantizan un efectivo decremento en el costo de ejecución de los programas finales. En ocasiones nuestra única guía es un conjunto de *heurísticas* bajo las cuales, y esperanzadoramente, el programa final tendrá el incremento de eficiencia deseado.

Pregunta 6 *¿Cuáles son los límites teóricos de optimización alcanzables vía la transformación de programas?*

Normalmente esta pregunta depende del problema a mano y no sólo del sistema de transformación de programas utilizado. Algunos programas que tiene una formulación de orden exponencial pueden transformarse en programas de orden lineal, lo que para propósitos prácticos es más que suficiente (el ejemplo clásico en la literatura es del cálculo de los números de Fibonacci). Cabe señalar, sin embargo, que algunos algoritmos tienen una complejidad inherente que no es posible mejorar.

A continuación abundamos en la descripción de una clase específica de sistemas de transformación.

3.4 El sistema de transformación por desdoblado-doblado en programación lógica

La transformación de programas nos permite obtener programas eficientes a partir de programas ineficientes pero correctos. La transformación de programas es una herramienta soporte de la programación, tanto si ésta es declarativa [BD77] como procedural [CP89].

La idea esencial [BD77] de la transformación de programas consiste en crear una sucesión de programas

$$P_0, P_1, \ldots, P_n \tag{3.1}$$

de acuerdo con un conjunto de ciertas reglas $\mathcal{T}$, llamadas *reglas de transformación*. En esta sucesión si se ha dotado a P_0 de un cierto valor con respecto a una función semántica, el programa final, P_n, debe tener el mismo valor semántico bajo esta función y además poseer algunas cualidades deseables pero inicialmente ausentes en P_0. La preservación, en este sentido, de tal semántica es lo que denominamos la *preservación de corrección total*; normalmente las reglas sólo preservan la corrección parcial y frecuentemente es necesario dar una demostración explícita de la preservación de corrección total.

Normalmente, la propiedad principal del programa inicial P_0 consiste en la claridad de su corrección, en tanto que el programa final P_n es más eficiente y generalmente tiene una construcción complicada, de manera que la transformación de programas nos permite la *deducción* de algoritmos, es decir, la demostración de corrección de los algoritmos respecto a un significado deseado. Esta deducción se logra por medio de la aplicación, en cada paso de una sucesión de transformación, de alguna regla que preserva la semántica del programa inicial.

3.5 El método de desdoblado/doblado: detalles técnicos

Para realizar las derivaciones especializadas de nuestros algoritmos, utilizaremos la transformación de programas por *desdoblado/doblado (unfold/fold) en el contexto de la programación lógica* [Hog81, TS84]. Algunos estudios de este sistema de transformación se encuentran en [TS84] pero están expuestos con un formato moderno y resultados actualizados en [PP94, PP96a] y [PP98b]. En [Zhu94] están algunos resultados relacionados con las limitaciones de los sistemas de desdoblado/doblado, pero el autor omite en su tratamiento la importante regla de la introducción de definiciones, lo que equivale a descartar completamente todo tipo de transformación de programas guiada por el usuario.

Las siguientes son algunas de las operaciones principales que se utilizan para transfor-

mar un programa lógico [PP94]:

Regla 1 (Desdoblado.) *Sea P_k un programa tal que $P_k = \Gamma_1 \cup \{C\} \cup \Gamma_2$ en donde C es la cláusula*

$$H \leftarrow S_1, A, S_2 \ ,$$

Γ_1 y Γ_2 son subconjuntos de cláusulas de P_k, A es una literal positiva, y $S_1, S_2 \subset body(C)$. Supongamos que:

1. *en el subconjunto $\{B_1, \ldots, B_n\} \subset P_j$, en donde $n > 0$, y $0 \leq j \leq k$, las cláusulas B_i son tales que*

$$A \text{ unifica con } head(B_1)\theta_1 \ ,$$
$$\ldots \ ,$$
$$A \text{ unifica con } head(B_n)\theta_n \ ,$$

   y cada θ_i, para $i = 1, \ldots, n$, es el umg entre A y B_i, y

2. *C_i es la cláusula*

$$(H \leftarrow S_1, body(B_i), S_2)\theta_i \ ,$$

   para $i = 1, \ldots, n$. Al desdoblar C con respecto de la submeta A usando la definición *P_j derivamos las cláusulas $\{C_1, \ldots, C_n\}$ y obtenemos el nuevo programa:*

$$P_{k+1} = \Gamma_1 \cup \{C_1, \ldots, C_n\} \cup \Gamma_2$$

En consecuencia, el desdoblado es básicamente la realización de un paso de resolución de una cláusula p con cada una de las cláusulas que conforman la definición de una submeta que pertenezca al cuerpo de p.

A continuación definimos la regla del doblado. Tamaki y Sato [TS84] fueron unos de los primeros en estudiar el sistema de desdoblado/doblado en los programas lógicos. Estos autores consideraron esta operación de doblado con respecto a un predicado definido por una sola cláusula. En la formulación que ellos dieron del doblado había, sin embargo, un error lógico, que fue remediado por Gardner y Shepherdson en [GS91] y por Shepherdson en [She92]. Posteriormente, Gergatsoulis y Katzouraki [GK94] consideraron el doblado con respecto a predicados definidos por varias cláusulas. Más recientemente, Pettorossi, Proietti y Renault [PPR97a] explotaron este doblado "extendido" para derivar la parte de búsqueda del algoritmo de Knuth, Morris y Pratt. Nosotros también utilizaremos este tipo de doblado, con las enmiendas de Gardner y Shepherdson ya incorporadas, en lo que sigue:

Regla 2 (Doblado.) *Sea*

$$P_k = \Gamma_1 \cup \{C_1, \ldots, C_n\} \cup \Gamma_2$$

y

$$B = \{D_1, \ldots, D_n\}$$

en donde $B \subset P_j$*, para cierto índice* j *tal que* $0 \leq j \leq k$*. Supongamos que existen un átomo* A *y dos conjuntos de submetas* S_1 *y* S_2 *tales que para cada* i*, con* $1 \leq i \leq n$*, existe una sustitución* θ_i *que satisface las siguientes condiciones:*

1. C_i *es una variante de la cláusula*

$$H \leftarrow S_1, body(D_i), S_2 \ ,$$

2. $A = head(D_i)\theta_i$*,*

3. *para toda cláusula* D *de* P_j *que no está en* $\{D_1, \ldots, D_n\}$*,* $head(D)$ *no es unificable con* A*, y*

4. *para toda variable* X *en el conjunto* $var(D_i) \setminus var(head(D_i))$*, tenemos que* $X\theta_i$ *es una variable que no ocurre en* (H, S_1, S_2)*, y la variable* $X\theta_i$ *no ocurre en el término* $Y\theta_i$*, para cualquier* Y *que ocurre en* $body(D_i)$ *y que es diferente de* X*.*

Al doblar las cláusulas $C_1, \ldots, C_n$ utilizando las cláusulas $D_1, \ldots, D_n$ en P_j *derivamos la cláusula*

$$C : H \leftarrow S_1,\ A,\ S_2 \ ,$$

y obtenemos el nuevo programa

$$P_{k+1} = \Gamma_1 \cup \{C\} \cup \Gamma_2$$

El doblado es una operación casi inversa al desdoblado, en el sentido de que, si la aplicamos al transformar el programa P_k para obtener al programa P_{k+1} doblando las cláusulas $C_1, \ldots, C_n$ con respecto al predicado p, entonces al desdoblar P_{k+1} con respecto a p obtenemos el programa P_k (o más bien, una variante del programa P_k). En el caso proposicional, de hecho, el desdoblado y el doblado funcionan como inversos uno del otro: De $p \leftarrow a,\ b$ y de

$$a \leftarrow a_1 \tag{3.2}$$

$$a \leftarrow a_2 \tag{3.3}$$

obtenemos

$$p \leftarrow a_1,\ b \tag{3.4}$$

$$p \leftarrow a_2,\ b \tag{3.5}$$

pero también de

$$p \leftarrow a_1,\ \boxed{b} \tag{3.6}$$

$$p \leftarrow a_2,\ \boxed{b} \tag{3.7}$$

y de la cláusula (3.2) y (3.3) podemos obtener la cláusula $p \leftarrow a,\ b$. (Las submetas encerradas en un rectángulo indican la submeta en común que tienen las cláusulas (3.6) y (3.7).)

No obstante, si primero desdoblamos no siempre podemos doblar de tal manera que obtengamos el conjunto inicial de cláusulas (aún salvo variantes), como ejemplificamos a continuación, en un ejemplo dado por Pettorossi y Proietti en [PP98a]:

De $h \leftarrow g(X, b),\ r(X)$ y de $g(a, Y) \leftarrow p(Y)$ desdoblamos para obtener: $h \leftarrow p(b),\ r(a)$, pero si doblamos $h \leftarrow p(b), r(a)$ utilizando $g(a, Y) \leftarrow p(Y)$ obtenemos la cláusula $h \leftarrow g(a, b),\ r(a)$, que es diferente de $h \leftarrow g(X, b),\ r(X)$.

Una de las razones por las que la aplicación de la regla de doblado realiza algunas mejoras importantes en la eficiencia de un programa es la siguiente: el doblado fusiona *a lo ancho* los nodos que están en un mismo nivel en el árbol-SLD de derivación, evitando que algunas computaciones que son comunes a varias trayectorias de derivación provoquen una ineficiencia en nuestro programa (ver Figs. 3.1 y 3.2). Además, el doblado permite utilizar definiciones previas, por lo que podemos utilizar el doblado después de desdoblar, y así crear versiones recursivas y autocontenidas de algunos predicados.

Regla 3 (Introducción de una definición.) *La introducción de una definición consiste en la introducción de n cláusulas B_i, para $i = 1, \ldots, n$, para obtener un nuevo programa P_{k+1} y suponiendo que el símbolo de predicado de head(B) no pertenece a ningún programa $P_0, \ldots, P_k$. La introducción de una definición es* no recursiva *si todos los símbolos que ocurren en los cuerpos de las cláusulas B_i ocurren en P_k también.*

Dado que podemos aplicar la regla de introducción de una definición varias veces, una simple extensión a esta regla es permitir que varias definiciones sean introducidas a la vez.

Regla 4 (Eliminación de definiciones.) *Obtenemos un programa P_{k+1} al borrar del programa P_k las cláusulas que constituyen la definición del predicado q, q no ocurre en P_0 y en donde ningún predicado en P_k depende de q.*

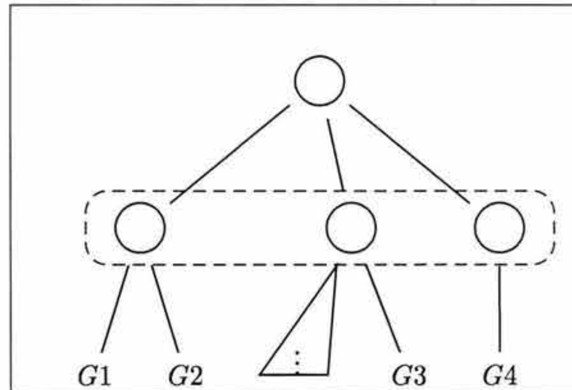

Figura 3.1: Una identificación de nodos que son comunes a un nivel.

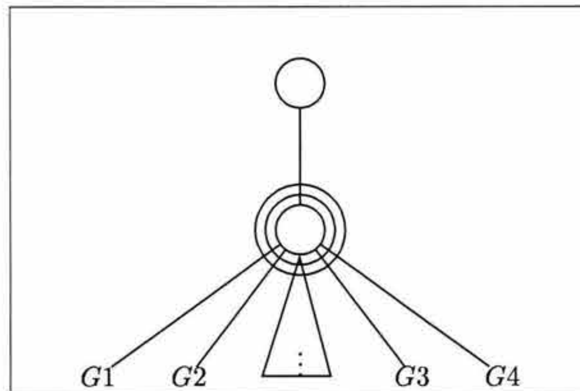

Figura 3.2: Efecto del doblado sobre los nodos que son comunes a un nivel.

Regla 5 (Reemplazo de submetas.) *La regla de reemplazo de submetas requiere una definición previa: Dos metas G_1 y G_2 son equivalentes con respecto a la semántica $\mathcal{S}$ y al programa P si y sólo si $\mathcal{S}[P, \leftarrow G_1] = \mathcal{S}[P, \leftarrow G_2]$. Una vez presentada esta definición, ahora enunciamos la regla de reemplazo de submetas: Sea $\mathcal{S}$ una función semántica, y G_1, G_2 dos metas equivalentes con respecto a $\mathcal{S}$ y P_k, y*

$$C : H \leftarrow S_1, G_1, S_2$$

una cláusula de P_k. Al reemplazar *G_1 por G_2 en la cláusula C derivamos la cláusula*

$$C' : H \leftarrow S_1, G_2, S_2$$

y obtenemos P_{k+1} de P_k al reemplazar C por C'.

Ahora bien, la regla de reemplazo de submetas depende de la función semántica $\mathcal{S}$, pero no de la cláusula en donde hacemos el reemplazo. A continuación formulamos una regla de reemplazo de submetas con respecto a *un conjunto de variables*, de tal manera que las submetas involucradas están relacionadas al resto de la cláusula vía estas variables.

Equivalencia de submetas con respecto a un conjunto de variables. Supongamos que el programa P_k consta de las cláusulas $C_1, \ldots, C_n$, y sea $\mathcal{S}$ una función semántica. Consideremos las siguientes dos cláusulas:

Cláusulas 1

$$D_1 : newp_1(X_1, \ldots, X_m) \leftarrow G_1$$

$$D_2 : newp_2(X_1, \ldots, X_m) \leftarrow G_2$$

en donde $newp_1$ y $newp_2$ son símbolos (distintos) de predicados que no ocurren en P_k, $V = \{X_1, \ldots, X_m\}$ es un conjunto de m variables, y G_1 y G_2 son dos submetas. G_1 y G_2 son *equivalentes con respecto a $\mathcal{S}$, P_k* y V, lo que escribimos como $G_1 \equiv_V G_2$, si y sólo si

$$\mathcal{S}[(C_1, \ldots, C_n, D_1), \leftarrow newp_1(X_1, \ldots, X_m)] =$$
$$\mathcal{S}[(C_1, \ldots, C_n, D_2), \leftarrow newp_2(X_1, \ldots, X_m)]$$

Regla 6 *El siguiente conjunto de reglas representa algunos de nuestros recursos de manipulación lógica:*

1. *rearreglo de cláusulas,*

2. *factorización,*

3. *borrado de cláusulas subsumidas, y*

4. *borrado de cláusulas con un cuerpo que falla finitamente.*

En [PP94] los autores definen la igualdad por $X = X \leftarrow$ en todo programa de la sucesión $P_0, \ldots, P_k$. Nosotros, en cambio, definimos la igualdad, para cada programa, por el conjunto de axiomas que conforman la teoría estándar de la igualdad, ya que pretendemos alcanzar un enfoque que involucre sólo cláusulas de Horn, aminorando los efectos laterales de la unificación.

Una vez que hemos definido las principales reglas que aplicamos en nuestro proceso de transformación, el siguiente objetivo complementario es considerar la corrección que las reglas preservan al pasar de un programa a otro.

Para las definiciones siguientes necesitamos algunas notaciones:

Sea $\mathcal{P}$ un conjunto de programas, $\mathcal{Q}$ un conjunto de preguntas, y

$$\mathcal{S}\mathcal{P} \times \mathcal{Q} \rightarrow (\mathbf{D}, \subseteq)$$

una función semántica.

Definición 1 (Corrección parcial de una sucesión de transformación) *Una sucesión de transformaciones $P_0, \ldots, P_k$ de programas en $\mathcal{P}$ es* parcialmente correcta *con respecto a $\mathcal{S}$si para toda pregunta Q en $\mathcal{Q}$, en donde Q contiene sólo símbolos de predicado que ocurren en P_0, tenemos que*

$$\mathcal{S}[P_k, Q] \subseteq \mathcal{S}[P_0, Q].$$

Definición 2 (Corrección total de una sucesión de transformación) *Una sucesión de transformaciones $P_0, \ldots, P_k$ de programas en $\mathcal{P}$ es* totalmente correcta *con respecto a $\mathcal{S}$si para toda pregunta Q en $\mathcal{Q}$, en donde Q contiene sólo símbolos de predicado que ocurren en P_0, tenemos que*

$$\mathcal{S}[P_k, Q] = \mathcal{S}[P_0, Q]$$

.

En particular, nos interesa sólo la semántica del *modelo mínimo de Herbrand de un programa P*, a la que denotamos por $M(P)$, y simbólicamente tenemos los dos casos que

corresponden a la noción de corrección: Dado un conjunto de átomos $\mathcal{A}$, entendemos la corrección total con respecto a $\mathcal{A}$ como

$$\forall A \in \mathcal{A} : M(P_0) \models A \Leftrightarrow M(P_k) \models A \tag{3.8}$$

y entendemos la corrección parcial como

$$\forall A \in \mathcal{A} : M(P_0) \models A \Leftarrow M(P_k) \models A \tag{3.9}$$

en donde P_0 es el programa inicial de una sucesión de transformación y P_k $(k \geq 1)$ es un programa de la sucesión.

Una regla de transformación es *parcialmente correcta* con respecto a la semántica $\mathcal{S}$ si para toda sucesión de transformación $P_0, \ldots, P_k$ la cual es parcialmente correcta con respecto a $\mathcal{S}$ y para todo programa P_{k+1} obtenido de P_k por una aplicación de esa regla, tenemos que la sucesión de transformaciones extendida $P_0, \ldots, P_k, P_{k+1}$ es parcialmente correcta con respecto a $\mathcal{S}$.

Una regla de transformación es *totalmente correcta* con respecto a $\mathcal{S}$ si para toda sucesión de transformación $P_0, \ldots, P_k$ la cual es totalmente correcta con respecto a $\mathcal{S}$ y para todo programa P_{k+1} obtenido de P_k por una aplicación de esa regla, tenemos que la sucesión de transformaciones extendida $P_0, \ldots, P_k, P_{k+1}$ es totalmente correcta con respecto a $\mathcal{S}$. En lo que sigue, por "corrección" nos referiremos a "corrección total".

Definición 3 *Una sucesión de transformación $P_0, \ldots, P_n$ construida usando un conjunto de reglas $\mathcal{R}$ es* reversible *si existe una sucesión de transformaciones*

$$P_n, Q_1, \ldots, Q_k, P_0 \ , \quad k \geq 0,$$

que puede construirse usando las reglas de $\mathcal{R}$.

Notación: $P_k \xrightarrow{R} P_{k+1}$ denota la aplicación de la regla R al programa P_k para obtener el programa P_{k+1}.

Un paso de transformación es reversible *si $P_k \xrightarrow{R_a} P_{k+1}$ y es el caso que existe una regla R_b tal que $P_{k+1} \xrightarrow{R_b} P_k$.*

Sea $\mathcal{S}$ una función semántica, y $\mathcal{R}$ un conjunto de reglas de transformación. Si P_1 y P_2 son dos programas dados, y $P_1 \xrightarrow{R_a} P_2$ es reversible $P_2 \xrightarrow{R_b} P_1$ con $R_b \in \mathcal{R}$ y siendo R_b una regla correcta, entonces $P_1 \xrightarrow{R_a} P_2$ es correcta.

Si las reglas en $\mathcal{R}$ son parcialmente correctas con respecto a $\mathcal{S}$, entonces cualquier sucesión reversible de transformaciones que usa reglas en $\mathcal{R}$ es correcta con respecto a $\mathcal{S}$.

Definición 4 (Doblado local o in-situ) *A un paso de doblado lo llamamos* doblado local *si, refiriéndonos a la regla 2, tenemos que:* $P_k = P_j$ *(tomamos las cláusulas para doblar del último programa) y*

$$\{C_1, \ldots, C_n\} \cap \{D_1, \ldots, D_n\} = \emptyset.$$

Lo relevante del doblado local es que es una regla reversible, y por lo tanto, correcta.

Teorema 1 (Corrección con respecto al modelo mínimo de Herbrand) *Supongamos que* $P_0, \ldots, P_n$ *es una sucesión de transformaciones de programas definidos y construidos por la aplicación de las siguientes reglas:*

1. *desdoblado,*
2. *doblado local,*
3. *introducción de definiciones,*
4. *eliminación de definiciones,*
5. *rearreglo de submetas,*
6. *simplificación por factorización de submetas,*
7. *reemplazo de submetas independientes, y*
8. *reemplazo de cláusulas.*

Entonces $P_0, \ldots, P_n$ *es correcta con respecto al modelo mínimo de Herbrand.*

Este es el Teorema 4.4.2 de [PP98b] llamado ahí Primer Teorema de Corrección con respecto al modelo mínimo de Herbrand, y que resume varios resultados de la literatura.

3.5.1 Ejemplos aplicados al problema de ordenamiento

Presentamos un par de ejemplos del sistema de transformación de programas lógicos recién expuesto; ambos ejemplos tratan con el *problema de ordenamiento*, una aplicación clásica de la transformación de programas.

Consideremos, pues, el problema de ordenamiento sobre los números enteros, junto con una relación de orden "a menor que b" (relación escrita por razones de claridad como $\leq$,

de donde podemos derivar por definición la notación $<, >$, y $\geq$, con el significado usual de estos símbolos). Formulemos el siguiente programa lógico que implementa un algoritmo de fuerza bruta bajo la semántica operacional de la resolución-SLD, y el cual resuelve el problema de ordenamiento de manera directa:

$$sort(L_1\,,\ L_2) \leftarrow perm(L_1, L_2),\ ord(L_2)$$

en donde $perm(L_1, L_2)$ se cumple si la lista L_2 es una permutación de la lista L_1, y $ord(L_2)$ se cumple si la lista L_2 es una lista ordenada de manera no-decreciente de acuerdo con la relación de orden $\leq$.

Dadas las definiciones de *perm* y *ord*, el algoritmo de fuerza bruta resuelve correctamente el problema de ordenamiento, pero es considerablemente ineficiente. Supongamos que una lista L tiene longitud n, y que L no tiene elementos repetidos. Dada que el número de permutaciones de L es $n!$, la complejidad en tiempo de este algoritmo es $O(n!)$, algo definitivamente impráctico, aún para valores no muy grandes de n. Nuestro objetivo, en lo siguiente, es formular algoritmos más eficientes que el algoritmo de fuerza bruta por medio de la transformación de programas lógicos.

Comenzamos presentando el algoritmo de ordenamiento de Tamaki y Sato [TS84].

Sea L una lista. Un conjunto de axiomas que define una la relación $perm(L_1, L_2)$, que es verdadera si L_2 es una permutación de L_1, es la siguiente:

Cláusulas 2 (Perm1, basada en *insert*)

$$perm([\,]\,,\ [\,]) \leftarrow \tag{3.10}$$

$$perm([A \mid L_1]\,,\ L_3) \leftarrow perm(L_1\,,\ L_2),\ insert(A\,,\ L_2\,,\ L_3) \tag{3.11}$$

$$insert(A\,,\ L\,,\ [A \mid L]) \leftarrow \tag{3.12}$$

$$insert(A\,,\ [B \mid L_1]\,,\ [B \mid L_2]) \leftarrow insert(A\,,\ L_1\,,\ L_2) \tag{3.13}$$

Para completar la definición de *sort*, necesitaremos también una posible definición de *ord*:

Cláusulas 3 (Ord1, lineal)

$$ord([\,]) \leftarrow \tag{3.14}$$

$$ord([A]) \leftarrow \tag{3.15}$$

$$ord([A, B \mid L]) \leftarrow A \leq B,\ ord([B \mid L]) \tag{3.16}$$

Denotemos por $\{Y\}$ el conjunto de los elementos de una lista Y. Otra definición de posible de *ord*, más abstracta que la previamente dada y basada en el conjunto de elementos de una lista, es la siguiente:

Cláusulas 4 (Ord2, por subconjuntos)

$$ord([\,]) \leftarrow \tag{3.17}$$

$$ord([A]) \leftarrow \tag{3.18}$$

$$ord([A \mid L]) \leftarrow min(A,\, L),\ ord(L) \tag{3.19}$$

en donde $min(A,\, L)$ es verdadero si A es una cota inferior de los elementos de la lista L.

Comencemos ahora una derivación de un algoritmo de ordenamiento dada por Tamaki y Sato en [TS84, página 135], basándonos en la definición de *perm* dada por Perm1 y Ord1, siguiendo las definiciones dadas por las cláusulas en 2 y 3. Las definiciones iniciales en las que basamos nuestras derivaciones se interrelacionarán estrechamente con los programas finales que obtendremos [Dar78].

Observemos otra vez nuestro algoritmo de fuerza bruta, renombrando *sort* a *sort_ts*:

Programa 1 (Programa de fuerza bruta para ordenamiento)

$$sort_ts(L_1,\, L_2) \leftarrow perm(L_1,\, L_2),\ ord(L_2) \tag{3.20}$$

Desdoblando *perm* en el cuerpo de la cláusula (3.20) obtenemos:

Programa 2

$$\begin{aligned} &sort_ts([\,],\, [\,]) \leftarrow ord([\,]) \\ &sort_ts([A \mid L_1],\, L_3) \leftarrow perm(L_1,\, L_2), \\ &\qquad insert(A,\, L_2,\, L_3), \\ &\qquad ord(L_3) \end{aligned} \tag{3.21}$$

En lo siguiente, desdoblaremos sin previo aviso algunas submetas como $ord([\,])$ en el cuerpo de la cláusula (3.21). Como señalan Tamaki y Sato, $insert(t_1, t_2, t_3) \wedge ord(t_3) \ \vdash\ ord(t_2)$, para términos aterrizados t_1, t_2, y t_3. Por lo tanto, agregamos la submeta $ord(L_2)$ en el cuerpo de la cláusula (3.21):

$$\begin{aligned} sort_ts([A \mid L_1],\, L_3) \leftarrow\ & ord(L_2), \\ & perm(L_1,\, L_2), \\ & insert(A,\, L_2,\, L_3), \\ & ord(L_3) \end{aligned} \tag{3.22}$$

Doblando las submetas $ord(L_2)$ y $perm(L_1,\, L_2)$ con respecto a *sort_ts*, obtenemos un nuevo programa más eficientemente ejecutable que el programa de fuerza bruta:

Programa 3 (Tamaki y Sato)

$$sort_ts([\,],\ [\,]) \leftarrow \tag{3.23}$$

$$\begin{aligned} sort_ts([A \mid L_1],\ L_3) \leftarrow\ & sort_ts(L,\ L_2), \\ & insert(A,\ L_2,\ L_3), \\ & ord(L_3) \end{aligned} \tag{3.24}$$

De lo anterior, hemos obtenido un programa que implementa un algoritmo de orden $O(n^3)$ de un algoritmo de orden $O(n!)$. Ahora bien, el problema con este programa bajo la regla de computación de izquierda a derecha tipo Prolog es el siguiente: Dada una lista $[A \mid L]$, primero ordenamos L, luego insertamos a A en alguna posición de L y, finalmente, verificamos si la lista resultante está ordenada. Así, tendríamos un mejor desempeño si probáramos el orden la lista resultante en términos de la *lista de entrada*, en lugar de hacerlo con respecto a la *lista de salida*, lo que nos lleva a consideraciones relacionadas con el *modo* de un predicado. Más aún, estaríamos en mejor posición si usáramos la lista de entrada como nuestra única lista de trabajo (ya que es inmediatamente conocida), lo que a su vez nos lleva a los temas de deducción parcial, de los que trataremos más adelante en este capítulo.

En lo que sigue derivaremos otro algoritmo de orden $O(n^2)$ del Prog. 3, el algoritmo conocido como *el algoritmo de ordenamiento por inserción*. La derivación de este algoritmo es parte de nuestras investigaciones en la transformación de programas aplicadas al tema de ordenamiento.

Comenzando con el Prog. 3, renombremos el predicado *sort_ts* a *inssort*:

Programa 4

$$inssort([\,],\ [\,]) \leftarrow \tag{3.25}$$

$$\begin{aligned} inssort([A \mid L],\ L_3) \leftarrow\ & inssort(L,\ Z), \\ & insert(A,\ Z,\ L_3), \\ & ord(L_3), \\ & Z = L_1 +\!\!+ L_2 \end{aligned} \tag{3.26}$$

en donde suponemos que a la lista Z la dividimos en dos sublistas: $L1$ y $L2$, y de aquí en adelante utilizamos $+\!\!+$ para denotar la operación de concatenación entre listas en notación infija.

Ahora utilizamos las siguientes propiedades. Primero, notamos que

$$ord(L_1 +\!\!+ [A] +\!\!+ L_2) \quad \vdash \quad ord(L_1) \wedge ord(L_2) \wedge L_1 \lhd A \wedge A \lhd L_2 \tag{3.27}$$

donde, si C es un número y L es una lista de números, $C \lhd L$ denota que $C < D$, para todo número D tal que $D \in \{L\}$, y $L \lhd C$ denota que $D < C$, para todo D tal que $D \in \{L\}$.

Ahora reescribimos la cláusula (3.26) como sigue:

$$\begin{aligned} \mathit{inssort}([A \mid L],\, L_3) \leftarrow\ & \mathit{inssort}(L,\, L_1 +\!\!+ L_2), \\ & \mathit{insert}(A,\, L_1 +\!\!+ L_2,\, L_3), \\ & \mathit{ord}(L_3) \end{aligned} \tag{3.28}$$

La inserción de A dada por la submeta $ins(A,\, L_1 +\!\!+ L_2,\, L_3)$ da $L_3 = L_1 +\!\!+ [A] +\!\!+ L_2$, y ya que $ord(L_3)$ es verdadero, creamos una nueva definición que coloca correctamente a A en la lista L_3:

Cláusulas 5

$$\mathit{filter}(A,\, [\,],\, [\,],\, [\,]) \leftarrow \tag{3.29}$$

$$\mathit{filter}(A,\, [B \mid L],\, [B \mid L_1],\, L_2) \leftarrow B < A \wedge \mathit{filter}(A,\, L,\, L_1,\, L_2) \tag{3.30}$$

$$\mathit{filter}(A,\, [B \mid L],\, L_1,\, [B \mid L_2]) \leftarrow A \leq B \wedge \mathit{filter}(A,\, L,\, L_1,\, L_2) \tag{3.31}$$

$\mathit{filter}(A,\, L,\, L_1,\, L_2)$ es verdadero si $L = L_1 \cup L_2$, $L_1 \lhd A$, y $A \lhd L_2$.

Reemplazando la submeta (3.27), tenemos:

Programa 5

$$\mathit{inssort}([\,],\, [\,]) \leftarrow \tag{3.32}$$

$$\begin{aligned} \mathit{inssort}([A \mid L_0],\, L_3) \leftarrow\ & \mathit{inssort}(L_0,\, Z), \\ & \mathit{filter}(A,\, Z,\, L_1,\, L_2), \\ & L_3 = L_1 +\!\!+ [A] +\!\!+ L_2 \end{aligned} \tag{3.33}$$

o, equivalentemente, expresando el programa 5 usando *append*

Programa 6 (Algoritmo de ordenamiento por inserción)

$$\mathit{inssort}([\,],\, [\,]) \leftarrow \tag{3.34}$$

$$\begin{aligned} \mathit{inssort}([A \mid L],\, L_3) \leftarrow\ & \mathit{inssort}(L,\, Z), \\ & \mathit{filter}(A,\, Z,\, L_1,\, L_2), \\ & \mathit{append}(L_1,\, [A],\, Y), \\ & \mathit{append}(Y,\, L_2,\, L_3) \end{aligned} \tag{3.35}$$

Este algoritmo por inserción resuelve el problema de ordenamiento en un orden de complejidad $O(n^2)$. Algunas optimizaciones menores son todavía posibles (por ejemplo, podemos usar listas diferencia en lugar de *append*), pero no las exploraremos, esperando que los anteriores ejemplos brinden una idea de la potencia y métodos de la transformación de programas.

Un estudio general de algunos programas del tipo generar-y-probar, al que pertenece el programa de ordenamiento por fuerza bruta, se halla en [Smi87]. Aplicaciones a la programación lógica del esquema generar-y-probar están en [Bsa92] (nuestras derivaciones de los algoritmos de ordenamiento que presentamos son un poco más abstractas que las de [Bsa92]). Otro estudio teórico y una clasificación de los algoritmos de ordenamiento, dentro del paradigma de la programación funcional, se encuentra en [Aug98].

3.5.2 Especialización de programas y deducción parcial

A continuación damos una descripción de lo que se conoce como *evaluación parcial* [FN88]:

Consideremos una función f con dos parámetros d_1 y d_2. Supongamos que d_1 es un parámetro conocido, y que d_2 es un parámetro desconocido. Realicemos todas las operaciones involucradas en f que dependan de nuestro conocimiento de d_1, y mantengamos intactas las que dependen de d_2. De este proceso, obtenemos una nueva función f_{d_1} que tiene la propiedad siguiente:

$$f_{d_1}(d_2) = f(d_1, d_2) \tag{3.36}$$

(La fórmula (3.36) es similar al teorema S_n^m de Kleene [Kle52, página 342], Teo. XXIII, Capítulo XII: *Partial recursive functions*, aunque aquí la evaluación parcial no se dirige a la optimización de algoritmos, sino que está dada en el contexto del estudio teórico de las funciones parciales recursivas.) Trasladamos ahora este formalismo al caso de la programación lógica.

La *especialización de programas lógicos* es una técnica para optimizar programas (no necesariamente por transformación) que aprovecha la información de un contexto específico en el que un programa se ejecutará, lo que involucra por ejemplo la inferencia de modos [DW98], la eliminación de no-determinismo [MNL90], o la terminación [CC92]; por consiguiente cuando especializamos programas hacemos algo más que deducción parcial.

La deducción parcial [LS91], en efecto, es un caso particular de la *especialización de programas*. En la deducción parcial de programas lógicos nuestro objetivo es aprovechar la instanciación previa a la llamada de un predicado, análogamente al conocimiento previo de d_1 en la ecuación (3.36). La particularidad del argumento generalmente nos lleva a obtener ciertas optimizaciones del programa inicial [Gal93].

3.5.3 El desdoblado determinístico

La regla de desdoblado tiene la virtud de reflejar directamente el proceso computacional dentro de un programa lógico, y además la importante propiedad de preservar corrección total, pero hay al menos seis temas que complican su aplicación. Estos son:

1. en qué cláusulas aplicar el desdoblado;
2. con respecto a qué submetas desdoblar;
3. con respecto a qué definición de la submeta aplicar el paso de desdoblado;
4. cuándo desdoblar;
5. la relación que existe entre un paso de desdoblado y las simplificaciones que podríamos realizar [Lis94, página 484]; y,
6. cuándo detener un proceso repetitivo de desdoblado.

Ya que el desdoblado es una regla que nos permite hacer explícitos ciertos procesos de cálculo, en el caso de una derivación de un programa pudiera darse el caso de que introduzcamos predicados de "alto nivel" de los que nos tengamos que deshacer totalmente en un paso posterior. Si alguno de los argumentos de tal predicado tiene un término que decrementamos en tamaño a cada paso, es finito (como en el caso de las listas), y hay un caso base bien definido, la eliminación de tal predicado por un desdoblado sistemático la llamamos un *desdoblado total*.

Ejemplo: Por desdoblado total podemos eliminar *member* con respecto a su definición usual

$$member(A, [A \mid Ls]) \leftarrow \tag{3.37}$$
$$member(A, [B \mid Ls]) \leftarrow member(A, Ls) \tag{3.38}$$

en la siguiente cláusula:

$$p(X) \leftarrow member(X1, [a, b]),\ X = X1 * c.$$

lo que resulta en:

$$p(X) \leftarrow X = a * c \tag{3.39}$$
$$p(X) \leftarrow X = b * c \tag{3.40}$$

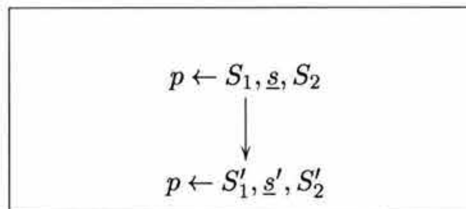

Figura 3.3: La idea principal en el desdoblado determinístico.

Note cómo cada vez que desdoblamos la lista Ls en $member(A, Ls)$, Ls decrementa su longitud en 1; el caso base es cuando $L = [\,]$, es decir, cuando la longitud de la lista L es 0.

En [GB91, página 323], se explican algunas de las virtudes del desdoblado determinístico. Definimos una *derivación determinística* como una aplicación cíclica, consecutiva, y finita de una sucesión de aplicaciones de la regla de desdoblado a una cláusula inicial. Las ventajas de las derivaciones determinísticas son:

1. el ahorro de pasos de resolución en el momento de ejecución del programa;
2. el dejar inalterada la conducta de la regla del retroceso del programa; y,
3. el no incremento el tamaño del programa (dado que no generamos cláusulas extras).

Un *trayectoria determinística* es una trayectoria en un árbol-SLD tal que a cada nodo de la trayectoria sólo corresponde otro nodo o ninguno.

En la Fig. 3.4 presentamos una trayectoria determinística y los nodos que son sucesores del nodo final en la trayectoria determinística, esto es, la trayectoria determinística y la etapa de prueba que permite obtener una derivación determinística.

3.6 Estrategias de transformación de programas

3.6.1 La estrategia S de Gallagher

En la estrategia S de Gallagher procedemos por *ciclos* de aplicaciones de algunas reglas de transformación. Dicha estrategia está establecida como sigue:

1. Inicialmente, nuestro programa tiene un predicado especializado con respecto a alguno o algunos de sus argumentos.

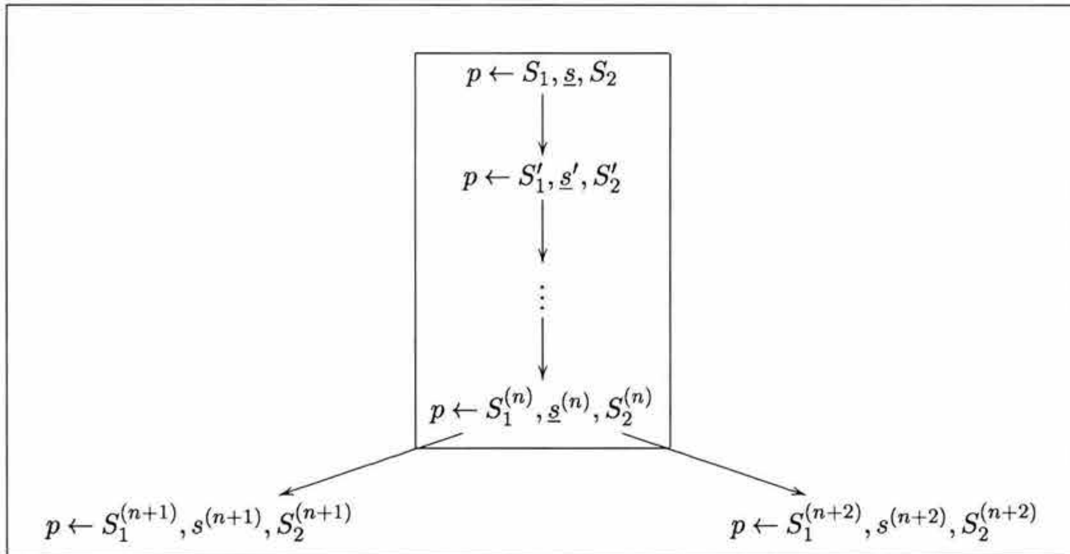

Figura 3.4: La idea principal en una derivación determinística.

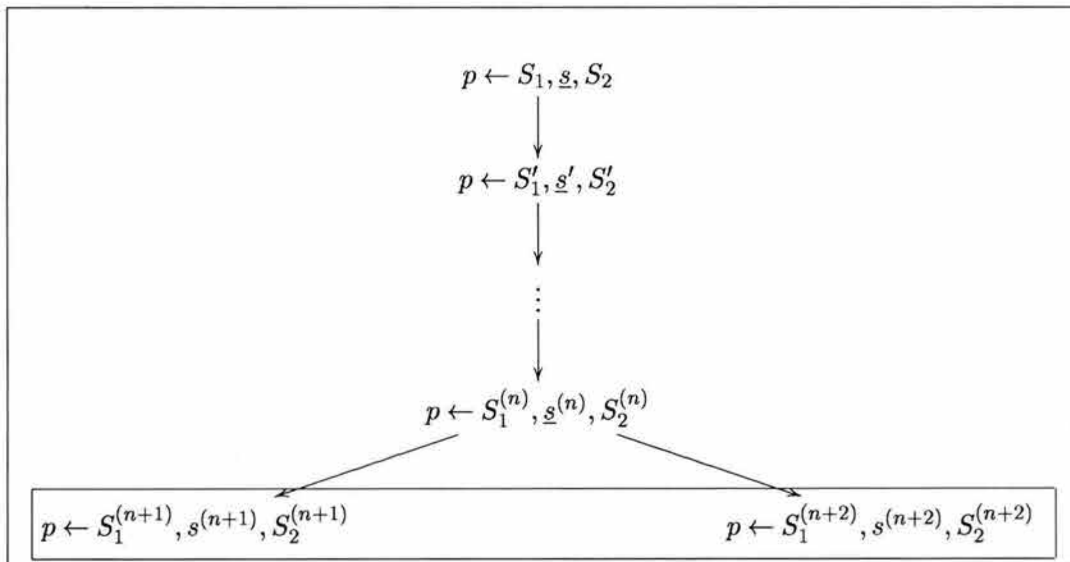

Figura 3.5: La idea principal del criterio para detener una derivación determinística.

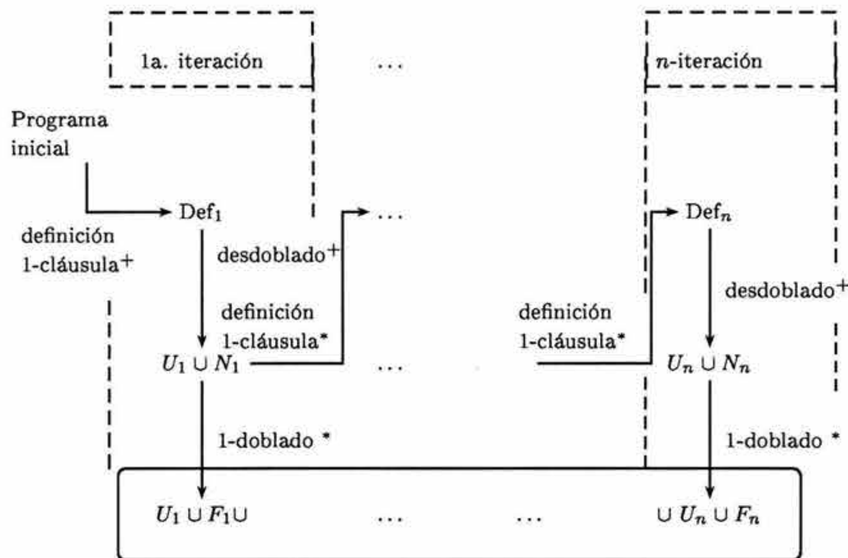

Figura 3.6: La estrategia S de especialización de Gallagher.

(a) Creamos una definición que conste *de una sola cláusula*;

(b) desdoblamos esta definición;

(c) creamos otras definiciones que *consten de una sola cláusula*;

(d) doblamos con las nuevas definiciones las cláusulas que alguna vez desdoblamos.

2. Finalmente, obtenemos un programa que consta de las cláusulas unitarias recolectadas de cada programa en nuestros pasos de transformación, así como de aquellas otras cláusulas que involucran nuevas definiciones y doblados.

En la Fig. 3.6 Pettorossi, Proietti y Renault [PPR97a] ilustran gráficamente los fundamentos y el desarrollo de esta estrategia. La estrategia S de deducción parcial es un ejemplo de una estrategia que optimiza un programa inicial de entrada y que es totalmente automática. Sin embargo, tiene el grave problema de que no puede tratar con el no-determinismo, pues los programas iniciales deben ser no-deterministas, y esto es algo relacionado con la limitación de que las definiciones deben constar de sólo una cláusula. Además, algunos problemas que involucran la introducción de restricciones en el cuerpo de las cláusulas quedan fuera de su alcance.

Normalmente, una medida intuitiva del poder de una estrategia dada está en su capacidad para pasar cierto tipo de pruebas. En el caso del casamiento de cadenas y la transformación de programas lógicos existes una prueba tradicional, conocida como la *prueba KMP*. La prueba KMP es una prueba ahora ya tradicional, que consiste en lo si-

guiente: el deductor parcial a probar debe generar una versión especializada del algoritmo de casamiento de cadenas KMP. Como un hecho, sucede que la estrategia S *no pasa* la prueba KMP. Para pasar esta prueba, y aún así conservar la esencia de la estrategia S, Pettorossi, Proietti y Renault idearon otra estrategia, a la que ellos llamaron la estrategia D, y que tratamos a continuación.

3.6.2 La estrategia D de Pettorossi, Proietti y Renault

La estrategia D de Pettorossi, Proietti y Renault [PPR97a] sigue los mismos lineamientos de la estrategia S de Gallagher, pero difiere de ésta en que en la estrategia D sí es posible hacer doblados utilizando varias cláusulas, y por consiguiente las definiciones recién introducidas pueden constar también de varias cláusulas.

La estrategia D está establecida como sigue:

1. Inicialmente, nuestro programa inicial es un programa especializado con respecto a alguno o algunos de sus argumentos.
    (a) Creamos una definición que conste de una *o varias* cláusulas;
    (b) desdoblamos esta definición;
    (c) creamos otras definiciones que consten de una *o varias* cláusulas;
    (d) doblamos (posiblemente varias cláusulas a la vez usando las nuevas definiciones) las cláusulas que alguna vez desdoblamos.
2. Finalmente, obtenemos un programa que consta de las las cláusulas unitarias básicos recolectadas de cada programa en nuestra sucesión de transformación, así como de aquellas otras cláusulas que involucran nuevas definiciones y doblados.

En la Fig. 3.7 Pettorossi, Proietti y Renault, en [PPR97a], ilustran gráficamente los fundamentos y el desarrollo de esta estrategia. Tal como está descrita, la estrategia D sí pasa con holgura la prueba KMP. Más adelante veremos que, sin embargo, la estrategia D también tiene limitaciones, y no pasa lo que hemos llamado la *prueba BM*, que consiste en que un deductor parcial pueda derivar una variante del algoritmo de casamiento de cadenas de Boyer y Moore. En un capítulo posterior trataremos este tema en detalle.

3.6.3 División en casos

La regla de *división en casos* consiste en la identificación y tratamiento de los distintos casos que un argumento de un predicado puede tomar.

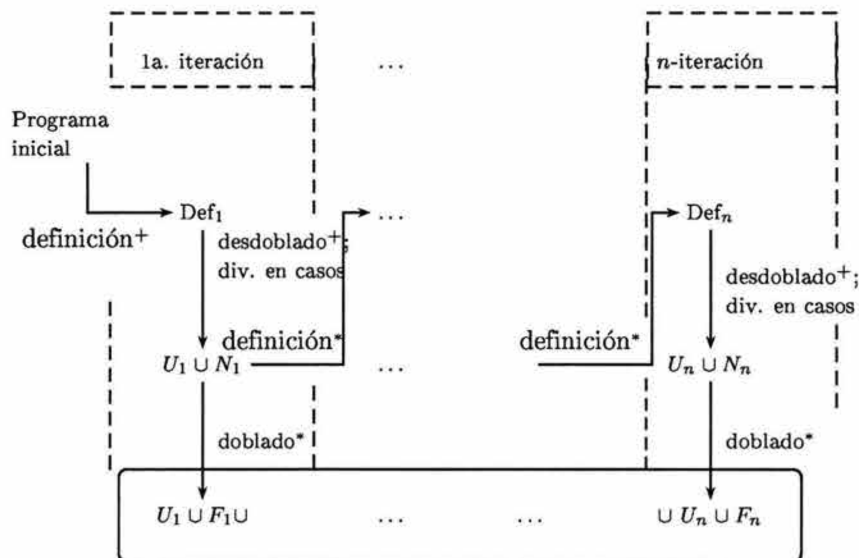

Figura 3.7: La estrategia D de Pettorossi, Proietti y Renault.

Sea $p(A_1, \ldots, A_n)$ un predicado de aridad n y centremos nuestra discusión en el argumento A_1, por decir. Supongamos que cuando resolvemos $p(A_1, \ldots, A_n)$ con respecto a una definición de p en la cláusula

$$s \leftarrow S_1,\ p(A_1, \ldots, A_n),\ S_n \tag{3.41}$$

(en donde debemos exigir a A_1 que aparezca también en la cabeza s, pues de no ser así los valores alternativos que A_1 puede tomar son irrelevantes) el argumento A_1 puede tomar m valores distintos, digamos $\{a_1, \ldots, a_m\}$, por lo que nuestra cláusula se transforma en las siguientes:

$$s \leftarrow A_1 = a_1,\ S_1,\ p(A_1, \ldots, A_n),\ S_2 \tag{3.42}$$

$$s \leftarrow A_1 = a_2,\ S_1,\ p(A_1, \ldots, A_n),\ S_2 \tag{3.43}$$

$$\vdots$$

$$s \leftarrow A_1 = a_m,\ S_1,\ p(A_1, \ldots, A_n),\ S_2 \tag{3.44}$$

siempre y cuando supongamos que A_1 no puede tomar ningún otro valor distinto de los del conjunto $\{a_1, \ldots, a_m\}$.

Si el número de los valores que A_1 puede tomar es infinito, y estos valores están tomados de un conjunto E, por decir, todavía podemos hacer algo con respecto a la división en casos de A_1, dependiendo de si existe o no una forma de particionar de manera finita a E.

La división en casos que realizamos depende a su vez y ahora de si A_1 pertenece a algún conjunto de la partición o no.

La introducción de ecuaciones está justificada por la teoría estándar de igualdad. Pero, por otro lado, también es necesario tratar los posibles casos en los que hay algún tipo de *información negativa*, expresada en nuestro contexto por medio de inecuaciones. A partir de la teoría de la igualdad de Clark, manejamos inecuaciones sin necesidad de utilizar negación, sino por medio de predicados que manejan en forma positiva la desigualdad de constantes y términos. Nuestros programas serán programas lógicos definidos aumentados con inecuaciones $t_1 \neq t_2$, en donde usamos la teoría de igualdad de Clark. Por lo tanto, la completa división en casos, de acuerdo con la teoría de igualdad de Clark, nos conduce a la siguiente transformación de cláusulas. De

$$s \leftarrow S_1,\ p(A_1,\ldots,A_n),\ S_2 \tag{3.45}$$

al aplicar división en casos, obtenemos:

$$s \leftarrow A_1 = a_1,\ S_1,\ p(A_1,\ldots,A_n),\ S_2 \tag{3.46}$$

$$s \leftarrow A_1 = a_2,\ S_1,\ p(A_1,\ldots,A_n),\ S_2 \tag{3.47}$$

$$\vdots$$

$$s \leftarrow A_1 = a_m,\ S_1,\ p(A_1,\ldots,A_n),\ S_2 \tag{3.48}$$

$$s \leftarrow A_1 \neq a_1,\ A_1 \neq a_2,\ \ldots, A_1 \neq a_m,\ S_1,\ p(A_1,\ldots,A_n),\ S_2 \tag{3.49}$$

Una manera concisa y a la vez abstracta de poner lo anterior es por medio de $\in$ y $\notin$:

$$s \leftarrow A_1 \in E,\ S_1,\ p(A_1,\ldots,A_n),\ S_2 \tag{3.50}$$

$$s \leftarrow A_1 \notin E,\ S_1,\ p(A_1,\ldots,A_n),\ S_2 \tag{3.51}$$

en donde las definiciones de $\in$ y $\notin$ difieren de las usuales:

Definición 1

$$A_1 \in [A_2 \mid L] \leftarrow A_1 = A_2 \tag{3.52}$$

$$A_1 \in [A_2 \mid L] \leftarrow A_1 \in L \tag{3.53}$$

Definición 2

$$A_1 \notin [\,] \leftarrow \tag{3.54}$$

$$A_1 \notin [A_2 \mid L] \leftarrow A_1 \neq A_2,\ A_1 \notin L \tag{3.55}$$

y en donde representamos al conjunto E por una lista (de preferencia sin elementos repetidos, ya que ellos sólo generarían cláusulas idénticas).

Dado que la división en casos que utilizamos está íntimamente relacionada con la introducción de ecuaciones, señalamos además que existe un nexo con la unificación. Básicamente, estamos introduciendo ecuaciones para dividir en casos y *para hacer explícitas las operaciones que quedan implícitas en la unificación.*

3.6.4 El desdoblado determinístico y el problema de la terminación

Las técnicas de desdoblado automático deben ser especialmente estrictas en lo que respecta al problema de la terminación. Cuando veamos el problema de casamiento de cadenas en detalle, notaremos que el desdoblado tiene dos formas de asegurar terminación, lo que hace que este desdoblado sea particularmente útil. Una de estas maneras es demostrando que un conjunto de ecuaciones e inecuaciones es *consistente* al tener al menos una solución. Otro criterio es más al estilo de la definición directa de desdoblado determinístico, y consiste en detener el desdoblado determinístico cuando, al ver un paso de resolución hacia adelante (al que llamamos "previsión") notamos que al desdoblar con respecto a una submeta dada obtenemos *dos* cláusulas (con lo que el doblado deja de ser *determinístico*).

Aunque una sola aplicación de la regla de desdoblado es un proceso automático, los problemas básicos a los que se enfrenta un proceso general de aplicaciones de la regla de desdoblado son:

1. dada una cláusula C, ¿qué átomo del cuerpo de C seleccionar para desdoblar?;
2. dado que seleccionamos un átomo A, y que las definiciones en un sistema de transformación por desdoblado/doblado *pueden variar de programa a programa*, ¿con respecto a qué definición debemos desdoblar A para obtener el siguiente programa?;
3. de existir una manera de desdoblar un átomo A, ¿cuántas veces debemos hacerlo durante una sucesión de aplicaciones del desdoblado determinístico?

En su momento, de acuerdo con la evolución de nuestra exposición, trataremos cada uno de estos temas.

3.6.5 El doblado

Actualmente en la literatura existen diversos conceptos de doblado. Uno de ellos es muy limitado, y toma una sola cláusula en la definición (dobla un conjunto de submetas con

respecto a una sola cláusula). Otro tipo de doblado, que ya tratamos en la Subsección 3.5, involucra tomar las submetas que están en la intersección del cuerpo de varias cláusulas, permitiendo con ello la posibilidad de doblar con respecto a una definición que conste de varias cláusulas. A este tipo de doblado tiene diversas propiedades, y una de las principales es eliminar el no-determinismo innecesario; otra propiedad, relacionada, es la de eliminar resoluciones redundantes.

La utilidad del doblado ha estado sujeta a cierta controversia, sobre todo en la versión del doblado en donde están involucradas varias cláusulas. Por ejemplo, algunos de los problemas son:

1. ¿Qué guía nuestra elección de las cláusulas a ser dobladas?
2. Dado que seleccionamos un conjunto de cláusulas a ser dobladas, ¿con respecto de qué definición debemos doblar?
3. ¿Cómo crear nuevas definiciones en el sentido previo?
4. Dado que las variables son locales a la cláusula en la que aparecen, ¿cómo estandarizar estas variables (con respecto a varias cláusulas) para lograr el doblado?
5. ¿Cómo identificar las submetas que son comunes al cuerpo de varias cláusulas?
6. Ahora, si existen diversos conjuntos de submetas que son comunes al cuerpo de varias cláusulas, ¿con respecto a qué conjunto intersectante debemos generar las nuevas definiciones?

Aún dando respuestas a estas preguntas, quedan, sin embargo, muchos detalles en lo específico a considerar. Uno de ellos es saber cuándo hemos desdoblado lo suficiente como para que las submetas redundantes afloren de manera inmediata. Otro problema es en qué orden se deben intercalar posibles simplificaciones previas a un doblado. Por lo tanto, *debemos tener un método para identificar (estandarizando previamente) cláusulas que son comunes al cuerpo de varias cláusulas, y estas submetas deben ser reconocidas sólo a nivel sintáctico;* al menos, este es el enfoque que hemos adoptado en nuestra investigación, como más adelante detallaremos.

3.7 Esquemas teóricos para la transformación de programas

Habiendo ya presentado un sistema específico de transformación de programas, a continuación discutimos uno de los problemas existentes de la transformación de programas en general.

La transformación de programas adolece de la falta de un adecuado formalismo que nos permita presentar de manera uniforme sus resultados, tanto teóricos como prácticos. Existen algunas propuestas para que la teoría de las categorías a brinde a las ciencias de la computación, en general, tal formalismo [Dyb86, BdM97] (ver también la Sección *Relevance of Categorical Logic for Computer Science* en [Poi86]). Sin embargo, las aplicaciones de la teoría de categorías a la metodología de la programación no son triviales, y prueba de ello es que cada aplicación de este tipo frecuentemente amerita una publicación. Alberto Pettorossi, uno de los líderes del área de la transformación de programas, ha publicado algún trabajo en el área de intersección entre programación y teoría de categorías [LP86, KLP87], pero su investigación ha derivado en ámbitos que siguen de cerca el cálculo de predicados formulado llanamente.

En efecto, existe una necesidad de crear esquemas genéricos para traducir en un alto nivel los desarrollos que se van dando de manera particular en diversos paradigmas [GB86], tal como las instituciones fueron introducidas en relación a los sistemas lógicos [Tar86]. Un trabajo posterior indica la posibilidad de ver las implementaciones como instituciones [BV87].

Aquí percibimos una ventaja de la familiaridad con las teorías matemáticas usuales, suponiendo que deseemos hacer que los resultados de nuestras investigaciones tengan un público amplio. Ver las listas en términos de 2-celdas [RS87], por ejemplo, no es algo estándar, pues requiere de un conocimiento de teoría de categorías inusual entre los programadores (a quienes en última instancia queremos influir).

No obstante, existen serias dudas acerca de qué tan amplio y tan general podría ser un formalismo que abarcara la transformación de programas como un todo. Los formalismos (cálculos) para la transformación de programas son abundantes ([Möl93a, Mee89, Pep93, BJJM98, dMS98], por citar unos pocos), pero estos se ubican en un alto nivel del tratamiento de los problemas, y descuidan los aspectos de implementación. Feather comentaba en 1986 en [Fea87] (3.3.2) que es claro que la naturaleza de la especificación, el lenguaje de programación utilizado, y los requerimientos de eficiencia influyen en la derivación de un programa por transformación, y que se conoce poco de cómo tratar de todos estos temas a la vez de una forma diferente a la particular. Y la situación actual no es muy distinta ahora. De hecho, tener un formalismo que abarcara todos estos materiales sería un logro important en las ciencias de la computación; y aún así, cabría notar que si bien tal formalismo puede organizar y estructurar nuestro conocimiento de estos temas no necesariamente sería una adecuada *herramienta de investigación*. Por ejemplo, la formulación de alto nivel de *merge* para el problema de ordenamiento en [Ryd86b, Ryd86a] es tal que nos impide conocer el importante tema de la implementación, para que lo presumiblemente habría que construir otra categoría y encontrar cómo interrelacionarlas. Por lo demás,

otras estructuras también serían posibles: por ejemplo, si bien las instituciones fueron introducidas en relación a los sistemas lógicos [Tar86], existe la posibilidad es tratar con las implementaciones como instituciones [BV87].

Mayor información acerca de los métodos, las estrategias y las tácticas de la transformación de programas que vimos en este capítulo pueden hallarse en [Fea87] en el caso de la transformación de programas en general, y en [PP96a], en el caso de la transformación de programas lógicos, en tanto que las tendencias a futuro (aún vigentes como tendencias) de la transformación de programas están en [PP96c]. Una discusión acerca de la síntesis, la derivación, la verificación y la transformación de programas (que pueden incluir negación) dentro del ámbito de la programación lógica se hallan en [PP02].

Capítulo 4

Algoritmos de casamiento de cadenas

En este capítulo presentaremos algunos algoritmos de casamiento de cadenas que posteriormente serán nuestros casos de estudio para la aplicación de la transformación de programas lógicos.

4.1 Introducción

Habiendo dado ya un panorama de las herramientas a nuestra disposición, a continuación planteamos el problema del *casamiento exacto de cadenas* y algunos de los algoritmos que se han ideado para resolverlo.

La búsqueda de cadenas en un texto es un problema ubicuo de la computación. Hallar cadenas es importante tanto en un archivo de texto ordinario como dentro de las largas cadenas formadas por el código genético. Generalmente la búsqueda se reduce a sólo encontrar la primera ocurrencia de un patrón p en un texto t, aún cuando la búsqueda de todas las ocurrencias del patrón p dentro de t también tiene su respectiva importancia. En esta tesis sólo tratamos el problema de hallar la primera ocurrencia de un patrón dentro de un texto, aún cuando el problema de casamiento múltiple es un resultado lateral que obtenemos manteniendo cierto no-determinismo en nuestros programas finales (determinismo que, por otra parte, podría ser eliminado con los métodos propuestos en [Rosar]). A continuación abordamos el problema de casamiento de cadenas con algunos algoritmos que son ya clásicos en la literatura.

Sea $\mathcal{A}$ un conjunto no vacío al que llamaremos *alfabeto*, y a cuyos elementos los llama-

remos *símbolos* o *letras*. Una *cadena* en $\mathcal{A}$ es una sucesión finita de símbolos pertenecientes a $\mathcal{A}$. Supondremos la existencia de la cadena vacía Λ. Denotaremos por $\mathcal{A}^*$ al conjunto de cadenas conformadas por símbolos de $\mathcal{A}$ y que incluye a Λ. En el contexto de la programación lógica, representamos una cadena por una *lista*. Una operación natural en las cadenas es la *concatenación*, operación que definimos a continuación:

Dado un alfabeto $\mathcal{A}$, definimos la operación de *concatenación* de cadenas $\mathcal{A}$ en $\mathcal{A}$:

$$++ : \mathcal{A}^* \times \mathcal{A}^* \rightarrow \mathcal{A}^*$$

tal que, para cadenas las S y T, $S ++ T$ es la cadena cuyos primeros elementos son aquellos de S seguidos por los de T. Junto con la ley de cerradura de $++$, $(\mathcal{A}^*, ++)$ es un monoide no-conmutativo con elemento neutro Λ.

Dada una cadena Q, denotaremos por $\{Q\}$ al *conjunto* de símbolos que componen a Q. Por consiguiente, los elementos en $\{Q\}$ no tienen un orden de ocurrencia, y además $\{Q\}$ no tiene elementos repetidos.

Notemos que siempre podemos obtener subsucesiones de una cadena. Nos interesa, particularmente, el siguiente tipo de subsucesiones:

Una *subcadena* s de una cadena t es una subsucesión de tal cadena y tal que los elementos que son adyacentes en la cadena son también adyacentes en la subcadena. Un *sufijo* de una cadena t es una subcadena s de t tal que $t = t_1 ++ s$ para una subcadena t_1 de t. Un prefijo t_1 de t es *prefijo propio de* t si $t_1 \neq t$. Similarmente, un *prefijo* de una cadena t es una subcadena u de t tal que $t = u ++ t_2$ para cierta subcadena t_2 de t. Un sufijo t_2 de t es *sufijo propio de* t si $t_2 \neq t$.

A continuación planteamos nuestro problema fundamental, al que llamamos *el problema de la subcadena*:

Suponemos dado un alfabeto $\mathcal{A}$. Dada una cadena p, y una cadena t, con $p, t \in \mathcal{A}^*$, queremos algoritmos que nos permitan determinar si p es una subcadena de t.

Nos referiremos a p como el *patrón*, y la cadena t como el *texto*. A continuación describiremos algunos algoritmos que resuelven este problema.

4.2 Algunos algoritmos de casamiento de cadenas

A continuación describimos algunos algoritmos que resuelven el problema de casamiento de cadenas, pero antes, convenimos en la siguiente terminología y notación:

Nos referiremos a un *casamiento* entre un par de cadenas cuando ambas cadenas coinci-

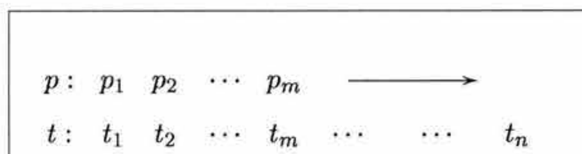

Figura 4.1: La dirección de los desplazamientos del patrón sobre el texto.

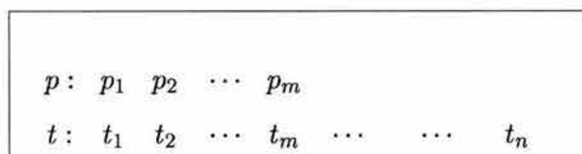

Figura 4.2: Configuración inicial entre el patrón y el texto.

den símbolo a símbolo, y a una *concordancia* entre dos símbolos si tales símbolos coinciden. Hablaremos de una *discordancia* refiriéndonos a un no-casamiento entre un símbolo y otro, y diremos *mal-casamiento* refiriéndonos al evento en que tenemos segmentos de cadenas que no casan. Denotamos a la subcadena a hallar por p, y al texto en donde queremos hallar esta subcadena por t. Supondremos que la longitud de p es m, y la longitud de t es n. Trataremos de asociar, siempre que sea posible, el índice i a la cadena t, y el índice j a la cadena p. Denotamos por s_k el k-ésimo símbolo de una cadena s.

4.2.1 El algoritmo de fuerza bruta

Damos a continuación una de las descripciones del *algoritmo de fuerza bruta* para resolver el problema de la subcadena. De manera directa, para determinar si la cadena p es una subcadena de la cadena t, hacemos lo siguiente:

Alineamos el patrón sobre el comienzo del texto, y probamos si el patrón es un prefijo del texto comparando cada símbolo del patrón con el correspondiente símbolo del texto (ver Fig. 4.1, Fig. 4.2, y Fig. 4.3; en la Fig. 4.3 hemos representado una posible comparación

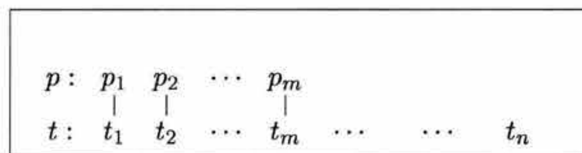

Figura 4.3: Posibles comparaciones entre los símbolos de patrón y del texto.

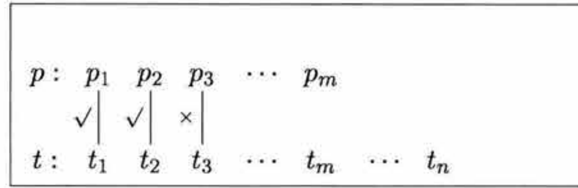

Figura 4.4: Algunas comparaciones símbolo a símbolo en una configuración inicial entre p y t.

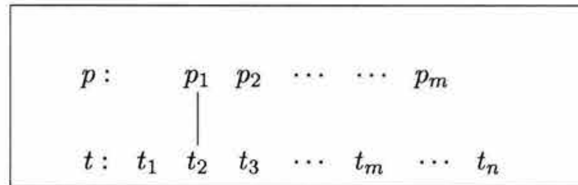

Figura 4.5: Próxima configuración del algoritmo de fuerza bruta.

con una barrita que enlaza los dos símbolos a comparar). Si todas las comparaciones son exitosas, entonces hemos probado que el patrón ocurre como prefijo del texto; de no ser así, existe una discordancia y en este caso desplazamos a la derecha el patrón sobre el texto un símbolo y comenzamos las comparaciones otra vez, desde el símbolo de más a la izquierda del texto, y además, como nota auxiliar, olvidándonos de todas las comparaciones previamente realizadas. Si todos los símbolos del patrón casan, hemos hallado una ocurrencia del patrón en el texto. Si no, repetimos el proceso hasta que alcancemos el fin del texto (realmente, cuando la longitud por casar del texto es menor que la longitud del patrón). En la Fig. 4.4 y la Fig. 4.5 presentamos algunas configuraciones que acontecen al momento de ejecutar este algoritmo: en la Fig. 4.4 hemos decorado las primeras dos comparaciones exitosas con un $\surd$, y la discordancia entre p_3 y t_3 (por decir) con una pequeña cruz, $\times$.

Una característica de este algoritmo es que si las discordancias ocurren cada vez que casi terminamos de comparar todos los símbolos del patrón con el texto, obtenemos su peor caso de desempeño, y tal peor caso es *cuadrático* con respecto al número de comparaciones realizadas. Esta situación, sin embargo, no ocurre con frecuencia, y el desempeño promedio del algoritmo de fuerza bruta, en la práctica, resulta ser *lineal*.

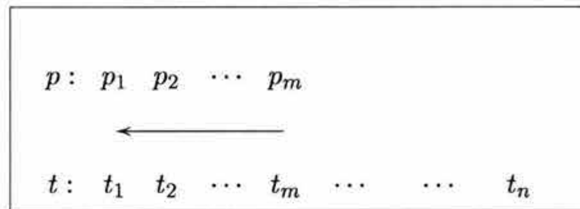

Figura 4.6: Dirección del proceso de casamiento símbolo a símbolo según el algoritmo BM.

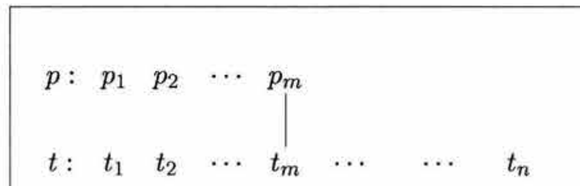

Figura 4.7: Inicio del casamiento símbolo a símbolo en el algoritmo BM.

4.2.2 El algoritmo de Boyer y Moore y algunas variantes

Haremos ahora una introducción a un algoritmo de gran importancia teórica y práctica, el *algoritmo de Boyer y Moore*, que es suficiente para nuestros propósitos (remitimos al lector a [Aho90, CR94, Ste94], o al artículo original de Boyer y Moore [BM77] para una información complementaria o todavía más detallada). En la Fig. 4.6 y en la Fig. 4.7 ilustramos dos de los puntos esenciales de este algoritmo. En particular, en la Fig. 4.6 ilustramos el hecho de que el algoritmo de Boyer y Moore intenta casar el patrón con el texto *derecha a izquierda*, a diferencia del algoritmo de fuerza bruta. Un ejemplo particular de un intento de casamiento inicial y una posterior configuración está dado en la Fig. 4.8. Nos referiremos al algoritmo de Boyer y Moore, de ahora en adelante, como el algoritmo BM.

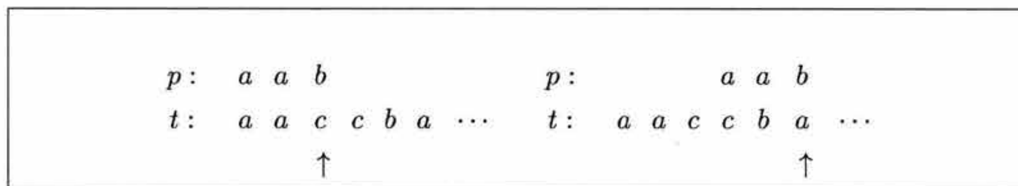

Figura 4.8: Ejemplo del funcionamiento del algoritmo BM.

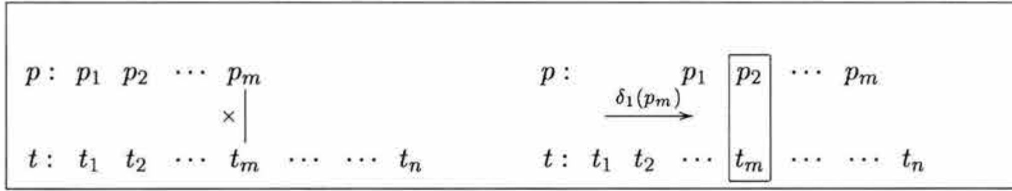

Figura 4.9: Desplazamiento dado por la función δ_1.

A continuación damos una descripción detallada del algoritmo BM.

Supongamos que ya hemos avanzado hasta cierta posición del texto, de tal manera que una posible alineación del patrón con el texto está entre t_{i-m} y t_i. Considere primero una discordancia entre p_m y t_i. Si t_i no ocurre en $\{p\}$ deslizamos el patrón m símbolos a la derecha. La próxima comparación es a continuación entre p_m y t_{i+m}, ya que un desplazamiento más pequeño obligaría a comparar p_m con un símbolo que ya sabemos no coincidirá con p_m. Ejemplificamos tal situación en la Fig. 4.8 para el caso del patrón aab y un texto con sus símbolos iniciales $aaccba$, pues notamos que c no ocurre en $\{aab\}(= \{a,b\})$.

Si, por otro lado, t_i sí ocurre en p, con una ocurrencia de más a la derecha en p_k, entonces alineamos t_i y p_k para posteriormente realizar un casamiento símbolo a símbolo (para lo cual es necesario desplazar el patrón sobre el texto $m-k$ símbolos), y reanudamos la prueba de casamiento comparando ahora p_m con t_{i+m-k}. Para determinar los valores de desplazamiento adecuados, por consiguiente, por medio de la siguiente función δ_1:

$$\delta_1(x) \quad = \quad \begin{cases} m & \text{si } x \notin \{p_1, \ldots, p_m\}\ , \\ m-k & k = \textit{máx}\{j \in \mathbb{N} \mid p_j = x\} \end{cases}$$

En la Fig. 4.9 ilustramos un caso particular de cómo aplicar esta función, con $p_2 = t_m$, y tal que p_2 es la ocurrencia de más a la derecha de t_m en p.

Consideremos ahora una concordancia entre p_m y t_i. Las comparaciones de los correspondientes símbolos del patrón y el texto pueden continuar derecha a izquierda o bien hasta que descubramos una ocurrencia completa del patrón en el texto o bien hasta que una discordancia ocurra entre p_j y $t_{i'}$, digamos. En este caso, el sufijo $p_{j+1} \cdots p_m$ es igual a la porción del texto $t_{i'+1} \cdots t_{i'+m-j}$, y además se cumple que $p_j \neq t_{i'}$.

Si $t_{i'}$ no ocurre p en absoluto, podemos desplazar el patrón $m-j$ posiciones a la derecha (para pasar la posición de $t_{i'}$), y la próxima comparación será entre p_m y $t_{i'+m}$. Note que finalmente incrementaremos el índice del texto en m posiciones.

Si, por otro lado, t_k sí ocurre en p, considere la ocurrencia de más a la derecha sobre

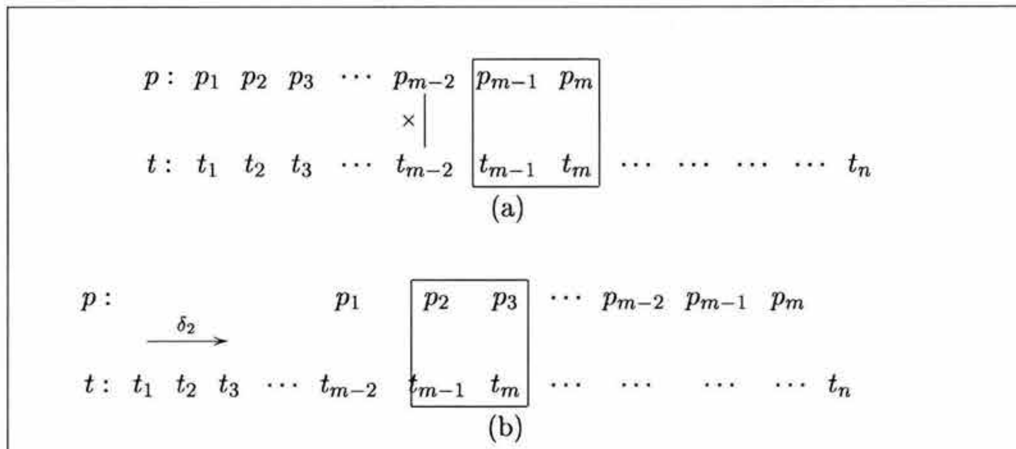

Figura 4.10: Desplazamiento dado por la función δ_2.

el patrón de tal símbolo, p_k. Hay dos posibilidades: p_k puede estar o a la izquierda o la derecha de p_j. En el caso en que p_k está a la izquierda, entonces alineamos a p_k y $t_{i'}$, y la próxima comparación será entre p_m y $t_{i'+m-k}$. Por consiguiente, podemos usar δ_1 para calcular el desplazamiento del índice del texto en ambos casos (este caso y aquel de alinear el símbolo de más a la derecha del patrón con el respectivo del texto). Si, sin embargo, p_k está a la derecha de p_j, entonces δ_1 produciría un valor negativo, lo que significaría un retroceso indeseable en el desplazamiento del patrón. Así, en este caso ignoramos a δ_1 y desplazamos el patrón un símbolo a la derecha, lo cual es equivalente a incrementar el índice del texto en $m - j + 1$ unidades.

En el caso de casamientos parciales, podríamos desplazar el patrón más símbolos que aquellos prescritos por δ_1. En lugar de determinar la ocurrencia dentro del patrón del símbolo $t_{i'}$ del texto que causó la discordancia, podemos determinar una reocurrencia del sufijo que ya ha casado. En este caso, $p_{j+1} \cdots p_m = t_{i-m+j+1} \cdots t_i$ y $p_j \neq t_{i-m+j}$. Si el sufijo $p_{j+1} \cdots p_m$ también aparece en P como una subcadena $p_{j+1-k} \cdots p_{m-k}$, con $p_{j-k} \neq p_j$, y esta es la ocurrencia de más a la derecha, entonces el desplazamos el patrón k símbolos a la derecha.

Puede también suceder que tal reocurrencia "desborde" el extremo izquierdo de p, esto es, un sufijo de $p_{j+1-k} \cdots p_{m-k}$ aparece como un prefijo de p, y este caso sucede cuando $k \geq j$. Un empalme adecuado del sufijo y prefijo respectivo garantizan que no perdemos ninguna ocurrencia del patrón en el texto.

Algoritmo 1

$inicializarBM(\delta_1, \delta_2, p);$
$i := m; j := m;$
while $(j > 0)$ **and** $(i \leq n)$ **do**
  if $t_i = p_j$
  then {intento de casamiento}
    begin $i := i - 1;\ j := j - 1$ **end**
  else {discordancia}
    begin $i := i + m(\delta_1(t_i), \delta_2(j));\ j := m$ **end**
if $j < 1$ **then** $i := i + 1$ **else** $i := 0$

Figura 4.11: El algoritmo de casamiento de cadenas de Boyer y Moore.

En resumen, la nueva información es calculada como sigue:

$$\begin{aligned} \delta_2(j) \ &= min\{ \ k + m - j \mid k \geq 1 \text{ y } (k \geq j \text{ o } p_{j-k} \neq p_j) \\ &\qquad \text{y } ((k \geq d \text{ o } p_{d-k} = p_d) \text{ para } j < d \leq m)\} \end{aligned}$$

En la Fig. 4.10 ilustramos algunas configuraciones que resultan de aplicar la función δ_2, en donde suponemos que $p_{m-1}p_m$ tiene una reocurrencia más a la derecha en P en p_2p_3 y $p_{m-2} \neq p_1$.

Notemos que el valor de δ_2 siempre produce un desplazamiento positivo. Consecuentemente, obteniendo el máximo de ambas δs, no sólo evitamos un desplazamiento negativo como posiblemente prescribiría δ_1, sino también movemos el patrón tantos símbolos como es posible, dados δ_1 y δ_2. La Fig. 4.11 muestra el algoritmo BM. En la presentación de este algoritmo hemos puesto, al principio, la función *inicializarBM*, que permite inicializar las tablas δ_1 y δ_2 (independientemente) una vez que ya conocemos p.

El cálculo de la función δ_1 y el de la función δ_2 dependen, como ya hemos presentado, del patrón p, sin referencia al texto t, por lo que el algoritmo BM es un campo particularmente apropiado para el estudio de la evaluación parcial del algoritmo de fuerza bruta cuando sólo conocemos p.

En cuanto a eficiencia, en el peor caso, el algoritmo BM tiene un desempeño de orden lineal, y en el caso promedio el algoritmo BM es sublineal.

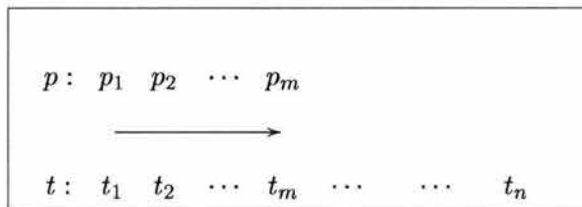

Figura 4.12: Dirección del intento de casamiento entre un patrón y un texto en el algoritmo KMP.

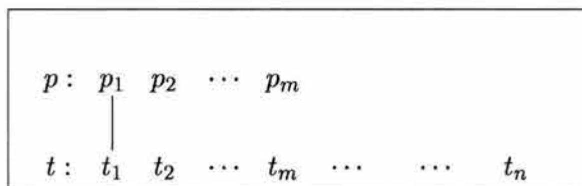

Figura 4.13: Inicio del casamiento símbolo a símbolo en el algoritmo KMP.

4.3 Los algoritmos MP y KMP

Una vez que hemos presentado una descripción del algoritmo BM, nos enfocamos al *algoritmo de Knuth, Morris y Pratt*, al que de ahora en adelante nos referiremos como el *algoritmo KMP*. Por razones de presentación, el algoritmo KMP es algunas veces precedido [CR94, Ste94] de una versión simplificada, usualmente llamada el *algoritmo de Morris y Pratt* y al que nos referiremos como el *algoritmo MP*, el cual utiliza el mismo código que el algoritmo KMP en su etapa de búsqueda, pero que difiere de éste en cuanto a la etapa de preprocesamiento. Aquí seguiremos esta misma presentación también.

4.3.1 El algoritmo de Morris y Pratt

En los algoritmos MP y KMP también ubicamos el patrón en la posición de más a la izquierda del texto (Fig. 4.12). A diferencia del algoritmo BM, sin embargo, los algoritmos MP y KMP inician el proceso de casamiento símbolo a símbolo entre el patrón y el texto de izquierda a derecha [KMP77] (ver Fig. 4.13).

Similarmente al algoritmo de fuerza bruta, el algoritmo MP intenta casar el patrón con el texto comparando los símbolos respectivos de izquierda a derecha. Si una discordancia acontece entre p_j y t_i, digamos, sabemos que $p_1 \cdots p_{j-1} = t_{i-j+1} \cdots t_{i-1}$. Si existe un prefijo propio y maximal del segmento del patrón ya casado $p_1 \cdots p_{j-1}$, de longitud $k-1$,

por ejemplo, de modo que este prefijo es igual a un *sufijo* del segmento del patrón ya casado, como a continuación ilustramos

$$p_1 \cdots p_{k-1} = p_{j-k+1} \cdots p_{j-1} \tag{4.1}$$

entonces podemos desplazar el patrón de tal forma que el prefijo ocupe el espacio previamente ocupado por el prefijo, i.e. k símbolos.

En el ejemplo siguiente, existe una discordancia entre $p_j = e$ y $t_i = d$:

$$\begin{array}{lcccccccccc} p: & a & b & c & d & a & b & c & e & \\ t: & a & b & c & d & a & b & c & d & \cdots \\ & & & & & & & & \uparrow & \end{array}$$

El segmento del patrón que ya ha casado es $p_1 \cdots p_{j-1} = a\ b\ c\ d\ a\ b\ c$, el cual tiene un prefijo propio y maximal $p_1 \cdots p_{k-1} = a\ b\ c$ igual al sufijo $p_{j-k+1} \cdots p_{j-1}$ de tal segmento del patrón. Desplazamos el patrón $k = 4$ símbolos:

$$\begin{array}{lcccccccccccc} p: & & & & & a & b & c & d & a & b & c & e \\ t: & a & b & c & d & a & b & c & d & \cdots & & & \\ & & & & & & & & \uparrow & & & & \end{array}$$

Una tabla f registra el número de símbolos que debemos desplazar el patrón con respecto al texto, y lo registra en función del símbolo del patrón que causó la discordancia:

$$\begin{aligned} & f(1) = 0 \\ & f(j) = máx\{i \mid i < j,\ p_1 \cdots p_{i-1} = p_{j-i+1} \cdots p_{j-1}\} \end{aligned}$$

Podemos calcular esta tabla como sigue: Alineamos a la cadena $p_1 \cdots p_{j-1}$ con una copia de sí misma y, a continuación, repetidamente deslizamos una copia con respecto a la otra hasta que todos los símbolos que se traslapan casen, o hasta que no quede ninguno. Notemos que alineamos inicialmente ambas copias en tal forma que p_1 corresponde a p_2 [Ste94, página 16]. Consideremos, por ejemplo, el patrón $p = abcdabce$, en donde $p_8 = e$. Existen tres símbolos que se traslapan y que casan en la siguiente configuración:

$$\begin{array}{ccccccccccc} a & b & c & d & a & b & c & & & & \\ & & & & a & b & c & d & a & b & c \end{array}$$

por lo que, $f(8) = 3 + 1 = 4$.

4.3.2 El algoritmo de Knuth, Morris y Pratt

El algoritmo MP ya representa en sí un avance con respecto al funcionamiento del algoritmo de fuerza bruta, pero es posible hacer algo mejor, lo que logramos agregando una restricción en forma de una inecuación. Detallamos la idea como sigue.

Considere el siguiente ejemplo:

$$\begin{array}{lllll} p: & a & a & b & \\ t: & a & b & c & \cdots \\ & & \uparrow & & \end{array}$$

Después de la discordancia de los símbolos señalados por la flecha (la segunda a del patrón y la b del texto), el algoritmo MP dicta un desplazamiento de un símbolo:

$$\begin{array}{llllll} p: & & a & a & b & \\ t: & a & b & c & \cdots & \\ & & \uparrow & & & \end{array}$$

En seguida, la próxima comparación es entre la primera a del patrón y la b del texto, y dicha comparación fracasará también. Note, sin embargo, que bien pudimos haber sabido *a priori* de este fracaso, ya que el apuntador al texto no se ha movido, y nosotros ya conocíamos todo el patrón (y por lo tanto el símbolo del patrón que causó la discordancia). Podemos, entonces, mejorar la tabla MP fortaleciendo la condición (4.1) en la forma

$$p_1 \cdots p_{k-1} = p_{j-k+1} \cdots p_{j-1} \text{ y } p_k \neq p_j \tag{4.2}$$

lo que nos conducirá, como veremos, al algoritmo KMP.

Llamaremos *next* a una tabla parecida a la de f, pero que satisface la condición adicional (4.2).

Consideremos el casamiento parcial

$$\begin{array}{lllll} t_1 & t_2 & \cdots & t_{j-1} & x \\ p_1 & p_2 & \cdots & p_{j-1} & p_j \end{array}$$

en donde $x \neq p_j$. El problema ahora es conocer qué tanto podemos desplazar el patrón p en el texto t de manera que no perdamos ninguna ocurrencia intermedia.

El valor de *next*(j) es entonces igual al mayor índice k, $k < j$, que satisfaga (4.2), si es que existe. Si ningún índice k tal existe, entonces *next*(j) es igual a 0. El ejemplo para

aab, tal como el algoritmo KMP lo trata, nos permite avanzar dos símbolos en lugar de uno solo:

$$\begin{array}{rcccc} p: & & a & a & b \\ t: & a & b & c & \cdots \\ & & & \uparrow & \end{array}$$

Esto lo logramos utilizando una función *next*, que nos permite avanzar lo suficiente correctamente. La definición de *next* es como sigue:

$$next(j) = máx\{i \mid p_1 : p_{j-1} = p_1 : p_{i-1} \text{ y } p_i \neq p_j\}$$

Ahora seleccionamos el mayor índice i menor que j que satisfaga la alineación siguiente (en el sentido de que los símbolos de arriba coincidan exactamente con los que están abajo):

$$\begin{array}{cccccccc} p_1 & p_2 & \cdots & p_k & \cdots & \cdots & p_{j-1} & p_j \\ & & & & p_1 & \cdots & p_{i-1} & p_i \end{array}$$

en donde $p_i \neq p_j$ (si tal i no existe, ponemos $next(j) = 0$).

Para construir la función *next*, requerimos la construcción de $f(j)$ que, como ya hemos visto, es el mayor índice i menor que j tal que la alineación siguiente se cumple:

$$\begin{array}{ccccccc} p_1 & p_2 & \cdots & p_k & \cdots & \cdots & p_{j-1} \\ & & & & p_1 & \cdots & p_{i-1} \end{array}$$

o bien

$$\begin{array}{cccccc} p_1 & p_2 & \cdots & p_{j-i+1} & \cdots & p_{j-1} \\ & & & p_1 & \cdots & p_{i-1} \end{array}$$

y tenemos que $f(j) \geq 1$ cuando $j > 1$, y $f(1) = 0$ (por convención).

La definición formal de *next* es

$$next(j) = \begin{cases} f(j) & \text{si } p_j \neq p_{f(j)} \\ next(f(j)) & \text{si } p_j = p_{f(j)} \end{cases} \tag{4.3}$$

Para comprender esta formulación de *next*, notemos que la primera rama de la condicional anterior nos obliga a poner

$$next(j) = máx\{i \mid f(j) \text{ y } p_i \neq p_j\}$$

En la otra rama suponemos que $p_j = p_i$:

$$\begin{array}{ccccccc} p_1 & p_2 & \cdots & p_k & \cdots & p_{j-1} & p_j \\ & & & p_1 & \cdots & p_{i-1} & p_i \end{array}$$

Algoritmo 2

```
inicializarKMP(next, p);
i := 1; j := 1;
while (i ≤ n) and (j ≤ m) do
    begin
        while (j > 0) and (p_j ≠ t_i) do j := next(j);
        i := i + 1;
        j := j + 1;
    end
if j > m then i := i - m else i := 0
```

Figura 4.14: El algoritmo de Knuth, Morris y Pratt.

lo que significa que todavía podemos desplazar aún más el subpatrón, hasta, digamos, un valor i' $(i' < i)$:

$$\begin{array}{cccccccc} p_1 & p_2 & \cdots & p_k & \cdots & p_{j-1} & p_j \\ & & & p_1 & \cdots & p_{i'-1} & p_{i'} \end{array}$$

de tal forma que $p_j \neq p_{i'}$. Pero de acuerdo con el siguiente diagrama, $p_i \neq p_{i'}$:

$$\begin{array}{cccccccc} p_1 & p_2 & \cdots & p_k & \cdots & \cdots & p_{j-1} & p_j \\ & & & p_1 & \cdots & \cdots & p_{i-1} & p_i \\ & & & & p_1 & \cdots & p_{i'-1} & p_{i'} \end{array}$$

La importante conclusión es que $next(j) = next(f(j))$ por transitividad.

La Fig. 4.14 presenta el código para el algoritmo KMP, en donde al principio del algoritmo hemos puesto el procedimiento *inicializarKMP* que permite construir la tabla *next* una vez conocido el patrón p.

La importancia del algoritmo KMP radica en que ha permitido a los investigadores de transformación de programas utilizarlo como una prueba del poder de algunos evaluadores parciales, como veremos más adelante. En efecto, en el algoritmo KMP sólo es necesario conocer el patrón p sin necesariamente conocer el texto t, como ya hemos observado al construir la función *next*, lo que es un caso ideal para un evaluador parcial. El desempeño promedio de este algoritmo es lineal.

Capítulo 5

Derivación de la parte de búsqueda de una variante del algoritmo BM

En el capítulo previo hemos presentado el problema de la ocurrencia de un patrón en un texto (problema de la subcadena) y algunos de los algoritmos que lo resuelven; abordamos ahora el *problema de derivar* algunos algoritmos que intentan casar el patrón con el texto de derecha a izquierda, de acuerdo con el método ideado por Boyer y Moore.

5.1 Derivación de la parte de búsqueda del algoritmo BM

En esta sección tratamos con la derivación de la parte de búsqueda de una variante del algoritmo BM. En efecto, ya hemos visto que algunos algoritmos de casamiento de cadenas constan de dos partes: una de ellas es relativa al preprocesamiento del patrón, y otra más está relacionada con la búsqueda del patrón en sí en tiempo de ejecución. Hemos realizado nuestra derivación *manualmente*, es decir, que la estrategia que utilizamos para derivar la variante del algoritmo BM está guiada por el usuario, y utiliza en diversos grados la intuición del programador. En cada paso, una motivación apropiada nos conduce a la aplicación de cierta regla en lo específico.

Nuestro repertorio de reglas será esencialmente el dado en el Capítulo 3, con las modificaciones ahí presentadas. Ya hemos visto, por ejemplo, que en lugar de aplicar la regla de división en casos con respecto a una sustitución $\{X/a\}$, nosotros aplicaremos algunas veces esta regla con respecto a una ecuación $X = a$, así derivando dos cláusulas: una con

$$T = (S_1 \mathbin{+\!\!+} aab) \mathbin{+\!\!+} S_3 = S_2 \mathbin{+\!\!+} S_3$$

$$S_2 = S_1 \mathbin{+\!\!+} aab$$

Figura 5.1: Una representación de la ocurrencia del patrón aab dentro del texto T.

la submeta $X = a$ y otra con la submeta $X \neq a$ (colocadas en sus respectivos cuerpos). Esta variación de la regla de división en casos nos permitirá dar un tratamiento uniforme a las cláusulas que después serán utlizadas para realizar doblados. Posteriormente mostraremos la interacción entre la teoría de estándar de igualdad y la teoría de igualdad de Clark (que amplía la teoría de igualdad axiomatizando la noción de desigualdad vía inecuaciones) y la introducción de ecuaciones e inecuaciones. Otra diferencia en nuestro uso de las reglas de [PP98b, PPR97a] es que nosotros agregaremos *factorización* [Kow79b] para eliminar submetas subsumidas (y también repetidas). Tales variaciones en las reglas citadas son lógicamente correctas, y forman parte de un repertorio que la Lógica nos provee, pero para nosotros resultaron particularmente útiles. Finalmente, el desdoblado determinístico [Gal93] del Capítulo 3 jugará un papel relevante en el resto de nuestro trabajo. Especializaremos nuestra medida de eficiencia al número de comparaciones símbolo a símbolo entre dos cadenas.

Como especificación nosotros tomamos un programa que implementa el algoritmo de fuerza bruta en Prolog. Aprovecharemos la correspondencia natural que existe entre una lista de n símbolos $[p_1, \ldots, p_n]$ y una cadena $p_1 \cdots p_n$ también de n símbolos, con la lista vacía $[\,]$ haciendo el papel de la cadena vacía ε (la cadena que carece de símbolos). De momento, en pro de la claridad de nuestra presentación, nos concentraremos en un patrón particular: aab. A pesar de que este patrón tiene símbolos repetidos, esperamos que los correspodientes pasos de nuestra derivación sean identificables y generalizables a un patrón de tamaño y forma arbitrarios; aún así, el lector puede consultar la sección 5.3, en donde presentamos el esquema general de derivación.

Comenzaremos desde el siguiente programa, que es uno de los programas que implementan un algoritmo de fuerza de bruta para casar cadenas:

$$s_{aab}(T) \leftarrow append(S_1, [aab], S_2),\ append(S_2, S_3, T) \tag{5.1}$$

como ilustramos en la Fig. 5.1, y en donde asociamos $\mathbin{+\!\!+}$ a la izquierda.

A continuación derivaremos un programa intermedio que bien podría haber sido nuestro programa inicial, dado lo intuitivo de su formulación. No obstante, hemos seguido

el lineamiento de [PPR97a] y consideramos el programa siguiente como un paso intermedio para obtener nuestros programas finales. De todas formas, el programa siguiente es no-determinístico, y traduce la especificación previa a otra basada en prefijos. Será útil referirnos a este otro programa, derivado casi inmediatamente desde la especificación mencionada, como si fuera la especificación inicial. Para obtener tal programa, primero eliminamos el uso de *append* como sigue. Desdoblamos (5.1) seleccionando su submeta izquierda, y con respecto a la definición usual dada a *append*:

$$s_{aab}(T) \leftarrow append([aab], S_3, T) \tag{5.2}$$

$$s_{aab}(T) \leftarrow append(X, [aab], Z),\ append([A \mid Z], S_3, T) \tag{5.3}$$

A continuación, desdoblamos (5.2):

$$s_{aab}([a \mid T']) \leftarrow append([ab], S_3, T')$$

Ahora desdoblamos la cláusula resultante:

$$s_{aab}([aa \mid T'']) \leftarrow append([b], S_3, T'')$$

Aplicamos de nuevo la regla de desdoblado:

$$s_{aab}([aab \mid T''']) \leftarrow append([\,], S_3, T''')$$

Y desdoblamos una última vez:

$$s_{aab}([aab \mid L]) \leftarrow$$

La secuencia de pasos de desdoblado comenzando con (5.2) tiene las siguientes propiedades interesantes:

1. cada uno de los pasos de desdoblado es *determinístico* [Gal93] en el sentido de que de cada uno de tales pasos derivamos exactamente una cláusula, y
2. desdoblamos todas las cláusulas seleccionando una submeta con el mismo símbolo de predicado.

Puesto que hallaremos frecuentemente tales secuencias de pasos de desdoblado en lo que sigue, introduciremos una abreviación. Denotaremos tal secuencia de pasos de desdoblado por:

$$s_{aab}(T) \leftarrow \underline{append([aab], S_3, T)}_4 \tag{5.2}$$

ESTA TESIS NO SALE
DE LA BIBLIOTECA

que significará que determinísticamente desdoblamos la cláusula de interés (en este caso (5.2)) seleccionando la submeta subrayada, y en donde el subíndice indica la longitud de tal secuencia de cláusulas derivadas (4, en este caso). También, diremos que tal secuencia de cláusulas es una derivación "determinística". Por lo tanto, definimos una *derivación determinística* comenzando en una cláusula C, seleccionando una submeta A, y usando una definición D, como una secuencia maximal $M = C_0, C_1, \ldots, C_l$ de cláusulas tales que cada C_i, para $0 \leq i < l$, tiene una submeta particular con el mismo símbolo de predicado que A, y C_{i+1} es derivada desde C_i por desdoblado determinístico con respecto a D. La *longitud* de la derivación determinística M es l. Nos aseguraremos de la finitud de una derivación determínistica con un paso adicional (lookahead), al que llamamos *una previsión*, que no forma parte de la derivación determinística en sí misma, por medio del cual verificamos que una cláusula generaría, por desdoblado, otras dos cláusulas más. Esta técnica para detener una sucesión de pasos que involucran desdoblados determinísticos es común en la literatura, pero posteriormente veremos que existe un tipo de derivaciones determinísticas que aprovechan información obtenida al resolver *previamente* algunas submetas (una provisión).

Habiendo ya desdoblado (5.2), podemos deshacernos de *append* al desdoblar, primero, a (5.3), seleccionando la submeta de la derecha:

$$s_{aab}([A \mid T]) \leftarrow append(X, [aab], Z),\ append(Z, S_3, T) \tag{5.4}$$

y entonces *doblando* (5.4) usando (5.1). Luego, la especificación se transforma en el programa que implementa una variante del *algoritmo de fuerza bruta bajo Prolog*:

Programa 7 (Programa de fuerza bruta bajo Prolog)

$$s_{aab}([aab \mid L]) \leftarrow \tag{5.5}$$

$$s_{aab}([A \mid L]) \leftarrow s_{aab}(L) \tag{5.6}$$

Para el caso particular del Programa 5.5, los axiomas de la teoría estándar de la igualdad 2.4 toman la siguiente forma:

$$X = X \leftarrow \tag{5.7}$$

$$X = Y \leftarrow Y = X \tag{5.8}$$

$$X = Z \leftarrow X = Y,\ Y = Z \tag{5.9}$$

$$[X_1 \mid X_2] = [Y_1 \mid Y_2] \leftarrow X_1 = Y_1,\ X_2 = Y_2 \tag{5.10}$$

$$s_{aab}(X) \leftarrow s_{aab}(Y),\ X = Y \tag{5.11}$$

en donde aplicamos la sustitutividad de predicados para s_{aab}. Estos axiomas nos permiten introducir ecuaciones en el programa de fuerza bruta basado en prefijos como sigue: Primero, resolvemos (5.5) con sustitutividad de predicados (5.11), así obteniendo:

$$s_{aab}(X) \leftarrow \underline{X = [aab \mid L]}_3 \tag{5.12}$$

A continuación, desdoblamos (5.12) utilizando la sustitutividad de funciones (5.10), lo que nos conduce a la siguiente secuencia de tres aplicaciones del desdoblado determinístico:

$$s_{aab}([A_1 : A_3 \mid Y]) \leftarrow a = A_1,\ a = A_2,\ b = A_3,\ L = Y$$

Ahora, aplicando simetría (5.8) y reflexividad (5.7), obtenemos:

$$s_{aab}([A_1 : A_3 \mid L]) \leftarrow A_1 = a,\ A_2 = a,\ A_3 = b \tag{5.13}$$

O bien, siguiendo la notación presentada en la subsección 2.5, la cláusula (5.13) queda como:

$$s_{aab}([A_1 : A_3 \mid L]) \leftarrow A_1 = a,\ A_2 = a,\ A_3 = b \tag{5.14}$$

Para un caso general, la cláusula (5.14) queda como sigue:

$$s_{a_1:a_n}([A_1 : A_n \mid L]) \leftarrow A_1 = a_1,\ \ldots,\ A_n = a_n \tag{5.15}$$

o bien, para evitar recargar la notación, y suponiendo nuestro patrón $p = p_1{:}p_m$, utilizamos el predicado s en lugar de $s_{p_1:p_m}$.

Necesitaremos también referirnos a símbolos individuales en la cláusula recursiva (5.6). Posteriormente, aplicaremos a (5.6) la sustitución $\{L/[A_2, A_3 \mid L]\}$. Notamos primero que la aplicación de esta sustitución no afecta la completitud del programa (requerir que el texto tenga al menos tantos símbolos como el patrón no modifica el modelo mínimo de Herbrand). Segundo, esta sustitución, la cual en general sería $\{L/[A_2, \ldots, A_m \mid L]\}$, depende sólo de la longitud del patrón y no de ninguna otra propiedad de tal cadena.

Programa 8

$$s_{aab}([A_1 : A_3 \mid L]) \leftarrow A_3 = b,\ A_2 = a,\ A_1 = a \tag{5.16}$$

$$s_{aab}([A_1 : A_3 \mid L]) \leftarrow s_{aab}([A_2 A_3 \mid L]) \tag{5.17}$$

Finalmente, para resaltar el hecho de que estamos procediendo de *derecha a izquierda* hemos invertido el orden de las ecuaciones en (5.16). El caso general, correspondiente a un patrón de tamaño m, $p = p_1 \cdots p_m$, lo podemos obtener fácilmente:

$$s_{aab}([A_1 : A_m \mid L]) \leftarrow A_m = p_m,\ \ldots,\ A_1 = p_1 \tag{5.18}$$

$$s_{aab}([A_1 : A_m \mid L]) \leftarrow s_{aab}([A_2 : A_m \mid L]) \tag{5.19}$$

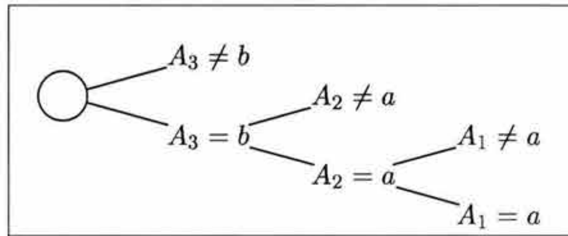

Figura 5.2: División en casos.

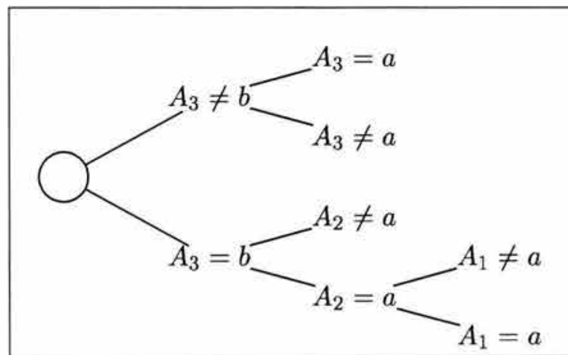

Figura 5.3: Posibilidades de la variable de más a la derecha.

Ahora comenzaremos con la derivación propiamente dicha. Para simplificar nuestra presentación, derivaremos una versión más débil que el algoritmo BM, en el sentido de que el programa resultante desplazará el patrón sobre el texto menos símbolos que este algoritmo cada vez que existe una discordancia. La ventaja de dicha versión débil del algoritmo BM es la brevedad, a la vez que ilustramos las ideas principales de manera directa.

La Fig. 5.2 presenta el proceso sistemático de la división de casos que necesitamos para considerar cada sufijo del patrón junto con el casamiento parcial apropiado.

Aplicando reiteradamente la regla de división en casos a la cláusula (5.17) obtenemos el siguiente programa:

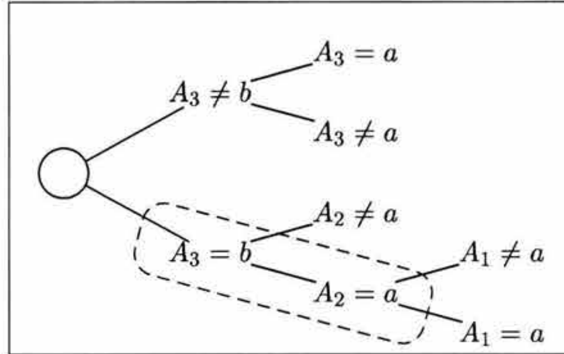

Figura 5.4: La trayectoria correspondiente a un casamiento parcial.

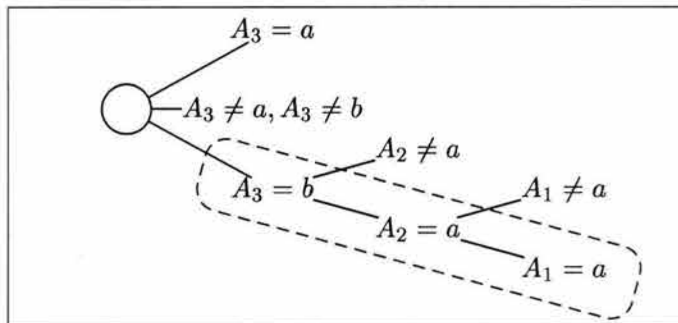

Figura 5.5: La trayectoria correspondiente a un casamiento total.

Programa 9

$$s_{aab}([A_1 : A_3 \mid L]) \leftarrow A_3 = b,\ A_2 = a,\ A_1 = a \tag{5.20}$$

$$s_{aab}([A_1 : A_3 \mid L]) \leftarrow A_3 \neq b,\ s_{aab}([A_2 A_3 \mid L]) \tag{5.21}$$

$$s_{aab}([A_1 : A_3 \mid L]) \leftarrow A_3 = b,\ A_2 \neq a,\ s_{aab}([A_2 A_3 \mid L]) \tag{5.22}$$

$$s_{aab}([A_1 : A_3 \mid L]) \leftarrow A_3 = b,\ A_2 = a,\ A_1 \neq a,\ s_{aab}([A_2 A_3 \mid L]) \tag{5.23}$$

$$s_{aab}([A_1 : A_3 \mid L]) \leftarrow A_3 = b,\ A_2 = a,\ A_1 = a,\ s_{aab}([A_2 A_3 \mid L]) \tag{5.24}$$

Estas cláusulas cubren los siguientes casos:

1. una discordancia inmediata (5.21),
2. el casamiento de sufijos (casamientos parciales del patrón) (5.22), (5.23), o
3. un casamiento completo (5.20), (5.24).

En particular la cláusula (5.24) cubre el caso de una ocurrencia total pero también la preparación del programa para encontrar más ocurrencias del patrón en el texto, algo que normalmente los especializadores en la literatura no realizan. Diremos que este programa tiene una forma *sufijo-triangular*.

Por ahora, nosotros no sólo deseamos saber si el símbolo de más a la derecha del patrón y el correspondiente del texto casan, sino también explorar un punto crucial del algoritmo BM, que consiste en determinar si tal símbolo del texto *ocurre en el patrón*. Para este propósito, aplicamos la regla de división en casos a (5.20) con respecto a las submetas $A_3 \in [aab]$ y $A_3 \notin [aab]$, y para enfatizar que no deseamos listas sino conjuntos, usamos para $\in$ y $\notin$ la siguiente notación: $A_3 \in \{aab\}$ y $A_3 \notin \{aab\}$, respectivamente, para señalar que tomamos el *conjunto de los símbolos de la lista* $[aab]$ *en* $\{aab\}$:

Cláusulas 6

$$s_{aab}([A_1 : A_3 \mid L]) \leftarrow A_3 \neq b,\ A_3 \in \{aab\}, \tag{5.25}$$

$$s_{aab}([A_2 A_3 \mid L]) \tag{5.26}$$

$$s_{aab}([A_1 : A_3 \mid L]) \leftarrow A_3 \neq b,\ A_3 \notin \{aab\}, \tag{5.27}$$

$$s_{aab}([A_2 A_3 \mid L]) \tag{5.28}$$

(En realidad, las metas $A_3 \in \{aab\}$ y $A_3 \notin \{aab\}$ representan una manera abreviada de una utilización extensiva de la regla de división en casos.)

Las definiciones de $\in$ y $\notin$ son:

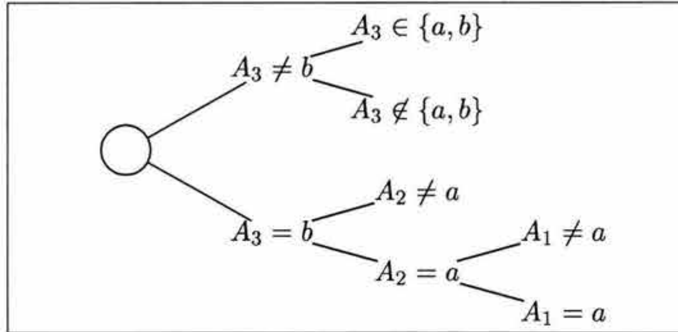

Figura 5.6: Posibilidades de A_3 dado un mal casamiento en b.

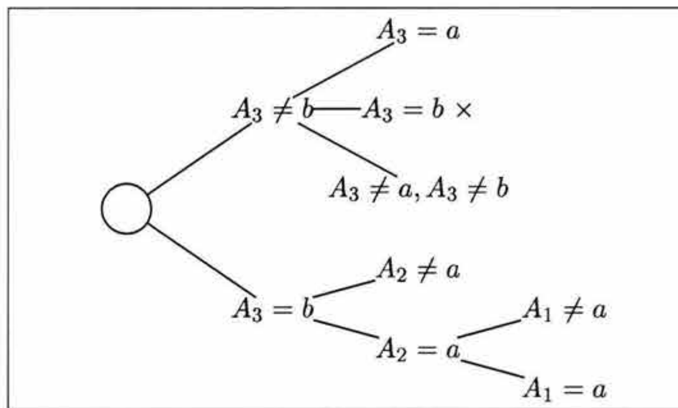

Figura 5.7: Eliminación de $\in$ y $\notin$.

Cláusulas 7

$$X_1 \in [X_2 \mid Ys] \leftarrow X_1 = X_2 \tag{5.29}$$
$$X_1 \in [X_2 \mid Ys] \leftarrow X_1 \in Ys \tag{5.30}$$

$$X_1 \notin [\,] \leftarrow \tag{5.31}$$
$$X_1 \notin [X_2 \mid Ys] \leftarrow X_1 \neq X_2,\ X_1 \notin Ys \tag{5.32}$$

A continuación, desdoblamos las cláusulas (5.26) y (5.28) con respecto a las definiciones de $\in$ y $\notin$. Eliminamos las cláusulas resultantes con un cuerpo que falla (por ejemplo, eliminamos las cláusulas con las submetas $A_3 \neq b$ y $A_3 = b$; ver Fig. 5.7, en donde marcamos con $\times$ este par de submetas contradictorias). Similarmente, simplificamos las submetas que están repetidas utilizando factorización.

Programa 10

$$s_{aab}([A_1 : A_3 \mid L]) \leftarrow A_3 = b,\ A_2 = a,\ A_1 = a \tag{5.33}$$

$$\begin{aligned}
&s_{aab}([A_1 : A_3 \mid L]) \leftarrow A_3 \neq b,\ A_3 = a,\ \underline{s_{aab}([A_2 A_3 \mid L])}_0\\
&s_{aab}([A_1 : A_3 \mid L]) \leftarrow A_3 \neq b,\ A_3 \neq a,\ \underline{s_{aab}([A_2 A_3 \mid L])}_2\\
&s_{aab}([A_1 : A_3 \mid L]) \leftarrow A_3 = b,\ A_2 \neq a,\ \underline{s_{aab}([A_2 A_3 \mid L])}_2\\
&s_{aab}([A_1 : A_3 \mid L]) \leftarrow A_3 = b,\ A_2 = a,\ A_1 \neq a,\ \underline{s_{aab}([A_2 A_3 \mid L])}_2\\
&s_{aab}([A_1 : A_3 \mid L]) \leftarrow A_3 = b,\ A_2 = a,\ A_1 = a,\ \underline{s_{aab}([A_2 A_3 \mid L])}_2
\end{aligned} \tag{5.34}$$

En el siguiente paso, desdoblamos determinísticamente cada cláusula con respecto a (5.16) y (5.17), y repetimos este proceso con las cláusulas resultantes tantas veces como sea posible. Los desdoblados determinísticos que realizamos necesitan una menor previsión para concluir que realmente sólo una cláusula es derivada a la vez (básicamente, en nuestro caso, debemos verificar que a una cláusula, de las dos que se generan, la podemos eliminar inmediatamente). Sin embargo, tal previsión es normal, y es considerado en la literatura relacionada al desdoblado determinístico estándar [Gal93].

Desdoblando la cláusula (5.34), por ejemplo, con respecto a (5.16) produce la siguiente cláusula

$$s_{aab}([A_1 : A_4 \mid L]) \leftarrow A_3 \neq b,\ A_3 \neq a,\ A_4 = b,\ A_3 = a,\ A_2 = a$$

la cual tiene un cuerpo que falla y la descartamos. Alternativamente, el siguiente conjunto de ecuaciones e inecuaciones

$$\{A_3 \neq b,\ A_3 \neq a,\ A_4 = b,\ A_3 = a,\ A_2 = a\}$$

no tiene solución.

El desdoblado con respecto a (5.17), a su vez, produce la siguiente cláusula:

$$s_{aab}([A_1 : A_3 \mid L]) \leftarrow A_3 \neq b,\ A_3 \neq a,\ s_{aab}([A_3 \mid L])$$

Este proceso puede ser realizado una vez más, produciendo la siguiente cláusula:

$$s_{aab}([A_1 : A_3 \mid L]) \leftarrow A_3 \neq b,\ A_3 \neq a,\ s_{aab}(L)$$

Como ya hemos visto, otra forma de manejar la previsión consiste en determinar si un conjunto de ecuaciones e inecuaciones tiene solución. Ya que en la cláusula (5.17) no existe ninguna ecuación, podemos utilizar esta cláusula en toda aplicación de la regla de desdoblado. Detenemos cada proceso o secuencia de desdoblados cuando un conjunto de ecuaciones e inecuaciones, que son el resultado de utilizar (5.16), tiene una solución (es decir, un desdoblamiento nos daría *dos* cláusulas y sería entonces no-determinístico). Note que cada vez que utilizamos la cláusula (5.17) podemos pensar que el patrón está siendo desplazado un símbolo a la derecha, por lo que, intuitivamente, estos desdoblados mueven el patrón tantos símbolos como es posible, dada la respectiva satisfacción de las restricciones contenidas en el cuerpo de cada cláusula en forma de ecuaciones e inecuaciones.

El programa resultante, junto con (5.16), es:

Programa 11

$$\begin{array}{l}
s_{aab}([A_1A_2A_3 \mid L]) \leftarrow \boxed{A_3 = b},\ A_2 = a,\ A_1 = a \\
s_{aab}([A_1A_2A_3 \mid L]) \leftarrow A_3 \neq b,\ A_3 = a,\ s_{aab}([A_2A_3 \mid L]) \\
s_{aab}([A_1A_2A_3 \mid L]) \leftarrow A_3 \neq b,\ A_3 \neq a,\ s_{aab}(L) \qquad (5.35) \\
s_{aab}([A_1A_2A_3 \mid L]) \leftarrow \boxed{A_3 = b},\ A_2 \neq a,\ s_{aab}(L) \\
s_{aab}([A_1A_2A_3 \mid L]) \leftarrow \boxed{A_3 = b},\ A_2 = a,\ A_1 \neq a,\ s_{aab}(L) \\
s_{aab}([A_1A_2A_3 \mid L]) \leftarrow \boxed{A_3 = b},\ A_2 = a,\ A_1 = a,\ s_{aab}(L)
\end{array}$$

Suponiendo una ejecución de arriba a abajo, y de izquierda a derecha (tipo Prolog), la cláusula (5.35), por ejemplo, la leemos proceduralmente como: "si el tercer símbolo del texto no aparece en el patrón, entonces desplace el patrón tres posiciones".

Tal como está el programa anterior, existe una enorme cantidad de comparaciones que son innecesarias. Por ello, a continuación definimos nuevos predicados para aplicar la regla del doblado, con la guía en nuestra derivación de evitar el no-determinismo innecesario y la ineficiencia latente al resolver dos o más veces algunas submetas.

Todas las cláusulas a ser dobladas tienen una submeta encerrada en un rectágulo. Las submetas *dentro* del rectángulo son las mismas en todas esas cláusulas, y representan las submetas que, de manera redundante, y por lo tanto ineficiente, tienen que ser resueltas. Las submetas *fuera* del rectángulo en cada una de tales cláusulas aparecerá como el cuerpo de una cláusula de un nuevo predicado.

Continuando con la derivación, definimos el nuevo predicado $match1$:

Cláusulas 8

$$match1(A_1, A_2, A_3, L) \leftarrow A_2 = a,\ A_1 = a$$
$$match1(A_1, A_2, A_3, L) \leftarrow A_2 \neq a,\ s_{aab}(L)$$
$$match1(A_1, A_2, A_3, L) \leftarrow A_2 = a,\ A_1 \neq a,\ s_{aab}(L)$$
$$match1(A_1, A_2, A_3, L) \leftarrow A_2 = a,\ A_1 = a,\ s_{aab}(L)$$

Así, podemos inferir lo siguiente:

Cláusulas 9

$$s_{aab}([A_1 A_2 A_3 \mid L]) \leftarrow A_3 = b,\ match1(A_1, A_2, A_3, L) \tag{5.36}$$
$$s_{aab}([A_1 A_2 A_3 \mid L]) \leftarrow \boxed{A_3 \neq b},\ A_3 = a,\ s_{aab}([A_2 A_3 \mid L])$$
$$s_{aab}([A_1 A_2 A_3 \mid L]) \leftarrow \boxed{A_3 \neq b},\ A_3 \neq a,\ s_{aab}(L)$$

(Note que si desdoblamos (5.36) con respecto a $match1$ obtendríamos el programa previo. Este es un caso particular del doblado local o *in situ*. Ver Capítulo 3.)

En una forma similar, ahora definimos otro predicado, que llamaremos *table*; este predicado determinará el tamaño del desplazamiento cuando la comparación con el símbolo de más a la derecha del patrón falle. Para tal símbolo, entonces, el predicado *table* desempeña el papel de la función δ_1 del algoritmo BM.

Después de doblar, obtenemos:

Cláusulas 10

$$table(A_1, A_2, A_3, L) \leftarrow A_3 = a,\ s_{aab}([A_2 A_3 \mid L]) \tag{5.37}$$
$$table(A_1, A_2, A_3, L) \leftarrow A_3 \neq a,\ s_{aab}(L) \tag{5.38}$$

$$s_{aab}([A_1 : A_3 \mid L]) \leftarrow A_3 = b,\ match1(A_1, A_2, A_3, L)$$
$$s_{aab}([A_1 : A_3 \mid L]) \leftarrow A_3 \neq b,\ table(A_1, A_2, A_3, L)$$

Observamos ahora que también podemos doblar en la definición de *match*1

Cláusulas 11

$$
\begin{aligned}
&match1(A_1, A_2, A_3, L) \leftarrow \boxed{A_2 = a},\ A_1 = a \\
&match1(A_1, A_2, A_3, L) \leftarrow A_2 \neq a,\ s_{aab}(L) \\
&match1(A_1, A_2, A_3, L) \leftarrow \boxed{A_2 = a},\ A_1 \neq a,\ s_{aab}(L) \\
&match1(A_1, A_2, A_3, L) \leftarrow \boxed{A_2 = a},\ A_1 = a,\ s_{aab}(L)
\end{aligned}
$$

con respecto a un nuevo predicado *match*2:

Cláusulas 12

$$
\begin{aligned}
&match2(A_1, A_2, A_3, L) \leftarrow \boxed{A_1 = a} \\
&match2(A_1, A_2, A_3, L) \leftarrow A_1 \neq a,\ s_{aab}(L) \\
&match2(A_1, A_2, A_3, L) \leftarrow \boxed{A_1 = a},\ s_{aab}(L) \\
&match1(A_1, A_2, A_3, L) \leftarrow A_2 = a,\ match2(A_1, A_2, A_3, L) \\
&match1(A_1, A_2, A_3, L) \leftarrow A_2 \neq a,\ s_{aab}(L)
\end{aligned}
$$

El proceso de doblado finaliza con la introducción de *match*3 y el respectivo doblado de *match*2 con respecto al predicado recién introducido:

Cláusulas 13

$$
\begin{aligned}
&match3(L) \leftarrow \\
&match3(L) \leftarrow s_{aab}(L) \\
&match2(A_1, A_2, A_3, L) \leftarrow A_1 = a,\ match3(L) \\
&match2(A_1, A_2, A_3, L) \leftarrow A_1 \neq a,\ s_{aab}(L)
\end{aligned}
$$

Obtenemos, por lo tanto, el siguiente programa final:

Programa 12

$$table(A_1, A_2, A_3, L) \leftarrow A_3 = a,\ s_{aab}([A_2 A_3 \mid L])$$
$$table(A_1, A_2, A_3, L) \leftarrow A_3 \neq a,\ s_{aab}(L)$$

$$s_{aab}([A_1 A_2 A_3 \mid L]) \leftarrow A_3 = b,\ match1(A_1, A_2, A_3, L)$$
$$s_{aab}([A_1 A_2 A_3 \mid L]) \leftarrow A_3 \neq b,\ table(A_1, A_2, A_3, L)$$

$$match1(A_1, A_2, A_3, L) \leftarrow A_2 = a,\ match2(A_1, A_2, A_3, L)$$
$$match1(A_1, A_2, A_3, L) \leftarrow A_2 \neq a,\ s_{aab}(L)$$

$$match2(A_1, A_2, A_3, L) \leftarrow A_1 = a,\ match3(L)$$
$$match2(A_1, A_2, A_3, L) \leftarrow A_1 \neq a,\ s_{aab}(L)$$

$$match3(L) \leftarrow$$
$$match3(L) \leftarrow s_{aab}(L)$$

El programa resultante captura la esencia del algoritmo BM. Existen, sin embargo, un número de detalles de implementación que debemos considerar. No vemos tales detalles como parte de la derivación en sí misma, si no más bien como un conjunto de suposiciones acerca de la implementación de nuestros programas lógicos.

Primero, suponemos una interpretación tipo Prolog, a saber, de arriba a abajo según las cláusulas, y de izquierda a derecha según las submetas. Similarmente, podemos intercalar adecuadamente algunos cortes (tipo Prolog) para manejar pruebas complementarias, de modo que evitemos duplicar comparaciones en el caso en que la primera comparación falla (los cortes, vistos de esta forma, forma parte del repertorio actual para transformar programas lógicos implementados en Prolog [PPR97a]).

La unificación, sin embargo, tal como está implementada en Prolog, no nos conviene, ya que esta operación accedería a tantos elementos de una lista como símbolos hay en el patrón en toda llamada diferente de una ecuación o inecuación. Las listas tampoco nos son adecuadas por una razón similar. Por lo tanto, en lugar de la unificación y las listas, tenemos que usar apuntadores y arreglos, de modo que la unificación sea implementada como un *goto* más ciertas operaciones con apuntadores.

Consideremos a continuación el predicado *table* (cláusulas (5.37) y (5.38)). Ya que este predicado determina el tamaño del desplazamiento, debemos tener la posibilidad de

acceder tal predicado en tiempo constante. En general, tendremos cláusulas de la forma

$$\begin{array}{ll} & \mathit{table}(A_1, \ldots, A_m, L) \leftarrow A_m = p_j,\ s_p([A_k, \ldots, A_m \mid L]) \\ \text{o} & \mathit{table}(A_1, \ldots, A_m, L) \leftarrow A_m \neq p_1,\ \ldots,\ A_m \neq p_m,\ s_p(L) \end{array}$$

en donde $k - 1$ es el número de símbolos que nosotros tenemos para desplazar el patrón. Logramos esta conducta (el tiempo de acceso constante) *desdoblando* las ecuaciones y las inecuaciones usando la teoría de igualdad de Clark (la cual tiene un axioma $p_i \neq p_j \leftarrow$ para todos los pares p_i, p_j de constantes que sean distintas.) Además, debemos suponer una implementación capaz de *indexar* este predicado en la ubicación de su m-ésimo argumento o introducir un predicado teniendo A_m en su primer argumento (ya que muchas implementaciones sí indexan en el primer argumento de los predicados que forman una definición, si tal argumento existe).

El programa entonces queda transformado en:

Programa 13

$$\begin{array}{l} s_{aab}([A_1A_2A_3 \mid L]) \leftarrow A_3 = b,\ \mathit{match1}(A_1, A_2, A_3, L) \\ s_{aab}([A_1A_2A_3 \mid L]) \leftarrow A_3 \neq b,\ \mathit{table}(A_1, A_2, A_3, L) \\ \mathit{match1}(A_1, A_2, A_3, L) \leftarrow A_2 = a,\ \mathit{match2}(A_1, A_2, A_3, L) \\ \mathit{match1}(A_1, A_2, A_3, L) \leftarrow A_2 \neq a,\ s_{aab}(L) \\ \mathit{match2}(A_1, A_2, A_3, L) \leftarrow A_1 = a,\ \mathit{match3}(L) \\ \mathit{match2}(A_1, A_2, A_3, L) \leftarrow A_1 \neq a,\ s_{aab}(L) \\ \mathit{match3}(L) \leftarrow \\ \mathit{match3}(L) \leftarrow s_{aab}(L) \\ \mathit{table}(A_1, A_2, a, L) \leftarrow s_{aab}([A_2, a \mid L]) \\ \mathit{table}(A_1, A_2, b, L) \leftarrow s_{aab}(L) \\ \mathit{table}(A_1, A_2, c, L) \leftarrow s_{aab}(L) \\ \qquad \vdots \\ \mathit{table}(A_1, A_2, z, L) \leftarrow s_{aab}(L) \end{array}$$

En el Capítulo 8 veremos que es posible derivar una variante del algoritmo BM que sólo toma en cuenta la tabla δ_2, pero tiene la ventaja de que esa derivación es automática, y además eliminamos los problemas de acceso a una lista al utilizar una base de datos de hechos formada por los símbolos del texto e índices.

El último punto que tratamos es el hecho de que el Programa 13 tiene todavía cierto no-determinismo. La versión de ocurrencias múltiples del problema de casamiento de cadenas es no-determinística por naturaleza, en el sentido de que tal problema puede tener más de una respuesta. Este no-determinismo, no obstante, nos permite hallar todas las ocurrencias del patrón en el texto, *aprovechando inclusive que el patrón ya ha sido hallado una vez previa a una siguiente.* En efecto, al encontrar una ocurrencia del patrón, y si en este patrón un sufijo es también un prefijo, la ubicación del nuevo intento de casamiento, de acuerdo con el programa residual obtenido, es tal que la ubicación del prefijo y del sufijo coinciden, por lo que el nuevo intento de casamiento ya tiene asegurado un desplazamiento óptimo para reencontrar el patrón, si existiera una reocurrencia del mismo.

Note también que el no-determinismo del programa final no involucra ramas fallidas, a diferencia del algoritmo de fuerza bruta. Podemos, en todo caso, utilizar los métodos desarrollados por Ueda [Ued87] y Tamaki [Tam87], que nos permiten recorrer determinísticamente espacios de búsqueda, para calcular todas las respuestas determinísticamente.

Resumiendo, hemos dado una derivación que consiste de:

1. una repetida aplicación la regla de división en casos, para obtener una forma sufijo-triangular:

$$
\begin{aligned}
s([A_1 : A_m \mid L]) &\leftarrow A_m = p_m, \ldots,\ A_1 = p_1 \\
s([A_1 : A_m \mid L]) &\leftarrow A_m = p_m, \ldots,\ A_{j+1} = p_{j+1}, \\
&\qquad A_j \neq p_j,\ s([A_2, \ldots, A_m \mid L]) \\
&\qquad\qquad \text{para } j = m, m-1, \ldots, 1 \\
s([A_1 : A_m \mid L]) &\leftarrow A_m = p_m, \ldots,\ A_1 = p_1, \\
&\qquad s([A_2 : A_m \mid L])
\end{aligned}
$$

2. una aplicación de la división en casos con respecto a $\in$ y $\notin$ al símbolo del patrón de más a la derecha:

$$
\begin{aligned}
s([A_1 : A_m \mid L]) &\leftarrow A_m \neq p_m, \\
&\qquad A_m \in \{p_1 : p_m\}, \\
&\qquad s([A_2 : A_m \mid L]) \\
s([A_1 : A_m \mid L]) &\leftarrow A_m \neq p_m, \\
&\qquad A_m \notin \{p_1 : p_m\}, \\
&\qquad s([A_2 : A_m \mid L])
\end{aligned}
$$

y un repetido desdoblado con respecto a $\in$ y $\notin$, que permita eliminar la ocurrencia de estos predicados,

3. la aplicación, de manera repetitiva, del desdoblado determinístico para permitir el avance del patrón, así como de la factorización de submetas comunes tantas veces como sea posible, y

4. la introducción de definiciones, también de manera repetitiva, y el doblado de cláusulas teniendo respectivamente las submetas $A_m = p_m$, $A_m \neq p_m$, $A_{m-1} = p_{m-1}$, $A_{m-1} \neq p_{m-1}$, $\ldots$, $A_1 = p_1$, y $A_1 \neq p_1$.

En la siguiente sección abundaremos en los detalles y en las variantes de este proceso.

5.2 Mayores detalles de nuestra derivación

Ahora relacionaremos nuestra derivación previa con el algoritmo BM en más detalle. Primero, veremos por qué es que nuestra derivación produce un programa más débil que el del algoritmo BM, al relacionar nuestra derivación con δ_1, y entonces corregiremos esta deficiencia. A continuación, mencionaremos cómo obtener una tabla que juega el papel de δ_2. Además, estableceremos algunos lemas en los cuales nos referiremos a las derivaciones determinísticas (i.e. secuencias maximales de desdoblados determinísticos) usando las cláusulas

Programa 14

$$s([p_1 : p_m \mid L]) \leftarrow \tag{5.39}$$

$$s([A \mid L]) \leftarrow s(L) \tag{5.40}$$

Abordamos a continuación en su mayor generalidad el problema de derivar una especificación alternativa que también resuelve el problema de casamiento de cadenas.

Definimos un predicado $occurs_in(P, T)$, cuya definición resuelve el problema del casamiento de cadenas y cuyo significado deseado es que $occurs_in(P, T)$ se cumple si P ocurre en T:

Programa 15

$$occurs_in(P, T) \leftarrow append(T_1, P, T_2),\ append(T_2, T_3, T) \tag{5.41}$$

en donde hemos utilizado la definición usual de *append*:

$$append([\,], Xs, Xs) \leftarrow \tag{5.42}$$

$$append([A \mid Xs], Ys, [A \mid Zs]) \leftarrow append(Xs, Ys, Zs) \tag{5.43}$$

Ahora definimos el predicado s_p que especializa el predicado *occurs_in* a la pregunta $occurs_in([p_1 : p_m], T)$ utilizando la siguiente secuencia de pasos:

$$s_p(T) \leftarrow \underline{occurs_in([p_1 : p_m], T)} \rightsquigarrow \{\!\![\text{ por (5.41) }]\!\!\}$$

$$s_p(T) \leftarrow \underline{append(T_1, [p_1 : p_m], T_2)},\ append(T_2, T_3, T) \rightsquigarrow$$

$$\left[\begin{array}{l} s_p(T) \leftarrow append([p_1 : p_m], T_3, T) \\ s_p(T) \leftarrow append(T_1, [p_1 : p_m], T_2),\ \underline{append([A \mid T_2], T_3, T)} \end{array}\right. \rightsquigarrow$$

$$\left[\begin{array}{l} s_p(T) \leftarrow append([p_1 : p_m], T_3, T) \\ s_p([A \mid T]) \leftarrow \boxed{append(T_1, [p_1 : p_m], T_2),\ append(T_2, T_3, T)} \end{array}\right. \rightsquigarrow \{\!\![\text{ doblado }]\!\!\}$$

$$\left[\begin{array}{l} s_p(T) \leftarrow \underline{append([p_1 : p_m], T_3, T)}_{m+1} \\ s_p([A \mid T]) \leftarrow s_p(T) \end{array}\right.$$

y al desdoblar *append* obtenemos el siguiente programa:

Programa 16

$$s_p([p_1 : p_m \mid L]) \leftarrow \tag{5.44}$$

$$s_p([A \mid L]) \leftarrow s_p(L) \tag{5.45}$$

De ahora en adelante, tomamos a este programa como nuestra especificación lógica que resuelve el problema del casamiento de cadenas.

A continuación introducimos las ecuaciones que nos permiten hacer las comparaciones símbolo a símbolo entre el patrón y el texto, y luego invertimos el orden de ocurrencia de las ecuaciones, de tal manera que enfatizamos el hecho de que, en el algoritmo BM y ahora, estamos casando el patrón con el texto *de derecha a izquierda*.

Desdoblando la cláusula (5.44), y usando (5.45) obtenemos:

$$s_p(X) \leftarrow \underline{X = [p_1 : p_m \mid L]}_m \rightsquigarrow \tag{5.46}$$

$$s_p([A_1 : A_m \mid L]) \leftarrow A_1 : A_m = p_1 : p_m,\ \underline{L = L} \rightsquigarrow \{\!\![\text{ reflexividad }]\!\!\} \tag{5.47}$$

$$s_p([A_1 : A_m \mid L]) \leftarrow A_1 : A_m = p_1 : p_m\ . \tag{5.48}$$

Al aplicar la sustitución $\theta = \{A/A_1, L/[A_2 : A_m \mid L_1]\}$ a (5.45), y al reordenar las submetas en el cuerpo de (5.48) obtenemos un programa el cual tiene una adecuada formulación para nuestros propósitos, ya que hemos hecho explícita la comparación de símbolos, y ahora tales comparaciones han tomado la forma de de un conjunto de ecuaciones que deben satisfacerce; finalmente, notemos que podemos ahora controlar el orden en el cual casamos el patrón con el texto.

Programa 17

$$s_p([A_1 : A_m \mid L]) \leftarrow A_m : A_1 = p_m : p_1 \tag{5.49}$$

$$s_p([A_1 : A_m \mid L]) \leftarrow s_p([A_2 : A_m \mid L]) \ . \tag{5.50}$$

Dado que sólo hemos realizado operaciones lógicas en el Programa 16 que nos brindan la equivalencia lógica con el Programa 17, procedemos al siguiente paso.

Observe que en el caso de casamiento parcial (i.e. de un sufijo) seguido por una discordancia, este algoritmo puede determinar (debido al preprocesamiento) si el símbolo actual del texto ocurre en el patrón. Sin embargo, nosotros sólo hemos aplicado la regla de división en casos con respecto a $\in$ y $\notin$ para el símbolo de más a la derecha del patrón. Por lo tanto, ahora tenemos que aplicar la regla de división en casos con respecto a estos predicados ($\in$ y $\notin$) a *todos* los símbolos del patrón (y en cada una de las posiciones respectivas), lo cual no hicimos antes tanto por brevedad como por nuestro deseo de explorar una variante sencilla del algoritmo BM. Ahora bien, es el momento de reemplazar el paso 2 por:

2'. la aplicación de la regla de división en casos con respecto a $\in$ y $\notin$ a todos los símbolos del patrón (y en las respectivas posiciones):

$$\begin{aligned}
s([A_1 : A_m \mid L]) \leftarrow\ & A_m : A_{j+1} = p_m : p_{j+1},\ A_j \neq p_j, \\
& A_j \in \{p_1 : p_m\}, \\
& s([A_2 : A_m \mid L]) \\
s([A_1 : A_m \mid L]) \leftarrow\ & A_m : A_{j+1} = p_m : p_{j+1},\ A_j \neq p_j, \\
& A_j \notin \{p_1 : p_m\}, \\
& s([A_2 : A_m \mid L]) \\
& \qquad \text{para } j = m, m-1, \ldots, 1
\end{aligned}$$

y desdoblando con respecto a $\in$ y $\notin$ (ver Fig. 5.8 y Fig. 5.9).

La mejora propuesta consiste, entonces, en sustituir $A_i \neq p_i$ por $A_i \notin [p_1 : p_m] \vee A_i \in [p_1 : p_m]$, para cada ecuación involucrada. (Al desdoblar, sin embargo, obtenemos tal

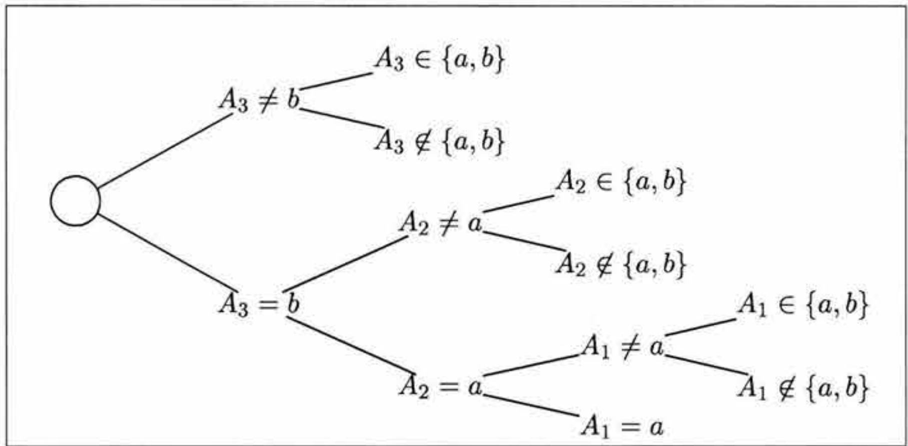

Figura 5.8: $\in$ y $\notin$ a todo nivel.

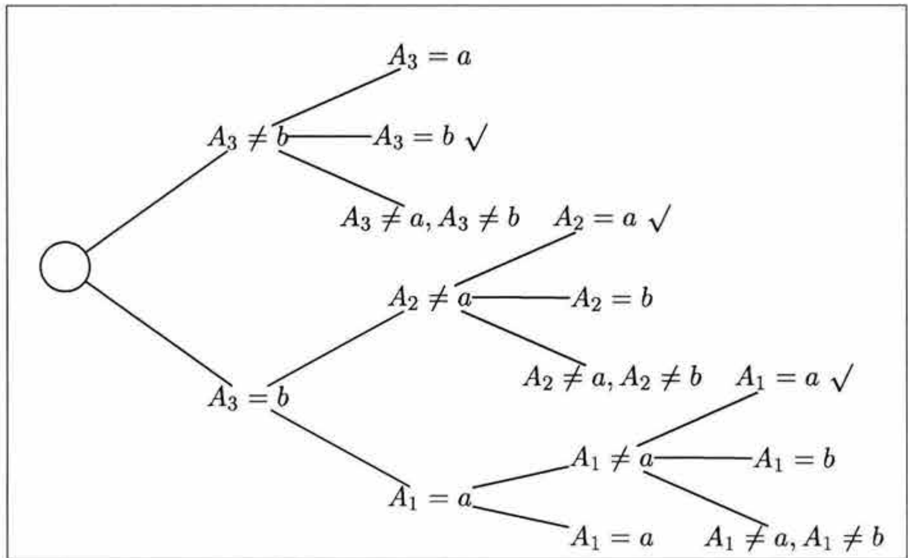

Figura 5.9: Eliminación de $\in$ y $\notin$.

profusión de casos que los programas se hacen textualmente muy grandes —para un patrón de tamaño m, existen al menos $m * (m-1)$ cláusulas.)

De hecho, si no consideramos el paso 1, entonces los pasos 2' y 3 producen δ_1, como a continuación establecemos en los siguientes dos lemas.

Primero, estableceremos una relación entre δ_1 y las cláusulas resultantes de desdoblar $\in$:

Teorema 2 *Supongamos que tenemos un patrón $p = p_1 \cdots p_m$, de tamaño m. Sea C una cláusula de la forma:*

$$s([A_1 : A_m \mid L]) \leftarrow A_j = p_k,\ s([A_2 : A_m \mid L])$$

en donde la ocurrencia de más a la derecha de p_k dentro del patrón p está a la izquierda del símbolo p_j del mismo patrón p. Entonces

$$det_unf(C) = \delta_1(p_k) - (m - j + 1) \ ,$$

en donde $det_unf(C)$ es la longitud de la derivación determinística comenzando en C, seleccionando la submeta $s([A_2, \ldots, A_m \mid L])$, y usando (5.49) y (5.50).

Demostración. Recordemos que la definición de δ_1 es:

$$\delta_1(x) = \begin{cases} m, & x \notin \{p_1, \ldots, p_m\}; \\ m - k, & k = máx\{j \in N \mid p_j = x\} \end{cases} \tag{5.51}$$

El símbolo p_k, de acuerdo con las hipótesis de nuestro teorema es tal que $k \in \{1, \ldots, j-1\}$, por lo que p_k está a la izquierda de p_j, y es algún símbolo de la región encerrada en el rectángulo:

$$p: \quad \boxed{p_1 \quad p_2 \quad \cdots \quad p_{j-1}} \quad p_j \quad \cdots \quad p_{m-1} \quad p_m \tag{5.52}$$

Inicialmente, tenemos la siguiente configuración, con las variables incluidas:

$$\begin{array}{cccccccc} p: & p_1 & p_2 & \cdots & p_j & \cdots & p_{m-1} & p_m \\ & & A_2 & \cdots & A_j & \cdots & A_{m-1} & A_m \end{array} \tag{5.53}$$

(En la submeta recursiva las variables unifican desde A_2 en adelante.)

Considerando la ecuación $A_j = p_k$, ahora depende de $k \in \{1, \ldots, j-1\}$ dentro del recuadro para realizar 0 o más desdoblados determinísticos. Consideramos dos casos: suponemos que o p_k está inmediatamente a la izquierda de p_j (una separación de 0 símbolos), o bien, en el segundo caso, que p_k y p_j están separados por 1 o más símbolos.

Para el primer caso, tenemos:

$$\begin{array}{lllllllllll} p: & p_1 & p_2 & \cdots & \cdots & p_{k-1} & p_k & p_j & \cdots & p_{m-1} & p_m \\ & & & & & & p_{j-1} & & & & \\ & & A_2 & \cdots & & & A_j & & \cdots & A_{m-1} & A_m \end{array} \quad (5.54)$$

por lo que $A_j = p_k$ $(p_{j-1} = p_k)$ se satisface y no es necesario realizar ningún desdoblado. Así que

$$det_unf(C) = 0 = \delta_1(p_k) - (m - j + 1), \quad (5.55)$$

y como la ocurrencia de más a la derecha de p_k está en $j - 1$, tenemos

$$(m - (j - 1)) - (m - j + 1) = m - j + 1 - m + j - 1 = 0. \quad (5.56)$$

Para el segundo caso, tenemos que existen d símbolos entre p_k y p_j, es decir, $p_k = p_{j-d}$ y $d > 1$ (estrictamente).

$$\begin{array}{lllllllllll} p: & p_1 & p_2 & \cdots & p_{j-d-1} & p_{j-d} & \cdots & \cdots & p_j & \cdots & p_{m-1} \quad p_m \\ & A_2 & \cdots\ \cdots & \cdots & \cdots & \cdots & & A_j & \cdots & A_{m-1} & A_m \\ & & & & & & & a_{j-1} & & & \\ & & & & & \longleftarrow d-1 \longrightarrow & & & & & \end{array} \quad (5.57)$$

(ya que $(j-1) - (j-d) = d-1$.) Entonces, para que la ecuación $A_j = p_k$ $(A_j = p_{j-d})$ se cumpla es necesario desdoblar $d - 1$ veces, por lo que

$$d - 1 = \delta(p_k) - (m - j + 1) = m - (j - d) - (m - j + 1) = d - 1$$

y la fórmula propuesta de nuevo es válida. ∎

Ahora hacemos algo parecido para las cláusulas resultantes de desdoblar $\notin$:

Corolario 1 (Avance en el caso de un símbolo que no pertenece a $\{P\}$) *Sea C una cláusula de la forma*

$$\begin{aligned} s([A_1 : A_m \mid L]) \leftarrow & A_j \neq a_1, \ \ldots, \ A_j \neq p_m, \\ & s([A_2 : A_m \mid L]) \end{aligned}$$

Tenemos, entonces, que $det_unf(C) = \delta_1(x) - (m - j + 1)$, en donde $x \notin \{p_1, \ldots, p_m\}$, y $det_unf(C)$ es la longitud de la derivación determinística comenzando en C, cuando seleccionamos a la submeta $s([A_2 : A_m \mid L])$, y usamos a (5.49) *and* (5.50).

Demostración. Para realizar la demostración de este corolario, construyamos la siguiente configuración:

$$\begin{array}{llllllllll} xp: & x & p_1 & p_2 & \dots & \dots & p_j & \dots & p_{m-1} & p_m \\ & & A_2 & \dots & \dots & \dots\ \dots & A_j & \dots & A_{m-1} & A_m \\ & \longleftarrow & & j-1 & \longrightarrow & & & & & \end{array} \tag{5.58}$$

Tenemos, entonces, que $j-1=\delta_1(x)-(m-j+1)=m-(m-j+1)=j-1$, lo que hace válida nuestra fórmula. ■

A continuación enfocamos nuestra atención a δ_2. Esta función resulta de los pasos 1 y 3 de la derivación (i.e. de aplicar el desdoblado determinístico a la forma sufijo-triangular).

Teorema 3 *Sea C una cláusula de la forma*

$$\begin{aligned} s([A_1:A_m\,|\,L]) \leftarrow A_m = p_m,\ \dots,\ & A_{j+1}=p_{j+1}, \\ & A_j \neq p_j, \\ & s([A_2:A_m\,|\,L]) \end{aligned}$$

Tenemos entonces que

$$det_unf(C)=\delta_2(j)-(m-j+1) \tag{5.59}$$

en donde $det_unf(C)$ es la longitud de la derivación determinística comenzando en C, seleccionando la submeta $s([A_2,\dots,A_m\,|\,L])$, y usando (5.49) *y* (5.50).

Demostración. Recordemos ahora la definición de δ_2:

$$\begin{aligned} &\delta_2(j)=min\{k+m-j\mid k\geq 1 \text{ y} \\ &(k\geq j \text{ o } p_{j-k}\neq p_j) \text{ y } ((k\geq d \text{ o } p_{d-k}=p_d) \text{ para } j<d\leq m)\} \end{aligned}$$

Para demostrar nuestra fórmula propuesta (5.59) desglosaremos en distintos casos la definición de δ_2, concluyendo que cada caso corresponde a cierta configuración del patrón consigo mismo. Antes, damos una definición. Sea x un símbolo y q una cadena. Llamaremos a xq un *semi-sufijo* del patrón p si q es un sufijo de p, y xq es tal que el elemento inmediatamente a la izquierda de q en p no coincide con x.

El caso en que todo un semi-sufijo está dentro del patrón. Suponemos, entonces, que $k<j$, por lo que $0<j-k$, y p_{j-k} está todavía dentro del patrón, lo que dividimos en dos subcasos:

$$\begin{array}{lllllllllll} p: & p_1 & p_2 & \cdots & p_{k-1} & p_k & \cdots & p_{j-1} & \boxed{p_j} & \cdots & p_{m-1} \quad p_m \\ & & & & & & & & \boxed{p_j} & \cdots & p_{m-1} \quad p_m \end{array} \tag{5.60}$$

y

$$\begin{array}{lllllllllll} p: & p_1 & p_2 & \cdots & p_{k-1} & p_k & p_{k+1}\cdots & \cdots & \boxed{p_j} & \cdots & p_{m-1} \quad p_m \\ & & & & & \boxed{p_j} & \cdots & p_{m-1} \quad p_m & & & \end{array} \tag{5.61}$$

(hemos encerrado en un cuadrito el símbolo p_j para señalar que ahí termina el semi-sufijo).

En la configuración (5.60) tenemos el siguiente conjunto de ecuaciones:

$$\{p_{j-1} \neq p_j,\ p_j = p_{j+1},\ p_{j+1} = p_{j+2},\ \ldots,\ p_{m-1} = p_m\} \tag{5.62}$$

En este caso, la separación entre p_j y p_k es de 0 símbolos, ya que suponemos que $p_k = p_{j-1}$. Dada la unificación de variables que realiza la llamada recursiva de la submeta s, todas las ecuaciones previamente enunciadas se satisfacen (junto con la inecuación), por lo que la longitud de la derivación determinística es 0. Sustituyendo en la fórmula (5.59), $0 = \delta_2(j) - (m - j + 1)$, por lo que $\delta_2(j) = (m - j + 1)$, pero $\delta_2(j) = k_0 + m - j$, en donde $k_0 = 1$, y comprobamos la igualdad propuesta.

Para el siguiente caso, representado en la configuración (5.61), tenemos lo siguiente: $p_{j-k} \neq p_j$ y $p_{d-k} = p_d$ para $j < d \leq m$. Nuestro trabajo ahora se reduce a medir qué costo tiene alinear p_j y p_{j-k}, para un índice k_0 que satisface la condición de minimalidad exigida por δ_2. Ahora bien, $k = j - (j - k)$, pero dado que ya hemos avanzado un símbolo, debemos tener entonces $k - 1$ desdoblados (la longitud de la derivación determinística), con lo que planteamos las ecuaciones siguientes:

$$det_unf(C) = \delta_2(j) - (m - j + 1) = k - 1 = k_0 + m - j - (m - j + 1) = k_0 - 1, \tag{5.63}$$

y ya que $k_0 = k$, tenemos que (5.59) también se satisface.

El caso de las ecuaciones positivas. En este caso, hay un prefijo del semi-sufijo que cae ya fuera del patrón:

$$\begin{array}{llllllllllll} p: & & & & p_1 & p_2 & \cdots & p_{k-1} & p_k & \cdots & \boxed{p_j} & \cdots \quad p_{m-1} \quad p_m \\ & \boxed{p_j} & \cdots & p_{m-1} \quad p_m & & & & & & & & \end{array} \tag{5.64}$$

Lo que resta ahora es verificar la concordancia del segmento sufijo que empalma con el prefijo del patrón. Para ello, es necesario considerar que $k \geq d$ implica que $0 \geq d - k$ o bien $d - k \leq 0$, por lo que, para cierta d en $j < d \leq m$ ya no tiene sentido realizar

comparaciones. Considerando un índice d_0 tal que $p_{d_0} = p_1$, $p_{d_0+1} = p_2$, $p_{d_0+m} = p_{d_0}$ (note que ahora tenemos únicamente ecuaciones), en donde $j < d_0 < m$, nuestro problema consiste en alinear p_1 con p_{d_0}. Para lograrlo, necesitamos desdoblar $d_0 - 1$ veces, $det_unf(c) = d_0 - 2$. ($d_0 - 1$ para el caso normal, y $(d_0 - 1) - 1$ para la llamada recursiva.) $d_0 - 1 = \delta_2(j) - (m - j + 1)$, y ya que $\delta_2(j) = d_0 - 1 + m - j$ tenemos que $det_unf(C) = d_0 - 1 + m - j - (m - j + 1) = d_0 - 2$.

El caso del semi-sufijo que está completamente fuera del patrón. Cuando $m - k < 0$, el semi-sufijo está completamente fuera del patrón:

$$\begin{array}{llllllllllllllll} p: & & & & p_1 & p_2 & \cdots & p_{k-1} & p_k & \cdots & \boxed{p_j} & \cdots & p_{m-1} & p_m \\ & \boxed{p_j} & \cdots & p_{m-1} & p_m \end{array} \tag{5.65}$$

$$\begin{array}{lllllllllllll} p: & p_1 & p_2 & \cdots & p_{k-1} & p_k & \cdots & p_{j-1} & \boxed{p_j} & \cdots & p_{m-1} & p_m \\ & & & & & & & \boxed{p_j} & \cdots & p_{m-1} & p_m \\ & \longleftarrow & & j-1 & & & \longrightarrow & \boxed{p_j} & \cdots & p_{m-1} & p_m \\ & & & & & & & \longleftarrow & m-j & \longrightarrow \end{array} \tag{5.66}$$

En total hay que desdoblar $j - 1 + (m - j) = det_unf(j) = m - 1$ veces. Ahora bien, en la fórmula $det_unf(C) = \delta_2(j) - (m - j + 1)$; esto quiere decir que $\delta_2(j) = m - 1 + m - j + 1 = 2 * m - j$. Comprobemos esto. De $k + m - j$ con $k \geq d$, $d \leq m$, el mínimo se obtiene para $k = m$, por lo que $\delta_2(j) = 2 * m - j$. ■

Observemos que en una lectura procedural de las cláusulas de estos lemas, suponiendo una interpretación (ejecución) de arriba a abajo y de izquierda a derecha, $det_unf(C) + 1$ es el número de símbolos que el patrón se desplazó. Cuando combinamos la restricciones produciendo δ_1 con las de δ_2, obtenemos desplazamientos posiblemente mayores que aquellos dados por el algoritmo BM. La razón es que requerimos $p_{j-k} = t_i$ para una completa reocurrencia, mientras que el algoritmo BM sólo requiere que $p_{j-k} \neq p_j$, lo cual es implicado por nuestra condición. Nuestra variante del algoritmo BM, sin embargo, tal como las de [Par90, Pep91], usa tablas más grandes que las del algoritmo BM.

Ahora estamos listos para presentar otro de los resultados de este capítulo. Sea *BM'* el programa que resulta de aplicar los pasos 1, 2', 3, y 4 a la especificación de fuerza bruta especializada a un patrón.

Teorema 4 *En todo mal-casamiento (incluida una discordancia), el programa BM', ejecutado en Prolog, desplaza el patrón al menos tantos símbolos como el algoritmo BM lo*

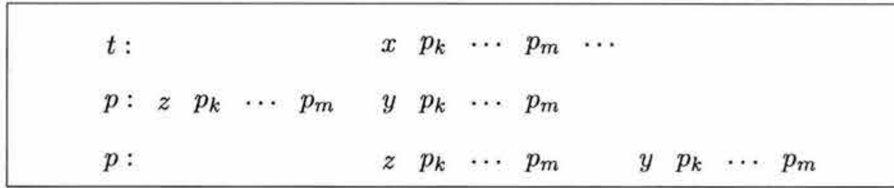

Figura 5.10: Caso en que y es comparado con z.

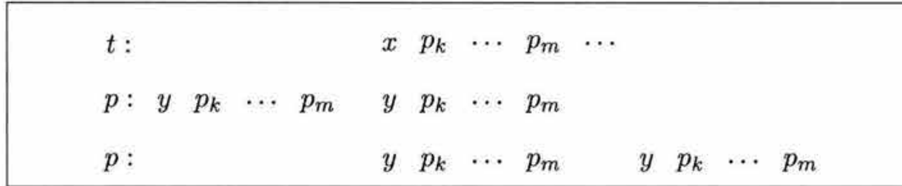

Figura 5.11: Una alineación redundante (pues $y \neq x$).

hace.

Demostración. Esto es válido, ya que la condición de reocurrencia total ($cp_{j-1} \cdots p_m$, y c es algún elemento del alfabeto, $c \neq p_j$) de segmentos del patrón implica a la condición original de Boyer y Moore: $xp_{j-1} \cdots p_m$ y $x \neq p_j$. ■

5.2.1 Observaciones

Notemos que hemos derivado una variante del algoritmo BM que puede realizar menos comparaciones que el algoritmo original [BYCG94]. La razón es la siguiente:

Considere un mal-casamiento en la posición dada por el símbolo y ($y \neq x$), situación ilustrada en la Fig. 5.10. Como podemos observar, nuestro patrón tiene una subcadena $p_k \cdots p_m$, y la primera ocurrencia de esta subcadena es precedida por el símbolo z. Si suponemos que $z = y$, tendremos la configuración ilustrada en la Fig. 5.11, en donde podemos observar que obtendremos otro fracaso al intentar casar nuestro patrón con el segmento actual del texto. El lector encontrará mayores detalles de este fenómeno en [Gus99, página 19].

Un ejemplo para el cual el algoritmo BM y el programa BM' exhiben diferentes comportamientos es el que aparece en [BM77], con

$$p = at\text{-}that$$

y

$$t = which\text{-}finally\text{-}halts.\text{--}at\text{-}that\text{-}point \ .$$

Para encontrar el patrón p en el texto t hay que hacer 14 comparaciones, de acuerdo con Boyer y Moore. En contraste, en el algoritmo BM' sólo tenemos que hacer 12 comparaciones. En esta variante, BM', sin embargo, hay que hacer un mayor preprocesamiento, por lo que habría que tomar una decisión entre invertir en un mayor preprocesamiento u obtener una mejora en la ejecución.

5.3 Derivación formal de la parte de búsqueda del algoritmo BM

Damos a continuación una descripción del proceso en general. Nuestro objetivo es formalizar algunos pasos previos.

Consideremos la siguiente forma triangular que corresponde al programa intermedio que obtenemos para el patrón $P = p_1 \cdots p_m$:

Programa 18

$$s_p([A_1 : A_m \mid L]) \leftarrow A_m : A_1 = p_m : p_1 \tag{5.67}$$

$$\left[\begin{array}{l} s_p([A_1 : A_m \mid L]) \leftarrow A_m \neq p_m,\ s_p([A_2 : A_m \mid L]) \\ s_p([A_1 : A_m \mid L]) \leftarrow A_m = p_m,\ A_{m-1} \neq p_{m-1},\ s_p(A_2 : A_m \mid L) \\ \qquad\qquad\qquad \vdots \\ s_p([A_1 : A_m \mid L]) \leftarrow A_m : A_2 = p_m : p_2,\ A_1 \neq p_1,\ s_p([A_2 : A_m \mid L]) \end{array}\right. \tag{5.68}$$

$$s_p([A_1 : A_m \mid L]) \leftarrow A_m : A_1 = p_m : p_1,\ s_p([A_2 : A_m \mid L])\ . \tag{5.69}$$

Como ya hemos visto, el propósito de esta aplicación de la división en casos es hacer explícitos los lugares en donde existe una falla en un intento de casamiento total. Nuestro objetivo al hacer estas fallas explícitas es aprovechar los no-casamientos y los casamientos parciales, pues cada cláusula en este programa corresponde al caso de o un casamiento total ((5.67), (5.69)), o a un casamiento parcial o a un no-casamiento inicial (conjunto de cláusulas (5.68)).

Con el objetivo de considerar valores alternativos al símbolo que causó el no-casamiento en la posición i-ésima, ahora realizamos una división en casos, *pero sólo para la variable* A_m ($A_m \in \{p_1 : p_m\}$ y $A_m \notin \{p_1 : p_m\}$), por lo que el Programa (18) se transforma en el siguiente:

Programa 19

$$s_p([A_1 : A_m \mid L]) \leftarrow A_m : A_1 = p_m : p_1 \quad (5.70)$$

$$s_p([A_1 : A_m \mid L]) \leftarrow A_m \neq p_m,\ A_m \in \{p_1 : p_m\},\ s_p([A_2 : A_m \mid L]) \quad (5.71)$$

$$s_p([A_1 : A_m \mid L]) \leftarrow A_m \neq p_m,\ A_m \notin \{p_1 : p_m\},\ s_p([A_2 : A_m \mid L]) \quad (5.72)$$

$$\vdots$$

$$s_p([A_1 : A_m \mid L]) \leftarrow A_m : A_2 = p_m : p_2,\ A_1 \neq p_1,\ s_p([A_2 : A_m \mid L]) \quad (5.73)$$

$$s_p([A_1 : A_m \mid L]) \leftarrow A_m : A_2 = p_m : p_2,\ A_1 \neq p_1,\ s_p([A_2 : A_m \mid L]) \quad (5.74)$$

$$s_p([A_1 : A_m \mid L]) \leftarrow A_m : A_1 = p_m : p_1,\ s_p([A_2 : A_m \mid L])\ . \quad (5.75)$$

Concentrémonos temporalmente en la cláusula (5.71). Por un lado, desdoblamos esta cláusula con respecto al átomo $A_m \in \{p_1 : p_m\}$, lo que nos da el siguiente subconjunto de cláusulas:

$$s_p([A_1 : A_m \mid L]) \leftarrow A_m \neq p_m,\ A_m = p_1,\ s_p([A_2 : A_m \mid L]) \quad (5.76)$$

$$s_p([A_1 : A_m \mid L]) \leftarrow A_m \neq p_m,\ A_m = p_2,\ s_p([A_2 : A_m \mid L]) \quad (5.77)$$

$$\vdots \quad (5.78)$$

$$s_p([A_1 : A_m \mid L]) \leftarrow A_m \neq p_m,\ A_m = p_{m-1},\ s_p([A_2 : A_m \mid L])\ . \quad (5.79)$$

Notamos que podría haber algunas cláusulas duplicadas si en el conjunto $\{p_1, \ldots, p_m\}$ existen algunos símbolos duplicados. Suponemos que hemos simplificado tales cláusulas. Además, hemos borrado la cláusula

$$s_p([A_1 : A_m \mid L]) \leftarrow A_m \neq p_m,\ A_m = p_m,\ s_p([A_2 : A_m \mid L])$$

ya que la submeta $p_m \neq p_m$ falla inmediatamente.

Por otro lado, al desdoblar $A_m \notin \{p_1 : p_m\}$ obtenemos el nuevo conjunto de inecuaciones $A_m \neq p_1, \ldots, A_m \neq p_m$ en aquellas cláusulas en donde $\notin$ aparece.

A continuación desdoblamos s_p k_i veces, hasta que el respectivo símbolo p_i sea colocado en su posición correcta; si ya está en su posición correcta, no desdoblamos en absoluto, y por uniformidad, decimos que desdoblamos cero veces:

$$s_p([A_1 : A_m \mid L]) \leftarrow A_m \neq p_m,\ A_m = p_1,\ \underline{s_p([A_2 : A_m \mid L])}_{k_1} \quad (5.80)$$

$$s_p([A_1 : A_m \mid L]) \leftarrow A_m \neq p_m,\ A_m = p_2,\ \underline{s_p([A_2 : A_m \mid L])}_{k_2} \quad (5.81)$$

$$\vdots$$

$$s_p([A_1 : A_m \mid L]) \leftarrow A_m \neq p_m,\ A_m = p_{m-1},\ \underline{s_p([A_2 : A_m \mid L])}_{k_m} \quad (5.82)$$

El proceso clave aquí es la aplicación de las derivaciones determinísticas. En el transcurso de nuestro proceso, puede darse el caso de que tengamos que eliminar las submetas duplicadas.

Por lo tanto, obtenemos las siguientes cláusulas:

Cláusulas 14

$$s_p([A_1 : A_m \mid L]) \leftarrow A_m \neq p_m,\ A_m = p_1,\ s_p(\phi(k_1)) \tag{5.83}$$

$$s_p([A_1 : A_m \mid L]) \leftarrow A_m \neq p_m,\ A_m = p_2,\ s_p(\phi(k_2)) \tag{5.84}$$

$$\vdots$$

$$s_p([A_1 : A_m \mid L]) \leftarrow A_m \neq p_m,\ A_m = p_{m-1},\ s_p(\phi(k_{m-1})) \tag{5.85}$$

en donde $\phi(k_i)$ es una lista que, como expresión, mantiene las variables necesarias.

En particular en la cláusula

$$s_p([A_1 : A_m \mid L]) \leftarrow A_m \neq p_m,\ A_m \notin \{p_1 : p_m\},\ s_p([A_2 : A_m \mid L])$$

podemos desplazar el patrón sobre el texto la longitud total del patrón:

$$s_p([A_1 : A_m \mid L]) \leftarrow A_m \neq p_m,\ A_m \notin \{p_1 : p_m\},\ s_p(L)$$

Obtenemos, por consiguiente, las siguientes cláusulas:

Cláusulas 15

$$s_p([A_1 : A_m \mid L]) \leftarrow \boxed{A_m \neq p_m,}\ A_m = p_1,\ s_p(\phi(k_1)) \tag{5.86}$$

$$s_p([A_1 : A_m \mid L]) \leftarrow \boxed{A_m \neq p_m,}\ A_m = p_2,\ s_p(\phi(k_2)) \tag{5.87}$$

$$\vdots \tag{5.88}$$

$$s_p([A_1 : A_m \mid L]) \leftarrow \boxed{A_m \neq p_m,}\ A_m = p_{m-1},\ s_p(\phi(k_{m-1})) \tag{5.89}$$

$$s_p([A_1 : A_m \mid L]) \leftarrow \boxed{A_m \neq p_m,}\ A_m \notin \{a_1 : p_m\},\ s_p(L) \tag{5.90}$$

Todas estas cláusulas tienen una submeta en común: $A_m \neq p_m$, encerradas en un rectángulo, por lo que podemos definir un nuevo predicado para doblar varias cláusulas:

$$\mathit{table}([A_1 : A_m \mid L]) \leftarrow A_m = p_1,\ s_p(\phi(k_1)) \tag{5.91}$$

$$A_m = p_2,\ s_p(\phi(k_2)) \tag{5.92}$$

$$\vdots \tag{5.93}$$

$$A_m \notin \{a_1 : p_{m-1}\},\ s_p(L) \tag{5.94}$$

El doblado en las cláusulas 15 utilizando la definición de *table* produce la siguiente cláusula:

$$s_p([A_1 : A_m \mid L]) \leftarrow A_m \neq p_m,\ table([A_1 : A_m \mid L]) \tag{5.95}$$

Esta cláusula implícitamente la fórmula δ_1, excepto para el caso $\delta_1(p_m)$, pero el avance de al menos una unidad dado en la llamada recursiva nos asegura la corrección también para este caso (es decir, $\delta_1(p_m) = 1$).

Para estudiar el casamiento de un sufijo o de un casamiento parcial, estudiamos la siguiente cláusula:

$$s_p([A_1 : A_m \mid L]) \leftarrow A_m : A_{k+1} = p_m : p_{k+1},\ A_k \neq p_k,\ s_p([A_2 : A_m \mid L]) \tag{5.96}$$

Aquí, un sufijo de tamaño $m - (k+1)$ ($p_{k+1} \cdots p_m$) ya casó. Sin embargo, $A_k \neq p_k$ evita casar el patrón completamente. En la cláusula (5.96) necesitamos reencontrar dentro del patrón el sufijo $p_{k+1} : p_m$ con la condición adicional $A_k \neq p_k$ [BM77]. Ahora desdoblamos $s_p([A_2 : A_m \mid L])$ utilizando una derivación determinística. En cada paso de derivación, hacemos un nuevo intento para obtener un conjunto de ecuaciones consistente; el siguiente conjunto, por ejemplo, es el conjunto que resulta del primer intento para encontrar una solución para las variables involucradas, y corresponde al primer desdoblado determinístico de la llamada recursiva de s_p:

$$\{A_m : A_{k+1} = p_m : p_{k+1},\ A_k \neq p_k\} \cup \{A_2 : A_m = p_1 : p_{m-1}\} \tag{5.97}$$

Si tal conjunto de ecuaciones no tiene solución, desdoblamos determinísticamente otra vez; si la tiene, retornamos a la cláusula previa y detenemos la derivación determinística.

Ahora describimos cómo mecanizar nuestro proceso de doblado, dando un ejemplo. Supongamos que después de desdoblar determinísticamente una submeta, obtenemos la cláusula siguiente:

$$s_p([A_1 : A_m \mid L]) \leftarrow A_m : A_{k+1} = p_m : p_{k+1},\ A_k \neq p_k,\ s_p(\phi(k)) \tag{5.98}$$

$(m > k)$. En este momento, podemos garantizar que

$$A_m : A_{k+1} = p_m : p_{k+1} \tag{5.99}$$

y notamos además que $A_m = p_m$ es una ecuación común al cuerpo de varias cláusulas y podemos realizar un doblado tomando esta ecuación como referencia. La ecuación $A_m = p_m$ es, además, complemento de la inecuación $A_m \neq p_m$, por lo que al aplicar el doblado decrementamos el no-determinismo de nuestros programas. Posteriormente, *podemos aplicar otra vez la regla del doblado a la nueva definición*. A continuación, identificamos la submeta $A_{m-1} = p_{m-1}$ en la nueva definición y aplicamos el proceso del doblado nuevamente,

y así sucesivamente. Este proceso tiene que terminar, ya que el número de ecuaciones en común es decrementado en cada paso. Nuestro caso general es: terminamos el proceso de doblado cuando no tenemos ya ninguna submeta que común al cuerpo de varias cláusulas. (En nuestros programas, aquellas cláusulas que contienen a la inecuación $A_m \neq p_m$ las hemos dejado sin cambio, pero cabe notar que sí podríamos utilizarlas como candidatas para aplicarles la regla del doblado.)

5.4 Conclusiones

En este capítulo hemos abordado el problema de la derivación de la parte de búsqueda de varias variantes del algoritmo BM. Aunque nuestra derivación consta de partes que son mecanizables, para desarrollar el proceso general requerimos de la guía del programador en el orden de la aplicación de las reglas de transformación.

Nuestra derivación es una de las primeras que atacan el problema de derivar una variante de la parte de búsqueda del algoritmo BM. Esta derivación forma parte de algunos resultados que fueron aceptados vía el artículo [HR01]. Una derivación de una variante del algoritmo BM contemporánea a las nuestras es la de [ACDM01]. La ventaja de esta derivación sobre las nuestras radica en que los autores la realizaron por métodos automáticos, lo que en nuestro caso no fue así.

Existen además otros puntos objetables o debatibles en nuestras derivaciones previas:

1. El uso de listas provoca una ineficiencia al examinar los elementos, que no aparece de ningún modo en la formulación original del algoritmo BM.

2. La estrategia que utilizamos, a primera vista, parece enfocada completamente a la derivación de las variantes del algoritmo BM que presentamos.

3. Hemos realizado la eliminación del no-determinismo a nivel lógico, pero sería deseable formularla en términos de algún modelo de ejecución, como por ejemplo el de Prolog.

4. Las derivaciones que hacemos implementan en esencia algunas variantes ya conocidas del algoritmo BM. La obtención exacta, sin embargo, de cada variante está sujeta a ciertas características imperativas que nosotros no manejamos.

5. La estrategia que utilizamos, como ya mencionamos, no es automática. Sería deseable seguir el lineamiento de [ACDM01] e investigar bajo qué restricción sí obtenemos una estrategia mecanizable.

En los siguientes capítulos estableceremos algunas de las respuestas que hemos hallado en nuestra investigación para estas objeciones. En particular, en respuesta al punto número 1 anterior, evitaremos una especificación basada en listas, sustituyéndola por una basada en hechos, logrando con ello soslayar la crítica de la ineficiencia en el uso de listas.

En lo referente al punto número 2, para descentralizar nuestra estrategia de las fronteras de la derivación del algoritmo BM, en el siguiente capítulo derivamos (manualmente) el algoritmo KMP, y establecemos una aplicación del concepto de *reuso de desarrollos*, pues observaremos que algunas partes que nos sirvieron para derivar el algoritmo BM también nos servirán para derivar el algoritmo KMP.

En relación al aspecto lógico, en el punto número 3 que involucra el análisis de los casos $A = a$ y $A \neq a$, una forma de incoporar el determinismo en, por ejemplo, un intérprete de Prolog, es por medio de la introducción de cortes ([Dev90, página 281], Transformación 10.5).

En lo referente al punto número 4 es necesario señalar que la obtención de un algoritmo al pie de la letra puede ser algo no deseable, ya que muchos algoritmos sólo tienen una importancia histórica, o práctica pero limitada a casos muy especiales, no valiendo la pena derivarlos manualmente por carecer de características relevantes que aporten algo sustancial a la transformación de programas lógicos. Lejos de ello, si deseamos automatizar parte o todo el proceso de la derivación de un algoritmo en lo específico, es necesario frecuentemente sólo tomar en consideración las versiones simplificadas de los mismos, tema que tratamos en el siguiente punto.

Finalmente, en el punto número 5 la crítica de cómo limitar el algoritmo BM para obtener una derivación automática del mismo está dada en el capítulo final de esta tesis. Ahí veremos que si consideramos una variante únicamente basada en la función δ_2 sí es posible mecanizar tanto la derivación de la parte de búsqueda del algoritmo KMP como la derivación de una variante de la parte de búsqueda del algoritmo BM.

Capítulo 6

Reuso de desarrollos y una derivación de la parte de búsqueda del algoritmo KMP

En el capítulo previo hicimos el estudio de algunas derivaciones que conducen a distintas variantes del algoritmo BM. En este capítulo mostraremos cómo es posible reutilizar algunos segmentos de nuestro desarrollos previos para obtener el algoritmo MP y el algoritmo KMP.

6.1 Reutilización de desarrollos derivacionales

La *reutilización* es un concepto que aparece en diversos campos de la Computación. En el ámbito de la programación, por ejemplo, tiene su mayor y más útil ejemplificación en la programación orientada a objetos. En nuestro caso, tal concepto también resulta aplicable al tratar el problema de la derivación de *familias de algoritmos*, como detallaremos en capítulo 7.

6.2 Derivación de la parte de búsqueda del algoritmo KMP

Ahora derivaremos la parte de búsqueda del algoritmo KMP reutilizando partes de la derivación del algoritmo BM (Capítulo 5).

Usaremos nuevamente el patrón *aab* como un ejemplo que nos permita ilustrar los

puntos esenciales de esta nueva derivación. El algoritmo KMP intenta casar el patrón de izquierda a derecha, por lo que esta vez usaremos prefijos, en lugar de los sufijos del caso BM.

Comenzamos ahora con la siguiente especificación directa:

Programa 20

$$s'_{aab}([A_1 : A_3 \mid L]) \leftarrow A_1 = a,\ A_2 = a,\ A_3 = b \tag{6.1}$$

$$s'_{aab}([A_1 : A_3 \mid L]) \leftarrow s'_{aab}([A_2 A_3 \mid L]) \tag{6.2}$$

Notemos que las ecuaciones tienen un orden de ocurrencia textual de izquierda a derecha correspondiente al orden de ocurrencia de a, a, y b en el patrón aab (a diferencia de la derivación que hicimos de los algoritmos tipo BM, en esta ocasión no alteramos el orden de las ecuaciones en (6.1)). Similarmente, centramos ahora nuestro interés en estudiar los prefijos del patrón, en lugar de hacerlo con los sufijos, como fue en el caso de los algoritmos de tipo BM previamente analizados.

El apóstrofe en s'_{aab} es para señalar que ahora pretendemos derivar un algoritmo diferente al de la variante del algoritmo de Boyer y Moore.

Al aplicar una división en casos a (6.2) (ver Figura 6.1) obtenemos la siguiente forma *prefijo-triangular*:

Cláusulas 16

$$\begin{array}{l}
s'_{aab}([A_1 :_2 A_3 \mid L]) \leftarrow A_1 \neq a,\ \underline{s'_{aab}([A_2 A_3 \mid L])}_0 \\
s'_{aab}([A_1 : A_3 \mid L]) \leftarrow A_1 = a,\ A_2 \neq a,\ \underline{s'_{aab}([A_2 A_3 \mid L])}_1 \\
s'_{aab}([A_1 : A_3 \mid L]) \leftarrow A_1 = a,\ A_2 = a,\ A_3 \neq b,\ \underline{s'_{aab}([A_2 A_3 \mid L])}_0 \\
s'_{aab}([A_1 : A_3 \mid L]) \leftarrow A_1 = a,\ A_2 = a,\ A_3 = b,\ \underline{s'_{aab}([A_2 A_3 \mid L])}_2
\end{array} \tag{6.3}$$

Cada casamiento parcial y la discordancia apropiada (Figura 6.2), o un casamiento completo (Figura 6.3), está cubierto por alguna de las anteriores cláusulas. Además, también está cubierto el caso de encontrar las reocurrencias de aab dentro del texto, si tales reocurrencias existiesen.

A continuación aplicamos derivaciones determinísticas usando las cláusulas (6.1) y (6.2), obteniendo lo siguiente:

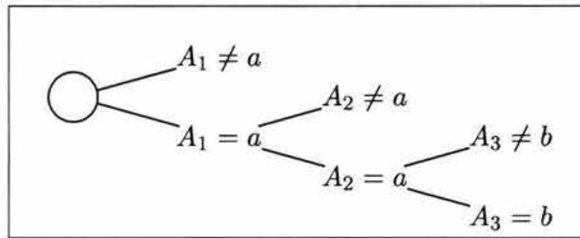

Figura 6.1: División sistemática de casos.

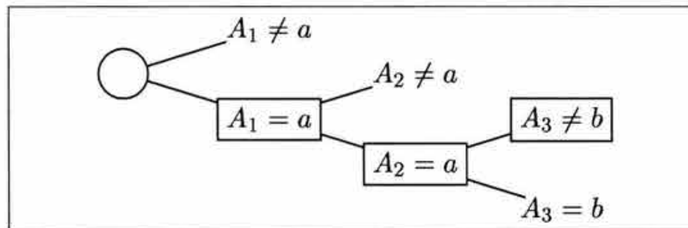

Figura 6.2: Casamiento parcial del patrón *aab* (subpatrón *aa*) dentro de un texto.

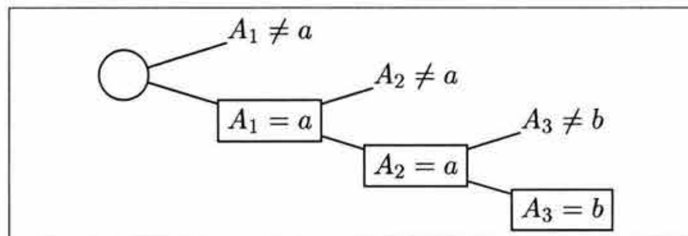

Figura 6.3: El hallazgo del patrón *aab* dentro de un texto.

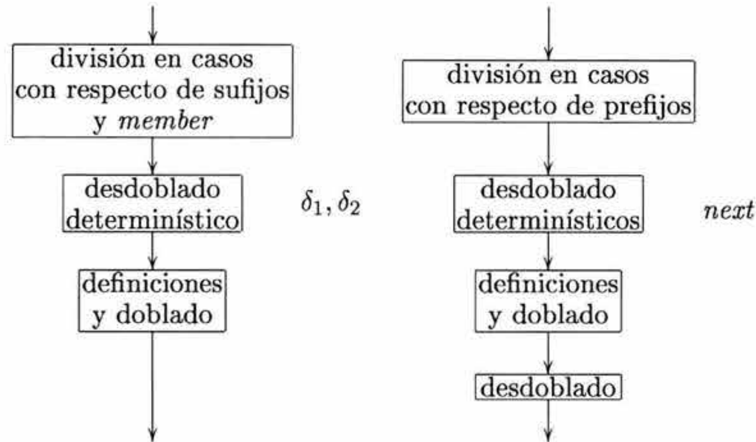

Figura 6.4: Derivaciones para BM (izquierda) y KMP (derecha)

Programa 21

$$s'_{aab}([A_1 : A_3 \mid L]) \leftarrow A_1 = a,\ A_2 = a,\ A_3 = b$$
$$s'_{aab}([A_1 : A_3 \mid L]) \leftarrow A_1 \neq a,\ s'_{aab}([A_2 A_3 \mid L])$$
$$s'_{aab}([A_1 : A_3 \mid L]) \leftarrow A_1 = a,\ A_2 \neq a,\ s'_{aab}([A_3 \mid L])$$
$$s'_{aab}([A_1 : A_3 \mid L]) \leftarrow A_1 = a,\ A_2 = a,\ A_3 \neq b,\ s'_{aab}([A_2 A_3 \mid L])$$
$$s'_{aab}([A_1 : A_3 \mid L]) \leftarrow A_1 = a,\ A_2 = a,\ A_3 = b,\ s'_{aab}(L)$$

Este programa no realiza los desplazamientos asociados a la parte de búsqueda del algoritmo KMP, ya que el apuntador sobre el texto puede retroceder (y además es no-determinístico). Sin embargo, este programa ya tiene la función *next* incorporada en él, como veremos más adelante, exhibiendo, bajo Prolog, una conducta intermedia entre la del algoritmo de fuerza bruta y la del algoritmo KMP. En efecto, en todo casamiento parcial, este programa desplaza el patrón el mismo número de símbolos tal como el algoritmo KMP lo haría, pero a continuación mueve el apuntador del texto (hacia atrás, cero o más posiciones) al símbolo correspondiente al primer símbolo del patrón.

En este punto termina el reuso de desarrollos que tomamos de las derivaciones que hicimos de las variantes del algoritmo BM. En lo que sigue, hay diferencias en la manera de hacer los desdoblados (Fig. 6.4). En el algoritmo BM este procedimiento es directo, pero en este caso hay una necesidad de hacer avanzar el apuntador lo suficiente, de tal forma que se le integre una *memoria* al algoritmo KMP, una de las principales diferencias

con el algoritmo original BM, ya que éste carece de memoria.

Ahora procederemos a definir nuevos predicados y a doblar en el programa actual, tal como hicimos en la derivación de algoritmo BM:

Programa 22

$$
\begin{aligned}
&s'_{aab}([A_1A_2A_3 \mid L]) \leftarrow A_1 = a,\ match1'(A_1, A_2, A_3, L) \\
&s'_{aab}([A_1A_2A_3 \mid L]) \leftarrow A_1 \neq a,\ s'_{aab}([A_2A_3 \mid L]) && (6.4) \\
&match1'(A_1, A_2, A_3, L) \leftarrow A_2 = a,\ match2'(A_1, A_2, A_3, L) \\
&match1'(A_1, A_2, A_3, L) \leftarrow A_2 \neq a,\ s'_{aab}([A_3 \mid L]) && (6.5) \\
&match2'(A_1, A_2, A_3, L) \leftarrow A_3 = b,\ match3'(A_1, A_2, A_3, L) \\
&match2'(A_1, A_2, A_3, L) \leftarrow A_3 \neq b,\ \underline{s'_{aab}([A_2A_3 \mid L])} && (6.6) \\
&match3'(A_1, A_2, A_3, L) \leftarrow \\
&match3'(A_1, A_2, A_3, L) \leftarrow \underline{s'_{aab}(L)} && (6.7)
\end{aligned}
$$

Note que las llamadas a s'_{aab} en (6.4) y (6.5) presentan la misma conducta bajo Prolog que la del algoritmo KMP, de tal manera que el apuntador no retrocede. Esto no es cierto para (6.6), sin embargo, ya que después de inspeccionar A_3, Prolog intentará hallar una ocurrencia del patrón comenzando en A_2.

Una manera de hacer que el apuntador del texto avance es desdoblando (6.6), como a continuación detallamos. Primero desdoblamos (6.6) con respecto a la definición actual s'_{aab}:

Cláusulas 17

$$
\begin{aligned}
&match2'(A_1, A_2, A_3, [A_4 \mid L]) \leftarrow A_3 \neq b,\ A_2 = a, \\
&\qquad\qquad \underline{match1'(A_2, A_3, A_4, L)} && (6.8) \\
&match2'(A_1, A_2, A_3, [A_4 \mid L]) \leftarrow A_3 \neq b,\ A_2 \neq a, \\
&\qquad\qquad s'_{aab}([A_3A_4 \mid L]) && (6.9)
\end{aligned}
$$

En (6.8), la submeta $A_3 \neq b$ aún no se recadena, porque A_3 otra vez se comparará en la llamada a $match1'$. Por lo tanto, debemos desdoblar a (6.8):

Cláusulas 18

$$
\begin{aligned}
match2'(A_1, A_2, A_3, [A_4 \mid L]) &\leftarrow A_3 \neq b,\ A_2 = a,\ A_3 = a, \\
&\quad match2'(A_2, A_3, A_4, L) \qquad (6.10) \\
match2'(A_1, A_2, A_3, [A_4 \mid L]) &\leftarrow A_3 \neq b,\ A_2 = a,\ A_3 \neq a, \\
&\quad s'_{aab}([A_4 \mid L]) \qquad (6.11)
\end{aligned}
$$

Notando que cuando $match2'$ se llama, A_2 ha unificado con a, por lo que borramos (6.9), y simplificamos en (6.10) y (6.11).

Ahora sólo resta tratar con las posibles reoccurrencias del patrón. El programa actual exhibe la misma conducta que el del autómata finito correspondiente [Per90, CR94] al algoritmo KMP, excepto por el hecho de que existe una transición Λ para cuando haya ocurrencias adicionales del patrón (6.7). Eliminando tal transición por desdoblado (6.7), llegamos a:

Programa 23

$$
\begin{aligned}
&s'_{aab}([A_1 A_2 A_3 \mid L]) \leftarrow A_1 = a,\ match1'(A_1, A_2, A_3, L) \\
&s'_{aab}([A_1 A_2 A_3 \mid L]) \leftarrow A_1 \neq a,\ s'_{aab}([A_2 A_3 \mid L]) \\
&match1'(A_1, A_2, A_3, L) \leftarrow A_2 = a,\ match2'(A_1, A_2, A_3, L) \\
&match1'(A_1, A_2, A_3, L) \leftarrow A_2 \neq a,\ s'_{aab}([A_3 \mid L]) \\
&match2'(A_1, A_2, A_3, L) \leftarrow A_3 = b,\ match3'(A_1, A_2, A_3, L) \\
&match2'(A_1, A_2, A_3, [A_4 \mid L]) \leftarrow A_3 = a,\ match2'(A_2, A_3, A_4, L) \\
&match2'(A_1, A_2, A_3, [A_4 \mid L]) \leftarrow A_3 \neq b,\ A_3 \neq a,\ s'_{aab}([A_4 \mid L]) \\
&match3'(A_1, A_2, A_3, L) \leftarrow \\
&match3'(A_1, A_2, A_3, [A_4 A_5 A_6 \mid L]) \leftarrow A_4 = a, \\
&\qquad\qquad match1'(A_4, A_5, A_6, L) \\
&match3'(A_1, A_2, A_3, [A_4 A_5 A_6 \mid L]) \leftarrow A_4 \neq a,\ s'_{aab}([A_5 A_6 \mid L])
\end{aligned}
$$

Similarmente a la derivación del algoritmo BM, el no-determinismo de este programa puede ser deseable con tal de encontrar todas las ocurrencias del patrón en el texto.

En resumen, a partir del conjunto de prefijos de un patrón, hemos logrado, según lo anterior, construir un conjunto de cláusulas que incorpora cada prefijo en forma de un conjunto de ecuaciones en el cuerpo de cada cláusula. Utilizando derivaciones determinísticas, entonces, hemos logrado derivar un algoritmo intermedio entre el algoritmo de

fuerza bruta y el algoritmo KMP. Finalmente, hemos tenido que hacer mayores desdoblados determinísticos para incorporarle al algoritmo KMP una memoria.

6.3 El desdoblado determinístico y el algoritmo KMP

Una vez presentado un ejemplo del proceso para obtener una derivación del algoritmo KMP, en esta sección presentamos los teoremas que ligan el desdoblado determinístico con dos funciones auxiliares, la función f y la función *next*. Empezamos tratando el algoritmo MP, una versión simplificada del algoritmo KMP dada por Morris y Pratt (versión históricamente previa al algoritmo KMP).

6.3.1 Caso MP

El teorema siguiente relaciona la función f del algoritmo MP y el desdoblado determinístico.

Recordemos la definición de la función f dada en la subsección 4.3.1:

$$f(1) = 0$$
$$f(j) = máx\{i \mid i < j,\ p_1 \cdots p_{i-1} = p_{j-i+1} \cdots p_{j-1}\}$$

Consideremos el siguiente patrón: *abcab*. El algoritmo de fuerza bruta para este patrón es:

$$s([A_1 : A_5 \mid L]) \leftarrow A_1 : A_5 = abcab \tag{6.12}$$
$$s([A_1 \mid L]) \leftarrow s(L) \tag{6.13}$$

Una forma triangular directa que involucre inecuaciones para derivar el algoritmo MP en este caso no tiene sentido. En lugar de dicha forma triangular, trabajamos con la siguiente forma triangular modificada:

$$s([A_1 : A_5 \mid L]) \leftarrow A_1 : A_5 = abcab \tag{6.14}$$
$$s([A_1 \mid L]) \leftarrow s(L) \tag{6.15}$$
$$s([A_1 \mid L]) \leftarrow A_1 = a,\ s(L) \tag{6.16}$$
$$s([A_1 : A_2 \mid L]) \leftarrow A_1 : A_2 = ab,\ \underline{s([A_2 \mid L])}_1 \tag{6.17}$$
$$s([A_1 : A_3 \mid L]) \leftarrow A_1 : A_3 = abc,\ \underline{s([A_2 : A_3 \mid L])}_2 \tag{6.18}$$
$$s([A_1 : A_4 \mid L]) \leftarrow A_1 : A_4 = abca,\ \underline{s([A_2 : A_4 \mid L])}_2 \tag{6.19}$$
$$s([A_1 : A_5 \mid L]) \leftarrow A_1 : A_5 = abcab,\ \underline{s([A_2 : A_5 \mid L])}_2 \tag{6.20}$$

La diferencia con una forma triangular directa es que en esta ocasión no tenemos las inecuaciones. (La introducción de una inecuación y una ecuación en cada paso, en analogía a lo que hicimos para el caso BM, nos llevaría a calcular $f(j+1)$ y $next(j)$.) La justificación lógica de las cláusulas adicionales es en este caso está fundamentada en la necesidad de hallar cada prefijo como un paso intermedio para hallar a todo el patrón. Note que este es un conjunto de cláusulas intermedio que tiene una estructura apropiada como programa lógico, pero *sin simplificar* ya que la cláusula (6.15) subsumiría a las demás cláusulas recursivas.

Teorema 5

$$f(j) = (j-1) - det_unf(C'_j) \tag{6.21}$$

en donde C_j *es la siguiente cláusula:*

$$s([A_1 : A_{j-1} \,|\, L]) \leftarrow A_1 : A_{j-1} = p_1 : p_{j-1},\; s([A_2 : A_{j-1} \,|\, L]) \tag{6.22}$$

Demostración. Primero, notemos que no es necesario trabajar con todo el patrón en esta ocasión, a diferencia del tratamiento que dimos para los algoritmos de tipo BM, pues ahora sólo necesitamos tratar con un segmento que abarque desde p_1 hasta p_{j-1}.

Para encontrar el valor de $f(j)$ en términos del número de desdoblados determinísticos aplicados a la cláusula C_j formulamos la siguiente configuración inicial:

$$\begin{array}{lllll} p_1 & p_2 & \cdots & p_{j-1} & \\ & p_1 & p_2 & \cdots & p_{j-1} \end{array} \tag{6.23}$$

Después de desdoblar determinísticamente, obtenemos la siguiente configuración:

$$\begin{array}{llllll} p_1 & p_2 & \cdots & p_l & \cdots & p_{j-1} \\ & & & p_1 & \cdots & p_{i-1} \end{array} \tag{6.24}$$

o bien

$$\begin{array}{llllll} p_1 & p_2 & \cdots & p_{j-i+1} & \cdots & p_{j-1} \\ & & & p_1 & \cdots & p_{i-1} \end{array} \tag{6.25}$$

en donde $i \leq j$. El número de desdoblados determinísticos que realizamos para alcanzar esta configuración fue:

$$j - i + 1 - 1 = j - i$$

Ahora bien $f(j) = i - 1$ (por definición), por lo que

$$det_unf(C_j) = j - i = j - 1 - (i - 1) = j - 1 - f(j)$$

es decir

$$f(j) = j - 1 - det_unf(C_j)$$

■

Para ejemplificar la fórmula de f en el caso que estamos tratando, primero desdoblamos las cláusulas (6.15, 6.16, 6.17, 6.18, 6.19, 6.20):

i	$f(i)$	Cláusula
1	0	$s([A_1 \mid L]) \leftarrow s(L)$
2	1	$s([A_1 \mid L]) \leftarrow A_1 = a,\ s(L)$
3	1	$s([A_1 : A_2 \mid L]) \leftarrow A_1 : A_2 = ab,\ s(L)$
4	1	$s([A_1 : A_3 \mid L]) \leftarrow A_1 : A_3 = abc,\ s(L)$
5	2	$s([A_1 : A_4 \mid L]) \leftarrow A_1 : A_4 = abca,\ s([A_4 \mid L])$
6	3	$s([A_1 : A_5 \mid L]) \leftarrow A_1 : A_5 = abcab,\ s([A_4 A_5 \mid L])$

Para $i = 3$, si C es la cláusula (6.17), tenemos

$$f(3) = i - 2 - det_unf(C) = 3 - 1 - 1 = 1$$

y para $i = 6$, si C es la cláusula (6.20), tenemos

$$f(6) = i - 1 - det_unf(C) = 5 - 2 = 3$$

6.3.2 Caso KMP

Tenemos ahora un teorema que enlaza las derivaciones determinísticas con la función *next*.

Teorema 6 *Sea C una cláusula de la forma*

$$\begin{aligned} s'([A_1, \ldots, A_m \mid L]) \leftarrow A_1 = p_1,\ \ldots,\ & A_{j-1} = p_{j-1}, \\ & A_j \neq p_j, \\ & s'([A_2, \ldots, A_m \mid L]) \end{aligned}$$

Entonces, $det_unf(C) = j - next(j) - 1$, donde $det_unf(C)$ esta definido como es usual, y seleccionamos la submeta $s([A_2, \ldots, A_m \mid L])$ para desdoblarla con respecto a (6.1) y (6.2).

Este teorema es consecuencia de (i) la definición de *next* y (ii) las derivaciones determinísticas con respecto a (6.1) y (6.2), las cuales desplazan a la derecha el segmento del patrón representado por la submeta seleccionada, hasta que un prefijo propio y maximal casa con un sufijo de tal segmento del patrón. Alternativamente, una demostración directa basada en el conteo del número de desdoblados determinísticos y en la definición de *next* también es posible. ∎

Considere, como ejemplo, a la tabla *next* para el patrón *aab*:

j	1	2	3
p_j	a	a	b
$next(j)$	0	0	2

Consideremos por ejemplo el caso de (6.3), para el $j = 2$. El número de desdoblados determinísticos aplicados a esta cláusula es

$$j - next(j) = 2 - 0 - 1 = 1$$

6.4 Conclusiones

Concluimos este capítulo como sigue:

1. Existen algunos desarrollos que son comunes a la derivación de las variantes tipo BM y tipo KMP, con lo que podemos afirmar que nuestro proceso previo que nos sirvió para derivar variantes del algoritmo BM tiene una aplicabilidad mayor de la inicialmente esperada, por lo que tal vez nuestros métodos son aplicables a otro tipo de problemas que no son de cadenas; la idea, en lo específico, que nos guió para realizar esta derivación de acuerdo con el concepto de *reuso de desarrollos* nació, básicamente, del artículo de Partsch y Völker [PV91];

2. Dada la formulación de la variante BM_{12*}, podemos obtener una variante más del algoritmo KMP que desplace el patrón de manera exacta en cada ocasión; en efecto, en cada ocasión de un no-casamiento, digamos en la posición del patrón p_3 y con el símbolo del texto t_k (para un patrón de longitud mayor que 3, digamos) el algoritmo KMP sólo exige que en la próxima alineación se tenga, para un símbolo t_i, que $t_i \neq t_k$ (para evitar que de nuevo se tenga el mismo mal-casamiento previo, lo que en [CR94, 35] se llama un *borde etiquetado (tagged border)*. La manera exacta de desplazar el patrón sería exigiendo que $t_i = p_3$, lo que en el caso general nos conduce naturalmente al *algoritmo de Simon* [CR94, página 42].

3. Nuestra introducción inmediata de las ecuaciones e inecuaciones, y la posterior aplicación de las derivaciones determinísticas contrasta con la derivación que Pettorossi, Proietti y Renault hicieron del algoritmo KMP.

Abundemos más en el último punto. La introducción inmediata de las ecuaciones e inecuaciones, y la posterior aplicación de las derivaciones determinísticas que nosotros utilizamos para derivar el algoritmo KMP es diferente de la que Pettorossi, Proietti y Renault hicieron del algoritmo KMP, que está basada en la *construcción por subconjuntos de un autómata finito determinístico a partir de un autómata finito no-determinístico*. Tal estrategia tiene una naturaleza *cíclica*, en el sentido de que la introducción de ecuaciones y los desdoblados apropiados se van realizando de manera gradual, en lugar de realizar esta introducción de ecuaciones y los desdoblados de manera inmediata, como hicimos nosotros. En contraste, la introducción de ecuaciones en nuestro método tiene un carácter manual (aunque para este caso particular es posible parametrizar una serie de reglas de transformación de programas para que, al menos en parte, los programas residuales sean obtenibles automáticamente). Notablemente, una justificación adicional para la utilización de nuestro método es la facilidad con la que hemos demostrado las correspondencias entre las funciones auxiliares de los algoritmos de cadenas que tratamos y los programas intermedios o resultantes obtenidos. Para propósitos de comparación, en [ADR02] se observa que las demostraciones de corrección de los programas residuales se han evadido por largo tiempo, y en particular, en este trabajo se adopta un enfoque imperativo para tratar un problema que fue resuelto, al menos en teoría, de modo declarativo. Igualmente valioso de notar es el caso de [FKG02] en donde la demostración de corrección del algoritmo BM los autores la presentan vía una especie de forma sufijo-triangular, y con conceptos adicionales *ad hoc* que nosotros creemos son innecesarios si se aborda el problema con nuestro enfoque.

Debido a la importancia de la derivación de la parte de búsqueda del algoritmo KMP dada por Pettorossi, Proietti y Renault, en el siguiente apéndice desarrollamos tal derivación.

6.5 Apéndice: El algoritmo de Knuth, Morris y Pratt según Pettorossi, Proietti y Renault

A continuación mostramos el proceso que Pettorosi, Proietti y Renault desarrollan para derivar el programa residual correspondiente a la cadena *aab* y que implementa la parte de búsqueda del algoritmo de Knuth, Morris y Pratt. Nos referiremos a este proceso como *la derivación D del algoritmo KMP*.

La derivación D del algoritmo KMP, dada en [PPR97a] comienza con la siguiente especificación:

Programa 24

$$naive_match(P, S) \leftarrow append(Left, P, X),\ append(X, Right, S) \quad (6.26)$$

$$append([\,], Y, Y) \leftarrow \quad (6.27)$$

$$append([C \mid X], Y, [C \mid Z]) \leftarrow append(X, Y, Z) \quad (6.28)$$

Ahora especializamos con respecto al patrón *aab*:

$$match1(S) \leftarrow naive_match([a, a, b], S) \quad (6.29)$$

Primera iteración. Desdoblamos la cláusula (6.29) con respecto a *naive_match*:

$$match1(S) \leftarrow append(Left, [a, a, b], X),\ append(X, Right, S) \quad (6.30)$$

No desdoblamos aún más (6.29) debido a que cada átomo en su cuerpo unifica con más de una cabeza, pues *append* está definido por dos cláusulas. Con el propósito de uniformizar el proceso, es necesario crear una definición intermedia, que nos permita doblar la cláusula (6.30):

$$match2(S) \leftarrow append(Left, [a, a, b], X),\ append(X, Right, S) \quad (6.31)$$

Doblamos ahora la cláusula (6.30) con respecto a la cláusula doblante (6.31):

$$match1(S) \leftarrow match2(S) \quad (6.32)$$

Segunda iteración. La cláusula (6.32) formará parte del programa residual, pero *match2* requiere mayor elaboración, por lo que desdoblamos (6.31) con respecto a *append*:

$$match2([C \mid S]) \leftarrow C = a, append([a, b], Right, S) \quad (6.33)$$

$$match2([C \mid S]) \leftarrow append(Left, [a, a, b], X),\ append(X, Right, S) \quad (6.34)$$

A continuación aplicamos una división en casos en el cuerpo de (6.34), con respecto a $C = a$ en el cuerpo de la cláusula (6.33):

$$match2([C \mid S]) \leftarrow \boxed{C = a,}\ append([a, b], Right, S) \quad (6.35)$$

$$match2([C \mid S]) \leftarrow \boxed{C = a,}\ append(Left, [a, a, b], X),\ append(X, Right, S) \quad (6.36)$$

$$match2([C \mid S]) \leftarrow C \neq a,\ append(Left, [a, a, b], X),\ append(X, Right, S) \quad (6.37)$$

Hemos encerrado en un recuadro a la submeta en común $C = a$ de las cláusulas (6.35) y (6.36), y en la estrategia D debemos introducir una nueva definición:

$$match3(S) \leftarrow append([a,b], Right, S) \tag{6.38}$$
$$match3(S) \leftarrow append(Left, [a,a,b], X),\ \ append(X, Right, S) \tag{6.39}$$

Un doblado del conjunto de cláusulas (6.35) y (6.36), utilizando la definición dada por las cláusulas (6.38) y (6.39), y un doblado de la cláusula (6.37) utilizando la cláusula (6.31) produce el siguiente conjunto de cláusulas:

$$match2([C \mid S]) \leftarrow C = a,\ \ match3(S) \tag{6.40}$$
$$match2([C \mid S]) \leftarrow C \neq a,\ \ match2(S) \tag{6.41}$$

Tercera iteración. En la tercera iteración, desdoblamos las cláusulas (6.38) y (6.39), con respecto la definición de *append* (tomando el *append* de más a la izquierda):

$$match3([C \mid S]) \leftarrow C = a,\ \ append([b], Right, S) \tag{6.42}$$
$$match3([C \mid S]) \leftarrow C = a,\ \ append([a,b], Right, S) \tag{6.43}$$
$$match3([C \mid S]) \leftarrow append(Left, [a,a,b], X),\ \ append(X, Right, S) \tag{6.44}$$

De nuevo aplicamos la regla de división en casos en (6.44) utilizando la ecuación $C = a$:

$$match3([C \mid S]) \leftarrow C = a,\ \ append(Left, [a,a,b], X),\ \ append(X, Right, S) \tag{6.45}$$
$$match3([C \mid S]) \leftarrow C \neq a,\ \ append(Left, [a,a,b], X),\ \ append(X, Right, S) \tag{6.46}$$

Ahora identificamos a $C = a$ como una meta que es común al cuerpo del conjunto de cláusulas (6.47):

$$\left[\begin{array}{ll} match3([C \mid S]) \leftarrow \boxed{C = a,} & append([b], Right, S) \\ match3([C \mid S]) \leftarrow \boxed{C = a,} & append([a,b], Right, S) \\ match3([C \mid S]) \leftarrow \boxed{C = a,} & append(Left, [a,a,b], X), append(X, Right, S) \end{array}\right. \tag{6.47}$$
$$match3([C \mid S]) \leftarrow C \neq a,\ \ append(Left, [a,a,b], X),\ \ append(X, Right, S) \tag{6.48}$$

introducimos la siguiente definición:

$$match4(S) \leftarrow append([b], Right, S) \tag{6.49}$$
$$match4(S) \leftarrow append([a,b], Right, S) \tag{6.50}$$
$$match4(S) \leftarrow append(Left, [a,a,b], X),\ \ append(X, Right, S) \tag{6.51}$$

Doblamos algunas cláusulas de *match*3 utilizando la definición de *match*4:

$$match3([C \mid S]) \leftarrow C = a,\ match4(S) \tag{6.52}$$

$$match3([C \mid S]) \leftarrow C \neq a,\ match2(S) \tag{6.53}$$

Cuarta iteración. Desdoblando las cláusulas (6.49), (6.50), y (6.51):

$$match4([C \mid S]) \leftarrow C = b,\ append([\,], Right, S) \tag{6.54}$$

$$match4([C \mid S]) \leftarrow C = a,\ append([b], Right, S) \tag{6.55}$$

$$match4([C \mid S]) \leftarrow C = a,\ append([a,b], Right, S) \tag{6.56}$$

$$match4([C \mid S]) \leftarrow append(Left, [a,a,b], X),\ append(X, Right, S) \tag{6.57}$$

En esta ocasión aplicamos la regla de división en casos en dos ocasiones. En la primera ocasión, aplicamos esta regla con respecto a la ecuación $C = b$:

$$match4([C \mid S]) \leftarrow C = b,\ append(Left, [a,a,b], X),\ append(X, Right, S) \tag{6.58}$$

$$match4([C \mid S]) \leftarrow C \neq b,\ append(Left, [a,a,b], X),\ append(X, Right, S) \tag{6.59}$$

y en la segunda ocasión con respecto a $C = a$:

$$match4([C \mid S]) \leftarrow C = b,\ append(Left, [a,a,b], X),\ append(X, Right, S) \tag{6.60}$$

$$\begin{aligned} match4([C \mid S]) \leftarrow\ & C \neq b,\ C = a,\ append(Left, [a,a,b], X),\\ & append(X, Right, S) \end{aligned} \tag{6.61}$$

$$\begin{aligned} match4([C \mid S]) \leftarrow\ & C \neq b,\ C \neq a,\ append(Left, [a,a,b], X),\\ & append(X, Right, S) \end{aligned} \tag{6.62}$$

Después de una simplificación en el cuerpo de la cláusula (6.61), obtenemos los siguiente:

$$match4([C \mid S]) \leftarrow append(Left, [a,a,b], X),\ append(X, Right, S) \tag{6.63}$$

(ya que $a \neq b$ es una submeta que se resuelve inmediatamente). Ahora *match*4 se transforma en:

$$\left[\begin{array}{l} match4([C \mid S]) \leftarrow \boxed{C = b,}\ append([\,], Right, S) \\ match4([C \mid S]) \leftarrow \boxed{C = b,}\ append(Left, [a,a,b], X), append(X, Right, S) \end{array}\right. \tag{6.64}$$

$$\left[\begin{array}{l} match4([C \mid S]) \leftarrow C = a,\ append([b], Right, S) \\ match4([C \mid S]) \leftarrow C = a,\ append([a,b], Right, S) \\ match4([C \mid S]) \leftarrow append(Left, [a,a,b], X),\ append(X, Right, S) \end{array}\right. \tag{6.65}$$

$$\begin{aligned} match4([C \mid S]) \leftarrow\ & C \neq b,\ C \neq a,\\ & append(Left, [a,a,b], X),\ append(X, Right, S) \end{aligned} \tag{6.66}$$

Para doblar el conjunto de cláusulas (6.64) introducimos la siguiente definición:

$$match5(S) \leftarrow append([\,], Right, S) \tag{6.67}$$

$$match5(S) \leftarrow append(Left, [a, a, b], X),\ append(X, Right, S) \tag{6.68}$$

en tanto que el conjunto de cláusulas (6.65) lo doblamos utilizando como cláusulas doblantes a *match4* mismo, y a la cláusula (6.66) la doblamos utilizando la cláusula (6.31). Doblando, obtenemos:

$$match4([C \mid S]) \leftarrow C = b,\ match5(S) \tag{6.69}$$

$$match4([C \mid S]) \leftarrow C = a,\ match4(S) \tag{6.70}$$

$$match4([C \mid S]) \leftarrow C \neq b,\ C \neq a,\ match2(S) \tag{6.71}$$

Quinta iteración. Desdoblando *match5*, obtenemos:

$$match5(S) \leftarrow \tag{6.72}$$

$$match5([C \mid S]) \leftarrow C = a,\ append([a, b], Right, S) \tag{6.73}$$

$$match5([C \mid S]) \leftarrow append(Left, [a, a, b], X),\ append(X, Right, S) \tag{6.74}$$

De acuerdo con la estrategia D, descartamos las cláusulas (6.73) y (6.74) por medio de la regla de subsumción.

Nuestro programa final es:

Programa 25

$$match1(S) \leftarrow match2(S) \tag{6.75}$$

$$match2([C \mid S]) \leftarrow C = a,\ match3(S) \tag{6.76}$$

$$match2([C \mid S]) \leftarrow C \neq a,\ match2(S) \tag{6.77}$$

$$match3([C \mid S]) \leftarrow C = a,\ match4(S) \tag{6.78}$$

$$match3([C \mid S]) \leftarrow C \neq a,\ match2(S) \tag{6.79}$$

$$match4([C \mid S]) \leftarrow C = b,\ match5(S) \tag{6.80}$$

$$match4([C \mid S]) \leftarrow C = a,\ match4(S) \tag{6.81}$$

$$match4([C \mid S]) \leftarrow C \neq b,\ C \neq a,\ match2(S) \tag{6.82}$$

$$match5(S) \leftarrow \tag{6.83}$$

(Es posible un mayor pos-procesamiento, al introducir cortes para eliminar las ecuaciones.)

Capítulo 7

Una taxonomía de algoritmos de casamiento de cadenas

La idea de *reuso* aparece en la programación en diversos contextos. Esta idea generalmente está asociada con dos intereses principales: a) con un decremento en el costo del desarrollo de programas y, b), con la confiabilidad de los programas resultantes. En este capítulo desarrollamos la idea de una taxonomía de que permite el reuso de algunas partes de las derivaciones de algunos algoritmos de casamiento de cadenas.

7.1 Reuso de desarrollos y familias de algoritmos

Uno de los conceptos que presentamos en el capítulo 6, precisamente, fue el de *reuso de desarrollos*. A continuación presentamos dos ventajas de un reuso de desarrollos en la transformación de programas y desde el punto de vista del desarrollo de *familias de algoritmos*.

Por un lado, la reutilización de desarrollos en el proceso de derivación de los algoritmos nos permite integrar y clasificar diversos algoritmos dentro de un *árbol taxonómico de derivación*. En particular, por medio de este árbol acentuamos las similitudes y las diferencias entre los algoritmos involucrados, contribuyendo con ello a una mejor comprensión de las decisiones de diseño que se tomaron para lograrlos.

Por otro lado, esta reutilización nos es útil para indicar que nuestras herramientas no están limitadas a la derivación de las variantes del algoritmo BM, sino que tienen una aplicación que abarca también el caso del algoritmo KMP, cuya derivación especializada es una de las pruebas tradicionales para sopesar la fuerza de los evaluadores parciales. Tal

ampliación de la aplicabilidad de nuestras técnicas nos permite suponer que tal vez haya otros problemas que podamos resolver por métodos análogos.

A continuación presentamos algunas de las variantes del algoritmo BM que hemos derivado para conformar un árbol taxonómico de algoritmos de casamiento de cadenas.

7.2 Variantes del algoritmo BM

7.2.1 Una variante menor

La variante que hemos obtenido en el capítulo 5 no corresponde exactamente al algoritmo de Boyer y Moore, tal como el ejemplo dado en la subsección 5.2.1 muestra. El proceso general que hemos descrito corresponde a una implementación de la función $\delta_1(X)$ sólo para el caso en que tenemos $X = A_m$, mientras para el caso de casamientos parciales la función δ_2 está completamente implementada en términos de los desplazamientos que debe ofrecer. Llamaremos a esta variante la variante BM_{12}.

7.2.2 La variante de Partsch

A partir de la siguiente observación obtenemos una variante de este algoritmo: si aplicamos una división en casos en cada inecuación $A_i \neq p_i$ tal que

- $A_i \in \{p_1 : p_m\}$;
- $A_i \notin \{p_1 : p_m\}$;

obtenemos la implementación de una variante que dieron Partsch y Stomp en [PS90].

La cláusula (5.96) se transforma ahora en una serie de cláusulas de la forma:

$$s_p([A_1 : A_m \mid L]) \leftarrow A_m : A_{k+1} = p_m : p_{k+1},\ A_k = p_i,\ s_p([A_2 : A_m \mid L]) \tag{7.1}$$

en donde $i \neq k$, y se tiene que satisfacer el siguiente conjunto de ecuaciones en el proceso de desdoblado determinístico:

$$\{A_m : A_{k+1} = p_m : p_{k+1},\ A_k = p_i,\} \cup \{A_2 : A_m = p_1 : p_{m-1}\} \tag{7.2}$$

o bien,

$$\begin{aligned} s_p([A_1 : A_m \mid L]) \leftarrow A_m : A_{k+1} = p_m : p_{k+1}, \\ A_k \neq p_1,\ \ldots A_k \neq p_m,\ s_p([A_2 : A_m \mid L]) \end{aligned} \tag{7.3}$$

A esta variante, que aplica $\in$ y $\notin$ a toda inecuación $A_i \neq p_i$, la llamamos BM_{12*}.

7.2.3 La variante de Horspool

La derivación de nuestra variante en donde sólo aplicamos división en casos a $A_m \neq p_m$ nos permite obtener implícitamente la función δ_1, aunque ya hemos obtenido una fórmula que relaciona explícitamente las derivaciones determinísticas con la función δ_1 (ecuación dada en el teorema 2). Consideremos ahora la situación en donde *no desdoblamos* en absoluto las cláusulas del tipo:

$$s_p([A_1 : A_m \mid L]) \leftarrow A_m : A_{k+1} = p_m : p_{k+1},\ A_k \neq p_k,\ s_p([A_2 : A_m \mid L]) \quad (7.4)$$

de manera que *no* realizamos de manera inmediata un doblado que nos permita eliminar el no-determinismo. Debido a que la submeta $s_p([A_2 : A_m \mid L])$ garantiza un avance de un símbolo en toda ocasión, la corrección de nuestro programa está garantizada. Supongamos ahora que en el evento de que $A_k \neq p_k$ *determinamos los avances por medio del predicado table para el valor que tome* A_k. Suponiendo los predicados auxiliares apropiados para determinar este avance, obtenemos la variante del algoritmo BM dada en [Hor80]. Llamaremos a esta variante BM_1.

En conclusión, en la variante de Horspool descartamos definitivamente a la función δ_2, y sólo utilizamos la función δ_1 para hacer los respectivos desplazamientos.

La importancia de la variante de Horspool radica en que es una variante práctica para alfabetos como el del español o el del inglés y tiene un costo bajo de preprocesamiento, expresado en nuestro formalismo al evitar desdoblar las cláusulas que conducirían a la construcción de la función δ_2.

7.3 La variante de Apostolico, Galil, y Giancarlo

En la variante de Apostolico, Galil, y Giancarlo[AG86] del algoritmo BM la problemática principal a resolver es la *ausencia de memoria del algoritmo BM.* En efecto, si bien los avances en el algoritmo BM en algunas ocasiones son dramáticos, este algoritmo tiene una ineficiencia básica asociada: la recomparación de símbolos (ineficiencia que no tiene, como veremos, el algoritmo de Knuth, Morris y Pratt).

En nuestras derivaciones es directo ver en dónde se dan estas recomparaciones: Una vez que en la cláusula

$$s_p([A_1 : A_m \mid L]) \leftarrow A_m : A_{k+1} = p_m : p_{k+1},\ A_k \neq p_k,\ s_p([A_2 : A_m \mid L])$$

desdoblamos (de acuerdo con la variante BM_{12}, por ejemplo), generamos una nueva

cláusula

$$s_p([A_1 : A_m \mid L]) \leftarrow A_m : A_{k+1} = p_m : p_{k+1},\ A_k \neq p_k,\ s_p(\phi(k)) \tag{7.5}$$

Ahora bien, existen dos casos de acuerdo a la estructura de $\phi(k)$:

1. si $\phi(k) = L$, el desdoblado ha eliminado toda variable A_i y la llamada recursiva generará nuevas variables que permitan intentar un nuevo casamiento;

2. si $\phi(k) = [A_k : A_m|L]$, es decir, si en la estructura de $\phi(k)$, como término, existe al menos una variable A_i, después de la aplicación de una derivación determinística, en la llamada recursiva *las variables* $A_k : A_m$ *tendrán que renombrarse y provocarán, vía la llamada recursiva asociada y las ecuaciones del cuerpo de la cláusula resolvente de entrada*, un nuevo conjunto de recomparaciones de otra manera innecesarias.

Como ejemplo, consideremos el siguiente patrón: *abbab*. Después de formular la siguiente configuración

$$\begin{array}{l} a\,b\,b\,a\,b \\ \;\;a\,b\,b\,a\,b \end{array}$$

la derivación determinística se detendrá con la siguiente otra configuración:

$$\begin{array}{l} a\,b\,b\,a\,b \\ \;\;\;\;\;\;a\,b\,b\,a\,b \end{array}$$

En la etapa en sí de búsqueda, con un texto como *abcabbab*, obtendremos, de acuerdo con el valor de avance de la función δ_2 lo siguiente:

En el primer intento de casamiento, tenemos la siguiente configuración:

$$\begin{array}{cccccccc} a & b & c & a & b & b & a & b \\ & & & \uparrow & \uparrow & & & \\ a & b & b & a & b & & & \end{array}$$

en donde $c \neq b$, y por lo tanto es necesario avanzar el patrón como sigue:

$$\begin{array}{cccccccc} a & b & c & a & b & b & a & b \\ & & & \uparrow\uparrow & \uparrow\uparrow & \uparrow & \uparrow & \uparrow \\ & & & a & b & b & a & b \end{array}$$

Esta vez encontramos el patrón, pero las dobles flechas indican que tuvimos que comparar dos veces algunos símbolos, lo que representa una ineficiencia inherente al algoritmo BM.

Como se puntualiza en [AG86], cuando el algoritmo BM desplaza el patrón a la derecha, el algoritmo no retiene ninguna información de los símbolos ya comparados. Así,

cada variante previa que hemos dado (y el algoritmo BM en sí mismo) tiene el problema de hacer algunas comparaciones innecesarias. En la siguiente variante del algoritmo BM, guardamos un registro de las subcadenas del patrón que ya han sido casadas con las correspondientes subcadenas del texto, y explotamos tal registro posteriormente, durante un nuevo intento de casamiento. El método, por lo tanto, explota una información extraída de las subcadenas del patrón que ya casaron con las subcadenas del texto en previos intentos. Esta variante, además, halla todas las ocurrencias del patrón en el texto, realizando a lo más $2n - m + 1$ comparaciones de símbolos. De hecho, con este método, no necesitamos examinar ningún símbolo dos veces.

La cláusula (5.82) nos permite reanudar de manera eficiente (después de aplicarle una derivación determinística) el proceso del casamiento del patrón que sigue a la detección de una ocurrencia del patrón en el texto. En el momento de procesar la llamada recursiva de esta cláusula obtenemos el procedimiento de Galil para detectar traslapes consecutivos de ocurrencias del patrón en el texto.

$$\begin{array}{l} p: p_1 \;\ldots\; p_{s-1}\; p_s \;\ldots\; p_m \\ t:\; t_1 \;\ldots\; t_{s-1}\; t_s \;\ldots\; t_m \end{array}$$

Supongamos, en la configuración previa, que $t_s : t_m = p_s : p_m$, pero $t_{s-1} \neq p_{s-1}$. De acuerdo con la regla de desplazamiento dada por δ_2 puede haber tres casos diferentes:

1. $p_s : p_m$ no ocurre otra vez dentro de $p_1 : p_m$, y entonces podemos desplazar el patrón sobre el texto como sigue:

   $$\begin{array}{ll} p: & \qquad\qquad\qquad\qquad p_1 \;\ldots\; p_{s-1}\; p_s \;\ldots\; p_m \\ t: & t_1 \;\ldots\; t_{s-1}\; t_s \;\ldots\; t_m \end{array}$$

   En este caso no es necesario ningún registro de comparaciones previas, pues el algoritmo avanza lo suficiente para *tener* que hacer nuevas comparaciones;

2. $p_s : p_m$ tiene un sufijo el cual es también un prefijo del segmento del texto $t_s : t_m$:

   $$\begin{array}{ll} p: & \qquad\qquad\quad p_1 \;\ldots\; p_{s-1}\; p_s \;\ldots\; p_m \\ t: & t_1 \;\ldots\; t_{s-1}\; t_s \;\ldots\; t_m \end{array}$$

   lo que Apostolico y Giancarlo llamaron "traslape de bordes" (border match).

Para evitar comparaciones innecesarias, entonces, tratamos cada caso de traslape como un caso especial, y sólo necesitamos tratar con el caso en que hay al menos una variable. Ya que existen a lo más m sufijos del patrón, tratamos entonces con a lo más m casos

especiales. Aún más, algunos sufijos (ímplicitos en la llamada recursiva) de la misma longitud pueden analizarse como un sólo caso particular.

Para evitar la recomparación de símbolos, definimos algunos nuevos predicados que tienen la parte desconocida de $[A_{(s-1)d} : A_m \mid L]$, es decir, L. Aplicando la regla de división en casos sobre los nuevos predicados, creamos nuevas formas (sub)-triangulares, que nos permiten tratar separadamente cada sub-caso de mal-casamiento. Si existe un mal-casamiento, llamamos a s_p (nivel superior) para tratar con una posible reocurrencia. Más aún, es posible hacer algunos desdoblados determinísticos extras y, finalmente, podemos aplicar el doblado sistemático a la forma sub-triangular para reducir el no-determinismo.

Detallemos el procedimiento. Tomemos un programa residual obtenido de tipo BM_{12} (sin la aplicación de $\in$ ni de $\notin$ a todo nivel). Primero, creamos nuevas definiciones: de

$$[A_{(s-1)d} : A_m \mid L] = [A_{s-1} : A_m B_{m-s} : B_m \mid L]$$

obtenemos nuevos predicados

$$match([B_{m-s} : B_m \mid L]) \leftarrow s([_ : _B_{m-s} : B_m|L])$$

en donde $[_:_]$ es una sublista de $m - (s-1)$ variables anónimas. Ahora bien, lo que necesitamos es analizar desde B_{m-s} a B_m en la próxima llamada de s. Nuestra próxima tarea es definir nuevos predicados para tratar con cada caso separadamente.

$$s_1([B_{m-s} : B_m|L]) \leftarrow s([_ : _B_{m-s} : B_m|L]) \tag{7.6}$$

Ahora aplicamos la regla de división en casos, pero sólo a $B_{m-s} : B_m$; luego, aplicamos desdoblado determinístico; finalmente, doblamos para decrementar el no-determinismo.

El costo de preprocesamiento, como vemos, es muy alto, pues existe una profusión de casos de subcadenas a tratar, y la consecución de esta variante es sólo por su valor teórico.

A esta variante la llamamos BM_{12r} (la r por el *registro* que hacemos de las comparaciones).

7.3.1 Variantes del algoritmo KMP

El algoritmo KMP tiene algunas variantes tales como el algoritmo MP y el algoritmo de Simon [CR94]. El algorimo de MP, en efecto, fué históricamente previo al algoritmo KMP. Hemos tratado la variante MP en el capítulo 6. Aunque existen otras variantes del algoritmo KMP, o bien de híbridos que están basados en el algoritmo de Boyer y Moore y el algoritmo KMP, un estudio detallado de estas variantes y de estos híbridos lo hemos dejado como trabajo a futuro.

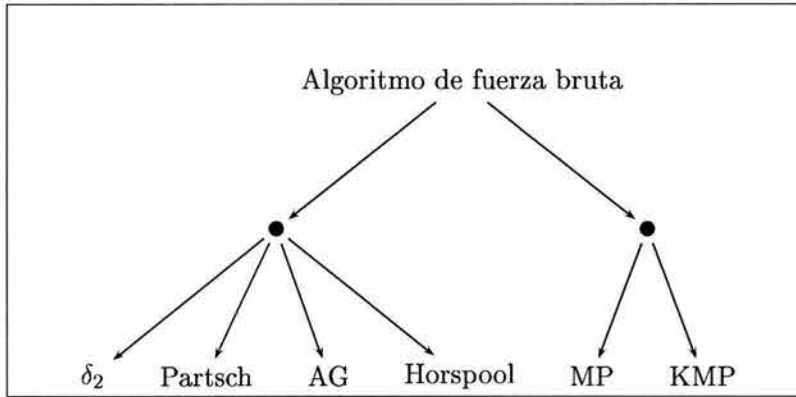

Figura 7.1: Una taxonomía transformacional de algunos algoritmos de casamiento de cadenas.

7.4 Acerca del diseño y desarrollo de una taxomía de algoritmos de casamiento de cadenas

En la Fig. 7.1 damos una representación taxonómica de algunos algoritmos de casamiento de cadenas. En esta figura un nodo interno es representado como un círculo si es una representación intermedia de la forma de un programa, pero no es un programa final en sí mismo ya sea por poseer un no-determinismo innecesario o una ineficiencia inherente. Un arco de un nodo a otro señala que al programa asociado con el primer nodo le aplicamos una sucesión de reglas de transformación para producir el programa del segundo nodo. Cada bifurcación de una trayectoria señala que hemos tomado una decisión de diseño para producir un subconjunto de algoritmos. El nodo interno del que parte la bifurcación actua como un ancestro de las hojas, y cada una de las hojas representa un miembro de una familia de algoritmos que siguen un método dado, ya sea el de Boyer y Moore, o el de Knuth, Morris, y Pratt, como explicamos en la Fig. 7.2.

En la Fig. 7.2 tenemos todas los hojas que en principio es posible obtener desde el programa inicial, y cada arco representa una sucesión de transformación. Además, en la Fig. 7.2 presentamos una taxonomía de los algoritmos que hemos derivado en esta tesis, clasificados de manera genérica, y omitiendo por transitividad los programas intermedios. Los métodos transformacionales, por consiguiente, tienen como resultado lateral ayudarnos a clasificar nuestro cuerpo actual de conocimientos.

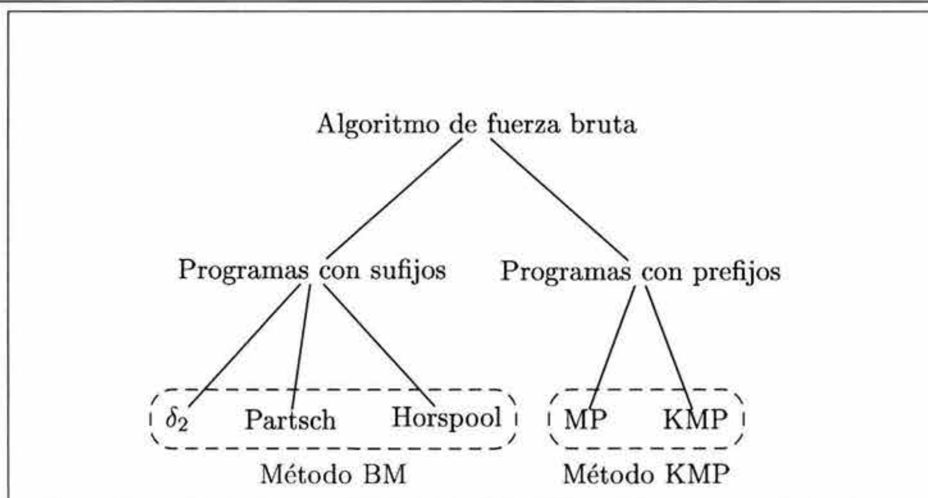

Figura 7.2: Una clasificación taxonómica de algunos algoritmos de casamiento de cadenas.

7.5 Conclusiones

Los desarrollos adicionales para alcanzar algunas variantes del algoritmo de Boyer y Moore, o del algoritmo de Knuth, Morris y Pratt, corresponden a características particulares de estas variantes, por lo que podemos dotarnos de un *criterio* para clasificar algunos algoritmos, lo que nos conduce naturalmente a un *árbol taxonómico de derivación (Fig. 7.1)*, cuya raíz es un programa directo o de fuerza bruta, cuyas bifurcaciones representan *decisiones de diseño*, y los arcos representan *sucesiones de transformación*, en tanto que una hoja de éste árbol representa un algoritmo en lo específico; en [CD80] y [Dar78] se aplica este tipo de clasificación a los algoritmos de ordenamiento, mientras que en [WZ96] se presenta una taxonomía de algunos algoritmos de casamiento de cadenas; ambos artículos, sin embargo, utilizan herramientas matemáticas que difícilmente serían implementables en un sistema transformador de programas automático o semi-automático.

Capítulo 8

La estrategia D'

La idea básica de la deducción parcial aparece en diversos contextos: por ejemplo, un programa específico puede verse como código que está adaptado para dar solución al problema para el que fue diseñado; en efecto, la ejecución de un programa específico es una especialización del uso del intérprete o del compilador. En este capítulo abordaremos algunas consecuencias de particularizar un programa cuando conocemos de antemano un argumento de uno de sus predicados.

8.1 Introducción

La deducción parcial (o evaluación parcial, en el contexto de programación funcional) es una técnica de derivación de programas basada en la *especialización* del programa con respecto a cierto parámetro [LS91, Kom92, Jon96, Leu98b, Leu98a].

El objetivo de la deducción parcial es la derivación *automática* de programas *eficientes* [Gal93, PPR97a], entendiendo por *automático* lo que está libre de intervención de parte de la usuaria o del usuario, y por *programa eficiente* un programa que tiene una mejora en ejecución con respecto al programa original, y en términos de alguna función de costo, lo que en el caso de la programación lógica frecuentemente involucra la eliminación del no-determinismo del programa inicial [PPR97b, PP98a]. La deducción parcial es un caso particular de la transformación de programas, ya que la deducción parcial requiere una aplicación sistemática de un conjunto de reglas dado, mientras que en la transformación de programas es posible que el usuario intervenga para guiar la aplicación de estas reglas. En ambos casos, es fundamental que las reglas preserven la semántica del programa inicial, pero mientras que en la transformación general de programas muchos pasos son guiados por

la intuición del programador (quien tiene que inventar ciertos predicados eureka, aunque existen algunos intentos de mecanizar la invención de estos predicados [PP90]), el objetivo básico en la especialización de programas es la *generación automática* de programas eficientemente ejecutables, aún a pesar de que la eficiencia obtenida no necesariamente sea la mejor de entre las existentes.

8.1.1 El algoritmo BM y la deducción parcial

Tradicionalmente, el algoritmo de casamiento de cadenas de Knuth, Morris y Pratt ha sido el caso de estudio típico en evaluación y deducción parcial [CD89, PV91, Smi91, PPR97a, SGJ96, ADR02]. Particularmente, Pettorossi, Proietti y Renault, en [PPR97a], ilustran su sistema de deducción parcial "disyuntiva" con una estrategia que deriva la parte de búsqueda del algoritmo de casamiento de cadenas de Knuth, Morris y Pratt. La suposición básica para tal deducción es que tenemos especializado un algoritmo de fuerza bruta no-determinístico de casamiento de cadenas con respecto a un patrón específico.

8.1.2 El caso de la derivación automática de la parte de búsqueda de variante del algoritmo BM

El algoritmo KMP es importante por su naturalidad y su elegancia teórica, pero en la práctica no es significativamente mejor que el algoritmo de fuerza bruta [Ste94, página 10]. En contraste, el algoritmo BM y algunas de sus variantes, son considerablemente más eficientes. Investigaciones recientes, sin embargo, señalan el hecho de que los evaluadores o deductores parciales requieren algunas modificaciones para tratar el caso BM [ACDM01, FKG02], y que para tratar el algoritmo BM con la mayor aproximación posible son necesarias algunas técnicas manuales de derivación [PS90, Pep91, HR01], aunque, como hemos observado en nuestras investigaciones que siguieron a [HR01], algunas partes de la derivación que involucran las funciones auxiliares δ_1 y δ_2 pueden realizarse automáticamente.

Por consiguiente, existen dos puntos de vista para abordar el problema de la mecanización de la derivación de un algoritmo de tipo BM:

- En un primer enfoque, nuestro objetivo es derivar manualmente un algoritmo tipo BM que se aproxime lo más posible al algoritmo original BM.
- En un segundo enfoque, nos limitamos a una variante del algoritmo BM que represente una mejora del algoritmo de fuerza bruta y que podamos obtener automáticamente.

En este capítulo seguimos el segundo enfoque y damos una generalización común de la estrategia de [PPR97a] que deriva la parte de búsqueda del algoritmo KMP, y la estrategia de [HR01] que deriva la parte de búsqueda de una variante del algoritmo BM, y específicamente, la variante que sólo involucra a la función δ_2. La generalización, entonces, es capaz de derivar el algoritmo KMP, por un lado, y por otro, es capaz de derivar la variante del algoritmo BM que sólo incorpora a la función δ_2. Esta generalización es mínima en el sentido de que al menos incluye ambas derivaciones, y además nos permite afirmar que tenemos una *estrategia* pues tenemos una sucesión bien definida de aplicaciones de reglas que se aplican a más de un algoritmo [PP02].

La generalización que proponemos tiene al menos tres conceptos novedosos que la caracterizan:

1. En contraste con la estrategia de D de Pettorossi, Proietti y Renault, nosotros omitimos ciertas aplicaciones de la regla de desdoblado en algunos pasos intermedios de la derivación;

2. también con contraste con la estrategia D, una importante modificación que hacemos es la selección de las variables que nos servirán para la aplicación de la regla de división en casos, pues a diferencia del algoritmo KMP, en donde siempre dividimos en casos con respecto a la cabeza de una lista, en el caso BM es necesario tener a disposición todas las variables del patrón especializado;

3. finalmente, desarrollamos una nueva variante del desdoblado determinístico, que explora la posibilidad de desdoblar una submeta A de acuerdo con un conjunto de submetas que habrán tenido éxito para cuando A sea considerada (provisión). En contraste, el enfoque tradicional de "ver hacia adelante" (previsión) nos permite decidir si desdoblamos una submeta A con base en una prueba anticipada.

8.1.3 Motivación de la estrategia D'

La motivación para crear esta nueva estrategia surgió de una analogía con el algoritmo KMP, y de dos problemas que nos encontramos en nuestras derivaciones de las variantes del algoritmo BM. La analogía con el algoritmo KMP viene de la siguiente observación: la construcción de la función δ_2 del algoritmo BM es similar a la construcción de la tabla *next* del algoritmo KMP, difiriendo en la dirección de concordancia símbolo a símbolo: si la dirección es de derecha a izquierda en el caso KMP, o de derecha a izquierda en el caso BM. En cuanto a los problemas que hallamos en nuestras derivaciones de los algoritmos que siguen el método BM dadas en el capítulo 5 (así como la derivación presentada en [HR01]) los formulamos como sigue.

Por un lado, al utilizar listas, el tiempo de acceso a los elementos apropiados no es constante, sino lineal. En esta ocasión, en lugar de utilizar listas, representamos el texto como un conjunto finito de cláusulas unitarias, pues tales conjuntos en muchos sistemas de programación lógica se implementan como arreglos de sólo lectura, a los que se accede en tiempo constante.

Por otro lado, las complicaciones propias del algoritmo BM nos condujeron a derivarlo manualmente, de manera que la estrategia que adoptamos para derivar el algoritmo BM está adaptada a cada una de la derivaciones de sus variantes y del algoritmo BM mismo. Al ligar ahora la estrategia que nos sirvió para derivar las variantes del algoritmo BM con la estrategia de Pettorossi, Proietti y Renault, ganamos cierta generalidad en relación a los algoritmos que podemos derivar sin intervención del (de la) usuario(a), a cambio de que estos algoritmos sean variantes algo limitadas de las versiones originales (en este caso, la versión original del algoritmo BM).

Al corregir las dos deficiencias señaladas, enfatizamos algunos aspectos. Primero, observemos que al utilizar conjuntos finitos de cláusulas unitarias que simulan arreglos, no perdemos el aspecto declarativo de la programación lógica, pues no estamos simulando la asignación destructivo, ya que sólo utilizamos el arreglo como de sólo-lectura, sin alterar los componentes del mismo. Segundo, la modificación que hacemos del orden, cantidad, y tipo de aplicaciones de las reglas que nos sirvieron para derivar el algoritmo BM, es con el objetivo de aproximarnos a los lineamientos de la estrategia D de Pettorossi, Proietti y Renault, de manera que la modificación resultante sea automática. Al final de este capítulo, damos algunas demostraciones de la corrección de los programas resultantes al aplicar nuestra estrategia.

8.2 Algoritmos de casamiento de cadenas

Ya hemos descrito los fundamentos de algunos algoritmos de casamiento de cadenas en capítulo 4, pero ahora describimos en detalle la variante del algoritmo BM que nos permitió acercarnos a la estrategia D de Pettorossi, Proietti y Renault. Llamaremos a nuestra estrategia la *estrategia D'*.

Podemos, entonces, precalcular el número de símbolos que necesitamos desplazar el patrón como sigue:

$$\begin{aligned}\delta(j) \;=\; & min\{k+m-j \mid k \geq 1 \\ & \quad y\ (k < j \Rightarrow p_{j-k} \neq p_j) \\ & \quad y\ ((k < d \Rightarrow p_{d-k} = p_d) \quad \forall d\ [j < d \leq m])\} \end{aligned} \tag{8.1}$$

Esta fórmula que no es otra cosa que la correspondiente a la función δ_2 del algoritmo BM, ligeramente modificada para adaptarla a nuestro nuevo contexto (note la introducción de las implicaciones en lugar de las disyunciones).

Si la reocurrencia no desborda el patrón, entonces $p_{j-k} \neq p_j, p_{j-k+1} = p_{j+1}, \ldots, p_{m-k} = p_m$, lo que resulta del segundo componente de la conjunción de variar d desde $j+1$ a m en el tercer componente de la conjunción. Si la reocurrencia desborda el patrón, entonces $p_1 = p_{k+1}, \ldots, p_{m-k} = p_m$, que resulta de variar d desde $k+1$ hasta m en el tercer componente de la conjunción (si ninguna ecuación se satisface, $k = m$).

Como ya hemos visto, Boyer y Moore [BM77] utilizan, además de la función δ, otra función, la función δ_1 basada en el símbolo actual t_i del patrón, cuya utilidad está justificada porque en muchas ocasiones incrementa la longitud del desplazamiento, y calculan el máximo de ambas funciones cuando ocurre una discordancia. Algunas veces, sin embargo, este criterio contribuye poco al desempeño general del algoritmo BM (por ejemplo, si el alfabeto es pequeño [CR94, página 59]), aunque cabe señalar que en alfabetos como el del español sí resulta útil. La variante que tratamos ahora omite la construcción de la función δ_1 y el objetivo es la derivación automática de un algoritmo más eficiente que el algoritmo de fuerza bruta (ver Fig. 8.1). La derivación de [FKG02]), posterior a la que dimos en [HR01], también limita el algoritmo BM, y obtiene los programas residuales que sólo incorporan la función δ_2.

8.3 Deducción parcial disyuntiva y la estrategia D

8.3.1 La deducción parcial disyuntiva

La deducción parcial "disyuntiva" [PPR97a] emplea una regla de doblado que permite que las definiciones consten de varias cláusulas, que operan sobre disyunciones de conjunciones de literales. Tal como se ilustra en [PPR97a], este doblado que consta de varias cláusulas en la definición doblante (inicialmente propuesto en [GK94]) es capaz de decrementar el no-determinismo de algunos programas, y estos autores derivan la parte de búsqueda del algoritmo KMP desde una especificación directa y no-determinística. Un ejemplo de la aplicación de esta estrategia para el patrón aab está en el Apéndice del capítulo 6.

Pettorossi, Proietti y Renault mecanizan la aplicación de las reglas de transformación por medio de una estrategia que ellos llamaron *la estrategia D*, que ya hemos presentado en la subsección 3.6.2, y es mostrada en forma de seudo-código en la Fig. 8.2 y gráficamente en la Fig. 3.7.

```
inicialización(p, δ);
i := m; j := m;
while (j > 0) and (i ≤ n) do
    if t_i = p_j
    then
        begin
            i := i − 1;
            j := j − 1
        end  {casamiento parcial}
    else
        begin
            i := i + δ(j);
            j := m
        end  {discordancia}
if j < 1 then i := i + 1 else i := 0
```

Figura 8.1: Una variante del algoritmo de Boyer y Moore.

Como ya hemos comentado en la subsección 3.6.2, la estrategia D generaliza la estrategia S de Gallagher [Gal93] en el sentido de que en la estrategia S no está permitido el doblado utilizando varias cláusulas, mientras que en la estrategia D sí lo está, ya que aquí sí es posible crear definiciones que consten de varias cláusulas. El resultado es que algunas computaciones que están en un mismo nivel en el árbol de derivación son *fusionadas* y por ello la aplicación de la regla de doblado mejora la eficiencia de los programas. La estrategia D, además, guarda una estrecha relación con el algoritmo de la construcción por subconjuntos que determiniza un autómata finito no-determinístico, y esta eliminación de no-determinismo es precisamente una de las características más importantes que hacen útil a la estrategia D, o a estrategias parecidas.

Ya en algunas ocasiones hemos considerado la importancia del desdoblado determinístico. Algunas veces, junto con el doblado determinístico, utilizamos una previsión para decidir si una submeta la podemos considerar determinística, al desdoblar las submetas que tendrán éxito después de A. En lugar de esta previsión, nosotros utilizaremos una *provisión*, al acumular submetas que habrán tenido éxito antes de que consideremos a la submeta A.

Cada ciclo de la estrategia D comprende lo siguiente:

$i := 0$; $Def_0 := \{C\}$; /* C es la especificación (cláusula especializada) */
repeat
(a) $Cs := case_split(det_unfold(unfold_once(Def_i)))$;
(b) $Def_{i+1} := required_defs(\bigcup_{j=0}^{i} Def_j, Cs)$;
(c) $F_{i+1} := fold(\bigcup_{j=0}^{i+1} Def_j, Cs)$;
(d) $U_{i+1} := unit_clauses(Def_{i+1})$;
$i := i + 1$
until $Def_i = \{\}$;
$P_{residual} := \bigcup_{j=1}^{i} U_j \cup \bigcup_{j=1}^{i} F_j$

Figura 8.2: La estrategia D.

(a) exactamente un desdoblado, posiblemente no-determinístico, de cada cláusula en Def_i, seguido por cero o más aplicaciones de desdoblados determinísticos, división en casos, subsumción, y simplificación de inecuaciones;

(b) cero o más introducción de definiciones;

(c) cero o más doblados; y

(d) la extracción de las cláusulas unitarias.

8.3.2 La estrategia D'

En la próxima sección veremos que una estrategia con la que derivamos la parte de búsqueda de una variante del algoritmo BM es similar a la estrategia D, excepto en los siguientes puntos:

i. omitimos el desdoblado posiblemente no-determinístico si una cierta condición (dada más adelante) se cumple;

ii. primero aplicamos la regla de división en casos, y luego el desdoblado determinístico tantas veces como sea posible (aplicación de derivaciones determinísticas)

iii. realizamos la aplicación de la regla de división en casos con respecto a la ocurrencia textual (de derecha a izquierda o de izquierda a derecha) del patrón en el cuerpo de una cláusula; y,

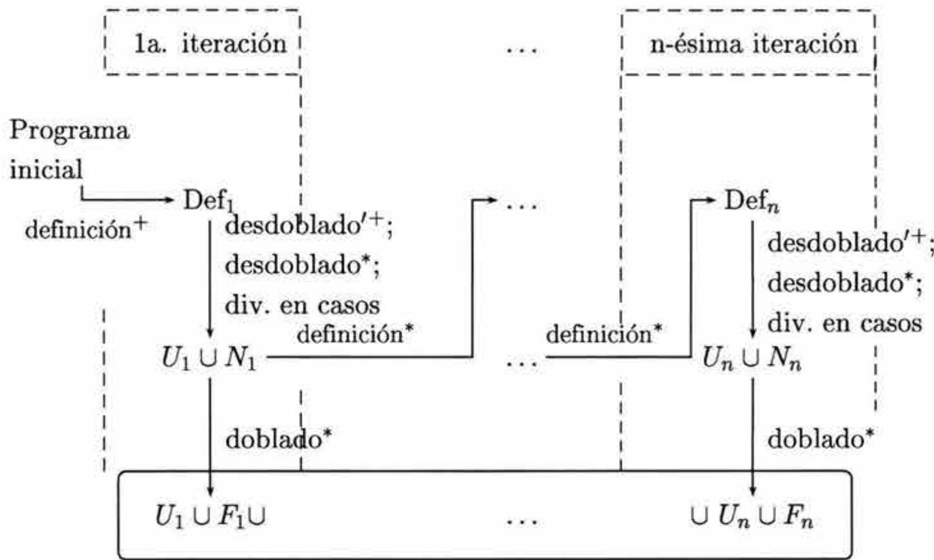

Figura 8.3: La estrategia D' representada gráficamente.

iv. el criterio para detener las derivaciones determinísticas es con respecto a las afirmaciones acumuladas previamente (provisión en lugar de previsión).

Para obtener una generalización común, nombrada por nosotros la *estrategia D'*, reemplazamos (a) por:

(a') $Cs := det_unfold'(case_split(det_unfold(possibly_unfold_once(Def_i))))$;

donde *det_unfold'* utiliza afirmaciones acumuladas, y *possibly_unfold_once* desdobla una vez sólo si una condición que evite una profusión de casos se cumple. Damos tal condición en detalle más adelante. Presentamos la estrategia D' en forma algorítmica en la Fig. 8.4. Damos una representación gráfica de la estrategia D' parecida a la de la Fig. 3.7 del Capítulo 3 en la Fig. 8.3, y en donde por "desdoblado$'$" señalamos un posible desdoblado no determinístico, mientras con "desdoblado*" indicamos la aplicación de cero o más desdoblados determinísticos.

$i := 0$; $Def_0 := \{C\}$; /* C es la especificación (cláusula especializada) */
repeat
(a') $Cs := det_unfold'(case_split(det_unfold(possibly_unfold_once(Def_i))))$;
(b) $Def_{i+1} := required_defs(\bigcup_{j=0}^{i} Def_j, Cs)$;
(c) $F_{i+1} := fold(\bigcup_{j=0}^{i+1} Def_j, Cs)$;
(d) $U_{i+1} := unit_clauses(Def_{i+1})$;
$i := i + 1$
until $Def_i = \{\}$;
$P_{residual} := \bigcup_{j=1}^{i} U_j \cup \bigcup_{j=1}^{i} F_j$

Figura 8.4: La estrategia D'.

8.4 Una derivación de la parte de búsqueda de una variante del algoritmo BM

Conceptualmente, nuestra derivación consiste en lo siguiente:

(a) la adición de restricciones vía la regla de división en casos, y

(b) la repetida aplicación del desdoblado determinístico hasta que las restricciones se satisfagan.

Para conformarnos a una estrategia cíclica, sin embargo, será más conveniente alternar ambos procesos incrementalmente.

Representamos el texto por el predicado $t(I, X)$ que deseamos que se cumpla cuando el símbolo en la posición I del texto es X. Las implementaciones estándar de Prolog, por ejemplo, normalmente indexan los predicados con respecto a la posición del argumento más a la izquierda, de manera que las submetas $t(I, X)$ que tienen un primer argumento constante tendrán éxito en tiempo constante. Para agilizar un poco nuestra notación, abreviaremos por medio de $t(J{+}d, X)$ a I *is* $J + d$, $t(I, X)$.

Como ejemplo, supongamos el patrón *abb*. Una especificación entonces podría ser la siguiente:

Programa 26

$$new_0(I, K) \;\leftarrow\; entre(I, J, K),\; t(J{+}2, b),\; t(J{+}1, b),\; t(J, a)$$

en donde $entre(I, J, K)$ se cumple si y sólo si $I \leq J \leq K$, y en donde notamos el orden textual de las submetas $t(J{+}2, b)$, $t(J{+}1, b)$, $t(J, a)$.

La pregunta apropiada para este programa es: $\leftarrow new_0(1, N)$, que permite instanciar a N a la posición en donde ocurre el patrón (de existir el patrón en el texto). Por concisión, y para evitar tratar con un desborde por la derecha en la búsqueda del patrón, sin embargo, preferimos una variante de esta especificación que es no-terminante, al omitir un argumento del predicado *entre* (el cual nombraremos *leq*):

Programa 27

$$new_0(I) \leftarrow leq(I, J),\ t(J{+}2, b),\ t(J{+}1, b),\ t(J, a) \tag{8.2}$$

donde $leq(I, J)$ se cumple si y sólo si $I \leq J$:

$$\begin{aligned} &leq(I, I) \leftarrow \\ &leq(I, J) \leftarrow leq(I{+}1, J) \end{aligned}$$

Primera iteración. Al desdoblar la cláusula (8.2) obtenemos (explicaremos más adelante las anotaciones)):

$$new_0(I) \ \leftarrow\ \boxed{t(I{+}2, b)}^{*},\ t(I{+}1, b),\ t(I, a) \tag{8.3}$$

$$new_0(I) \ \leftarrow\ leq(I{+}1, J),\ t(J{+}2, b),\ t(J{+}1, b),\ t(J, a) \tag{8.4}$$

Aplicando una división en casos a (8.4) con respecto a $t(I{+}2, b)$ obtenemos:

$$new_0(I) \ \leftarrow\ \boxed{t(I{+}2, b)}^{*},\ leq(I{+}1, J),\ t(J{+}2, b),\ t(J{+}1, b),\ t(J, a) \tag{8.4a}$$

$$new_0(I) \ \leftarrow\ not(t(I{+}2, b)),\ leq(I{+}1, J),\ t(J{+}2, b),\ t(J{+}1, b),\ t(J, a) \tag{8.4b}$$

La primera diferencia con respecto a la estrategia D aparece aquí, pues la estrategia D no prescribe tal desdoblado. Sin embargo, una característica importante del algoritmo BM, como ya hemos visto, es la posibilidad de desplazar el patrón con respecto al texto en más de un símbolo en caso de un mal-casamiento, y obtenemos tal desplazamiento al desdoblar la cláusula (8.4b) (ya que t es desconocido, todos los desdoblados son con respecto a *leq*):

$$new_0(I) \leftarrow not(t(I{+}2, b)),\ t(I{+}3, b),\ t(I{+}2, b),\ t(I{+}1, a) \tag{8.4b$'$}$$

$$new_0(I) \leftarrow not(t(I{+}2, b)),\ leq(I{+}2, J),\ t(J{+}2, b),\ t(J{+}1, b),\ t(J, a) \tag{8.4b$''$}$$

Puesto que (8.4b′) tiene submetas contradictorias, podemos borrar la cláusula (8.4b′), de modo que este paso de desdoblado es, en efecto, determinístico. Sólo un desdoblado

determinístico es posible, ya que otro desdoblado derivaría dos cláusulas, ninguna de las cuales tiene submetas contradictorias. El desdoblado determinístico no puede ser aplicado a (8.4a). Para doblar las cláusulas (8.3) y (8.4a) con respecto de $t(I{+}2, b)$ (óvalos etiquetados con $*$), definimos:

$$new_1(I) \leftarrow \underline{t(I{+}1, b)}^{**},\ t(I, a) \tag{8.5}$$

$$new_1(I) \leftarrow leq(I{+}1, J),\ t(J{+}2, b),\ t(J{+}1, b),\ t(J, a) \tag{8.6}$$

Ahora doblamos las cláusulas (8.3) y (8.4a) utilizando las cláusulas (8.5) y (8.6), de lo que obtenemos:

$$new_0(I) \leftarrow t(I{+}2, b),\ new_1(I) \tag{8.7}$$

Al doblar (8.4b″) utilizando (8.2) obtenemos:

$$new_0(I) \leftarrow not(t(I{+}2, b)),\ new_0(I{+}2) \tag{8.8}$$

Segunda iteración. Una segunda diferencia con respecto a la estrategia D acontece en este punto, en donde tal estrategia prescribe un paso de desdoblado (posiblemente no-determinístico). La realización de tal paso, no obstante, nos llevaría a crear un no-determinismo indeseable. Para ver esto, supongamos que desdoblamos (8.6):

$$new_1(I) \leftarrow \underline{t(I{+}3, b)},\ t(I{+}2, b),\ t(I{+}1, a) \tag{8.6′}$$

$$new_1(I) \leftarrow leq(I{+}2, J),\ t(J{+}2, b),\ t(J{+}1, b),\ t(J, a) \tag{8.6″}$$

Las cláusulas (8.5) y (8.6′) tienen dos submetas esencialmente distintas (submetas subrayadas) las cuales generan una diversidad (profusión) de casos durante la aplicación de la regla de división de casos. Para evitar desdoblar, entonces, nosotros hemos ideado el criterio siguiente: *si al desdoblar una cláusula que define un predicado obtenemos una submeta de más a la izquierda A diferente de la submeta de más a la izquierda B de otra cláusula que define el mismo predicado, tal que* $\leftarrow A, B$ *puede ser éxitosa, entonces no desdoblamos.* Por ejemplo: $\leftarrow t(I{+}3, b),\ t(I{+}1, b)$ puede tener éxito, pero no así $\leftarrow t(I{+}2, b),\ t(I{+}2, a)$ debido a la funcionalidad de t con respecto a su primer argumento.

Al dividir en casos, en el cuerpo de la cláusula (8.6), con respecto de $t(I{+}1, b)$, obtenemos:

$$new_1(I) \leftarrow t(I{+}1, b),\ leq(I{+}1, J),\ t(J{+}2, b),\ t(J{+}1, b),\ t(J, a) \tag{8.6a}$$

$$new_1(I) \leftarrow not(t(I{+}1, b)),\ leq(I{+}1, J),\ t(J{+}2, b),\ t(J{+}1, b),\ t(J, a) \tag{8.6b}$$

Como en la primera iteración, ahora desdoblamos determinísticamente las cláusulas (8.6a) y (8.6b) si ello es posible. Esta vez, sin embargo, debemos desdoblar utilizando algunas

afirmaciones acumuladas; en particular, aquí usamos el hecho de que $t(I{+}2,b)$ se cumple (cf. (8.7)). Ahora (8.6a) puede desdoblarse determinísticamente, pero (8.6b) no.

$$new_1(I) \leftarrow \boxed{t(I{+}1,b)}^{**},\; leq(I{+}2,J),\; t(J{+}2,b),\; t(J{+}1,b),\; t(J,a) \tag{8.6a'}$$

Para doblar las cláusulas (8.5) y (8.6a′) con respecto de $t(I{+}1,b)$ (óvalos etiquetados con $**$), debemos crear la siguiente definición:

$$new_2(I) \leftarrow \boxed{t(I,a)}^{***} \tag{8.9}$$

$$new_2(I) \leftarrow leq(I{+}2,J),\; t(J{+}2,b),\; t(J{+}1,b),\; t(J,a) \tag{8.10}$$

Ahora doblamos las cláusulas (8.5) y (8.6a′) utilizando como cláusulas doblantes a (8.9) y (8.10), por lo que obtenemos:

$$new_1(I) \leftarrow t(I{+}1,b),\; new_2(I) \tag{8.11}$$

Al doblar (8.6b) utilizando (8.2), obtenemos:

$$new_1(I) \leftarrow not(t(I{+}1,b)),\; new_0(I{+}1) \tag{8.12}$$

Tercera iteración.

Por la misma razón que antes, no desdoblamos (aún más). Por medio de una división en casos tratamos ahora a la cláusula (8.10) con respecto de $t(I,a)$ y obtenemos:

$$new_2(I) \leftarrow t(I,a),\; leq(I{+}2,J),\; t(J{+}2,b),\; t(J{+}1,b),\; t(J,a) \tag{8.10a}$$

$$new_2(I) \leftarrow not(t(I,a)),\; leq(I{+}2,J),\; t(J{+}2,b),\; t(J{+}1,b),\; t(J,a) \tag{8.10b}$$

A continuación desdoblamos determinísticamente las cláusulas (8.10a) y (8.10b), considerando que $t(I{+}2,b)$ y $t(I{+}1,b)$ se cumplen (como el conjunto actual de afirmaciones acumuladas):

$$new_2(I) \leftarrow \boxed{t(I,a)}^{***},\; leq(I{+}3,J),\; t(J{+}2,b),\; t(J{+}1,b),\; t(J,a) \tag{8.10a'}$$

$$new_2(I) \leftarrow not(t(I,a)),\; leq(I{+}3,J),\; t(J{+}2,b),\; t(J{+}1,b),\; t(J,a) \tag{8.10b'}$$

Para doblar las cláusulas (8.9) y (8.10a′) con respecto de $t(I,a)$ (óvalos etiquetados con $***$), introducimos la siguiente definición:

$$new_3(I) \leftarrow \tag{8.13}$$

$$new_3(I) \leftarrow leq(I{+}3,J),\; t(J{+}2,b),\; t(J{+}1,b),\; t(J,a) \tag{8.14}$$

Al doblar las cláusulas (8.9) y (8.10a′) utilizando como cláusulas doblantes a (8.13) y (8.14), obtenemos:

$$new_2(I) \leftarrow t(I,a),\ new_3(I) \tag{8.15}$$

Doblamos ahora a la cláusula (8.10b′) utilizando la cláusula (8.2), y obtenemos:

$$new_2(I) \leftarrow not(t(I,a)),\ new_0(I{+}3) \tag{8.16}$$

Cuarta iteración. Al doblar (8.14) utilizando (8.2), obtenemos:

$$new_3(I) \leftarrow new_0(I{+}3) \tag{8.17}$$

Ahora recolectamos las cláusulas que finalmente conforman nuestro programa residual. En efecto, el programa residual consiste de las siguientes cláusulas (8.7, 8.8, 8.11, 8.12, 8.15, 8.16, 8.13, 8.17):

Programa 28

$$\begin{array}{lcl}
new_0(I) & \leftarrow & t(I{+}2,b),\ new_1(I)\\
new_0(I) & \leftarrow & not(t(I{+}2,b)),\ new_0(I{+}2)\\
new_1(I) & \leftarrow & t(I{+}1,b),\ new_2(I)\\
new_1(I) & \leftarrow & not(t(I{+}1,b)),\ new_0(I{+}1)\\
new_2(I) & \leftarrow & t(I,a),\ new_3(I)\\
new_2(I) & \leftarrow & not(t(I,a)),\ new_0(I{+}3)\\
new_3(I) & \leftarrow & \\
new_3(I) & \leftarrow & new_0(I{+}3)
\end{array}$$

Observamos ahora que todas las cláusulas del programa residual son de la forma:

$$\begin{array}{lll}
 & new_{m-j}(I) \leftarrow \quad t(I{+}j{-}1,p_j),\ new_{m-j+1}(I) & j=m,\dots,1\\
\text{o} & new_{m-j}(I) \leftarrow not(t(I{+}j{-}1,p_j)),\ new_0(I{+}c(j)) & j=m,\dots,1 \qquad (8.18)\\
\text{o} & new_m(I) \leftarrow & \\
\text{o} & new_m(I) \leftarrow new_0(I{+}c(0)) & \qquad (8.19)
\end{array}$$

para alguna función c, relacionada con δ como sigue:

Teorema 7 (Revisitado) $c(j) = \delta(j) - m + j$.

Demostración. La afirmación de este teorema es una reformulación del teorema dado por la ecuación (5.59), pero el contexto de derivación automática actual nos lleva a considerar algunos detalles adicionales en lo que respecta a la demostración. Trataremos primero con los mal-casamientos. En las cláusulas de la forma (8.18), $c(j)$ es el número (acumulado) de veces que hemos aplicado desdoblados determinísticos. De manera equivalente, podemos realizar tales desdoblados al final, una vez que hemos derivado las siguientes cláusulas:

$$\begin{array}{rcl} new_0(I) & \leftarrow & t(I{+}m{-}1, p_m),\ new_1(I) \\ & \vdots & \\ new_{m-j-1}(I) & \leftarrow & t(I{+}j, p_{j+1}),\ new_{m-j}(I) \\ new_{m-j}(I) & \leftarrow & not(t(I{+}j{-}1, p_j)),\ leq(I{+}1, J),\ agree(J, 0) \end{array}$$

Aquí, y en lo que sigue $agree(K, l)$ abrevia a $t(K + m - 1, p_m), \ldots, t(K + l, p_{l+1})$.

A continuación aplicamos desdoblados determinísticos a las cláusulas de la forma:

$$new_{m-j}(I) \ \leftarrow \ \boxed{agree(I, j),}\ not(t(I{+}j{-}1, p_j)),\ leq(I{+}1, J),\ agree(J, 0) \qquad (8.20)$$

Utilizamos un rectángulo para indicar cuáles son las submetas que debemos considerar como afirmaciones acumuladas (provisión).

Supongamos que, comenzando desde la cláusula (8.20), podemos aplicar $c(j) - 1$ veces el desdoblado determinístico, de tal forma que del primer desdoblado determinístico derivamos:

$$new_{m-j}(I) \ \leftarrow \ \boxed{agree(I, j),}\ not(t(I{+}j{-}1, p_j)),\ agree(I{+}c(j), 0) \qquad (8.21)$$

$$new_{m-j}(I) \ \leftarrow \ \boxed{agree(I, j),}\ not(t(I{+}j{-}1, p_j)),\ leq(I{+}c(j){+}1, J),\ agree(J, 0)$$

Note que (8.21) no tiene ninguna submeta contradictoria (de otra manera podríamos realizar otro paso de desdoblado determinístico), así que doblamos la cláusula del paso previo a este último desdoblado, obteniendo:

$$new_{m-j}(I) \ \leftarrow \ \boxed{agree(I, j),}\ not(t(I{+}j{-}1, p_j)),\ new_0(I{+}c(j)) \qquad (8.22)$$

Distinguimos ahora dos casos:

1. (La inecuación se cumple.) Supongamos que para algún índice k tal que $1 \leq k < j$,

   $$p_{j-k} \neq p_j \text{ y } (p_{d-k} = p_d) \quad \forall d \ [j < d \leq m].$$

   Entonces (8.21), la primera cláusula sin contradicciones, tiene las siguientes submetas:

$$\begin{array}{llll} & t(I{+}m{-}1, p_m), & \ldots, \; t(I{+}j, p_{j+1}), & not(t(I{+}j{-}1, p_j)), \\ \ldots, & t(I{+}m{-}1, p_{m-k}), & \ldots, \; t(I{+}j, p_{j+1-k}), & t(I{+}j{-}1, p_{j-k}) \end{array}$$

Los desdoblados determinísticos se detienen con el menor índice de tales k. Luego,

$$\begin{aligned} c(j) &= min\{k \mid k \geq 1 \text{ y } p_{j-k} \neq p_j \text{ y } (p_{d-k} = p_d) \;\; \forall d \; [j < d \leq m]\} \\ &= \delta(j) - m + j \end{aligned}$$

2. (La inecuación no se cumple para ningún índice k y algunas ecuaciones sí se cumplen.) Supongamos que para algún índice k tal que $j \leq k \leq m$,

$$(k < d \Rightarrow p_{d-k} = p_d) \;\; \forall d \; [j < d \leq m]$$

(en donde $k = m$ si ninguna ecuación se cumple). Entonces (8.21) tiene las siguientes submetas:

$$\begin{array}{llll} & t(I{+}m{-}1, p_m), & \ldots, \; t(I{+}k, p_{k+1}), & \ldots, \; t(I{+}j{-}1, p_j), \; \ldots \\ \ldots, & t(I{+}m{-}1, p_{m-k}), & \ldots, \; t(I{+}k, p_1) & \end{array}$$

En consecuencia,

$$\begin{aligned} c(j) &= min\{k \mid k \geq 1 \text{ and } ((k < d \Rightarrow p_{d-k} = p_d) \;\; \forall d \; [j < d \leq m])\} \\ &= \delta(j) - m + j \end{aligned}$$

Similarmente, en la cláusula (8.19), $c(0) = min\{k \mid k \geq 1 \text{ and } ((p_{d-k} = p_d) \;\; \forall d \; [k < d \leq m])\} = \delta(0) - m + 0$ especifica el tamaño del desplazamiento en caso de un casamiento completo del patrón, habilitando al programa para encontrar de manera eficiente algunas otras ocurrencias del mismo patrón. ■

La estrategia D' es capaz de derivar la parte de búsqueda del algoritmo KMP, lo que es posible confirmar partiendo del siguiente programa:

$$new_0(I) \leftarrow leq(I, J), \; t(J, a), \; t(J{+}1, b), \; t(J{+}2, b)$$

8.5 Conclusiones

En este capítulo hemos extendido la estrategia D [PPR97a] en una nueva, a la que hemos llamado estrategia D'. La estrategia D' no sólo es capaz de derivar la parte de búsqueda del algoritmo de KMP, tal como lo hace la estrategia D, sino también la parte de búsqueda de una variante del algoritmo de BM.

La selección de la derivación de un programa que implementa el algoritmo KMP o la variante del algoritmo BM se transforma en algo tan simple como el orden textual de casamiento en el programa de fuerza bruta. El resto de las decisiones están en el derivador de programas. Yendo un paso más adelante, y evitando pensar en los casos particulares que un derivador automático puede atacar, lo rescatable de cualquier estrategia está en los principios genéricos en los que se basa, para posteriormente adecuar sus partes esenciales a la derivación de otros algoritmos.

En este sentido, esperamos que estemos contribuyendo a tener un conjunto completo de herramientas, en donde dado un algoritmo A, tengamos una estrategia T que pueda derivar A. Por el momento, sin embargo, nuestras investigaciones tienen que ser abductivas: dado un algoritmo A, y una estrategia T, si T no deriva A debemos crear una estrategia T' que derive sin problemas a A, y que incluya como subestrategia a T. Si A está definitivamente fuera del alcance de herramientas existentes, entonces nos planteamos otro problema: la derivación utilizando la estrategia T de un algoritmo A' parecido a A (es decir, una variante de A). Si todavía con T no podemos derivar a A', buscamos (abductivamente) una estrategia T' que derive a A'. De esta manera ganamos terreno en el conocimiento de nuestras potenciales herramientas teniendo como referencia a la estragia inicial T, y de paso, en la naturaleza de A. En este capítulo en particular, el algoritmo A es el algoritmo BM restringido a utilizar sólo la función δ_2, T es la estrategia D, y T' es la estragia D'.

Capítulo 9

Trabajo relacionado

En este capítulo damos un resumen de algunos trabajos relacionados con el nuestro, además de los trabajos que ya hemos tratado durante los capítulos previos.

9.1 Derivaciones del algoritmo BM

Entre las derivaciones de de la parte de búsqueda del algoritmo BM destaca la de Partsch y Stomp [Par90], ya que dan una derivación formal para un patrón arbitrario. La desventaja de su derivación es que, por un lado, ellos no tienen un modelo de ejecución definido, y por otro lado, la estrategia utilizada en su derivación requiere la introducción de varias etapas de tipo eureka. El lenguaje que utilizan en su derivación es un "lenguaje de amplio espectro", que está dotado de no-determinismo (en las etapas de la derivación) y de una lógica no necesariamente limitada a cláusulas. Con tales herramientas, deducen una fórmula similar a la de Boyer y Moore para $máx(\delta_1(t_i), \delta_2(j))$ (Fig. 4.11), en donde δ_1 y δ_2 están definidas por las expresiones que nosotros dimos en el capítulo 4.

Después de algunas manipulaciones algebraicas (similares a las que hacemos para eliminar *append*), su especificación (con una notación ligeramente diferente de la original) es:

$$s(k) = eq(k) \vee s(k+1)$$

en donde $eq(k)$ es verdadero si el patrón ocurre en la posición (de comienzo) k del texto, y $\vee$ es un "O secuencial", el cual evalúa su operando a la derecha sólo si su operando a la izquierda es falso (i.e. únicamente la primera ocurrencia del patrón es deseada). El uso que hacen Partsch y Stomp del no-determinismo es ilustrado por medio de la siguiente

función:

$$delta(k) = i \text{ si } \exists i\,(1 \leq i \leq m \;\&\; \neg eq(k+i))$$

(El no-determinismo, aquí, permite posponer la decisión de cuánto desplazar el patrón a la derecha si existe un mal-casamiento, hasta que tengamos más información disponible durante el desarrollo de la derivación.)

Pepper [Pep91] también da una derivación formal de la parte de búsqueda del algoritmo BM para una cadena arbitraria, usando técnicas similares a las de Partsch y Stomp. Su derivación, sin embargo, resulta más limpia que la de [Par90], ya que él evita el uso excesivo de índices al nombrar adecuadamente las subcadenas en cuestión.

El trabajo de Partsch y Völker [PV91] es particularmente relevante para nosotros, así como el de Darlington [Dar78], ya que estos autores tienen como interés *relacionar* diversos tipos de algoritmos que resuelven un problema dado. En efecto, Partsch y Völker reusan varios desarrollos de [Par90] para obtener el proceso de búsqueda del algoritmo KMP. Como en [Par90], ellos llegan a una expresión similar a la de δ_2 del algoritmo BM, pero que corresponde a la expresión de *next*. Darlington [Dar78], por otro lado, describe una taxonomía para algunos algoritmos (los más relevantes en la literatura) que resuelven el problema del ordenamiento, en un trabajo que intenta deducir de manera lógica cada algoritmo de acuerdo con las definiciones de alto nivel que resuelven directamente (utilizando permutaciones) el problema del ordenamiento.

Otro trabajo que también deriva la parte de búsqueda de los algoritmos KMP y una variante de BM que sólo involucra la función δ_1 es el de Amtoft, Consel, Danvy, y Malmkjær [ACDM01]. Estos investigadores emplean un algoritmo de fuerza bruta (especificación) aumentado con un "cache", que mantiene un registro de lo que es conocido acerca del texto, y que es similar a la memoria cache de algunos procesadores de hardware. El cache es accedido en preferencia al texto, de modo que el texto es accedido sólo si la información necesaria no está ya presente en el cache, y en tal caso, el cache es actualizado. El cache almacena tanto información positiva (casamientos parciales), como información negativa (casos de mal-casamiento o discordancias). Dependiendo si el patrón es recorrido de izquierda a derecha, o de derecha a izquierda, un evaluador parcial deriva, usando el cache, la parte de búsqueda del algoritmo KMP o de una de las variantes del algoritmo BM (que sólo involucra la función δ_1).

Las investigaciones previas de Hogger [Hog79, Hog81] nos son también interesantes, ya que Hogger utiliza la programación lógica para intentar derivar algunos algoritmos de casamiento de cadenas. Comenzando desde una especificación directa de problema del casamiento de cadenas, primero deriva una especificación alternativa, la cual es utilizada, por ejemplo, en [Gal93]. Entonces, Hogger deriva algunos predicados de alto nivel corres-

pondientes a los algoritmo KMP y BM pero no deriva las tablas de desplazamientos que estos métodos utilizan. Por ejemplo, el algoritmo BM que deriva es:

$$
\begin{aligned}
s(x,y) &\leftarrow \mathit{length}(x,n),\ s'(x,y,n,1) \\
s'(x,y,i,k) &\leftarrow i < 1 \\
s'(x,y,i,k) &\leftarrow m(u,i,x),\ m(u,i+k-1,y), \\
&\qquad s'(x,y,i-1,k) \\
s'(x,y,i,k) &\leftarrow m(u,i,x),\ m(v,i+k-1,y),\ u \neq v, \\
&\qquad \mathit{length}(x,n),\ \mathit{displace}(x,i,v,j), \\
&\qquad s'(x,y,n,k+j)
\end{aligned}
$$

en donde $m(u,i,x)$ expresa que el símbolo u está en la posición i en la cadena x. No deriva, sin embargo, el predicado *displace*. El trabajo de Hogger, no obstante, es importante en el sentido de que brinda los primeros pasos para obtener, dentro de la programación lógica, las derivaciones formales de algunos algoritmos de casamiento de cadenas para un patrón arbitrario.

Una derivación de la variante del algoritmo de Boyer y Moore que sólo utiliza la función δ_2 es la de [FKG02], en donde los autores derivan también el algoritmo KMP. Su técnica de derivación utiliza evaluación parcial y la noción de "Generalized Partial Computation" (GPC). Su derivación tiene que construirse sobre la base de que existe una función que realiza casamientos parciales en un casador de patrones de fuerza bruta. En este artículo también hay una demostración de la corrección del programa residual (en el sentido de que en efecto implementa los avances propios de la función δ_2). Creemos, sin embargo, que nosotros escribimos la función que anticipa casamientos parciales por medio de nuestros programas de forma triangular, y en cuanto a la demostración, está reflejada esencialmente en nuestra misma derivación.

Nuestra derivación del estado de preprocesamiento del algoritmo KMP es quizás más cercana a la de Pettorossi, Proietti y Renault [PPR97a]. Ambas derivaciones son similares con respecto a la otra en el sentido de que involucran la utilización de, esencialmente, las mismas operaciones. Una diferencia radica, no obstante, en el *orden* en el cual las operaciones se aplican. Similarmente a nuestra derivación, la de [PPR97a] utiliza división en casos, desdoblados determinísticos y no-determinísticos, así como la introducción de definiciones y el doblado de varias cláusulas. Sin embargo, en lugar de aplicar la misma regla varias veces, sin aplicar ninguna otra regla a cada paso, la derivación de [PPR97a] está compuesta de *ciclos*, cada uno de los cuales aplica todas las reglas. Cada ciclo comienza con un desdoblado no-determinístico, seguido por repetidas aplicaciones del desdoblado determinístico tantas veces como sea posible. A continuación, la derivación aplica la regla

de división en casos, y define un nuevo predicado que es utilizado para doblar. El proceso completo termina cuando todas las cláusulas tienen distintas cabezas.

9.2 Derivaciones del algoritmo KMP

Algunas derivaciones del algoritmo KMP están en el espíritu del refinamiento de expresiones [Mor90] o en el diseño de invariantes de ciclos [vdW87] y por lo tanto caen fuera de nuestro enfoque transformacional de la derivación de programas. En cambio, una de las primeras derivaciones del algoritmo KMP que provocó el interés de la comunidad de la transformación de programas fue la de Consel y Danvy [CD89]. En esta derivación, los autores desarrollan sus programas utilizando el lenguaje de programación funcional Scheme. Una de las objeciones que podría tener esta derivación del algoritmo KMP es el programa inicial. El programa inicial tiene dos copias del patrón y ambas tienen que actuar entre sí conforme se da el proceso de casamiento, lo que equivale aproximadamente a una simulación de no-determinismo o a la regla de retroceso en Prolog. La parte difícil, entonces, es saber qué agregar a un programa de fuerza bruta para que "la situación sea ideal para un evaluador parcial" [CD89, página 82].

Siguiendo las ideas de transformación de programas dentro de la programación funcional, otras derivaciones importantes son las de [GK93], y de [SGJ96]. En la frontera entre transformación de programas y la demostración automática de teoremas, encontramos también la derivación del algoritmo KMP dada en [FN88].

En programación lógica, la derivación dada en [Kur88] el programa inicial carece de declaratividad, pues ya involucra un acumulador, en tanto que las derivaciones de Gallagher [Gal93] y de Smith [Smi91] adolecen del defecto de comenzar con algoritmos de fuerza bruta determinísticos, y además inicialmente introducen ecuaciones e inecuaciones de manera injustificada. Sin embargo, Smith intuye, creemos que apropiadamente, la estrecha relación del algoritmo KMP y la satisfacción de restricciones, además de generalizar el problema de casamiento de cadenas a otros un poco más complicados, como los de casamiento en dominios finitos, casamientos de listas de términos, y casamiento múltiple de patrones, además de intuir también la utilidad en la programación de lógica de considerar las fallas como una fuente de información en la optimización de programas, vía el método de continuaciones. Por nuestra parte, ya hemos visto que nosotros aprovechamos las fallas vía la introducción explícita de ecuaciones e inecuaciones y casamientos parciales.

Capítulo 10

Conclusiones

Hemos resumido algunas conclusiones de nuestro trabajo al final de cada capítulo, tratando de reflejar de manera inmediata el material involucrado. Como complemento, en este capítulo damos las conclusiones generales que derivamos de esta tesis.

10.1 Aportaciones

Pensamos que las aportaciones que esta tesis provee a la comunidad científica de la transformación de programas son las siguientes:

10.1.1 Programación declarativa

En la filosofía de la programación por transformación existe una búsqueda constante para hallar metodologías que sean declarativas en el nivel de las especificaciones, mientras que se busca la identificación del nexo que existe entre las derivaciones de distintos tipos de algoritmos. Nuestra aportación a la programación declarativa radica en la valoración que hemos dado al no-determinismo de los programas lógicos, lo que contribuye en cierta medida al fortalecimiento del paradigma de la programación lógica.

10.1.2 Reuso de desarrollos

La programación declarativa contribuye sustancialmente a la escritura de programas correctos, pero en el nivel de la especificación el programa lógico correspondiente tal vez no es eficientemente ejecutable. La transformación de programas contribuye en la progra-

mación en general al tratar de mantener la declaratividad de la especificación intentando manipular los programas que derivamos de esta especificación. Sin embargo, la invención de métodos particulares a la derivación de cierto algoritmo demerita el uso práctico de la transformación de programas; entonces, en lugar intentar adaptar en cada ocasión una sucesión de aplicaciones de reglas, lo que se busca son sucesiones generales que tengan como característica esencialmente la misma derivación para distintos algoritmos, aún cuando estos algoritmos no necesariamente se apeguen a los algoritmos ideados por los humanos. En nuestra opinión, nuestra contribución en esta dirección es doble: Primero, nosotros proveemos una derivación de la parte de búsqueda de algunas variantes del algoritmo de Boyer y Moore [BM77], lo que representa una contribución a los aspectos generales de la derivación de programas. Segundo, mostramos que podemos reutilizar algunos desarrollos de tal derivación para derivar la parte de búsqueda del algoritmo de casamiento de cadenas de Knuth, Morris y Pratt [KMP77], y esta reutilización nos permitió analizar en cierto detalle la generalidad de los métodos con los que derivamos las variantes del algoritmo de Boyer y Moore. Un paso más allá sería ocultar completamente al usuario toda la secuencia transformacional, y hacer que los sistemas computacionales tomaran las decisiones apropiadas de acuerdo con algún criterio, por ejemplo el ahorro de espacio en memoria o la velocidad de ejecución. En este sentido, al derivar la parte de búsqueda de algunas variantes del algoritmo BM, hemos mostrado la viabilidad de la transformación de programas lógicos en la práctica, y las derivaciones involucradas representan un paso más dentro de la corriente de derivación de algoritmos.

10.1.3 Contribuciones a la transformación de programas lógicos

En el aspecto manual, los resultados de los capítulos 5 y 6 fueron presentados en el congreso *Principles and Practice of Declarative Programming'01*, y fueron publicados en las memorias de este congreso [HR01]. Nuestras aportaciones, en el aspecto general de transformación de programas, son:

1. la derivación de la parte de búsqueda de algunas variantes del algoritmo BM;
2. un nuevo enfoque para la derivación de la parte de búsqueda del algoritmo KMP, contribuyendo con ello a nueva derivación de este algoritmo;
3. la adaptación de la idea de *reuso de desarrollos* al caso de la derivación de dos importantes algoritmos de casamiento de cadenas; y
4. la utilización del no-determinismo para realzar la declaratividad de una especificación, así como un método que permite la eliminación de este no-determinismo en lo relativo a la implementación.

10.1.4 Contribuciones a la deducción parcial de programas lógicos

Algunos de los resultados en nuestros intento de mecanizar una parte o todo el proceso de los algoritmos de cadenas que tratamos están en [HR03], que está inspirado en la idea de la derivación automática de algoritmos.

Técnicamente, además de haber derivado una variante del algoritmo de Boyer y Moore, importante en la práctica, creemos que tal derivación tiene la atractiva propiedad que sólo utilizamos desdoblados *determinísticos* [Gal93] (distintos de aquellos desdoblados con respecto a $\in$ y $\notin$, predicados que podríamos reemplazar por aplicaciones exhaustivas de la regla de división en casos), proveyéndonos de un criterio simple para detener el proceso de desdoblado. Además, hemos basado la detención de los procesos repetitivos de aplicaciones de desdoblados determinísticos en la noción de que podemos anticipar un paso en la computación sin incrementar excesivamente el tamaño de los programas intermedios.

Se considera frecuentemente que el método de Boyer y Moore es un caso representativo de aquellos algoritmos en donde la sencillez se sacrifica en favor de la eficiencia [Par90], y es por lo que hallamos sorprendente que nuestra derivación de la parte de búsqueda de una variante de tal algoritmo resultara tan natural y directa. Después de que agregamos algunas restricciones en la forma de ecuaciones e inecuaciones, obtenemos una versión no-determinística del algoritmo deseado al aplicar derivaciones determinísticas. Posteriormente, eliminamos tal no-determinismo al introducir nuevos predicados y realizar doblados al programa actual.

Realizamos la introducción de restricciones en una primera etapa al aplicar la regla de división en casos, lo que establece un paso preparatorio para el cálculo de las tablas que indican cuántos símbolos desplazar el patrón en el caso de un mal-casamiento o discordancia (funciones δ_1 y δ_2). Posteriormente, relacionamos las derivaciones determinísticas con el cálculo efectivo de tales tablas.

Ahora bien, aplicamos esencialmente la misma técnica para derivar la parte de búsqueda del algoritmo de Knuth, Morris y Pratt. Aunque las restricciones cambian, las derivaciones determinísticas (con las cuales calculamos la tabla llamada *next*), y el proceso de eliminación del no-determinismo, están presentes. Después de tales dos procesos que son comunes a ambos algoritmos, sin embargo, tenemos que realizar un desdoblado adicional que resulta del hecho de que el algoritmo de Knuth, Morris y Pratt mantiene un registro (memoria) de las comparaciones ya realizadas (puesto que el algoritmo de Boyer y Moore no registra ni almacena tales comparaciones [BYCG94]). Aún así, creemos que la identificación de las partes comunes en las derivaciones involucradas contribuye a la mecanización de las mismas.

10.2 Actualidad de nuestro trabajo

Pensamos que la derivación de una variante del algoritmo de Boyer y Moore es interesante por sí misma. Aunque el método de Knuth, Morris y Pratt ha sido estudiado más completamente desde el punto de vista de la derivación de programas, el método Boyer y Moore es más importante en la práctica [HS91, Ste94], pero esta importancia sólo recientemente se está reflejando en la comunidad de transformación de programas. La actualidad de [ACDM01, FKG02, ADR02] en el tema de la derivación de algoritmos de casamiento de cadenas avala esta afirmación, además de los artículos de [PS90] (que fue incluido como un ejemplo extendido en el libro [Par90]) y [Pep91] sobre el tema que se publicaron en la década de los noventa.

10.3 Trabajo a futuro

En lo concerniente al trabajo a futuro, a continuación presentamos algunas de las posibilidades que nuestro estado actual de investigación presenta como más prometedoras:

1. El estudio de la relación existente entre el algoritmo de unificación de Martelli y Montanari y la solución del conjunto de ecuaciones en cada desdoblado determinístico.

2. El estudio detallado de la relación existente entre la introducción de ecuaciones y la programación lógica con restricciones.

3. La investigación de cómo adaptar nuestros programas a la búsqueda de todas las ocurrencias de un patrón sobre un texto.

4. El estudio de la aplicación de nuestras propuestas al problema (en cierto sentido, dual) de, en esta ocasión, considerar al texto como dado (o fijo) y al patrón como desconocido (es decir, la evaluación parcial de algoritmo de casamiento de cadenas de fuerza bruta esta vez *con respecto al texto*), lo que conduciría a la derivación de algoritmos que permitirían un rápido reconocimiento de un patrón en un texto previamente dado (tal problema es solucionado por medio de los algoritmos de McCreight, Weiner, y Ukkonen [CR94]).

5. El estudio del caso del casamiento múltiple de patrones.

6. La investigación del caso del casamiento aproximado de patrones, lo que nos conduce naturalmente al área de programación dinámica.

7. El estudio del diseño de una estrategia automática que permita derivar una variante del algoritmo BM limitada, esta vez, a la función δ_1 [ACDM01].

8. El estudio de la aplicación de nuestras técnicas a problemas distintos de los de casamiento de cadenas (vemos algunas posibilidades en el área de la programación lógica con restricciones).

9. El estudio de los alcances, tanto teóricos como prácticos de la transformación de programas, junto con el establecimiento de algunas métricas que permitan conocer en qué sentido y en qué medida un programa es la versión mejorada de otro.

10. Otra vertiente de investigación muy interesante sería el diseño de un esquema teórico donde ubicar la transformación de programas; actualmente, el esquema básico varía de paradigma a paradigma, a pesar de que muchos resultados son genéricos y aplicables en distintos contextos. La teoría de categorías, una área de relativamente nueva creación en matemáticas, es una buena candidata para proveernos de tal esquema, pero otros esquemas de menor generalidad, tales como las lógicas modales, podrían ser un buen inicio [Pep87a].

Bibliografía

[ACDM01] T. Amtoft, C. Consel, O. Danvy, y K. Malmkjær. The abstraction and instantiation of string-matching programs. BRICS RS-01-12, DAIMI, Department of Computer Science, University of Aarhus, Aarhus, Dinamarca, 2001. Reporte técnico, `http://www.brics.dk/RS/01/12/`.

[ADR02] Mads Sig Ager, Olivier Danvy, y Henning Korsholm Rohde. On obtaining Knuth, Morris, and Pratt's string matcher by partial evaluation. En *Proceedings of ASIA-PEPM'02*, pp. 32–46, Aizu, Japón, septiembre de 2002. ACM.

[AG86] Alberto Apostolico y Raffaele Giancarlo. The Boyer–Moore–Galil string searching strategies revisited. *SIAM J. Comput.*, 15(1):98–105, febrero de 1986.

[Aho90] Alfred V. Aho. Algorithms for finding pattern in strings. En J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volumen B, pp. 256–300. Elsevier Science Publishers B. V., 1990.

[Apt97] Krzysztof R. Apt. *From Logic Programming to Prolog.* Prentice Hall, 1997.

[Aug98] Lex Augusteijn. Sorting morphisms. En S. Doaitse Swierstra, Pedro R. Henriques, y José N. Oliveira, editores, *Advanced Functional Programming. Third International School, AFP'98. Revised Lectures*, volumen 1608 de *Lecture Notes in Computer Science*, pp. 1–27. Springer-Verlag, Braga, Portugal, septiembre de 1998.

[Bac78] John Backus. Can programming be liberated from the von Newmann style? A functional style and its algebra of programs. *Communications of the Association for Computing Machinery*, 8(21):613–641, agosto de 1978.

[Bau79a] F.L. Bauer. From specification to implementation —the formal approach. En F.L. Bauer y M. Broy, editores, *Program Construction*, volumen 69 de *Lecture Notes in Computer Science*, capítulo III. Program Development by Transformation, pp. 235–236. Springer-Verlag, 1979.

[Bau79b] F.L. Bauer. Program development by stepwise transformations. the Project CIP. En F.L. Bauer y M. Broy, editores, *Program Construction*, volumen 69 de *Lecture Notes in Computer Science*, capítulo III. Program Development by Transformation, pp. 237–266. Springer-Verlag, 1979.

[BB79] F.L. Bauer y M. Broy, editores. *Program construction*. Springer-Verlag, 1979. Preface.

[BD77] Rod M. Burstall y John Darlington. A transformation system for developing recursive programs. *Journal of the Association for Computing Machinery*, 24(1):44–67, enero de 1977.

[BdM97] Richar Bird y Oege de Moor. *Algebra of Programming*. Series in Computer Science. Prentice Hall, 1997.

[BJJM98] Roland Backhouse, Patrick Jansson, Johan Jeuring, y Lambert Meertens. Generic programming — an introduction —. En S. Doaitse Swierstra, Pedro R. Henriques, y José N. Oliveira, editores, *Advanced Functional Programming. Third International School, AFP'98*, volumen 1608 de *Lecture Notes in Computer Science*, pp. 28–115. Springer-Verlag, Braga, Portugal, revised lectures edición, septiembre de 1998.

[BM77] Robert S. Boyer y J. Strother Moore. A fast string searching algorithm. *Communications of the Association for Computing Machinery*, 20(10), octubre de 1977.

[Bsa92] Khaled Bsaïes. A strategy for transforming generate and test logic programs. En *Symposium on Applied Computing: Proceedings of the 1992 ACM/SIGAPP Symposium on Applied computing: technological challenges of the 1990's*, pp. 563–572, Kansas, Missouri, United States, 1992.. ACM Press.

[BV87] C. Beierle y A. Voss. Viewing implementations as an institution. En David Pitt, Axel Poigné, y David Rydeheard, editores, *Category Theory and Computer Programming: Proceedings*, volumen 283 de *Lecture Notes in Computer Science*, pp. 196–218. Springer-Verlag, Guilford, Reino Unido, septiembre de 1987.

[BW88] Richard Bird y Philip Wadler. *Introduction to Functional Programming*. Series in Computer Science. Prentice Hall, 1988.

[BYCG94] R.A. Baeza-Yates, C. Choffrut, y G.H. Gonnet. On Boyer–Moore automata. *Algorithmica*, (12):268–292, 1994.

[CC92] Patrick Cousot y Radhia Cousot. Abstract interpretation and application to logic programs. *The Journal of Logic Programming*, 13:103–179, 1992.

[CD80] K.L. Clark y J. Darlington. Algorithm classification through synthesis. *The Computer Journal*, 23(1):61–65, 1980.

[CD89] Charles Consel y Olivier Danvy. Partial evaluation of pattern matching in strings. *Information Processing Letters*, (30):79–86, 1989.

[Cla78] Keith L. Clark. Negation as failure. En H. Gallaire y J. Minker, editores, *Logic and Databases*, pp. 293–322. Plenum Press, 1978.

[CP89] Jiazhen Cai y Robert Paige. Program derivation by fixed point computation. *Science of Computer Programming*, 11(3):197–261, abril de 1989.

[CR94] M. Crochemore y W. Rytter. *Text Algorithms*. Oxford University Press, 1994.

[Dar78] John Darlington. A synthesis of several sorting algorithms. *Acta Informatica*, 11:1–30, 1978.

[DB76] John Darlington y Rod M. Burstall. A system which automatically improves programs. *Acta Informatica*, 6:177–197, 1976.

[Dev90] Yves Deville. *Logic Programming. Systematic Program Development*. Addison-Wesley, 1990.

[Dij72] Edsger W. Dijkstra. *Structured Programming*, volumen 8 de *A.P.I.C. Studies in Data Processing*, capítulo Notes on Structured Programming. Academic Press, 1972.

[Dij75] E.W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM*, (18):453–457, agosto de 1975.

[Dij76] Edsger W. Dijkstra. *A Discipline of Programming*. Automatic Computation. Prentice-Hall, 1976.

[Dij79a] Edsger W. Dijkstra. Interplay Between Invention and Formal Techniques — The thinking programmer (summary). En F.L. Bauer y M. Broy, editores, *Program construction*, volumen 69 de *Lecture Notes in Computer Science*, capítulo I. The Thinking Programmer, p. 1. Springer-Verlag, 1979.

[Dij79b] Edsger W. Dijkstra. On the interplay between mathematics and programming. En F.L. Bauer y M. Broy, editores, *Program construction*, volumen 69 de *Lecture Notes in Computer Science*, capítulo I. The Thinking Programmer, pp. 35–46. Springer-Verlag, 1979.

[dMS98] Oege de Moor y Ganesh Sittampalam. Generic program transformation. En S. Doaitse Swierstra, Pedro R. Henriques, y José N. Oliveira, editores, *Advanced Functional Programming. Third International School, AFP'98. Revised Lectures*, volumen 1608 de *Lecture Notes in Computer Science*, pp. 116–149. Springer-Verlag, Braga, Portugal, septiembre de 1998.

[Doe94] Kees Doets. *From Logic to Logic Programming.* Foundation of Computing Series. The MIT Press, 1994.

[DW98] Saumya K. Debray y David S. Warren. Automatic mode inference for logic programs. *The Journal of Logic Programming*, 5(3):207–229, septiembre de 1998.

[Dyb86] P. Dybjer. Category theory and programming language semantics: an overview. En David Pitt, Samson Abramsky, Axel Poigné, y David Rydeheard, editores, *Category Theory and Computer Programming: Proceedings*, volumen 240 de *Lecture Notes in Computer Science*, pp. 165–181. Springer-Verlag, Guilford, Reino Unido, septiembre de 1986.

[Fea87] Martin S. Feather. A Survey and Classification of some Program Transformation Approaches and Techniques. En L.G.L.T. Meertens, editor, *Program specification and transformation*, IFIP, pp. 165–195. Elsevier Science Publishers B.V. (North–Holland), 1987.

[FKG02] Yoshihiko Futamura, Zenjiro Konishi, y Robert Glück. Automatic generation of efficient string matching algorithms by generalized partial computation. En *Proceedings of ASIA-PEPM'02*, Aizu, Japón, septiembre de 2002. ACM.

[Flo67] Robert W. Floyd. Nondeterministic algorithms. *Journal of the Association for Computing Machinery*, 14(4):636–644, octubre de 1967.

[FN88] Yoshihiko Futamura y Kenroku Nogi. Generalize partial computation. En D. Bjørner, A.P. Ershov, y N.D. Jones, editores, *Partial Evaluation and Mixed Computation*, pp. 133–151. Elsevier Science Publishers B.V. (North-Holland), 1988.

[Gal93] J.P. Gallagher. Tutorial on specialisation of logic programs. En *PEPM'93, ACM*, pp. 88–98, Copenhague, Dinamarca, junio de 1993.

[GB65] Solomon W. Golomb y Leonard D. Baumert. Bactrack programming. *Journal of the Association for Computing Machinery*, 12(4):516–524, octubre de 1965.

[GB86] J.A. Goguen y R.M. Burstall. A study in the foundations of programming methodology: Specifications, institutions, charters and parchments. En David Pitt, Samson Abramsky, Axel Poigné, y David Rydeheard, editores, *Category Theory and Computer Programming: Proceedings*, volumen 240 de *Lecture Notes in Computer Science*, pp. 313–332. Springer-Verlag, Guilford, Reino Unido, septiembre de 1986.

[GB91] John Gallagher y Maurice Bruynooghe. The derivation of an algorithm for program specialisation. *New Generation Computing*, 9(3/4):305–303, 1991.

[GK93] Robert Glück y Andrei V. Klimov. Occam's razor in metacomputation: the notion of a perfect process tree. En Patrick Cousot et al., editores, *Static Analysis*, volumen 724 de *Lecture Notes in Computer Science*, pp. 112–123, Padua, Italia, septiembre de 1993. Third International Workshop, WSA '93.

[GK94] Manolis Gergatsoulis y Maria Katzouraki. Unfold/fold transformations for definite clause programs. En Manuel Hermenegildo y Jaan Penjam, editores, *Programming Language Implementation and Logic Programming*, volumen 844 de *Lecture Notes in Computer Science*. 6th International Symposium, PLILP'94, Springer-Verlag, septiembre de 1994.

[Gri78] David Gries. On structured programming. En David Gries, editor, *Programming methodologies. A Collection of Articles by Member of IFIP WG2.3*, pp. 70–74. Springer-Verlag, 1978.

[Gri79] D. Gries. Current ideas in programming methodology. En F.L. Bauer y M. Broy, editores, *Program Construction*, volumen 69 de *Lecture Notes in Computer Science*, capítulo II. Program Verification, pp. 77–93. Springer-Verlag, 1979.

[GS91] P.A. Gardner y J.C. Shepherdson. Unfold/fold transformations of logic programs. En Lassez y Plotkin, editores, *Computational Logic*, capítulo 17. MIT Press, 1991.

[GS95] Robert Glück y Morten H. Sørensen. An algorithm of generalization in positive supercompilation. En John Lloyd, editor, *Logic Programming*, pp. 465–478. Proceedings of the 1995 International Symposium, The MIT Press, 1995.

[Gus99] Dan Gusfield. *Algorithms on Strings, Trees, and Sequences.* Cambridge University Press, 1999.

[Gut86] John Guttag. Notes on type abstraction. En N. Gehani y A.D. McGettrick, editores, *Software Specification Techniques*, International Computer Science Series, pp. 55–74. Adison–Wesley, 1986.

[Hoa78] C.A.R. Hoare. The engineering of software: a startling contradiction. En David Gries, editor, *Programming methodologies. A Collection of Articles by Member of IFIP WG2.3*, pp. 29–36. Springer-Verlag, 1978.

[Hog79] C.J. Hogger. Logical analysis of some string-matching algorithms. Reporte técnico, Imperial College of Science and Technology, Londres, Reino Unido, octubre de 1979.

[Hog81] C.J. Hogger. Derivation of logic programs. *Journal of the Association for Computing Machinery*, 28(2):372–392, abril de 1981.

[Hor80] Nigel R. Horspool. Practical fast searching in strings. *Software—Practice and experience*, 10(6):501–506, 1980.

[HR01] Manuel Hernández y David A. Rosenblueth. Development reuse and the logic program derivation of two string-matching algorithms. En *Proceedings of the 3rd International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming (PPDP'01)*, pp. 38–48, Florencia, Italia, septiembre de 2001.

[HR03] Manuel Hernández y David A. Rosenblueth. A disjunctive partial deduction of a right-to-left string-matching algorithm. *Information Processing Letters*, 87(5):235–241, septiembre de 2003.

[HS91] A. Hume y D. Sunday. Fast string searching. *Software —Practice and Experience*, 21(11):1221–1248, 1991.

[HSG98] Morten Heine, B. Sørensen, y Robert Glück. Introduction to supercompilation. En John Hatcliff, Torben Æ. Mogensen, y Petern Thiemann, editores, *Partial Evaluation: Practice and Theory*, volumen 1706 de *Lectures Notes of Computer Science*, pp. 247–270. Springer-Verlag, Copenhague, Dinamarca, junio–julio de 1998. DIKU 1998 International Summer School.

[Jon96] Neil D. Jones. An introduction to partial evaluation. *ACM Computing Surveys*, 28(3):480–503, septiembre de 1996.

[Kle52] Stephen C. Kleene. *Introduction to Metamathematics*. D. Van Nostrand Co., 1952.

[KLP87] S. Kasangian, A. Labella, y A. Pettorossi. Enriched categories for local and interaction calculi. En David Pitt, Axel Poigné, y David Rydeheard, editores, *Category Theory and Computer Programming: Proceedings*, volumen 283 de *Lecture Notes in Computer Science*, pp. 56–70. Springer-Verlag, Guilford, Reino Unido, septiembre de 1987.

[KMP77] Donald E. Knuth, James H. Morris, y Vaughan R. Pratt. Fast pattern matching in strings. *SIAM Journal of Computation*, 6(2):323–350, junio de 1977.

[Kom92] Jan Komorowski. An introduction to partial deduction. En A. Pettorossi, editor, *Meta-Programming in Logic*, volumen 649 de *Lecture Notes in Computer Science*, pp. 49–69, Upsala, Suecia, junio de 1992. Third International Workshop, META-92, Springer-Verlag.

[Kot85] Laurent Kott. Unfold/fold program transformations. En Maurice Nivat y John C. Reynolds, editores, *Algebraic methods in semantics*. Cambridge University Press, 1985.

[Kow79a] Robert Kowalski. Algorithm = logic + control. *Communications of the Association for Computing Machinery*, 22(7):424–436, julio de 1979.

[Kow79b] Robert Kowalski. *Logic for Problem-Solving*. North-Holland, 1979.

[Kur88] Peter Kursawe. Pure partial evaluation and instantiation. En D. Bjørner, A.P. Ershov, y N.D. Jones, editores, *Partial Evaluation and Mixed Computation*, pp. 283–298. Elsevier Science Publishers B.V. (North-Holland), 1988.

[Leu98a] Michael Leuschel. Advanced logic program specialisation. En *Partial Evaluation. Practice and Theory*, volumen 1706, pp. 271–292. Springer-Verlag, 1998.

[Leu98b] Michael Leuschel. Logic program specialisation. En John Hatcliff, Torben Æ. Mogensen, y Peter Thiemann, editores, *Partial Evaluation: Practice and Theory*, volumen 1706 de *Lecture Notes in Computer Science*, pp. 155–188. Springer-Verlag, Copenhague, Dinamarca, junio–julio de 1998. DIKU 1998 International Summer School.

[Lis94] Björn Lisper. Total unfolding: theory and aplications. *Journal of Functional Programming*, 4(4):479–498, octubre de 1994.

[Llo87] J.W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, segunda edición, 1987.

[LP86] A. Labella y A. Pettorossi. Categorical models of process cooperation. En David Pitt, Samson Abramsky, Axel Poigné, y David Rydeheard, editores, *Category Theory and Computer Programming: Proceedings*, volumen 240 de *Lecture Notes in Computer Science*, pp. 266–281. Springer-Verlag, Guilford, Reino Unido, septiembre de 1986.

[LS91] J.W. Lloyd y J.C. Shepherdson. Partial evaluation in logic programming. *The Journal of Logic Programming*, 11(3 & 4):217–242, octubre-noviembre de 1991.

[Mah88] Michael J. Maher. Equivalences of logic programs. En Jack Minker, editor, *Foundations of Deductive Databases and Logic Programming*, capítulo 16, pp. 627–658. 1988.

[McC63a] J. McCarthy. A basis for a mathematical theory of computation. En P. Braffort y D. Hirschberg, editores, *Computer Programming and Formal Systems*, pp. 33–70. North-Holland Publishing Company, 1963.

[McC63b] John McCarthy. Towards a mathematical science of computation. En C. M. Popplewell, editor, *Information Processing: Proceedings of IFIP 1962*, pp. 21–28. North-Holland Publishing Company, 1963.

[MDM94] Ali Mili, Jules Desharnais, y Fatma Mili. *Computer Program Construction*. Oxford University Press, 1994.

[Mee89] Lambert Meertens. Constructing a calculus of programs. En *Mathematics of Program Construction. Proceedings of the 375th Anniversary of the Groningen University International Conference*, volumen 375 de *Lecture Notes in Computer Science*, pp. 66–90. Groningen, Holanda, junio de 1989.

[MH88] Torben Æ. Mogensen y Carsten Kehler Holst. Terminology. En D. Bjørner, A.P. Ershov, y N.D. Jones, editores, *Partial Evaluation and Mixed Computation*, pp. 383–588. Elsevier Science Publishers B.V. (North-Holland), 1988.

[MNL90] K. Marriot, L. Naish, y J.-L. Lassez. Most specific logic programs. En *Annals of Mathematics and Artificial Intelligence*. 1990.

[Möl93a] Bernhard Möller. Algebraic calculation of graph and sorting algorithms. En B. Bjørner, M. Broy, y I.V. Pottosin, editores, *Formal Methods in Programming and their Applications*, volumen 735 de *Lecture Notes in Computer Science*, pp. 394–413, 1993.

[Möl93b] Bernhard Möller. Derivation of graph and pointer algorithms. En Bernhard Möller, Helmut Partsch, y Steve Schuman, editores, *Formal Program Development*, volumen 755. Springer-Verlag, 1993.

[Mor90] Joseph M. Morris. Programming by expression refinement: the KMP algorithm. En W.H.J. Feijen et al., editor, *Beauty Is Our Business: A Birthday Salute to Edsger W. Dijkstra*. Springer-Verlag, 1990.

[Pai96] R. Paige. Future directions in program transformations. *ACM Computing Surveys*, (28(4es)), 1996. `http://www.acm.org/pubs/citations/journals/surveys/1996-28-4es/a170-paige/`.

[Par78] David L. Parnas. On the design and development of program families. En David Gries, editor, *Programming methodologies. A Collection of Articles by Member of IFIP WG2.3*, pp. 342–361. Springer-Verlag, 1978.

[Par86] H. Partsch. Intervención de Partsch durante una sesión de discusión en la IFIP TC2/WG 2.1 Working Conference on Program Specification and Transformation. En *Program Specification and Transformation*, IFIP. Editado por L.G.L.T Meertens, 1987. Ed. North-Holland, 15–17 abril de 1986. Bad Tölz, FRG.

[Par90] Helmut Partsch. *Specification and Transformation of Programs*. Texts and Monographs in Computer Science. Springer-Verlag, 1990.

[Pep87a] Peter Pepper. Application of modal logics to the reasoning about applicative programs. En L.G.L.T. Meertens, editor, *Program specification and transformation*, IFIP, pp. 429–450. Elsevier Science Publishers B.V. (North–Holland), 1987.

[Pep87b] Peter Pepper. A simple calculus for program transformation (inclusive of induction). *Science of Computer Programming*, (9):221–262, 1987. North-Holland.

[Pep91] Peter Pepper. Literate program derivation: A case study. En M. Broy y M. Wirsing, editores, *Methods of Programming*, volumen 544 de *Lecture Notes in Computer Science*, pp. 101–124. Springer-Verlag, 1991.

[Pep93] Peter Pepper. Program development in an algebraic setting. En B. Möller, H. Partsch, y S. Schuman, editores, *Formal Program Development*, número 755 en Lecture Notes in Computer Science, pp. 225–262. Springer-Verlag, 1993.

[Per90] D. Perrin. Handbook of theoretical computer science. volumen B, capítulo 1.Finite automata, pp. 1–57. Elsevier Science Publishers B.V., 1990.

[Poi86] A. Poigné. Category theory and logic. En David Pitt, Samson Abramsky, Axel Poigné, y David Rydeheard, editores, *Category Theory and Computer Programming: Proceedings*, volumen 240 de *Lecture Notes in Computer Science*, pp. 103–142. Springer-Verlag, Guilford, Reino Unido, septiembre de 1986.

[PP90] Alberto Pettorossi y Maurizio Proietti. Synthesis of eureka predicates for developing logic programs. En N. Jones, editor, *ESOP'90: 3rd European Symposium on Programming (Proceedings)*, volumen 432 de *Lecture Notes in Computer Science*, pp. 306–325. Springer-Verlag, 15–18 mayo, 1990.

[PP94] Alberto Pettorossi y Maurizio Proietti. Transformation of logics programs: Foundations and techniques. *The Journal of Logic Programming*, (19–20):261–320, 1994.

[PP96a] Alberto Pettorossi y Maurizio Proietti. A comparative revisitation of some program transformation techniques. En O. Danvy, R. Glück, y P. Thiemann, editores, *Partial Evaluation (International Seminar)*, volumen 1110 de *Lecture Notes in Computer Science*, pp. 355–385. Springer-Verlang, Dagstuhl Castle, Alemania, febrero de 1996.

[PP96b] Alberto Pettorossi y Maurizio Proietti. Developing correct and efficient logic programs by transformation. *Knowledge Engineering Review*, 11(4):347–360, diciembre de 1996.

[PP96c] Alberto Pettorossi y Maurizio Proietti. Future directions in program transformation. *ACM Computing Surveys*, 28 (4es):171–es, diciembre de 1996.

[PP96d] Alberto Pettorossi y Maurizio Proietti. Rules and strategies for transforming functional and logic programs. *ACM Computing Surveys*, 28(2):360–414, 1996.

[PP98a] Alberto Pettorossi y Maurizio Proietti. Logic program transformation. Tutorial en IJCLP 98, Manchester, Reino Unido, 15-19 junio, 1998.

[PP98b] Alberto Pettorossi y Maurizio Proietti. Transformation of logic programs. En D.M. Gabbay, C.J. Hogger, y J.A. Robinson, editores, *Handbook of Logic in Artificial Intelligence and Logic Programming*, volumen 5, pp. 697–787. Oxford University Press, 1998.

[PP02] Alberto Pettorossi y Maurizio Proietti. Program derivation = rules + strategies. En A. Kakas y F. Sadri, editores, *Computational Logic: Logic Programming and Beyond (Essays in Honour of Robert A. Kowalski - Part I)*, volumen 2407 de *Lectures Notes in Artificial Intelligence*, pp. 273–309. Springer-Verlag, 2002.

[PPR97a] Alberto Pettorossi, Maurizio Proietti, y Sophie Renault. Enhancing Partial Deduction via Unfold/Fold Rules. En *Logic Program Synthesis and Transformation*, volumen 1207 de *Lecture Notes in Computer Science*. LOPSTR'96, Springer-Verlag, agosto de 1997.

[PPR97b] Alberto Pettorossi, Maurizio Proietti, y Sophie Renault. Reducing nondeterminism while specializing logic programs. En *Proceedings of the 24th Annual ACM Symposium on Principles of Programming Languages*, pp. 414–427. ACM Press, 1997.

[PS90] H.A. Partsch y F.A. Stomp. A fast pattern matching algorithm derived by transformational and assertional reasoning. *Formal Aspects of Computing*, 2:109–122, 1990.

[PV91] H.A. Partsch y N. Völker. Another case study on reusability of transformational development. En M. Broy y M. Wirsing, editores, *Methods of Programming (Selected Papers on the CIP-Project)*, volumen 544 de *Lecture Notes in Computer Science*, pp. 35–48. Springer-Verlag, 1991.

[RAE01] *Diccionario esencial de la Real Academia Española*. Espasa, segunda edición, 2001.

[Rosar] David A. Rosenblueth. Chain programs for writing deterministic metainterpreters. *Theory and Practice of Logic Programming*, (To appear).

[RS83] John H. Reif y William L. Scherlis. Deriving efficient graphs algorithms. En Springer-Verlag, editor, *Logic of programs (Proceedings 1983)*, volumen 164, pp. 421–441. Lecture Notes in Computer Science, 1983.

[RS87] D.E. Rydeheard y J.G. Stell. Foundations of equational deduction: A categorical treatment of equational proofs and unification algorithms. En David Pitt, Axel Poigné, y David Rydeheard, editores, *Category Theory and Computer Programming: Proceedings*, volumen 283 de *Lecture Notes in Computer Science*, pp. 114–139. Springer-Verlag, Guilford, Reino Unido, septiembre de 1987.

[Ryd86a] D.E. Rydeheard. Adjunctions. En David Pitt, Samson Abramsky, Axel Poigné, y David Rydeheard, editores, *Category Theory and Computer Programming: Proceedings*, volumen 240 de *Lecture Notes in Computer Science*, pp. 51–57. Springer-Verlag, Guilford, Reino Unido, septiembre de 1986.

[Ryd86b] D.E. Rydeheard. Functors and natural transformations. En David Pitt, Samson Abramsky, Axel Poigné, y David Rydeheard, editores, *Category Theory and Computer Programming: Proceedings*, volumen 240 de *Lecture Notes in Computer Science*, pp. 43–50. Springer-Verlag, Guilford, Reino Unido, septiembre de 1986.

[Sch67] Joseph R. Schoenfield. *Mathematical Logic.* Adison-Wesley, 1967.

[SGJ96] M.H. Sørensen, R. Glück, y N.D. Jones. A positive supercompiler. *Journal of Functional Programming*, 6(6):811–838, 1996.

[She92] J.C. Shepherdson. Unfold/fold transformations of logic programs. *Math. Struct. in Comp. Science*, 2:143–157, 1992.

[Smi87] D.R. Smith. On the design of generate-and-test algorithms: subespace generators. En L.G.L.T. Meertens, editor, *Program specification and transformation*, IFIP, pp. 207–220. Elsevier Science Publishers B.V. (North–Holland), 1987.

[Smi91] Donald A. Smith. Partial evaluation of pattern matching in constraint logic programming language. En *Proceedings of the Symposium on Partial Evaluation and Semantics-Based Program Manipulation, PEPM'91*, pp. 62–71, Connecticut, E.U.A., junio de 1991. ACM Press.

[Ste94] Graham A. Stephen. *String Searching Algorithms*, volumen 3 de *Lecture Notes Series on Computing.* World Scientific Publishing, 1994.

[Tam87] H. Tamaki. Stream-based compilation of ground I/O Prolog into committed-choice languages. En *Proc. Fourth International Conference on Logic Programming*, pp. 376–393, Melbourne, Australia, 1987.

[Tar86] A. Tarlecki. Bits and pieces of the theory of institutions. En David Pitt, Samson Abramsky, Axel Poigné, y David Rydeheard, editores, *Category Theory and Computer Programming: Proceedings*, volumen 240 de *Lecture Notes in Computer Science*, pp. 313–332. Springer-Verlag, Guilford, Reino Unido, septiembre de 1986.

[TS84] Hisao Tamaki y Taisuke Sato. Unfold/fold transformation of logic programs. *International Conference on Logic Programming*, pp. 127–138, 1984.

[Ued87] K. Ueda. Making exhaustive search programs deterministic. *New Generation Computing*, (5):29–44, 1987.

[vdW87] J. van der Woude. Playing with patterns, searching for strings. Computing Science Report 87-13, Eindhoven University of Technology, Holanda, 1987.

[Weg76] B. Wegbreit. Goal-directed program transformation. *IEEE Transactions on Software Engineering*, SE-2(2):69–80, 1976.

[WZ96] B.W. Watson y G. Zwaan. A taxonomy of sublinear multiple keyword pattern matching algorithms. *Science of Computer Programming*, (27):85–118, 1996.

[You79] E.N. Yourdon, editor. *Classics in software engineering*. Computing. Yourdon Press, 1979.

[Zhu94] Hong Zhu. How powerful are folding/unfolding transformations? *Journal of Functional Programming*, 4(1):89–112, 1994.

Índice de Materias

átomo, 23
  seleccionado, 27

afirmaciones
  acumuladas, 145
alfabeto, 31, 63
algoritmo
  de Knuth, Morris y Pratt, 70
  de Apostolico, Galil, y Giancarlo, 129
  de Boyer y Moore, 66
  de fuerza bruta, 65
  de fuerza bruta (en Prolog), 80
  de fuerza bruta (usando *append*), 78
  de Horspool, 129
  de Morris y Pratt, 70
  de Simon, 120
argumento, 24
aridad, 23
axioma
  de reflexividad, 30
  de simetría, 30
  de transitividad, 30

Backus, John, 10
base de Herbrand, 25
Bauer, F. L., 9
borde etiquetado, 120
Burstall, Rod M., 33

cabeza de una lista, 31
cabeza de una cláusula, 24
cadena, 63
  vacía, 63
casamiento, 64
cláusula, 24
  de Horn, 24
  definida, 24
  vacía, 24
cláusula de entrada, 27
Clark, Keith, 30
cola de una lista, 31
concatenación, 64
concordancia, 64
conjunto de ecuaciones e inecuaciones, 59
construcción por subconjuntos, 121
corrección
  parcial, 38
  total, 38
crisis de *software*, 9
cuerpo de una cláusula, 24

Darlington, John, 33
deducción
  de algoritmos, 38
deducción parcial, 51, 135
deducción parcial disyuntiva, 136
definición de un predicado, 24
definiciones
  no recursivas, 41
derivación
  de la parte de búsqueda del algoritmo KMP, 111
  del algoritmo MP, 117

determinística, 80
fallida, 28
derivación D
del algoritmo KMP, 121
derivación determinística, 80
derivación exitosa, 27
derivación-SLD, 27
derivaciones determinísticas, 53
desdoblado
determinístico, 53
problemas, 52
total, 52
desdoblado/doblado, 34
desempeño promedio
del algoritmo de fuerza bruta, 66
desempeño peor
del algoritmo de fuerza bruta, 66
del algoritmo BM, 70
desempeño promedio
del algoritmo BM, 70
del algoritmo KMP, 74
Dijkstra, E.W., 9
discordancia, 64
doblado
(como operación inversa), 40
extendido, 39, 140
local (in situ), 46
local o in situ, 88
y eficiencia, 41

eficiencia de un programa lógico, 29
equivalencia de programas lógicos, 29
equivalencia de programas, 43
equivalencia entre programas lógicos, 26
especialización de programas lógicos, 51
especificación (lógica), 29
estandarización aparte, 27
estrategia, 35
D, 56, 140
D', 138, 141
de Burstall y Darlington, 35
S, 53
evaluación parcial, 51, 135
expresión, 24
aterrizada, 24

fórmula verdadera, 25
factorización, 86
forma
prefijo-triangular, 112
sufijo triangular, 84
función *next* (KMP), 119
función f, 72
función f (KMP), 118
función next (definición), 73
fusión de computaciones, 140

hecho, 24
cláusula unitaria, 24
heurísticas, 37
Hoare, C.A.R, 9

instancia, 25
calculada, 27
interpretación
declarativa de un programa lógico, 26
procedural de un programa lógico, 28
interpretación de Herbrand, 25
introducción
de ecuaciones, 57

letras, 63
listas, 31
literal, 24
negativa, 24
positiva, 24
longitud

de una derivación, 27
de una derivación determinística, 80

método de transformación
por desdoblado/doblado, 38
MacCarthy, John, 11
mal-casamiento, 64
memoria del algoritmo KMP, 114
meta, 24
modelo, 25
modelo de Herbrand, 26
modelo mínimo de Herbrand, 26
monoide, 64

negación por falla, 31

Partsch, Helmut, 12
paso de derivación, 27
paso de resolución, 27
paso de transformación
reversible, 45
patrón, 64
Pepper,Peter, 12
prefijo, 64
pregunta, 24
previsión, 86, 137, 140
previsión (lookahead), 59, 80
problema
de la ocurrencia de una cadena, 64
problema del casamiento exacto de cadenas, 63
programa lógico, 24
(totalmente) correcto, 29
completo, 29
parcialmente correcto, 29
proposición, 24
provisión, 140
provisión (lookbehind), 80, 137
prueba
BM, 56
KMP, 55

Regla
de instanciación, 35
de desdoblado, 35
de doblado, 35
de introducción de definiciones, 35
regla
de eliminación de definiciones, 41
de reemplazo de submetas, 43
de desdoblado, 39
de división en casos, 56
de doblado, 40
de introducción de definiciones, 41
regla de selección, 27
regla de transformación
parcialmente correcta, 45
totalmente correcta, 45
reglas
accesorias, 44
reglas de transformación, 38
resolvente-SLD, 27
respuesta, 27
reuso de desarrollos, 114
reuso de desarrollos, 120

símbolo, 63
significado
de un programa lógico, 26
significado deseado, 29
subcadena, 64
submeta, 24
subsumción, 25, 78
sucesión
parcialmente correcta, 44
reversible, 45
totalmente correcta, 44
sucesión transformacional, 38

sufijo, 64
sustitución, 24
    más general, 25
sustitutividad en funciones, 30
sustitutividad en predicados, 30

término, 23
términos
    unificables, 25
tabla f, 72
tabla next, 73
taxonomía, 134
teoría de la igualdad, 30
    de Clark, 30
teoría estándar de igualdad, 30
teorema
    de corrección con respecto a la semántica
        del mínimo modelo de Herbrand,
        46
terminación
    de las derivaciones determinísticas, 59
texto, 64
transformación de programas, 38
trayectoria determinística, 53

unificador, 25
    más general (umg), 25
universo de Herbrand, 25

variante, 25
variante del algoritmo BM
    de Apostolico, Galil, y Giancarlo, 129
    de Horspool, 129
    de Partsch y Stomp, 128
verdad, 25