

UNIVERSIDAD DE DEUSTO

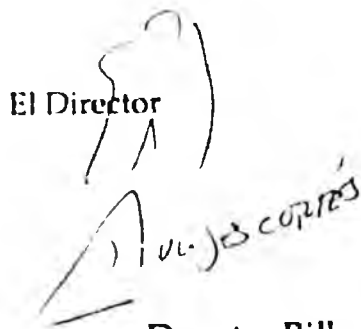
Facultad de Informática

IMPLEMENTACIÓN EFICIENTE
DE LA RECURSIVIDAD FINAL
EN LENGUAJES IMPERATIVOS
BASADA EN NUEVAS TÉCNICAS
DE COMPILACIÓN

Tesis doctoral presentada por D. Jesús Luis Díaz Labrador

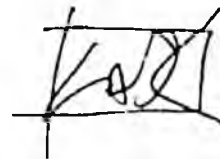
Dirigida por el Dr. D. Ulises Cortés García

El Director



Ulises Cortés

El Doctorando



Deusto, Bilbao, marzo de 1996

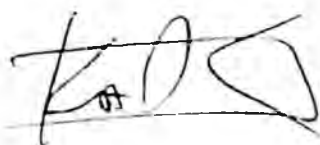
Deusto, a 11 de octubre de 1996

Atención:

La tesis ha sido impresa para su encuadernación en dos caras: las caras pares llevan la numeración de página en el borde izquierdo y las impares en el borde derecho.

Se han insertado hojas en blanco para suplir caras pares no impresas cuando el capítulo siguiente debía comenzar en hoja impar.

Sin más,



JOSU K. DÍAZ LABRADOR

A Karmele
y
a mis padres

You are like a hurricane
There's one in your eyes
NEIL YOUNG

TABULA GRATULATORIA

Quisiera expresar mi gratitud a la Universidad de Deusto por sostener la realización de este trabajo, poniendo a mi disposición numerosos medios a los que de otra forma no hubiera podido tener acceso. También a todas las personas que me han ayudado, tanto en el sentido literal de la palabra como por el simple, pero tan importante, apoyo emocional que me han prestado.

En primer lugar, es necesario, y no por ello menos sentido, que agradezca sinceramente a mi director, Ulises Cortés, de la Universitat Politècnica de Catalunya, la inestimable ayuda que ha hecho posible que este proyecto saliera adelante. Ulises ha sabido guiarme por caminos desconocidos para mí, y gran parte de mis ideas e intereses actuales se deben a sus orientaciones y estímulos. Es por ello que me siento afortunado por haberle conocido, y mucho más por haber sido merecedor de sus esfuerzos a lo largo de estos años.

He de agradecer especialmente, aun sin haber mantenido con ellos más que esporádicas "conversaciones electrónicas", a Chris Hanson, Guillermo Rozas y demás colaboradores del "SCHEME Project" del Massachusetts Institute of Technology (Department of Electrical Engineering and Computer Science) el desarrollo de MIT-SCHEME, la implementación del lenguaje SCHEME en el MIT. Este trabajo no habría resultado tan agradable, en su vertiente aplicada, de no haber contado con su sistema, y debo reconocer al MIT por permitir su libre distribución.

También quiero mostrar mi gratitud a José María Alcaide, de la Universidad del País Vasco, y a Carlos Sierra, del Centre d'Estudis Avançats de Blanes (CSIC), por la inestimable ayuda prestada en determinados momentos. Yolanda Herranz y Jesús Pastor, de la Universidad de Vigo, colaboraron en algunos detalles metodológicos.

Agradezco sinceramente a Anselmo del Moral, María José Gil, Andoni Eguiluz, Javier García Zubía, Monika Gómez, Pilar Gómez, Inés Jacob y

Javier Oliver, compañeros de la Facultad de Informática de la Universidad de Deusto, la lectura de partes más o menos extensas de este trabajo, y los valiosos comentarios que ayudaron indudablemente a mejorarlo. Quiero recordarles también, junto a Máximo Llaguno y Juan Luis Gutiérrez, decanos de la Facultad, y todos los demás compañeros, por su inestimable apoyo en todo momento.

A todos mis amigos y amigas debo reconocerles que hayan seguido regalándome su entrañable amistad, a pesar de los pocos ratos que les he podido dedicar.

A mis padres, Feli y Jesús, mis hermanos, Begoña e Ibón, mis abuelos, Feli y Manuel, y Manuela, y mis suegros, Carmen y Frutos, deseo dedicarles más que un agradecimiento, pues casi todo lo que soy se lo debo a ellos y constantemente me animan a seguir adelante. Y por último, desde el principio, a Kar mele: si no fuera por ti...

RESUMEN

El trabajo muestra una implementación eficiente (comportamiento dinámico en espacio constante) de los programas recursivos finales en lenguajes imperativos, más en concreto, en PASCAL. Ello se consigue mediante técnicas alternativas de compilación, en que se modifica el tratamiento genérico de las activaciones finales de subprogramas. Con esto, se obtiene como caso particular el comportamiento descrito para la recursividad final, ya sea directa o mutua, ya sea de procedimientos o de funciones. La técnica propuesta es más simple que otras estrategias conocidas para lenguajes como SCHEME (de características diferentes de PASCAL), aunque tiene el inconveniente de imponer cierta restricción en los programas recursivos finales que pueden ser optimados; sin embargo, se argumenta que esta restricción tiene escaso impacto en la programación habitual en el lenguaje. Finalmente, la presencia de la optimación de la recursividad final en PASCAL posibilita hábitos de programación más en consonancia con los criterios actualmente aceptados.

ABSTRACT

This work shows an efficient implementation (execution in constant space) of tail recursive programs written in imperative languages, more specifically, PASCAL. This is possible by means of new compilation strategies, in which the generic treatment of tail calls is modified. Thus, the efficiency of the particular case of tail recursion is obtained, either for procedures or functions, either for direct or indirect recursion. The proposed implementation method is simpler than known strategies for other languages, notably SCHEME (which have several important differences from PASCAL), although it has one drawback: it imposes a restriction in the set of tail recursive programs which can be optimized. Nonetheless, it is claimed that this restriction has little impact in the common programming in PASCAL. Tail recursion optimization is important, as it allows programming habits closer to the accepted programming methodologies.

ÍNDICE

ÍNDICE DE EPÍGRAFES.....	xv
Índice de algoritmos.....	xv
Índice de condiciones.....	xv
Índice de restricciones.....	xvi
Índice de programas.....	xvii
Índice de figuras	xx
Índice de tablas	xxii
1. INTRODUCCIÓN.....	1
1.1. Ámbito, motivación e interés del trabajo	1
1.2. Organización de la memoria	6
2. BASES TEÓRICAS Y ANTECEDENTES	9
2.1. Introducción	9
2.2. Recursividad y programación.....	10
2.2.1. Comportamiento de los programas recursivos.....	14
2.2.2. Iteratividad y recursividad.....	20
2.3. Soluciones (I): transformación de programas.....	25
2.4. Soluciones (II): recursividad final en SCHEME.....	34
2.4.1. Obtención clásica de la recursividad final en SCHEME	35
2.4.2. Recursividad final y vinculación por medio de una pila.....	44
2.5. Planteamiento del trabajo e hipótesis	50
2.5.1. Modelo dinámico estándar del lenguaje PASCAL	54
3. OPTIMACIÓN DE ACTIVACIONES FINALES RECURSIVAS DE PROCEDIMIENTOS	61
3.1. Introducción	61
3.1.1. Posibles optimaciones en PASCAL.....	61
3.2. Modelo dinámico optimado: activaciones finales recursivas de procedimientos	70

3.2.1. Definición de posición final en PASCAL.....	70
3.2.2. Optimación de la secuencia de llamada.....	74
3.2.3. Datos locales.....	82
3.2.4. Parámetros formales por valor.....	86
3.2.5. Parámetros por referencia.....	92
3.2.6. Parámetros procedimentales y funcionales.....	106
3.2.7. Activaciones de otros procedimientos o funciones.....	118
3.2.8. Saltos no locales.....	120
3.3. Consideraciones adicionales	121
4. OPTIMACIÓN DE ACTIVACIONES FINALES NO RECURSIVAS DE PROCEDIMIENTOS.....	127
4.1. Introducción.....	127
4.2. Modelo dinámico optimado: activaciones finales no recursivas de procedimientos	127
4.2.1. Posición final y nivel de anidamiento	128
4.2.2. Optimación de la secuencia de llamada.....	130
4.2.3. Datos locales, parámetros formales, otras activaciones y saltos no locales	140
4.3. Consideraciones adicionales	145
5. OPTIMACIÓN DE ACTIVACIONES FINALES DE FUNCIONES	155
5.1. Introducción.....	155
5.1.1. El concepto de función en PASCAL.....	156
5.1.2. Modelo dinámico estándar de las funciones en PASCAL.....	160
5.1.3. Posibles optimaciones.....	164
5.2. Modelo dinámico optimado: activaciones finales recursivas de funciones	171
5.2.1. Definición de activación final de función en PASCAL.....	172
5.2.2. Optimación de la secuencia de llamada.....	173
5.3. Modelo dinámico optimado: activaciones finales no recursivas de funciones.....	175
5.4. Consideraciones adicionales	184
5.5. Modelo dinámico optimado: activaciones finales mixtas de procedimientos.....	188
5.5.1. Definición de activación final mixta en PASCAL.....	188
5.5.2. Optimación de la secuencia de llamada.....	189

6. EVALUACIÓN DE LA OPTIMACIÓN.....	199
6.1. Introducción	199
6.2. Comportamiento dinámico optimado	199
6.2.1. Activaciones finales recursivas de procedimientos	206
6.2.2. Activaciones finales no recursivas de procedimientos	218
6.2.3. Activaciones finales recursivas y no recursivas de funciones.....	237
6.2.4. Comentarios finales	247
6.3. Requisitos estáticos de la optimación	249
6.3.1. Un compilador clásico de PASCAL	249
6.3.2. Detección de las activaciones optimables.....	254
6.4. Influencia de la elección del modelo dinámico del lenguaje en la optimación.....	268
6.4.1. Variaciones simples	269
6.4.2. Parámetros antes de la zona de control.....	272
6.4.3. Uso de instrucciones especializadas	277
6.4.4. Paso de parámetros	286
6.4.5. Consideraciones adicionales	292
6.5. El compilador optimador PASCAL-JK	293
7. RECURSIVIDAD FINAL Y PROGRAMACIÓN EN PASCAL	313
7.1. Introducción	313
7.2. Alcance de las restricciones de la optimación.....	313
7.3. Recursividad final y criterios de programación	319
8. CONCLUSIONES.....	337
Líneas futuras de investigación	341
9. BIBLIOGRAFÍA	343
APÉNDICE A. EXTRACTO DEL COMPILADOR PASCAL-JK	361
A.1. Introducción	361
A.2. Módulo de optimación	361
A.3. Módulo de generación de código.....	366

ÍNDICE DE EPÍGRAFES

ÍNDICE DE ALGORITMOS

Algoritmo 2.5.1. Secuencia de llamada estándar de un procedimiento en PASCAL.....	59
Algoritmo 2.5.2. Secuencia de retorno estándar de un procedimiento en PASCAL.....	60
Algoritmo 3.2.1. Secuencia de llamada optimada para una activación final recursiva de procedimiento.....	80
Algoritmo 4.2.1. Secuencia de llamada optimada para una activación final no recursiva de procedimiento.....	139
Algoritmo 4.2.2. Secuencia de llamada optimada para una activación final no recursiva de procedimiento (incluyendo el caso en que se invoca un parámetro formal procedimental).....	143
Algoritmo 5.1.1. Secuencia de llamada estándar para activaciones de funciones en PASCAL.....	162
Algoritmo 5.1.2. Secuencia de retorno estándar para activaciones de funciones en PASCAL.....	163
Algoritmo 5.2.1. Secuencia de llamada optimada para activaciones finales recursivas de funciones.....	175
Algoritmo 5.5.1. Secuencia de llamada optimada para activaciones finales mixtas.....	197
Algoritmo 6.4.1. Secuencia de llamada optimada para activaciones finales no recursivas cuando los parámetros formales se encuentran en la base del registro de activación.....	275

ÍNDICE DE CONDICIONES

Condición 3.2.1.	75
Condición 3.2.2.	76
Condición 3.2.3.	76
Condición 3.2.4.	77
Condición 3.2.5.	78

Condición 3.2.6.	78
Condición 3.2.7.	79
Condición 3.2.8.	81
Condición 3.2.9.	88
Condición 3.2.10.	90
Condición 3.2.11.	95
Condición 3.2.12.	111
Condición 4.2.1.	130
Condición 4.2.2.	131
Condición 4.2.3.	131
Condición 4.2.4.	132
Condición 4.2.5.	132
Condición 4.2.6.	132
Condición 4.2.7.	132
Condición 4.2.8.	137
Condición 4.2.9.	141
Condición 4.2.10.	143
Condición 5.5.1.	190
Condición 5.5.2.	191
Condición 6.3.1.	259

ÍNDICE DE RESTRICCIONES

Restricción 3.2.1.	73
Restricción 3.2.2.	99
Restricción 3.2.3.	116
Restricción 4.2.1.	128
Restricción 4.2.2.	129
Restricción 4.2.3.	144
Restricción 4.2.4.	144
Restricción 5.2.1.	173
Restricción 5.3.1.	176
Restricción 5.3.2.	176
Restricción 5.3.3.	181
Restricción 5.3.4.	182
Restricción 5.5.1.	189
Restricción 5.5.2.	189

ÍNDICE DE PROGRAMAS

Programa 2.4.1.	Un programa con valores funcionales en SCHEME.....	37
Programa 2.4.2.	Un programa con recursividad final no directa en SCHEME.....	42
Programa 3.1.1.	Un procedimiento recursivo final en PASCAL.....	62
Programa 3.1.2.	Procedimiento no recursivo equivalente al del Programa 3.1.1.....	63
Programa 3.1.3.	Un procedimiento con una activación no recursiva final en PASCAL.....	65
Programa 3.1.4.	Dos procedimientos mutuamente recursivos en PASCAL.....	67
Programa 3.2.1.	Un procedimiento recursivo final con datos locales en PASCAL.....	82
Programa 3.2.2.	Procedimiento no recursivo (¡no equivalente!) correspondiente al del Programa 3.2.1.....	83
Programa 3.2.3.	Un procedimiento recursivo final con dos parámetros formales por valor en PASCAL.....	86
Programa 3.2.4.	Procedimiento no recursivo (¡no equivalente!) correspondiente al del Programa 3.2.3.....	87
Programa 3.2.5.	Procedimiento no recursivo equivalente al del Programa 3.2.3.....	87
Programa 3.2.6.	Procedimiento no recursivo equivalente al del Programa 3.2.3 en PASCAL estricto.....	88
Programa 3.2.7.	Otro procedimiento recursivo final con dos parámetros formales por valor en PASCAL.....	89
Programa 3.2.8.	Procedimiento no recursivo equivalente al del Programa 3.2.7.....	90
Programa 3.2.9.	Procedimiento recursivo final con un parámetro por referencia en PASCAL.....	93
Programa 3.2.10.	Procedimiento no recursivo equivalente al del Programa 3.2.9.....	95
Programa 3.2.11.	Procedimiento recursivo final con dos parámetros por referencia en PASCAL.....	96
Programa 3.2.12.	Procedimiento no recursivo equivalente al del Programa 3.2.11.....	96
Programa 3.2.13.	Procedimiento recursivo final con un dato local como parámetro actual por referencia en PASCAL.....	98
Programa 3.2.14.	Procedimiento recursivo final con un dato local como parámetro actual por referencia en PASCAL.....	101
Programa 3.2.15.	Procedimiento recursivo final (con una activación final optimada) equivalente al del Programa 3.2.14.....	102
Programa 3.2.16.	Procedimiento no recursivo (con dos activaciones finales optimadas) equivalente al del Programa 3.2.14.....	104
Programa 3.2.17.	Un programa PASCAL que incluye un procedimiento con un parámetro formal procedimental.....	106
Programa 3.2.18.	Procedimiento recursivo final con un parámetro procedimental en PASCAL.....	109
Programa 3.2.19.	Procedimiento no recursivo equivalente al del Programa 3.2.18.....	110
Programa 3.2.20.	Procedimiento recursivo final con dos parámetros procedimentales en PASCAL.....	111
Programa 3.2.21.	Procedimiento no recursivo equivalente al del Programa 3.2.20.....	112
Programa 3.2.22.	Otro procedimiento recursivo final con un parámetro procedimental en PASCAL.....	113

Programa 3.2.23.	Procedimiento recursivo final con un parámetro procedimental en PASCAL (modificado a partir del Programa 3.2.22)	116
Programa 3.3.1.	Una función recursiva final en SCHEME (tomada de Kamin [1990: 442])	124
Programa 4.3.1.	Dos procedimientos finales mutuamente recursivos en PASCAL	146
Programa 4.3.2.	Dos procedimientos finales, uno local al otro, mutuamente recursivos en PASCAL (modificado a partir del Programa 4.3.1)	148
Programa 4.3.3.	Transformación del Programa 4.3.2 en un programa no recursivo equivalente	149
Programa 4.3.4.	Una variación minúscula del Programa 4.3.2	150
Programa 4.3.5.	Transformación del Programa 4.3.4 en un programa no recursivo ¡pero no equivalente!	151
Programa 5.1.1.	Un programa con una función recursiva final y una función no recursiva final en PASCAL (tomado de Jensen y Wirth [1991: 244])	165
Programa 5.1.2.	Función no recursiva equivalente a la del Programa 5.1.1	167
Programa 5.1.3.	Función con una activación no recursiva final de procedimiento	170
Programa 5.3.1.	Un programa con una supuesta activación final no recursiva de función	177
Programa 5.3.2.	Un programa con una activación de función que casi es final no recursiva	180
Programa 5.5.1.	Un programa con una activación final mixta en PASCAL	191
Programa 6.2.1.	Esquema del código objeto generado para un procedimiento directamente recursivo según el modelo dinámico estándar, usando la cadena estática	206
Programa 6.2.2.	Esquema del código objeto generado para un procedimiento directamente recursivo según el modelo dinámico optimado, usando la cadena estática	207
Programa 6.2.3.	Esquema del código objeto generado para un procedimiento directamente recursivo según el modelo dinámico estándar, usando el <i>display</i>	212
Programa 6.2.4.	Esquema del código objeto generado para un procedimiento directamente recursivo según el modelo dinámico optimado, usando el <i>display</i>	213
Programa 6.2.5.	Esquema del código objeto generado para una activación final casi recursiva según el modelo dinámico optimado, y usando la cadena estática	219
Programa 6.2.6.	Esquema del código objeto generado para una activación final no recursiva, según el modelo dinámico estándar, y usando la cadena estática	221
Programa 6.2.7.	Esquema del código objeto generado para una activación final no recursiva, según el modelo dinámico optimado, y usando la cadena estática	222
Programa 6.2.8.	Esquema del código objeto generado para una activación final casi recursiva, según el modelo dinámico optimado, y usando el <i>display</i>	225
Programa 6.2.9.	Esquema del código objeto generado para una activación final no recursiva, según el modelo dinámico optimado, y usando el <i>display</i>	226
Programa 6.2.10.	Esquema del código objeto generado para una activación final de un parámetro formal procedimental, según el modelo dinámico estándar, y usando la cadena estática	232

Programa 6.2.11.	Esquema del código objeto generado para una activación final de un parámetro formal procedimental local, según el modelo dinámico optimado, y con la cadena estática.....	233
Programa 6.3.1.	Esquema de la acción semántica de recorrido del árbol decorado de la parte de sentencias de un procedimiento que verifica las restricciones de posición final de las activaciones de procedimientos	262
Programa 6.3.2.	Esquema de la acción semántica de recorrido del árbol decorado de la parte de sentencias de una función que verifica las restricciones de posición final de las sentencias de asignación del valor de retorno de funciones.....	264
Programa 6.4.1.	Esquema del código objeto generado para las secuencias de llamada y retorno usando instrucciones especializadas de manejo de la pila máquina, según el modelo dinámico estándar, usando la cadena estática, y con la estructura del registro de activación de la Figura 6.4.4	282
Programa 6.4.2.	Esquema del código objeto generado para las secuencias de llamada y retorno usando instrucciones especializadas de manejo de la pila máquina y de subrutinas, según el modelo dinámico estándar, usando la cadena estática, y con la estructura del registro de activación de la Figura 6.4.4	284
Programa 6.5.1.	Implementación en SCHEME del subsistema de lectura como tipo abstracto de datos, basada en la signatura de la Figura 6.5.2.....	307
Programa 6.5.2.	Implementación en SCHEME de la generación de código para una sentencia de activación de procedimiento, sin optimación.....	309
Programa 6.5.3.	Implementación en SCHEME de la generación de código para una sentencia de activación final recursiva optimada de procedimiento o función	310
Programa 6.5.4.	Implementación en SCHEME de la generación de código para una sentencia de activación final no recursiva optimada de procedimiento o función	311
Programa 7.2.1.	Un procedimiento recursivo final para duplicar una lista ligada (solo con ligeras variaciones a partir del original de Rohl [1984: 26])	315
Programa 7.2.2.	Una función recursiva en "dos niveles" (original de Rohl [1984: 19])	317
Programa 7.3.1.	Implementación recursiva ineficiente de la función que calcula los números de Fibonacci	320
Programa 7.3.2.	Implementación recursiva eficiente de la función que calcula los números de Fibonacci	321
Programa 7.3.3.	Implementación iterativa eficiente de la función que calcula los números de Fibonacci (adaptada del original de Wirth [1986a: 139]).....	321
Programa 7.3.4.	Un procedimiento para construir árboles de expresiones (con ligeras variaciones a partir del original de Rohl [1984: 99-100]).....	323
Programa 7.3.5.	Un procedimiento para construir árboles de expresiones basado exactamente en la gramática subyacente	325
Programa 7.3.6.	Otro procedimiento para construir árboles de expresiones basado exactamente en la gramática subyacente, con procedimientos no anidados.....	327
Programa 7.3.7.	Estructura necesaria para los procedimientos de un analizador sintáctico descendente si la interdependencia entre las variables no admite el anidamiento	331

Programa 7.3.8. La función de Ackermann implementada en PASCAL con una función y con un procedimiento (solo con ligeras variaciones a partir del original de Rohl [1984: 171])	332
--	-----

ÍNDICE DE FIGURAS

Figura 2.2.1. Estructura clásica de un programa objeto en un lenguaje imperativo, con vinculación mediante una pila	17
Figura 2.2.2. Estructura típica del registro de activación de un subprograma en un lenguaje imperativo	18
Figura 2.4.1. (a) Estructura clásica de un programa objeto en el lenguaje LISP, con vinculación de datos en el montículo. (b) Registro de control, alojado en la pila de control. (c) Registro de ambiente, alojado en el montículo	39
Figura 2.4.2. Instantáneas sucesivas de los ambientes creados y liberados durante la ejecución del Programa 2.4.2	42
Figura 2.4.3. Estructura genérica de la pila de activaciones según Hanson [1990]	46
Figura 2.4.4. Instantáneas sucesivas de la pila de activaciones durante la ejecución del Programa 2.4.2, según la estrategia de vinculación en una pila de Hansen [1990]	48
Figura 2.5.1. Estructura del registro de activación de un procedimiento en PASCAL	55
Figura 2.5.2. Modo de acceso a la información local del registro de activación	57
Figura 3.1.1. Instantáneas sucesivas de la pila de control durante la ejecución del Programa 3.1.1 (a) en las llamadas, y (b) en los retornos	62
Figura 3.1.2. Instantáneas sucesivas de la pila de control durante la ejecución del Programa 3.1.2	63
Figura 3.1.3. Instantáneas sucesivas de la pila de control durante la ejecución del Programa 3.1.3	66
Figura 3.1.4. Instantáneas sucesivas de la pila de control durante la ejecución del Programa 3.1.3 después de la optimización de la llamada	67
Figura 3.1.5. Instantáneas sucesivas de la pila de control durante la ejecución del Programa 3.1.4 (a) en las llamadas, y (b) en los retornos	68
Figura 3.1.6. Instantáneas sucesivas de la pila de control durante la ejecución del Programa 3.1.4 cuando se ha efectuado la optimización	69
Figura 3.2.1. Enlaces de acceso en una cadena de activaciones recursivas	70
Figura 3.2.2. Instantánea de la pila de control durante una cadena de activaciones finales recursivas del procedimiento del Programa 3.2.9 (según el modelo dinámico estándar)	94
Figura 3.2.3. Instantánea de la pila de control durante una cadena de activaciones finales recursivas del procedimiento del Programa 3.2.9 (según el modelo dinámico optimado)	94
Figura 3.2.4. Instantánea de la pila de control durante una cadena de dos activaciones finales recursivas del procedimiento del Programa 3.2.11, según el modelo dinámico estándar	97

Figura 3.2.5.	Instantánea de la pila de control durante una cadena de activaciones finales recursivas del procedimiento del Programa 3.2.13, según el modelo dinámico estándar.....	98
Figura 3.2.6.	Instantáneas de la pila de control durante una cadena de activaciones finales recursivas del procedimiento del Programa 3.2.14, según el modelo dinámico estándar.....	101
Figura 3.2.7.	Más instantáneas de la pila de control durante una cadena de activaciones finales recursivas del procedimiento del Programa 3.2.14, (a) según el modelo dinámico estándar, y (b) según el modelo dinámico optimado.....	103
Figura 3.2.8.	Instantáneas de la pila de control durante una cadena de activaciones finales recursivas del procedimiento del Programa 3.2.14, según el modelo dinámico optimado modificado.....	105
Figura 3.2.9.	Instantáneas de la pila de control durante la ejecución del Programa 3.2.17 (a) después de la activación del procedimiento que incluye el parámetro formal procedimental, y (b) después de la activación del propio parámetro formal procedimental.....	107
Figura 3.2.10.	Instantánea de la pila de control durante una cadena de activaciones finales recursivas del procedimiento del Programa 3.2.18 (según el modelo dinámico estándar).....	109
Figura 3.2.11.	Instantánea de la pila de control durante una cadena de activaciones finales recursivas del procedimiento del Programa 3.2.18 (según el modelo dinámico optimado).....	110
Figura 3.2.12.	Instantánea de la pila de control durante una cadena de activaciones finales recursivas del procedimiento del Programa 3.2.22 (según el modelo dinámico estándar).....	114
Figura 3.2.13.	Otra instantánea de la pila de control durante una cadena de activaciones finales recursivas del procedimiento del Programa 3.2.22 (según el modelo dinámico estándar).....	115
Figura 3.2.14.	Instantánea de la pila de control durante una cadena de activaciones finales recursivas del procedimiento del Programa 3.2.22 (según el modelo dinámico optimado).....	115
Figura 3.2.15.	Instantáneas de la pila de control durante una cadena de activaciones finales recursivas del procedimiento del Programa 3.2.23, (a) según el modelo dinámico estándar, y (b) según el modelo dinámico optimado.....	117
Figura 4.2.1.	Enlace de acceso de un procedimiento activado no recursivamente sujeto a la Restricción 4.2.2.....	129
Figura 4.2.2.	Situaciones de la pila y del <i>display</i> después de la activación no recursiva final de un procedimiento con el mismo nivel de anidamiento que su llamador (a) según el modelo dinámico estándar, y (b) según el modelo dinámico optimado.....	136
Figura 4.2.3.	Situaciones de la pila y del <i>display</i> después de la activación no recursiva final de un procedimiento con nivel de anidamiento inferior a su llamador (a) según el modelo dinámico estándar y (b) según el modelo dinámico optimado.....	137
Figura 5.1.1.	Estructura del registro de activación de una función en PASCAL.....	160
Figura 5.1.2.	Instantáneas sucesivas de la pila de control durante la ejecución del Programa 5.1.1 (a) en las llamadas, y (b) en los retornos.....	166
Figura 5.1.3.	Instantáneas sucesivas de la pila de control durante la ejecución del Programa 5.1.2.....	168

Figura 6.2.1.	Comparación entre los costes (usando el criterio 2b) de las distintas clases de activaciones finales de procedimientos, (a) usando la cadena estática, y (b) usando el <i>display</i>	228
Figura 6.2.2.	Comparación entre las razones de optimación de las distintas clases de activaciones finales de procedimientos, (a) usando la cadena estática, y (b) usando el <i>display</i>	229
Figura 6.2.3.	Comparación (a) entre los costes (usando el criterio 2b), y (b) entre las razones de optimación de las distintas clases de activaciones finales de funciones, usando la cadena estática.....	244
Figura 6.2.4.	Comparación entre los costes (usando el criterio 2b) de las distintas clases de activaciones finales de procedimientos y funciones, usando la cadena estática.....	245
Figura 6.2.5.	Comparación entre las razones de optimación de las distintas clases de activaciones finales de procedimientos y funciones, usando la cadena estática.....	246
Figura 6.3.1.	Estructura típica de un compilador de PASCAL.....	254
Figura 6.4.1.	Variación del modelo dinámico en que se señala estrictamente el comienzo de cada registro de activación.....	269
Figura 6.4.2.	Estructura del registro de activación en que la zona de parámetros se sitúa antes de la zona de control (reelaborado a partir de Aho, Sethi y Ullman [1990: 411, figura 7.8]).....	272
Figura 6.4.3.	Instantáneas de la situación requerida en la pila de control en el modelo dinámico optimado durante una activación final no recursiva de procedimiento.....	274
Figura 6.4.4.	Estructura del registro de activación en que la zona de parámetros se sitúa antes de la zona de control (variación de la Figura 6.4.2).....	281
Figura 6.5.1.	Estructura del subsistema de lectura del compilador PASCAL-JK.....	305
Figura 6.5.2.	Signatura del tipo abstracto de datos que refleja el estado de lectura.....	306

ÍNDICE DE TABLAS

Tabla 6.2.1.	Costes de un repertorio de instrucciones del lenguaje máquina genérico, según dos criterios: (1) el coste definido según Aho, Sethi y Ullman [1990: 536-537], (2) el coste exacto de las instrucciones equivalentes del MC68020 [Motorola 1985: 9/11-9/28] bajo suposición de (a) "mejor caso", y (b) "peor caso".....	205
Tabla 6.2.2.	Coste de una secuencia de llamada recursiva final de procedimiento (usando la cadena estática) con el modelo dinámico estándar (los números de instrucciones se refieren al Programa 6.2.1).....	208
Tabla 6.2.3.	Coste de una secuencia de llamada recursiva final de procedimiento (usando la cadena estática) con el modelo dinámico optimado (los números de instrucciones se refieren al Programa 6.2.2).....	209
Tabla 6.2.4.	Razón de optimación (usando la cadena estática) para una llamada final recursiva de procedimiento.....	209

Tabla 6.2.5.	Coste de una secuencia de retorno de procedimiento (usando la cadena estática) en los dos modelos dinámicos (los números de instrucciones se refieren al Programa 6.2.1).....	210
Tabla 6.2.6.	Razón de optimación (usando la cadena estática) para un proceso completo de activación recursiva final de un procedimiento.....	210
Tabla 6.2.7.	Razón de optimación (usando el <i>display</i>) para una llamada final recursiva de procedimiento	213
Tabla 6.2.8.	Coste de una secuencia de retorno de procedimiento (usando el <i>display</i>) en los dos modelos dinámicos (los números de instrucciones se refieren al Programa 6.2.3).....	214
Tabla 6.2.9.	Razón de optimación (usando el <i>display</i>) para un proceso completo de activación recursiva final de procedimiento.....	214
Tabla 6.2.10.	Razón de optimación (con el <i>display</i> en memoria) en un proceso de activación recursiva final de procedimiento.....	214
Tabla 6.2.11.	Coste de una secuencia de llamada final casi recursiva optimada, usando la cadena estática (los números de instrucciones se refieren al Programa 6.2.5).....	220
Tabla 6.2.12.	Razón de optimación (usando la cadena estática) para una activación final casi recursiva.....	220
Tabla 6.2.13.	Coste de una secuencia de llamada final no recursiva usando la cadena estática, según el modelo dinámico estándar (los números de instrucciones se refieren al Programa 6.2.6)	223
Tabla 6.2.14.	Coste de una secuencia de llamada final no recursiva usando la cadena estática, según el modelo dinámico optimado (los números de instrucciones se refieren al Programa 6.2.7)	223
Tabla 6.2.15.	Razón de optimación (usando la cadena estática) para una activación final no recursiva.....	223
Tabla 6.2.16.	Primeros valores de la razón de optimación (usando la cadena estática) para una activación final no recursiva de procedimiento según la diferencia de anidamiento	224
Tabla 6.2.17.	Razón de optimación (usando el <i>display</i>) para una activación final casi recursiva	225
Tabla 6.2.18.	Coste de una secuencia de llamada final no recursiva usando el <i>display</i> , según el modelo dinámico optimado (los números de instrucciones se refieren al Programa 6.2.9).....	227
Tabla 6.2.19.	Razón de optimación (usando el <i>display</i>) para una activación final no recursiva	227
Tabla 6.2.20.	Resumen de las razones de optimación para las distintas clases de activaciones finales de procedimientos	229
Tabla 6.2.21.	Coste de una secuencia de llamada final (usando la cadena estática) de un parámetro formal procedimental, según el modelo dinámico estándar (los números de instrucciones se refieren al Programa 6.2.10).....	234
Tabla 6.2.22.	Coste de una secuencia de llamada final (usando la cadena estática) de un parámetro formal procedimental local, según el modelo dinámico optimado (los números de instrucciones se refieren al Programa 6.2.11)	234
Tabla 6.2.23.	Razón de optimación (usando la cadena estática) para una activación final no recursiva de un parámetro formal procedimental del llamador.....	235
Tabla 6.2.24.	Coste de una secuencia de llamada final (usando la cadena estática) de un parámetro formal procedimental no local, según el modelo dinámico optimado (los números de instrucciones se refieren al Programa 6.2.12)	235

Tabla 6.2.25. Razón de optimación (usando la cadena estática) para una activación final de un parámetro formal procedimental no local.....	236
Tabla 6.2.26. Primeros valores de la razón de optimación (usando la cadena estática) para una activación final de un parámetro formal procedimentalno local, según la distancia léxica con el llamador.....	236
Tabla 6.2.27. Costes de las instrucciones de manejo de <i>bits</i> para la verificación del centinela de definición de un dato, según los criterios habituales.....	239
Tabla 6.2.28. Costes del proceso de asignación de los centinelas de definición de los datos locales y el valor de retorno, según los criterios habituales.....	240
Tabla 6.2.29. Coste de una secuencia de retorno de función (usando la cadena estática) en los dos modelos dinámicos	240
Tabla 6.2.30. Razón de optimación tentativa (usando la cadena estática) para un proceso completo de activación recursiva final de una función.....	241
Tabla 6.2.31. Coste de la operación de copia del valor de retorno en una activación final de función, según el tamaño del valor de retorno	242
Tabla 6.2.32. Costes de las distintas clases de secuencias de llamada para activaciones fmales de funciones, usando la cadena estática	243
Tabla 6.2.33. Razones de optimación (usando la cadena estática) para las distintas clases de activaciones finales de una función	243
Tabla 6.2.34. Primeros valores de la razón de optimación (usando la cadena estática) para una activación final no recursiva de función según la diferencia de anidamiento.....	244
Tabla 6.4.1. Coste de la secuencia de llamada recursiva estándar de procedimiento (usando la cadena estática) cuando se utilizan instrucciones de manejo de la pila máquina y de subrutinas (entre paréntesis aparecen los valores originales, cuando no se usan esas instrucciones, de la Tabla 6.2.2 y la Tabla 6.2.5).....	285

1. INTRODUCCIÓN

1.1 ÁMBITO, MOTIVACIÓN E INTERÉS DEL TRABAJO

El trabajo doctoral que aquí se presenta está enmarcado en el área de *lenguajes de programación*, y más en concreto en lo que se refiere a su *implementación y compilación*. A grandes rasgos, la principal motivación del mismo es la aparente "contradicción" existente entre la teoría y la praxis en el uso cotidiano de un lenguaje de programación tradicional. En efecto, la teoría de la programación hace gran hincapié en considerar el concepto de *recursividad* como uno de los dos mecanismos fundamentales para la expresión de los programas, siendo el otro la utilización de construcciones específicas de control que realizan bucles, lo que llamaremos *iteratividad*.

Sin embargo, aún hoy sigue existiendo la impresión de que los lenguajes de programación "no se llevan bien" con los programas recursivos, es decir, que las virtudes que se asignan a las soluciones recursivas (claridad, legibilidad, alta abstracción, facilidad de verificación y análisis formal) quedan empañadas por el inconveniente de ser, según parece, soluciones ineficientes tanto espacial como temporalmente.

Nuestra conclusión es que se produce una especie de "contradicción de estrategias" en el programador, situación que es especialmente grave, como es obvio, en los estadios iniciales del aprendizaje de la programación: primero se le adiestra con énfasis en una técnica de programación, la recursividad, que luego se califica, con mayor o menor insistencia, durante sus mismos estudios, de ineficiente. El estudiante consigue desarrollar, aunque sea inconscientemente, un cierto "rechazo" a la recursividad (en mayor o menor medida dependiendo del énfasis que se haya puesto en la presentación de sus

virtudes y desventajas), rechazo que, en cualquier caso, los compiladores de los lenguajes habituales se encargan de fortalecer en la práctica profesional.

Si el lector admite como válido el anterior análisis, estará probablemente de acuerdo en que deben realizarse esfuerzos para poner remedio a esta situación. Dadas como hemos dicho las buenas cualidades de la recursividad para acomodarse a una teoría moderna de la programación que entiende que esta tarea no consiste *solo* en "escribir programas", sino también en practicar su análisis, verificación, derivación, mantenimiento, *etc.*, y dado también que la iteratividad no parece acomodarse tan bien en todos los casos a tal teoría, el objetivo ha de ser mejorar el principal inconveniente de los programas recursivos, es decir, mejorar su rendimiento o comportamiento dinámico.

Desde luego, somos conscientes de que estamos planteando una cuestión antigua en Informática, que incluso algunas personas podrían decir, y nosotros lo admitimos, que ha sido resuelta ya, de diferentes modos, a lo largo de los últimos veinte años. Sin embargo, y esta es una postura que puede resultar chocante, mantenemos que han sido casi todas ellas soluciones "parciales", sin que ello desmerezca para nada la gran importancia de las mismas. Mantenemos, en efecto, la parcialidad de tales soluciones desde la perspectiva del impacto real que han causado en general en la metodología y uso cotidiano de la programación.

La postura anterior está basada en una cuestión metodológica que creemos importante reseñar sin pérdida de tiempo. Es un proceso bastante común la generalización de los resultados del trabajo científico en un cierto campo. Sin embargo, a menudo, esta generalización olvida detalles determinantes que, al ser considerados, impedirían la misma, o al menos la complicarían. Un ejemplo conocido es lo ocurrido en inteligencia artificial: la euforia inicial a que condujeron los resultados de muchos proyectos, unida a la facilidad para vislumbrar el objetivo deseable ("máquinas pensantes"), hizo que se olvidara que tales trabajos al fin y al cabo resolvían mayoritariamente *toy problems*, lo que llevó a una fase posterior de cierto desencanto al no obtenerse un acercamiento al objetivo final ni una mejora sustancial de resultados en un espacio amplio de tiempo¹.

¹ Winograd, T. (1991) «Máquinas pensantes: ¿son posibles?, ¿lo somos?», *Revista de Occidente* 119, abril: 113-150.

Los conocedores del problema de la ineficiencia de los programas recursivos probablemente recuerden que se han hecho grandes esfuerzos orientados a la eliminación de la recursividad. Nuestra postura es que se conocen estrategias para la transformación de esquemas recursivos muy variados (para empezar, no todos), pero mantenemos que la mayor parte de ellas son estrategias heurísticas, es decir, de difícil mecanización en algunos casos. Gran parte de la literatura consultada (como veremos ya en el cuerpo de la tesis) presenta estas estrategias con ayuda de ejemplos, plantea la generalización y concluye que ello puede mecanizarse (cuando operaciones concretas de tal generalización quedan ciertamente sin la precisión necesaria). Algunos de estos estudios han dado lugar a sistemas (no del todo) automáticos que se aplican a lenguajes con unas características muy restrictivas. Es en este sentido en el que pensamos que, siendo estos trabajos fundamentales para el desarrollo de la metodología que ahora se reconoce como necesaria en la teoría de la programación, no se puede afirmar que hayan resuelto *completamente* el problema.

De la misma manera, el lenguaje SCHEME, descendiente de LISP, es capaz de proporcionar eficiencia a los programas que siguen el esquema más sencillo de recursividad, lo que se llama *recursividad final*. Ello no es por una búsqueda específica de tal clase de recursividad, sino gracias a su estrategia de implementación, notablemente separada de la que utilizan lenguajes imperativos tradicionales como PASCAL. Entonces, en SCHEME puede identificarse recursividad final con iteratividad. Alguien podría decir: ya se ha demostrado que *un* lenguaje de programación puede hacer que los programas recursivos finales sean eficientes, por lo que eso debe ser cierto de *cualquier* lenguaje de programación. Las diferencias entre PASCAL y SCHEME, no siendo tantas como las que existen entre PASCAL y el lenguaje FORTRAN de 1960, sí nos parecen suficientes como para que lo anterior no nos parezca obvio.

Esta es la motivación del presente trabajo. Los lenguajes de programación tradicionales, imperativos, de los que PASCAL es el paradigma, no cuentan aún con herramientas de ninguna clase, de uso extendido y cotidiano, destinadas a mejorar la ineficiencia que parecen tener los programas recursivos. Nuestro objetivo es incorporar una de esas posibles herramientas al lenguaje.

Una "tesis", en su sentido más amplio, es "una proposición que se mantiene con razonamientos". La presencia de una proposición

(sintácticamente completa) no es, desde hace muchos años, habitual en los títulos de los proyectos doctorales, lo cual sería deseable en la medida que esta clase de trabajo debe responder a una serie de criterios como interés, certeza, espíritu crítico, etc.. Por ello, no queremos olvidar esta acepción, y presentamos un título que, recogiendo el oficial, es más acorde con lo dicho:

"Gran parte de los programas recursivos finales en PASCAL pueden ejecutarse de manera eficiente como resultado de técnicas de compilación alternativas"

Desgajaremos esta proposición para presentar las ideas concretas que se van a encontrar en este trabajo.

En primer lugar, nos dedicaremos a los *"programas recursivos finales"*. Hemos dicho que la recursividad final es la clase más simple de esquema recursivo. Pero no solamente es esa la razón para elegir esta clase de programas recursivos. El objetivo último de la metodología de programación aludida en un párrafo anterior consiste justamente en la obtención de programas recursivos finales a partir de aquellos que no lo son. Con ello, la conversión de la recursividad final en iteratividad se convierte en una especie de "cuello de botella" de tal metodología, pues ese proceso se tiene por sencillo y casi obvio, y mostraremos que no lo es tanto (al considerar ciertos detalles de los lenguajes de programación). Consiguiendo la implementación eficiente de los programas recursivos finales, desaparece el cuello de botella y se evita el último paso obligado de la metodología de programación, lo que fundamenta el interés de este trabajo.

Segundo, mantenemos que *"pueden ejecutarse de manera eficiente"*. Ello significa dos cosas. Por un lado, no va a existir un consumo creciente de la memoria durante el proceso recursivo: este se ejecutará en un espacio constante, básicamente el mismo que precisaría el programa iterativo equivalente. Por otro lado, en muchos casos el coste temporal de los procesos recursivos finales será bastante menor que en la implementación clásica, y lo que es más importante, muy cercano al coste temporal del programa iterativo correspondiente. Son claras las implicaciones que esto puede tener sobre el estilo de programación, como es conocido en SCHEME, y amén de lo dicho en el párrafo anterior.

Ello se consigue *"como resultado de técnicas de compilación alternativas"*, es decir, se adopta la perspectiva existente en el lenguaje SCHEME para resolver el problema, en la cual la propia estrategia de implementación del lenguaje lleva a

una compilación que tiene el efecto reseñado sobre los programas recursivos finales. En nuestro caso, esto significa realizar una modificación de la implementación clásica del lenguaje, modificación que, sin embargo, no es drástica, sino que pretendemos que pueda ser adoptada fácilmente por compiladores ya existentes. La técnica para conseguir el objetivo marcado se entiende meramente como una *optimización* dentro del compilador, pero que no impondrá fuertes requisitos en la estructura de este (o al menos, no más de los que se precisan para practicar otras optimaciones ya habituales), ni supondrá una disminución notable de su eficiencia.

Lo comentado hasta ahora debe completarse diciendo que el término "*programa recursivo final*" ha de entenderse en un sentido laxo. No solamente van a ser los programas en que existe una recursividad directa los que resulten mejorados, sino también programas en que existe recursividad indirecta (o mutua). Y no solamente los programas que sean estrictamente recursivos finales en este último sentido, sino en general, los programas que contengan llamadas recursivas finales, aunque contengan, por ejemplo, otras llamadas recursivas no finales. En este último caso, sin embargo, no será cierto que no se produce el crecimiento de la memoria necesaria conforme avanza el proceso, pero las necesidades serán menores. Tal resultado es debido a que, como en SCHEME, las técnicas que solucionan el problema son más generales que una mera búsqueda específica de posibles esquemas recursivos, al estar orientadas a un tratamiento alternativo de toda llamada final, esté o no inmersa en un proceso recursivo (lo que por otro lado, es posible que ni siquiera llegue a conocer el compilador).

Elegimos "*PASCAL*" como caso paradigmático del grupo de lenguajes imperativos, suficientemente rico como para presentar una serie de elementos que se supone pueden influir en el problema, elementos que están presentes en la mayoría de lenguajes de la clase considerada (el lenguaje C, por ejemplo, evidentemente no sería adecuado para este propósito). Por otro lado, el lenguaje PASCAL es todavía uno de los lenguajes más utilizados en los ámbitos educativo e industrial, con lo que se añade un interés práctico al trabajo.

Finalmente, son "*gran parte*" de los programas recursivos finales los que tendrán el comportamiento dinámico descrito, pero lamentablemente *no todos*. Este resultado es importante, pues no se habla en el lenguaje SCHEME de que exista parcialidad alguna en el tratamiento de los programas recursivos finales (apoyando por cierto nuestra hipótesis de que la aplicación de las ideas de

SCHEME al lenguaje PASCAL no era trivial). Encontraremos que existen varias restricciones, es decir, casos en que el tratamiento alternativo de las llamadas finales no es posible con nuestra técnica. Debemos investigar las causas de dichas restricciones y analizar el impacto que tendrán en el objetivo propuesto, aunque ya podemos anticipar que solo una de ellas puede considerarse realmente como una restricción de cierta fuerza.

El proyecto doctoral se plasma, a efectos prácticos, en la construcción de un compilador de PASCAL con las características reseñadas. El compilador que se ha construido es un prototipo, sin ánimo de ser competitivo todavía. Se restringe al subconjunto del lenguaje directamente relacionado con el problema (pero son pocos los aspectos excluidos) y genera un código virtual que es posteriormente interpretado. Se ha escrito en SCHEME, lo cual también es un aspecto novedoso a reseñar, pues se ha encontrado que este lenguaje se muestra muy adecuado para el diseño de compiladores, al menos en la fase de prototipado (y por consiguiente, también, de cara a la docencia de compiladores).

A continuación, antes de entrar en lo que es el cuerpo de la tesis doctoral, presentamos sinópticamente su estructura y contenido.

1.2 ORGANIZACIÓN DE LA MEMORIA

La memoria que presenta el trabajo doctoral realizado se puede dividir en tres grandes partes, que configuran los estadios a través de los que se desarrolló el proyecto: estudio de los campos involucrados en el problema y concreción definitiva del trabajo, desarrollo del mismo, y obtención de los resultados. Estas partes se estructuran a su vez en los capítulos que se presentan a continuación.

El capítulo 2 recoge de una manera más detallada la presentación del problema, profundizando en las cuestiones que se han avanzado en esta introducción. Primero, se realiza un análisis crítico de ciertas actitudes que, aún hoy, existen al respecto de la dicotomía "iteratividad *versus* recursividad", y de la conclusión de que los programas recursivos son ineficientes. Luego, se examinan en detalle las dos soluciones que se han barajado para este problema.

Se termina con la concreción del trabajo, las hipótesis de partida, y los objetivos específicos que nos proponemos alcanzar.

El desarrollo de la técnica alternativa de compilación de PASCAL que nos permitirá mejorar las prestaciones de los programas recursivos finales se distribuye en tres capítulos, teniendo en cuenta que en PASCAL los subprogramas (agentes de la recursividad) pueden ser de dos tipos: procedimientos o funciones. El capítulo 3 estudia el caso más sencillo de recursividad final directa de procedimientos, basándose para ello en las técnicas *ad hoc* conocidas de eliminación de la recursividad. Las conclusiones que se obtienen se pueden aplicar, sin embargo, a todos los demás casos. El capítulo 4 analiza el caso más general de llamada final de procedimiento no directamente recursiva. El capítulo 5 se dedica al mismo propósito que los dos anteriores pero en lo referente a las funciones.

La evaluación de todos los aspectos involucrados en la técnica desarrollada se recoge en los capítulos 6 y 7. El primero de ellos comienza con el análisis del comportamiento dinámico de los programas optimados, tanto desde el punto de vista espacial como temporal. Después se estudian los requisitos que han de cumplir los compiladores de PASCAL, en cuanto a su estructura, para poder llevar a cabo la optimación. A continuación se considera un amplio elenco de posibles variaciones de las hipótesis adoptadas en relación con la implementación del lenguaje, estudiando su influencia en los resultados conseguidos. Finalmente, se describen las características más significativas del compilador de PASCAL que se ha construido.

El capítulo 7 representa la entrada en la "recta final" del trabajo. Se discutirán algunos aspectos relacionados con la cuestión genérica de la programación, tal como se han avanzado al principio de esta introducción. Se examinará el alcance de las restricciones y se enmarcará la optimación en la metodología de programación comúnmente aceptada hoy en día.

La memoria se completa con las conclusiones finales del trabajo, que constituyen el capítulo 8, la bibliografía utilizada o de relevancia para el proyecto, y un apéndice en el que se incluye parte del código fuente del compilador, concretamente el módulo de detección de las llamadas optimables y el módulo de generación de código.

Sin más, pasamos ya al cuerpo de la tesis, comenzando, como hemos dicho, por la presentación del estudio preliminar y la fijación definitiva de los objetivos e hipótesis de trabajo.

2. BASES TEÓRICAS Y ANTECEDENTES

2.1 INTRODUCCIÓN

Este primer capítulo del trabajo presenta el campo en el que se enmarca, la motivación del mismo, los objetivos generales y específicos y las hipótesis de partida para acometer su desarrollo en profundidad. Hemos considerado conveniente extendernos en algunos aspectos, pero no con el ánimo de cansar al lector, sino para que se comprenda bien nuestra posición en ciertas "lecturas críticas" de algunos trabajos que nos preceden, con la misma motivación genérica que el nuestro (críticas que, hemos de dejar claro, en ningún momento han de verse como una infravaloración de su importancia).

La primera sección a continuación presenta las ideas generales de la recursividad como técnica de programación. Se llega a la conclusión de que todavía hoy "recursividad" es entendido como sinónimo de "ineficiencia", aunque tal resultado es puesto en tela de juicio mediante diversos argumentos. Las dos secciones siguientes muestran sendas soluciones que existen al problema anterior. La primera, la transformación de programas, es importante por haber dado origen a una metodología de programación, aceptada actualmente hasta el punto de su inclusión definitiva en los estudios de Informática; tal solución trata de adoptar un enfoque general del problema, pero mantendremos que falla a la hora de producir una mecanización del proceso destinado a mejorar las prestaciones de los programas recursivos.

La segunda solución, existente en el lenguaje de programación SCHEME, aborda el caso más simple de recursividad, pero desde la perspectiva de la implementación del lenguaje, como mero efecto de la compilación. A pesar de resultar un caso simple, constituye justamente el objetivo de la solución

anterior, y por ende, de la metodología de programación reseñada, con lo que se puede evitar el recurso a la iteratividad, paso final de otra forma obligado del método. En este estado de cosas se planteará definitivamente el trabajo, en la sección que finaliza el capítulo.

2.2 RECURSIVIDAD Y PROGRAMACIÓN

Es ya casi un lugar común decir que recursividad y programación son dos conceptos profundamente relacionados, además desde diversas perspectivas. Desde el punto de vista teórico, la *recursividad* es el hilo conductor que une a todos los intentos, llámense máquina de Turing, funciones recursivas de Gödel y Kleene, cálculo *lambda* de Church, *etc.*, de capturar el significado de "algoritmo" o "proceso efectivo de cálculo" [Davis 1965; Gómez y Díaz Labrador 1995]. Si se entiende simplemente (aunque quizá no se recojan todas las posibles implicaciones) que la *programación* es el conjunto de técnicas que permiten la expresión de algoritmos para la solución de problemas mediante el computador, es indudable que equiparamos el concepto de *lenguaje de programación* (o más en general deberíamos decir "herramientas de programación") con el de máquina de Turing, función recursiva, y demás, por lo que la recursividad, teóricamente entendida, ha de ser también aquí el hilo conductor.

Sin embargo, así como la recursividad se manifiesta (sintácticamente, podríamos decir) en la máquina de Turing mediante la posibilidad de efectuar transiciones cíclicas en un mismo estado o entre diferentes estados, en las funciones recursivas a través de un operador primitivo, *etc.*, en los lenguajes de programación se manifiesta de dos modos principales, cada uno *alter ego* de los ejemplos teóricos elegidos: primero, por la posibilidad de expresar la solución a un problema mediante una construcción que implica la realización cíclica de un determinado proceso (*programa iterativo*), y segundo, por la posibilidad de que el nombre dado a la solución aparezca en la propia definición (*programa recursivo*). Es importante recalcar que estamos refiriéndonos al aspecto puramente sintáctico de los programas: la calificación de un programa como iterativo o recursivo surge a partir de las características observables del texto

del mismo. Por lo tanto, cuando exista posibilidad de confusión, deberíamos referirnos a la recursividad como concepto matemático, o a la recursividad como propiedad sintáctica de ciertos programas; normalmente, el término "recursividad" se tomará ya desde ahora por esta última acepción.

Desde el punto de vista propiamente técnico de la programación, por lo tanto, se ofrecen dos mecanismos alternativos para la expresión de las soluciones a los problemas. "Alternativo" no significa en este caso "excluyente", pues un programa puede presentar (en la mayoría de los lenguajes de programación) simultáneamente ambas propiedades, iteratividad y recursividad. Significa más bien una clara diferenciación entre dos *estilos* de programación. Parece que no podemos escapar a una vieja polémica, pero lamentablemente debemos entrar en ella si queremos dar sentido a este trabajo.

Es importante empezar recordando que cada mecanismo es en sí mismo suficiente: cualquier programa podría escribirse utilizando solo iteratividad, como parece demostrar el teorema de Boehm y Jacopini [1966], y cualquier programa podría escribirse utilizando solo recursividad, según McCarthy [1960]. Sin embargo, ciertos argumentos no son demasiado válidos, a nuestro entender. Niklaus Wirth, por ejemplo, afirma:

«The fact that implementations of recursive procedures on essentially non-recursive machines exist proves that for practical purposes every recursive program can be transformed into a purely iterative one» [Wirth 1986a: 140]

Bajo nuestro punto de vista, las "máquinas esencialmente no recursivas" muestran que tanto la iteratividad como la recursividad son, "a efectos prácticos", innecesarias. Wirth parece entender que, en un lenguaje de programación como PASCAL, si un programa no es recursivo sintácticamente, entonces el programa es iterativo. Nosotros no lo entendemos así, es decir, no consideramos que, siguiendo con PASCAL, la sentencia condicional sea una construcción iterativa. De hecho, McCarthy [1960] demostró realmente que cualquier programa podría escribirse con condicionales, secuencias y llamadas a procedimientos (mecanismo para la expresión de la recursividad en el programa). Y es trivial decir que cualquier programa podría escribirse en PASCAL solo con condicionales y el salto incondicional, y nosotros no diríamos que tal programa es iterativo o recursivo: simplemente, sería un feo programa.

La visión actual de la programación entiende que se trata de una tarea compleja en la que el hecho de "escribir un programa" es solo una parte (y diríamos pequeña) de la misma. La corrección de los programas, planteada como una necesidad durante años por autores como Hoare, Dijkstra, el propio Wirth (en descargo de la "crítica" anterior) y otros muchos, ha acabado reconociéndose como uno de los criterios fundamentales que ha de guiar la programación. El método de *derivación formal* de programas de Dijkstra [1976], en el que la construcción del programa y su verificación se producen simultáneamente, se ha enriquecido notablemente desde entonces, y se ha integrado definitivamente en los estudios de Informática [Berlioux y Bizard 1986; Manber 1989; Díaz de Ilarraza y Lucio 1990; Dahl 1992; Balcázar 1993]. Es a la luz de esta filosofía en la que hay que situar ahora el viejo dilema entre iteratividad y recursividad.

Los lenguajes de programación deben entonces ofrecer facilidades para expresar los programas que se consiguen según esta metodología moderna. Se están haciendo grandes esfuerzos para incorporar un número cada vez mayor de facilidades [Botella, Burgués y Franch 1988; Franch, Burgués y Botella 1993; Franch y Burgués 1993], en sistemas que más bien habría que calificar de "herramientas de programación", como hemos hecho antes. Pero si reservamos el concepto de "lenguaje de programación" para su sentido clásico (o sea, PASCAL, LISP, etc.), los dos mecanismos reseñados son los que se encuentran a disposición del programador para la expresión de los programas, y son los que se encuentran, en diversos grados, en los diferentes textos reseñados. Sin olvidar ninguna de las técnicas, lo cierto es que algunos presentan un enfoque principalmente iterativo [Dromey 1989; Cormen, Leiserson y Rivest 1990], otros un enfoque primordialmente recursivo [Berlioux y Bizard 1986; Balcázar 1993], y los más un término medio entre ambas filosofías [Wirth 1986a; Sedgewick 1988; Manber 1989; Díaz de Ilarraza y Lucio 1990; Dahl 1992; Eguíluz 1993], pero el denominador común de todos ellos es el uso de una notación más o menos asimilable a un lenguaje como PASCAL. Mención aparte debería hacerse de Abelson, Sussman y Sussman [1985], todo recursividad de la primera a la última página, pero por una razón distinta: usa LISP.

Pocos textos entre los reseñados que utilizan el lenguaje PASCAL o alguna notación cercana, hasta donde llega nuestro conocimiento de los mismos, son explícitos acerca de los aspectos del dilema que nos ocupa, y ello parece deberse

a que se adopta la misma postura que al final será la nuestra: iteratividad y recursividad son dos medios útiles para la expresión de programas y no hay razón para desechar ninguno. Sin embargo, siguiendo el rastro de la propia clasificación que se ha realizado podremos encontrar otros datos que nos interesan.

En general, en la totalidad de los textos que tocan este asunto, las construcciones iterativas parecen surgir de una forma natural, intuitiva, y que no necesita mayores comentarios (ni ejemplos) que el habitual *simil culinario* ("cocer durante diez minutos"); rápidamente, se pasa a presentar todas sus variedades (*while*, *repeat*, *for*), y a recalcar lo bueno que es disponer de las construcciones iterativas por permitir que un programa de longitud limitada pueda expresar un proceso mucho más largo, quizá no terminante. Por contra, la presentación de la recursividad parece requerir bastantes más explicaciones: casi todos coinciden en que es un concepto que puede resultar paradójico, difícil de aprehender, *etc.* (todo tiene su lado bueno: aquí se han producido gloriosas manifestaciones pictóricas del concepto, por ejemplo la de Wirth [1986a: 135]), y para el cual se necesita acudir a las matemáticas (relaciones de recurrencia, definiciones recursivas, y demás) y presentar al menos dos o tres ejemplos, entre los cuales el factorial, los números de Fibonacci o la exponenciación son casos obligados.

Después de presentar el concepto de recursividad, se pasa a justificar su necesidad. El argumento más utilizado es que la recursividad (sintáctica) es apropiada cuando el proceso a realizar, o los datos involucrados, están descritos de forma recursiva: se habla de la mayor "naturalidad" de ciertas soluciones recursivas, y se dice explícitamente que las soluciones iterativas correspondientes serían oscuras y artificiosas. También es común mencionar que en los programas recursivos es, por lo general, más fácil analizar su rendimiento y verificar su corrección.

Por último, la mayoría de los autores terminan la presentación del concepto de recursividad aludiendo explícitamente, y con gran énfasis en muchos casos, a la ineficiencia que parece estar asociada a este mecanismo de programación. El caso extremo de esta idea está presente en el siguiente extracto de un reciente libro de texto introductorio a la programación, justificando la necesidad de conocer bien el lenguaje de programación que se utiliza:

«Muchos lenguajes ofrecen herramientas que, utilizadas en ciertas situaciones, pueden ser muy provechosas para el programador, pero que usadas sin ningún control pueden malgastar grandes cantidades de recursos o comportar errores lógicos. Un ejemplo *claro* de esto es el uso de la recursividad (informalmente diríamos que es el uso de lo que se está definiendo en la propia definición, por ejemplo al definir una acción o una función), que bien usada puede ser una herramienta de programación muy potente, pero que usada innecesariamente puede provocar un fuerte incremento en el tiempo de ejecución» [Castro *et al.* 1993: 122; la cursiva es nuestra]

Es decir, sin haber siquiera presentado el concepto de recursividad (de hecho, creemos haber recogido en la cita anterior la única aparición de tal palabra en la obra, y no tenemos constancia de que contenga ningún programa recursivo), ya existe una preocupación por *aclarar* de una vez por todas que “lo que sea eso” puede llegar a ser ineficiente. Otros textos son menos drásticos en esta advertencia (al menos definen y justifican la recursividad), pero la filosofía común se mantiene:

- Sedgewick [1988] dedica diez páginas a la presentación de la recursividad (51-60), incluyendo soluciones sintácticamente iterativas equivalentes a los ejemplos que propone, y las cinco siguientes (61-65) a las estrategias para su eliminación (después de haber mostrado solo cinco programas recursivos), lo que recomienda vivamente;
- Wirth [1986a] presenta los conceptos básicos en tres páginas (135-137) y rápidamente pasa a una sección titulada literalmente «WHEN NOT TO USE RECURSION» (138-140), donde incluye los ejemplos del factorial y los números de Fibonacci.

Y eso que estos textos son de los que más espacio dedican posteriormente a la recursividad como técnica de programación. Parece necesario, por tanto, estudiar las razones de la ineficiencia de la recursividad.

2.2.1 Comportamiento de los programas recursivos

Es un hecho bien conocido (desde que se comprende el concepto de recursividad en relación con la programación, como hemos podido comprobar) que los programas recursivos tienen un principal inconveniente: en palabras

llanas, se asocia la idea de un programa recursivo con un proceso que consume memoria y no es eficiente temporalmente. Sin embargo, la cuestión no puede reducirse a la anterior aseveración: el viejo dilema "iteratividad *versus* recursividad" ha producido numerosas páginas apoyando una y otra concepción de la programación, y aún en estos días, parecen seguir existiendo dos posturas, no diríamos tanto que enfrentadas, pero sí al menos *diferentes*, no solo en cuanto al estilo de programación, sino también en cuanto al estilo de lenguaje que propicia tal tarea (cuestión obviamente vinculada a la primera, pero no tan *directamente* como parece).

Por un lado, los lenguajes descendientes de ALGOL, conocidos como *lenguajes imperativos*, siguen basando mayoritariamente su filosofía de la programación en las construcciones iterativas (sintácticamente hablando). Por otro lado, el lenguaje LISP y sus descendientes, junto con toda la gran familia de *lenguajes funcionales*, abogan por un uso más intensivo de la recursividad, o más en general, del concepto de subprograma (procedimiento, función o como quiera llamarse) para expresar el control. En otro plano, existe la impresión (el mito, podríamos decir) de que los lenguajes imperativos producen código eficiente, mientras que los lenguajes como LISP y en general del paradigma funcional logran altas cotas de expresividad y abstracción a costa de un comportamiento ciertamente peor en la mayoría de los casos. Es un error, sin embargo, a nuestro entender, establecer un paralelo entre ambas cuestiones, es decir, culpar exclusivamente al uso más intenso de soluciones recursivas de estas diferencias de comportamiento dinámico.

La implementación tradicional de un lenguaje imperativo muestra las causas de estas ideas acerca del pretendido peor comportamiento de los programas recursivos. Nos extenderemos en presentar los detalles de esta implementación, pues serán necesarios en secciones posteriores de este estudio, y en general, constituyen uno de los puntos de partida de nuestro trabajo, por lo que será útil fijar la terminología que se usará a lo largo de esta tesis.

Los programas que produce un compilador de un lenguaje imperativo presentan una serie de características que englobaremos bajo la denominación genérica de *modelo dinámico* del lenguaje. Esto incluye diferentes aspectos como la estructura del propio programa objeto, el modo y el mecanismo de vinculación de los datos, los procesos de activación y terminación de los subprogramas, *etc.* Todo ello está recogido en cualquier texto clásico de

compilación [Aho, Sethi y Ullman 1990; Fischer y LeBlanc 1988], o de lingüística de la programación [MacLennan 1987; Sethi 1989].

La estructura básica de un programa objeto consiste en un conjunto de instrucciones ejecutables (el código) y un almacén para los datos. Las entidades más características de los lenguajes de programación (y no solo nos estamos refiriendo a los lenguajes imperativos) son el *bloque*, agente principal del encapsulamiento de los datos necesarios por una parte del programa, y el *subprograma*, agente del control y de la modularidad o separación de tareas en el programa. Ambas entidades suelen ser muchas veces un solo elemento del lenguaje, como ocurre en PASCAL: los bloques son definidos precisamente por los subprogramas; que se puedan definir bloques de otras formas (como en ALGOL) no es importante para nuestro propósito, así que supondremos el caso de PASCAL. Por lo tanto, entenderemos el concepto de subprograma en su doble vertiente de gestor de datos y agente del control en el programa.

Un programa se puede entender entonces como un proceso de sucesivas activaciones y terminaciones de subprogramas. Cada activación de un subprograma supone la aparición de nuevos datos, justamente los que se definen en el bloque establecido por tal subprograma. Del mismo modo, en los lenguajes imperativos al uso, la terminación de un subprograma supone no solo el salto a otro punto del código, sino también la desaparición de los datos que contiene como bloque. Las activaciones y terminaciones de los subprogramas no ocurren en secuencia sino de forma anidada: un subprograma puede activar a otro (incluso a él mismo), por lo que en un momento determinado, pueden estar activados sin terminación varios subprogramas (aunque solo el último se está ejecutando realmente, y los demás se encuentran a la espera de que el siguiente termine). Cada subprograma debe conocer entonces una dirección de retorno a la que saltar cuando termine, y puede necesitarse mantener un número arbitrario de direcciones de retorno (una por cada subprograma sin terminar). Es fácil entender que estas direcciones de retorno pueden acumularse en forma de pila: cuando un subprograma es activado, se acumula una dirección de retorno, mientras que cuando un subprograma termina, desapila la dirección de retorno y salta a ella. Así surge una de las estructuras básicas de los programas, que al albergar información relativa al flujo de instrucciones que ha de ejecutarse, se denomina *pila de control*.

La activación y terminación de un subprograma también provoca, como se ha dicho, la aparición y desaparición, respectivamente, del conjunto de datos del bloque definido por el subprograma, lo que llamaremos el *ambiente* (preferiríamos esa palabra antes que "entorno" para el inglés *environment*). Por ello, es posible asociar la misma estructura anterior, la pila, con el almacén necesario para dicho ambiente. Así, la pila llamada antes de control contendrá información relativa específicamente al control (direcciones de retorno), pero también la información de ambiente (los datos), así que convendría mejor llamarla *pila de activaciones*. En esta pila, cada activación de un subprograma acumula una entidad, conocida como *registro de activación*, que contiene ambas clases de información; la terminación del subprograma desapila el registro de activación de la cumbre de la pila, saltando a la dirección de retorno en él contenida. Por ello, la estructura clásica definitiva de un programa objeto en un lenguaje imperativo es la que se muestra en la Figura 2.2.1, expresión de la estrategia de almacenamiento de los datos conocida como *vinculación mediante una pila* (en inglés, *stack allocation*).

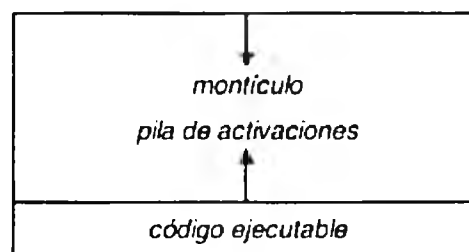


Figura 2.2.1 Estructura clásica de un programa objeto en un lenguaje imperativo, con vinculación mediante una pila

El *montículo* (en inglés *heap*) es el almacén de los datos que surgen dinámicamente durante la ejecución del programa, y que no son definidos estáticamente en los bloques. En principio, no es una estructura esencial para nuestro propósito en un lenguaje imperativo (aunque sí en lenguajes como LISP).

Una serie de detalles de esta implementación son precisos. Primero, se supone que existe dinámicamente un apuntador global que en todo momento señala al registro de activación que está en la cumbre la pila de activaciones (dependiendo de las características de los lenguajes, puede necesitarse más de un apuntador para este propósito). Segundo, a veces es necesario que cada

registro de activación conozca la situación del que le precede (hacia la base de la pila): un elemento del registro de activación, llamado *enlace de control*, cumple tal propósito; el seguimiento sucesivo de estos apuntadores da lugar a la *cadena dinámica*. Tercero, cada subprograma puede acceder a datos no locales (no definidos en su bloque); para ello, los lenguajes imperativos observan habitualmente *reglas de ámbito léxico*, que determinan los datos accesibles a partir de la estructura textual (estática) del programa fuente. Entonces, los datos accesibles pueden no encontrarse dinámicamente en el registro de activación precedente, pero es posible determinar estáticamente el desplazamiento relativo que ha seguirse: el *enlace de acceso* es otro elemento de un registro de activación, que señala al registro de activación más cercano (hacia la base de la pila) del superior léxico; el seguimiento sucesivo de estos apuntadores da lugar a la *cadena estática* (existe otra estrategia para este propósito, conocida como *display*, pero la diferencia no es esencial). Con todo ello, se obtiene la composición típica de un registro de activación, aunque su estructura exacta puede variar respecto a la presentada en la figura siguiente.

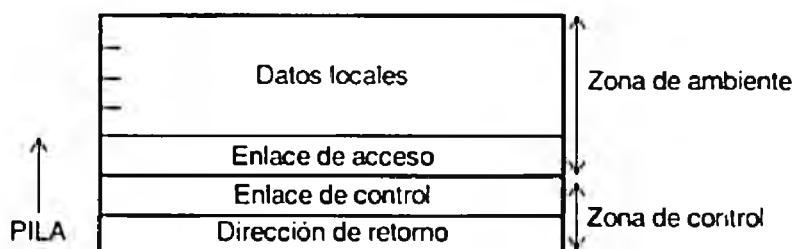


Figura 2.2.2 Estructura típica del registro de activación de un subprograma en un lenguaje imperativo

La activación de un subprograma supone la acumulación de un nuevo registro de activación en la pila y el salto a la primera instrucción del subprograma. La terminación de una activación supone la desaparición del registro de activación de la cumbre de la pila y el salto a la dirección de retorno en él contenida. Denominaremos *subprograma llamador* a aquel que genera la activación (invoca o contiene la llamada), y *subprograma receptor* al que es activado (recibe la llamada o es llamado). La construcción y desaparición del registro de activación correspondiente a cada llamada es normalmente responsabilidad compartida entre el subprograma llamador y el receptor, denominándose *secuencia de llamada* al conjunto de operaciones que producen la

activación y *secuencia de retorno* a las que la terminan. Más adelante, al fijar las hipótesis de trabajo, se ofrecerán con todo detalle estas cuestiones.

Con estas ideas básicas acerca de la implementación de los lenguajes imperativos, estamos preparados para retomar la discusión planteada al principio de la sección: entender la razón del peor comportamiento de los programas recursivos. El agente de la recursividad es la activación de un subprograma, básicamente el mismo subprograma que genera la activación, lo que denominaremos *recursividad directa* (aunque también existe como caso más general la *recursividad indirecta*, cuando hay ya una activación del subprograma que es llamado, pero no es la que efectúa la llamada). Cada activación consume un espacio de la pila de activaciones, y exige un cierto tiempo de proceso (secuencias de llamada y retorno). El proceso recursivo puede extenderse hasta una profundidad arbitraria, con lo que el espacio necesario y el tiempo consumido dan lugar a ese peor comportamiento conjunto.

Ahora bien, peor comportamiento ¿con respecto a qué? El otro agente del flujo del programa en los lenguajes de programación imperativos es la *sentencia de control*. Nos interesan particularmente las sentencias iterativas, agentes de la iteratividad, aunque el argumento se aplica a la sentencia condicional igualmente. El hecho es que la ejecución de una sentencia iterativa no precisa la acumulación de ningún registro de activación, y el control se resuelve por doquier mediante una de las instrucciones menos costosas que incorpora cualquier máquina, la instrucción de ruptura de secuencia (*jump*, para diferenciarla de la sentencia de salto *goto*, aunque conceptualmente sea lo mismo). Siguiendo el argumento anterior, parece que el programa iterativo se ejecutará en un espacio constante (puesto que no se acumula información en la pila de activaciones). En cuanto al tiempo, aun suponiendo que se haga un mismo número de "ciclos" que llamadas en el caso recursivo, el factor de ineficiencia que resulta de comparar la instrucción *jump* y las complejas secuencias de llamada y de retorno provoca a efectos prácticos una ejecución más rápida del proceso iterativo en cualquier caso. Además, la recursividad, en su concepción más general, es un proceso de "ida y vuelta", mientras que la iteratividad es solo de "ida": una vez que se llega al punto culminante, el proceso termina, con lo que queremos decir que la sucesiva terminación de las activaciones del proceso recursivo debe tenerse en cuenta también a la hora de evaluar su comportamiento temporal.

Este es el análisis que habitualmente se realiza para decir que los programas recursivos son ineficientes. Es indudable que parte de lo que hemos dicho es verdad, pero otra parte simplemente no lo es. La discusión de estas ideas, y la vuelta al problema "iteratividad *versus* recursividad", continúa en el siguiente apartado.

2.2.2 Iteratividad y recursividad

La idea general a que muchas veces se reduce (en el sentido filosófico del término) lo anterior es la siguiente: la iteratividad sintáctica significa "computación en espacio constante" y "eficiente en tiempo", mientras que la recursividad sintáctica implica "consumo creciente de la memoria" e "ineficiencia temporal". Se comprende en este caso que la actitud reduccionista haya sido tan criticada en muchos otros planos científicos y humanistas. Porque el dilema no puede simplificarse hasta ese extremo. El error está, ciertamente, en creer que el consumo de memoria de un programa se establece a través del número de registros de activación que se acumulan durante su ejecución.

En general, los problemas tienen, teóricamente, una *complejidad*, tanto espacial como temporal, y es difícil retrotraerse a ella. Si se considera, por ejemplo, un proceso asimilable al recorrido inorden de un árbol binario, es imposible que pueda realizarse sin ayuda de una pila [Bird 1977a; Rohl 1984: 172]. Es decir, las soluciones iterativas de los problemas complejos tienen, en general, consumos de memoria y de tiempo similares (en orden de magnitud) a los programas recursivos correspondientes. Otra cosa es que, cuando se programa la solución iterativa, se recoja explícitamente en esa pila solamente la información imprescindible, mientras que la solución recursiva (que evita el que nos preocupemos de la pila, lo que tiene que redundar necesariamente en una mayor abstracción y legibilidad del programa) añade toda una serie de información, que siendo necesaria, provoca la aparición de un factor de crecimiento adicional.

Tomemos un ejemplo muy significativo, a nuestro entender, de esta idea. Wirth [1986a], tal como hemos señalado anteriormente, reconoce el enorme potencial de la recursividad como herramienta para la programación (de hecho, en descargo de lo que va a ser una crítica, es uno de los autores que más la ha

promovido), pero todavía mantiene una postura cercana a la discutida: rápidamente aborda una sección sobre los casos en que no debe utilizarse. Allí se obtienen dos conclusiones, a saber:

«algorithms which by their nature are recursive rather than iterative should be formulated as recursive procedures» [ibid.: 140]

por un lado, y:

«avoid the use of recursion when there is an obvious solution by iteration» [ibid.: 140]

por otro. Es útil investigar qué le hace llegar a esta última conclusión. Primero, esquemas recursivos lineales como:

«P ≡ IF B THEN S; P END
P = S; IF B THEN P END» [ibid.: 138]

Luego, como no, el factorial. Pero sobre todo nos interesa el siguiente ejemplo: los números de Fibonacci. La versión recursiva que se ofrece es la habitual:

« PROCEDURE Fib(n: INTEGER): INTEGER;
BEGIN
IF n = 0 THEN RETURN 0
ELSIF n = 1 THEN RETURN 1
ELSE RETURN Fib(n-1) + Fib(n-2)
END
END Fib» [ibid.: 139]

en que el número de llamadas (y por lo tanto el espacio y el tiempo consumidos) crece exponencialmente. Acto seguido se ofrece la solución iterativa:

«the Fibonacci numbers can be computed by an iterative scheme that avoids the recomputation of the same values by use of auxiliary variables such that $x = \text{fib}_i$ and $y = \text{fib}_{i+1}$.

$i := 1; x := 1; y := 0;$
WHILE $i < n$ DO $z := x; x := x + y; y := z; i := i + 1$ END» [ibid.: 139]

que ciertamente se ejecuta en un espacio constante (cuatro enteros).

El ejemplo es importante para enseñar que no ha de adoptarse sin más la primera solución que se divisa para un problema, pero tal máxima se aplica tanto a las soluciones iterativas como a las recursivas, es decir, es ortogonal a nuestro problema. De ahí que se ponga énfasis en el análisis del coste espacial y

temporal de los programas, independientemente de la técnica de construcción, y en la obligación de intentar soluciones más eficientes cuando se encuentran tasas de crecimiento no lineales.

Es cierto que la solución recursiva para los números de Fibonacci es altamente ineficiente, pero es que existen otras soluciones recursivas mejores, y lo que es más importante, esas soluciones recursivas aparecen en el proceso de comprensión del problema, en el que la solución iterativa solo es el último paso. Concretamente, la idea de evitar la recomputación de los mismos valores una y otra vez conduce primero a la necesidad de que la función de Fibonacci "conozca" los dos valores previos:

```
PROCEDURE FibAux(n: INTEGER; x, y: INTEGER): INTEGER;
BEGIN
  IF n = 1 THEN RETURN x
  ELSE RETURN Fib(n-1, x + y, x)
END
END FibAux
```

y entonces definir la función como:

```
PROCEDURE Fib(n: INTEGER): INTEGER;
BEGIN
  IF n = 0 THEN RETURN 1
  ELSE RETURN FibAux(n, 1, 0)
END
END Fib
```

Solo habiendo entendido este paso podemos llegar ahora a la solución iterativa con seguridad. En esta solución recursiva la complejidad espacial se ha reducido considerablemente: se trata de una complejidad lineal. Lo que es más importante: la complejidad temporal de las dos soluciones (iterativa y recursiva) es exactamente la misma, solo que los detalles prácticos de la implementación de los lenguajes de programación hacen que el factor de linealidad de la solución recursiva sea mayor, con lo que parece, en efecto, más lenta.

Es en este punto, creemos, en el que deben establecerse las comparaciones del dilema "iteratividad *versus* recursividad": los *programas iterativos más simples* se ejecutan en un espacio constante, y con gran eficiencia temporal, mientras que los *programas recursivos más simples* presentan, en la implementación tradicional, una complejidad espacial lineal, y un peor

comportamiento temporal (causado exclusivamente por los detalles necesarios de la implementación, pues la tasa de crecimiento es la misma). Hacer la comparación entre programas iterativos de consumo constante y programas recursivos "voraces" con el mismo propósito es tan poco correcto como plantear la comparación entre un programa iterativo mal diseñado, con una complejidad espacial y temporal alta, y un programa recursivo simple equivalente. ¿Es más fácil escribir un programa recursivo ineficiente que uno iterativo de las mismas características? La respuesta es probablemente positiva, lo admitimos, pero el programador ha de ser consciente de la complejidad de sus programas, como hemos dicho. Además, la propia pregunta lleva su reverso: si es más fácil escribir un programa recursivo ineficiente, ¿no será porque en general es más fácil escribir un programa recursivo?

Ya tan lejos en el tiempo como en 1977, Guy L. Steele hacía una crítica similar [Steele 1977a], esta vez situándose en la perspectiva del lenguaje LISP. Su postura le permitió llegar a una solución que es antecesora directa de nuestro trabajo venidero, y que deberemos examinar con detalle en una sección posterior de este capítulo. Pero ya debemos anticipar algunos de los argumentos aducidos por Steele. Concretamente, hemos visto que ante dos programas, uno iterativo y otro recursivo, de la misma complejidad temporal, existe un factor de ineficiencia en el último, debido a que la operación resuelta con un simple salto incondicional (*goto*) en la solución iterativa se corresponde con la activación y terminación de un subprograma. Evidentemente, cuanto más costosa sea la implementación de las secuencias de llamada y de retorno en el modelo dinámico del lenguaje, tanto más ineficiente será la solución recursiva. Sin embargo, Steele afirma que:

«The difficulty with the GOTO statement and the procedure call is characterized as a conflict between abstract programming concepts and concrete language constructs» [Steele 1977a]

es decir, que si la recursividad es un concepto fundamental en la programación, deben hacerse esfuerzos para que los programas recursivos tengan un comportamiento atractivo, y no simplemente prohibirlos o restringir su uso a causa de su actual mal comportamiento.

La filosofía que tenemos sobre el concepto de programación parece acercarse más a la que presentan autores como Berlioux y Bizard [1986] o Bálcazar [1993]. Iteratividad y recursividad son dos herramientas poderosas

para la expresión de programas: ninguna ha de ser olvidada; esta idea también se puede atribuir a autores a los que hemos realizado críticas, como Wirth [1986a] o Segdewick [1988]. La diferencia es que el proceso de programar se entiende como una *derivación* (Bálcazar [1993: xii] habla de «calcular los programas»), en la que el esquema más habitual de trabajo consiste en comenzar por obtener una solución recursiva, junto con su verificación completa; este primer programa suele ser sencillo de obtener, y es seguro; ahora se analiza (o se compara con esquemas recursivos conocidos), y si se observa que tiene una alta complejidad, se entra en un proceso de *transformación* que, manteniendo la verificación realizada (como si fuera una demostración matemática), permite disminuir aquella. El objetivo final deseable es, efectivamente, un programa recursivo lineal o uno iterativo, pero ello puede ser más o menos difícil. La manera de realizar estas transformaciones se examinará en la próxima sección, pues justamente surgió a partir de los estudios para la eliminación de la recursividad.

La derivación directa de programas iterativos también es posible, pero los autores citados la reservan para los casos más simples (en que la transformación de recursivo a iterativo sería casi directa), y ambos la presentan *después* de haber mostrado la derivación de programas recursivos. Hay otra coincidencia significativa en la concepción de las construcciones iterativas: tanto Berlioux y Bizard [1986] como Bálcazar [1993] utilizan exclusivamente una sola construcción de esa clase (los primeros la sentencia *while* de PASCAL y el segundo una sintaxis más cercana al *cond* de LISP). Y ello no parece deberse a un ánimo de simplificar la verificación de otras sentencias iterativas presentes, por ejemplo, en PASCAL:

«La mayoría de los lenguajes de programación ofrecen varias formas de denotar una misma construcción, a veces con diferencias sutiles. Por ejemplo, suele haber varias instrucciones de bucle: WHILE, REPEAT, FOR... Lejos de verlo como una ventaja, consideramos que es una causa de errores: el bucle REPEAT, que no puede efectuarse cero veces, reduce flexibilidad sin ofrecer nada a cambio; el bucle FOR, que reduce automáticamente una cota (pero no se sabe si antes o después), dificulta el razonamiento, y facilita olvidar la reducción de la cota en un bucle WHILE que se escriba poco después. Aconsejamos elegir *una sola instrucción* que corresponda a bucles, y usarla sistemáticamente, prescindiendo de las otras posibilidades que ofrezca el lenguaje; lo mismo puede decirse de las alternativas, y de otras instrucciones» [Bálcazar 1993: 405]

En definitiva, se extrae una conclusión similar, en cierto modo, a la de Steele: las consideraciones de eficiencia, siendo importantes como son, lo cual no se niega, no han de prevalecer desde el primer momento en una metodología de programación que intenta aportar a los programas toda una serie de características deseables, como claridad, abstracción, seguridad y, por supuesto, también eficiencia. La cuestión no es que ciertas propiedades de entre las anteriores deban observarse antes que otras: todas son igual de necesarias, y justamente la metodología de programación más correcta será aquella que las desarrolle armónicamente. En este contexto, iteratividad y recursividad son dos mecanismos de expresión igual de necesarios para seguir una tal metodología y aportar a los programas las características deseadas. Pero según se reconoce en casi todas las ocasiones, parece que claridad, abstracción y seguridad son conceptos que tienden a facilitarse con las soluciones recursivas, mientras que es la eficiencia lo que sale mejor parado en las soluciones iterativas.

¿Es posible mejorar la claridad, abstracción y dificultad de verificación de muchos programas iterativos? La recursividad surgió en programación como técnica para facilitar esa mejora, así que la pregunta queda contestada. Ahora bien, ¿es posible mejorar la ineficiencia de los programas recursivos? Esta cuestión se ha abordado desde distintas perspectivas, y se han encontrado algunas soluciones. El estudio de las mismas es el objeto de las dos secciones siguientes.

2.3 SOLUCIONES (I): TRANSFORMACIÓN DE PROGRAMAS

Desde principios de la década de los 70 ha existido una gran actividad en el campo de la eliminación de la recursividad, o sea, la obtención de programas sintácticamente iterativos equivalentes a programas sintácticamente recursivos. Esta actividad comenzó mostrando primordialmente técnicas *ad hoc*, pero pronto se obtuvieron resultados más generales que ampliaron las áreas de aplicación, entroncándose en el campo más amplio conocido como *transformación de programas*. Este campo continua activo [Souquières 1993], pero quizás atendiendo más a otras áreas de interés. Presentaremos las ideas

generales que motivan la transformación de programas, para restringirnos después al área específica de eliminación de la recursividad.

El objetivo último de la transformación de programas proviene de la necesidad de una formalización del proceso de construcción de programas [Feather 1987]. En ese sentido, podría decirse que es una metodología de programación, y llamarla *programación transformacional*, como hacen Partsch y Steinbrüggen [1983]: la construcción de un programa empieza con una especificación, formalizada de alguna manera, que se modifica sucesivamente mediante *transformaciones* que mantienen cierta equivalencia semántica con la especificación original, y finaliza en un programa ejecutable (o admisible para un compilador). Por supuesto, esta filosofía de síntesis de programas, aplicada al desarrollo de sistemas informáticos, se ha integrado últimamente en el campo de la *ingeniería del software* [Botella, Burgués y Franch 1988; Franch, Burgués y Botella 1993; Franch y Burgués 1993; Souquières 1993].

Restringiéndose al desarrollo de algoritmos concretos o programas "sencillos", más que a la construcción de grandes sistemas informáticos (o en todo caso, como parte de ello), encontramos la aplicación primigenia de esta filosofía, que es la que nos interesa principalmente. En ella se reconoce explícitamente el potencial de la recursividad como mecanismo prácticamente único para expresar de un modo sencillo las soluciones algorítmicas a los problemas. Tales soluciones recursivas son "sencillas" en el sentido de que es relativamente fácil llevar a cabo una demostración de la corrección de la solución, es decir, que el programa o algoritmo es correcto para todas las entradas. Sin embargo, normalmente estos algoritmos son "complejos" en el sentido computacional del término: exigen grandes recursos de espacio y/o tiempo, y no solo desde el punto de vista teórico, sino en su implementación, tal como hemos observado en la sección precedente. Entonces, es altamente interesante disponer de transformaciones que puedan aplicarse a la solución original (ya demostrada correcta), que mantengan su significado y que produzcan un programa de menor complejidad, iterativo, a ser posible. Esta filosofía se ha elevado en los últimos años a la metodología de la programación discutida hace un rato, que se ha integrado definitivamente en los primeros estadios de la enseñanza de la programación. En lo sucesivo en esta tesis, el término "transformación de programas" se restringirá a este último significado.

Una de las características primordiales de esta filosofía es la de mantener tanto la solución original como las sucesivas transformaciones en el mismo lenguaje de programación. Es decir, se entiende que la obtención de eficiencia en los programas recursivos ineficientes solo puede llevarse a cabo sin abandonar el propio lenguaje de alto nivel en el que se expresan los algoritmos. La compilación, siendo también en cierto modo una "transformación", se ve entonces como la parte final del proceso, que produce un programa ejecutable de una manera completamente automática, pero que (quizá por ello) no es capaz de rebajar la complejidad de la solución en ningún caso.

Cronológicamente, como hemos dicho, la motivación principal de los primeros trabajos fue exclusivamente la ineficiencia de los procesos recursivos (incluso, algunos lenguajes de amplio uso en aquella época no admitían la recursividad). Pueden encontrarse ya varios trabajos a partir del año 1970, que citamos a partir de Bird [1977a], como los de A.K. Chandra (1972), J. Darlington y R.M. Burstall (1973), M. Patterson y C.E. Hewitt (1970) y H.R. Strong (1971). Todos ellos, según el propio Bird [*op. cit.*], se restringían a los casos más sencillos de programas recursivos, los que podían convertirse en programas no solo iterativos, sino con un comportamiento limitado en cuanto a los requisitos de espacio. Pronto se demostró que este "ideal" no era posible de alcanzar, y que la eliminación de la recursividad en casos más generales requería el uso en un modo u otro de una pila, con lo que el programa podía ser sintácticamente iterativo, pero su comportamiento no lo era (si por "comportamiento iterativo" entendemos la ejecución en espacio constante, en consonancia con la discusión de la sección precedente).

Bird [1977a, 1977b] y Auslander y Strong [1978], entre otros, proponen métodos para la eliminación de la recursividad en programas más complejos. Pero existe una característica común en todos estos primeros trabajos: se presentan por medio de ejemplos más o menos simples, o esquemas particulares de recursividad, dando reglas heurísticas (no demasiado claras en algunas ocasiones) para su aplicación en otros casos. Por ejemplo, Bird [1977a] presenta diversas soluciones para el esquema recursivo:

"proc $S(x)$;
 if px then $N(x)$; $S(fx)$; $S(gx)$; $M(x)$ fi" [Bird 1977a]

dependiendo de la presencia de $N(x)$ o de $M(x)$ o de ambas.

El esquema más sencillo de recursividad lineal, lo que denominaremos *recursividad final*, era ya conocido. En la notación de Bird:

```
proc S(x);
  if px then N(x); S(fx) fi
```

podía sustituirse por:

```
proc S(x);
  L: if px then N(x); x := fx; goto L fi
```

es decir, por una actualización de los parámetros y un salto al comienzo del procedimiento. Como la sentencia de salto incondicional no está bien vista, conviene transformar lo anterior en:

```
proc S(x);
  while px do begin N(x); x := fx end
```

Las palabras exactas de Bird son:

«If the last action of a procedure p before it returns is to call procedure q , then simply goto the beginning of procedure q instead.

»This useful rule, which is easy to justify, is discussed in Knuth [5]; it also works when $q = p$ [...]. When parameters are present they must be reassigned just before the jump is made» [Bird 1977a]

En principio, parece que se consideraba un esquema resuelto, o al menos, que no causaba mayores problemas, y cuya solución podía aplicarse en otros esquemas, como el anteriormente citado cuando $M(x)$ era nulo.

Las soluciones de Bird [1977a] al esquema de recursividad doble se basaban en el uso de una (o más) pilas, aunque no solo en el sentido en que se implementa la recursividad en el modelo dinámico asociado al lenguaje, sino como registro de futuras "obligaciones" (Rohl [1984: 172] da crédito de ello a Knuth). Lo curioso es que el caso general se resolvía normalmente en un algoritmo de peor comportamiento, sobre el que se "observaban" ciertos detalles que eliminaban algunas operaciones o simplificaban otras, con lo que se obtenía un algoritmo mejorado, pero como estamos viendo a costa de convertir el proceso en no mecánico.

En definitiva, la eliminación de la recursividad se llevaba a cabo por técnicas completamente heurísticas, como confirman Auslander y Strong [1978]:

«These techniques can be applied to a program without an understanding of its purpose. However, they are complex enough so that we are inclined to teach them as tools for programmers rather than try to mechanize them as an option in an optimizing compiler» [Auslander y Strong 1978]

Por otro lado, Auslander y Strong usaban sin restricciones la sentencia `goto`, pero en general, sin decirlo abiertamente, se preferían las técnicas que no los tenían y se basaban en las sentencias de control "estructuradas". Así, comentando la solución más eficiente del esquema planteado, Bird dice:

«Unfortunately, [el programa] (17) does contain a single `goto`, but this may be regarded as a minor blemish» [Bird 1977a]

con lo que se añade otra restricción a las técnicas, obviamente.

Es preciso señalar que el énfasis en la eficiencia (sobre otras características de los programas) puede ser comprensible en el año 1977, por razones obvias. Sin embargo, estas ideas no dejan de tener actualidad. Sedgewick [1988: 62-65] transforma el recorrido recursivo preorden de un árbol binario ($M(x)$ nulo en el esquema de Bird [1977a]) en un programa no iterativo, pues no aparece ninguna sentencia iterativa: en su lugar, un procedimiento de diez líneas contiene cuatro sentencias con etiqueta y cinco sentencias `goto`, además de que debe usar una pila. Admite que es «a rather opaque program» [Sedgewick 1988: 63]. Entonces, afirma que las sentencias `goto` «can be "mechanically" removed to yield a more structured piece of code» [*ibid.* (comillas en el original)]; entre las operaciones que "mecánicamente" realiza están la identificación de dos estructuras `while` anidadas a partir de los saltos incondicionales, la conversión de una de ellas en una sentencia `repeat` con el consiguiente movimiento de parte del código, la conversión a un solo bucle (para lo cual incluye más operaciones de acumulación en la pila) y, finalmente, la adición de condicionales para no acumular en la pila subárboles nulos. Como conclusión, Sedgewick [1988: 65] insiste en la posibilidad de aplicar estas operaciones "mecánicas" a cualquier algoritmo recursivo.

Por otro lado, existe otra cuestión a considerar, y no solo para estas primeras soluciones, sino en general para toda conversión de la recursividad que suponga el uso explícito de una pila (en realidad, casi cualquier programa recursivo salvo los lineales). Preocupándose de la misma cuestión que nosotros, Loudon [1987] propone una observación de la eficiencia temporal de algoritmos recursivos y no recursivos en PASCAL. Elige el recorrido recursivo inorden de

un árbol binario. Inmediatamente, al intentar la solución no recursiva que usa una pila, se plantea la cuestión en la que queremos incidir:

«Should one adhere to the Abstract Data Type approach in the use of a stack, which is itself expensive in terms of procedure calls, or should one abandon structured programming and use in-line code for maximal run-time efficiency?» [Louden 1987]

La cuestión no es vana, y sus implicaciones son más profundas de lo que a simple vista parece: exponen una vez más el choque que se produce entre claridad y abstracción, por un lado, y eficiencia, por otro. Tal cuestión no ha sido clarificada, hasta donde llega nuestro conocimiento, en ninguno de los trabajos de la "primera época" de la transformación de programas, ni en los que reseñaremos a continuación. Y el hecho es que la versión recursiva mejoraba la versión no recursiva basada en tipos abstractos de datos en un factor que iba desde el 35% en un IBM-PC hasta el 45% en un VAX, mientras que usar código *inline* en la versión no recursiva mejoraba el programa recursivo en un 20%.

El alejamiento de los "métodos" *ad hoc* había comenzado ya en la época de Bird [1977a], por varios autores, principalmente Burstall y Darlington [1977], aunque con anterioridad al año referido. Aquí encontramos ya plenamente la filosofía transformacional en el sentido de que se enfatiza la necesidad de que las modificaciones que se introduzcan en los programas mantengan la equivalencia semántica con el original. Ello es posible si la transformación se basa en una serie de *reglas de transformación*, cuya aplicación haya sido mostrada correcta en cualquier caso. Entonces, el proceso se ve como una especie de demostración (es significativo que la misma metodología haya sido utilizada para sistemas automáticos de demostración, como el de Boyer y Moore [1984]).

Burstall y Darlington [1977] muestran dos clases de operaciones de gran interés en el proceso transformacional, que son la base de todas las transformaciones conocidas en la actualidad:

- la *inmersión* o *generalización* (*embedding* en inglés) es la búsqueda de una función más general que la función recursiva de partida; la función de *inmersión* tiene normalmente argumentos adicionales que, convenientemente sustituidos, producen la función original;
- el *desplegado* y *plegado* (*unfold-fold* en inglés) es la sustitución de una llamada por su cuerpo o valor y viceversa, respectivamente.

El método más sencillo que emplea esas reglas o técnicas tiene como objetivo la obtención de una función recursiva final equivalente a una función recursiva lineal (una sola llamada recursiva) no final. Para ello, se busca una inmersión de la función original, y entonces se aplican las reglas de plegado y desplegado junto con las propiedades de los otros elementos que ocurren en la definición de la función original, hasta conseguir una definición recursiva final para la función de inmersión. El método es ya clásico y se encuentra actualmente en textos orientados a estudiantes de los primeros cursos de los estudios de informática, como el de Balcázar [1993].

Existen dos inconvenientes o "puntos débiles" del método. Primero, no es inmediatamente aparente cuál ha de ser la función de inmersión; Burstall y Darlington [1977] hablan explícitamente de un "eureka" para conseguir este propósito. Segundo, tampoco es demasiado clara la secuencia de plegados y desplegados que ha de seguirse, y lo que es más importante, depende directamente de las propiedades (asociatividad, elemento neutro) de otras funciones que aparezcan en el cuerpo de la función original; muchas veces, estas funciones son operadores estándar como la suma y la multiplicación, pero no se aclara cómo actuar si se trata de otras funciones definidas por el programador.

Debido a ello, como señalan Partsch y Steinbrüggen [1983], casi todos los sistemas de transformación de programas recursivos son *interactivos*, en el sentido de que requieren cierta ayuda del usuario para la consecución de algunas metas, como los "eurekas" de Burstall y Darlington [1977]. Arsac y Kodratoff [1982] muestran una estrategia que mejora un poco este problema; la estrategia es «suficientemente sistemática como para ser implementada» (en sus propias palabras), pero en cualquier caso precisa de un «conjunto de heurísticas» (también literalmente).

Por otra parte, casi todos los sistemas definen lenguajes muy restringidos; el sistema original de Burstall y Darlington [1977] se basa en el lenguaje NPL ("*Non Procedural Language*"), que es aplicativo (no tiene efectos secundarios). Incidentalmente, hay que decir que estos estudios han dado lugar a una de las escuelas más activas en la definición e implementación de lenguajes funcionales, la escuela de Edimburgo, de donde han surgido descendientes de NPL como HOPE [Bailey 1990] o ML [Milner, Tofte y Harper 1990].

Finalmente, en general, las técnicas se aplican a funciones, y no a procedimientos [Díaz de Ilarraza y Lucio 1990: 187]; hay una razón para ello, en consonancia con la ausencia de efectos secundarios: las funciones pueden mantenerse en el dominio de las expresiones, matemáticamente "transparentes", mientras que en el concepto de procedimiento está prácticamente implícita la idea de efecto secundario.

El estudio de esquemas recursivos más generales que los lineales es, por supuesto, más problemático. Trabajos más recientes como los de Partsch y Pepper [1986], Cohen [1990], Hoogerwoord [1990a, 1990b] y Balcázar [1993, 1994], entre otros, presentan aproximaciones más o menos generales a esquemas recursivos múltiples, pero siguen existiendo diversas dificultades, a nuestro entender, tanto en la mecanización del proceso, como en la aplicación a un lenguaje imperativo como PASCAL. Partsch y Pepper [1986] explícitamente advierten que la notación funcional simplifica notablemente las manipulaciones del programa recursivo. Sin embargo, Balcázar [1993] realiza un loable esfuerzo por presentar el repertorio de transformaciones en un lenguaje próximo a los lenguajes imperativos habituales.

Como conclusión de este estudio de la eliminación de la recursividad mediante transformación de programas, mantenemos las siguientes ideas:

- la filosofía transformacional supone una metodología necesaria (como reconocen los planes de estudios ya en vigor) en la programación, entendiendo este proceso como una derivación verificada de algoritmos (es decir, no solo teniendo como objetivo una solución meramente, sino una solución completamente garantizada);
- la recursividad es el mecanismo único para la expresión de una primera solución a un problema; reconociendo que las soluciones intuitivamente más claras son normalmente más complejas en el sentido algorítmico, y que en general la implementación de la recursividad es ineficiente, han de realizarse esfuerzos para la obtención de programas equivalentes de menor complejidad;
- las transformaciones sucesivas se producen siempre en el mismo lenguaje, siendo por lo tanto un proceso de características cualitativa y estructuralmente distintas de las que posee un compilador (concretamente, no observando el decremento en la complejidad de un algoritmo como una *optimación*);

- se considera que el esquema de recursividad lineal final es el objetivo último deseable del proceso, aunque ello suponga el uso necesario en una u otra de medida de una pila en la transformación de esquemas recursivos más generales; una vez obtenida la recursividad final, el paso a un programa sintácticamente iterativo se da por obvio en la literatura técnica (aunque también como efecto de una última transformación), es decir, se ha identificado "sintácticamente iterativo" con "sintácticamente recursivo final";
- las tácticas utilizadas en las transformaciones incluyen mayoritariamente la inmersión y el plegado y desplegado, junto con propiedades concretas de otros elementos que intervienen en el problema (constantes, operadores);
- las estrategias de transformación de esquemas recursivos diversos están muy elaboradas, pero, primero, no incluyen *todos* los esquemas recursivos posibles, y segundo, no dejan de ser estrategias mayormente heurísticas, cuya mecanización es tanto más improbable cuantos más sean los factores que se admiten en los esquemas recursivos correspondientes, concretamente, cuanta mayor diversidad de aspectos contenga el lenguaje en el que se escriben los algoritmos.

Quizá esta última conclusión pueda ser entendida como una crítica: nada más alejado de nuestra intención. Reconocemos el gran esfuerzo que se ha hecho en este campo, y la importancia de la filosofía de "cálculo de programas" en la enseñanza de la programación, establecida como ya se ha mencionado. Quizá lo que ocurre se asemeja, salvando las distancias, a la situación producida en inteligencia artificial: la facilidad para concebir el fin deseable (en este caso, la conversión *mecánica* de *cualquier* esquema recursivo) ha chocado con las enormes dificultades inherentes de la tarea, y es por ello que los resultados obtenidos pueden parecer escasos si se comparan con ese fin deseable. Es el momento, como lo fue para la inteligencia artificial, de ser más modestos, no tan ambiciosos, percibiendo la gran complejidad del problema, y reconocer entonces lo *mucho* que se ha obtenido: hasta ahora, es lo único de que disponemos.

2.4 SOLUCIONES (II): RECURSIVIDAD FINAL EN SCHEME

El lenguaje de programación SCHEME es *propiamente recursivo final*. En el documento que define el lenguaje se dice expresamente:

«Implementations of Scheme are required to be properly tail-recursive. This allows the execution of an iterative computation in constant space, even if the iterative computation is described by a syntactically recursive procedure. Thus with a tail-recursive implementation, iteration can be expressed using the ordinary procedure-call mechanics, so that special iteration constructs are useful only as syntactic sugar» [Clinger y Rees 1991: 3]

Por lo tanto, SCHEME soluciona los mismos problemas que la clase más sencilla de transformación de programas recursivos, que se han estudiado en la sección 2.3, identificando como allí el concepto de “programa recursivo final” con “programa iterativo”, pero siguiendo una filosofía radicalmente distinta. De hecho, la ejecución de funciones recursivas finales en espacio constante es un caso particular (quizá el más importante por sus implicaciones) de un resultado más general, consistente en utilizar un mecanismo alternativo de llamada en el caso de una activación final de función. Es decir, la consecución de procesos iterativos a partir de programas recursivos finales no se realiza mediante una búsqueda específica de la recursividad y una transformación de la misma, sino como mero resultado de la implementación del lenguaje y de la generación de código correspondiente a las llamadas a los subprogramas que se realiza durante la compilación. Nos detendremos entonces en la presentación de la estrategia utilizada en la implementación habitual de SCHEME para conseguir este tratamiento de la recursividad final.

El lenguaje de programación SCHEME es definido por primera vez por Sussman y Steele [1975]. A la hora de encuadrarlo en una “familia” de lenguajes, se está de acuerdo en considerarlo un “dialecto” del lenguaje de programación LISP [McCarthy 1960; Cortés y Sierra 1987], y de hecho, existen en un primer contacto con el lenguaje SCHEME grandes semejanzas con LISP, que por supuesto se ponen de manifiesto en la mayoría de las estrategias para su implementación. Así, las primeras revisiones del lenguaje presentan en sus títulos frases como «a dialect of Lisp» [Steele y Sussman 1975] o «an uncommon Lisp» [Clinger 1985]. Sin embargo, no queremos dejar de mencionar que parece existir un sentimiento entre los participantes en las siguientes revisiones de lenguaje [Rees y Clinger 1986; Clinger y Rees 1991] en reconocer la importancia

del lenguaje ALGOL en la concepción de SCHEME, hecho que se manifiesta en el propio título de los documentos mencionados (que quiere rememorar el de los históricos informes que definieron ALGOL [Naur 1960, 1963]) y en la dedicatoria explícita que se hace a la memoria de dicho lenguaje.

En cierto modo, así como ALGOL supuso un punto de inflexión en la historia de la informática, SCHEME presenta una serie de características con las que pretende convertirse, salvando las distancias, en un lenguaje igual de paradigmático: entre otras, citaremos el tratamiento de los subprogramas como entidades de "primera clase" (es decir, que pueden manejarse exactamente igual que datos de cualquier otro tipo, incluyendo ser pasadas como parámetros o devueltas como valor de retorno de otros subprogramas), el manejo explícito de continuaciones (no presente en ningún otro lenguaje de programación), la distinción entre aritmética exacta e inexacta (que independiza los tipos de datos numéricos de la representación máquina que se utilice) y la propia recursividad final. SCHEME, siguiendo a ALGOL y todos los lenguajes "imperativos" herederos de este, tiene ámbito léxico, al contrario que LISP, que observa las reglas de ámbito dinámico. Esta característica ha sido ya adoptada por la mayor parte de implementaciones de LISP, principalmente COMMON LISP [Steele 1990]: de hecho, hoy existe aún menos diferencia entre SCHEME y COMMON LISP, pero solo a resultas de que los estudios que condujeron al primero se han aplicado sucesivamente en las revisiones del segundo.

2.4.1 Obtención clásica de la recursividad final en SCHEME

La implementación de SCHEME sigue en principio, en lo que al tratamiento de la recursividad final respecta, similares estrategias a las de LISP, tal como se conocían desde la década de los setenta, recogidas por ejemplo en la obra de Allen [1978]. Así, si se elige el subconjunto de LISP que es similar a un lenguaje imperativo como ALGOL o PASCAL, la estrategia de implementación podría ser exactamente la misma que se presentó en el apartado 2.2.1: vinculación por medio de una pila, en la que se acumula información para cada activación de una función, que incluye información de control (la dirección de retorno) y el ambiente de dicha función [*ibíd.*: 289-300]. Nótese que esta

estrategia sirve para ese subconjunto de LISP tanto en interpretación como en compilación (un dilema muy habitual en LISP no siempre bien entendido).

La presencia de argumentos funcionales, es decir, funciones que son pasadas como parámetros a otras funciones, y que son posiblemente activadas dentro de estas últimas (problema conocido en la comunidad LISP como *downward funarg*), no requiere una modificación especial de este esquema (además de que existe en lenguajes imperativos como PASCAL), teniendo en cuenta que el argumento funcional se representa por un *cierre léxico* (o *closure* [ibíd.: 141]): combinación de un punto de entrada a la función pasada como argumento funcional y un ambiente en el que buscar sus datos. Cuando el argumento funcional es activado, se extiende el ambiente asociado a dicho argumento (que está presente más abajo en la pila) con su propio ambiente y se salta al punto de entrada a la función. La vinculación por medio de una pila sigue siendo perfectamente válida, y si este problema fue alguna vez tal se debió más a las reglas de ámbito dinámico imperantes en LISP en aquel entonces.

Lo que sí produce la imposibilidad de aplicar este esquema de vinculación mediante una pila es la presencia en el lenguaje de *valores funcionales* completos, esto es, la consideración de las funciones como entidades de "primera clase", con la posibilidad de ser integradas en estructuras de datos o de ser devueltas como valor de retorno por otras funciones, en definitiva, de ser tratadas como cualquier otro dato del programa. Evidentemente, que se posibilite el uso de valores funcionales (en la comunidad LISP, *upward funarg*) es el primer requisito (aunque no único) de un lenguaje que pretenda adherirse al paradigma de la programación funcional, y no es este el momento de explicar la necesidad de tal característica para conseguir los más altos grados de abstracción (*vide*, por ejemplo, Backus [1978]).

La principal consecuencia del uso sin restricciones de valores funcionales es que el ambiente asociado a un bloque o función puede sobrevivir a su propia terminación. Clarifiquemos esta cuestión. Consideramos que tanto ALGOL (y descendientes como PASCAL o ADA) como LISP o SCHEME son lenguajes con *estructura de bloques*, pues permiten la construcción anidada de bloques (no así el lenguaje C, por ejemplo). También consideramos, al igual que en el apartado 2.2.1, que los bloques, como entidades que definen los datos presentes en un ambiente, se establecen a través de los subprogramas.

Entonces, en un lenguaje sin valores funcionales se cumple que el tiempo de vida (durante la ejecución de un programa) de un ambiente se corresponde exactamente con el tiempo de activación del subprograma o bloque que define dicho ambiente. Por ello, la información que es necesaria dinámicamente para la ejecución puede englobar en una sola entidad la información de control (relativa al proceso de activación, como la dirección de retorno) y la información de ambiente, de donde surge el concepto de registro de activación y su estructura típica en lenguajes de la familia de ALGOL, recogido en la Figura 2.2.2. Además, dada la estructura en que lógicamente se desenvuelven las activaciones de los subprogramas (un árbol), el uso de una pila es adecuado para mantener dichos registros de activación.

Ahora bien, si una función puede devolver como objeto de retorno a su vez otra función (valores funcionales completos), su ambiente puede sobrevivir a la terminación de su activación. Consideremos el ejemplo del siguiente programa.

```
(define f
  (lambda (x y)
    (define g
      (lambda (z)
        (write (* x y z))))
    g))      ; f devuelve la función g como valor de retorno

(let ((vf (f 7 13)))    ; vinculación del valor funcional
    (vf 25))           ; aplicación del valor funcional
```

Programa 2.4.1 Un programa con valores funcionales en SCHEME

Cuando se activa la función *f* mediante *(f 7 13)*, se crea un valor funcional que es devuelto como resultado. El valor funcional sigue siendo un cierre léxico, formado con el punto de entrada a la función *g* y el ambiente de la misma. Nótese que al establecer el nombre *vf* en la forma *let*, la activación de *f* ha terminado. Sin embargo, el ambiente de *f* no puede perderse, pues es parte del ambiente de *g*, que debe activarse al ser llamado el valor funcional. Como resultado, la presencia de valores funcionales en un lenguaje obliga a la diferenciación explícita entre el manejo de la información de control (relativa a los procesos de activación de los subprogramas) y el manejo de la información del ambiente.

La solución a esta cuestión consiste evidentemente en separar ambas clases de información. Bobrow y Wegbreit [1973] propusieron una estrategia válida para el ámbito dinámico de LISP, que fue adaptada después por Steele [1977b] para el ámbito léxico. Básicamente, la idea es que la información de control se continua manejando en una pila (pues las activaciones de un programa siguen dándose en forma de árbol), con lo que efectivamente sería adecuado el nombre de *pila de control*, pero los ambientes pueden aparecer y desaparecer de una manera distinta: pueden quedar ambientes "abiertos" (cuando se crea un valor funcional) que han de mantenerse después de la terminación de las activaciones que los crearon. En ausencia de valores funcionales, los ambientes estarán dispuestos como en una pila, pero cuando aquellos aparecen, el ambiente de la cumbre ha de mantenerse (junto con los ambientes de sus superiores léxicos, siguiendo la cadena estática que une unos ambientes con otros), mientras que otros más cercanos a la "base" de la estructura pueden desaparecer, con lo que esta se vería "fragmentada" [*op. cit.*].

El modo más sencillo de solucionar este problema es situando los ambientes en el almacén de memoria dinámica o montículo (*vide* Figura 2.2.1). LISP tiene un tratamiento diferente al de los lenguajes imperativos de los datos creados dinámicamente durante el curso del programa, debido al intensivo uso que se hace de ello. Concretamente, los lenguajes de la familia de ALGOL suelen tener *vinculación dinámica explícita*, lo que significa que deben darse operaciones explícitas para la creación y destrucción de la memoria que alojará (en el montículo) los datos dinámicos. Por contra, LISP y SCHEME tienen *vinculación dinámica implícita*, al ser la *lista* uno de los tipos de datos básicos y fundamentales del lenguaje: no existen operaciones particulares para la creación y destrucción de la memoria dinámica, sino que esto ocurre simplemente como resultado implícito de operaciones de manejo de los datos. Por ello, estos lenguajes precisan un sistema dinámico de manejo del montículo que "conozca" la memoria no usada. Aunque existen diversos mecanismos, el más habitual es el *recolector de basura* (en inglés, *garbage collector*), que no se preocupa por la recuperación de la memoria no usada, hasta que se termina el espacio libre, momento en el cual suspende la ejecución del programa para recuperar la memoria que en ese momento no está en uso.

Entonces, la creación de los ambientes se realiza en el montículo. Cuando un ambiente ha sido finalizado, no quedan apuntadores a él, por lo que su

memoria pasará a ser libre cuando se ejecute el recolector de basura (o mecanismo alternativo). Si se produce un valor funcional, quedará un apuntador a un cierto ambiente (junto con todos los superiores léxicos, a través del enlace de acceso), por lo que tal información no se liberará aunque se ejecute el recolector de basura [Allen 1978: 301]. Y es por ello que se dice que los lenguajes LISP y SCHEME tienen *vinculación en el montículo* (o *heap allocation*), y no solamente por ser la lista el tipo de dato básico del lenguaje.

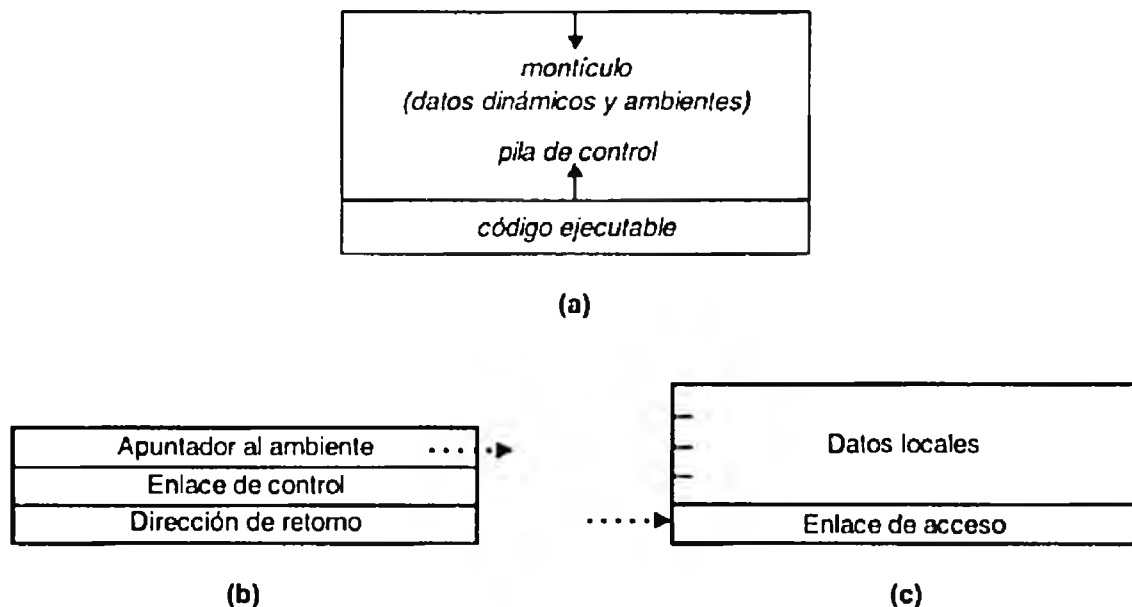


Figura 2.4.1 (a) Estructura clásica de un programa objeto en el lenguaje LISP, con vinculación de datos en el montículo.
 (b) Registro de control, alojado en la pila de control.
 (c) Registro de ambiente, alojado en el montículo

Incidentalmente, es preciso comentar que la posibilidad que ofrece SCHEME de manejo explícito de continuaciones establece un paralelismo con los resultados anteriores pero ahora en lo relativo precisamente al control. Los argumentos de continuación (*downward continuation*, es decir, continuaciones pasadas como parámetros a funciones) permiten seguir manejando la información de control en una pila, en la medida que la invocación del argumento de continuación no supone más que la adopción de un estado de control anterior; son por lo tanto, similares a los saltos no locales de PASCAL, que se implementan eliminando cierto número de registros de activación de la pila de control.

Sin embargo, los valores de continuación (*upward continuation*, es decir, continuaciones devueltas como resultado de funciones o almacenadas en estructuras de datos, convertidas por tanto, en objetos de "primera clase") no permiten que la información de control se maneje ya como una pila: la invocación del valor de continuación puede darse después de que se haya abandonado el contexto dinámico en que la continuación fue creada [Friedman, Wand y Haynes 1992: 312]. Para ello, la información de control debe situarse también en el montículo: si no existen valores de continuación, la información aparecerá en forma de pila, pero si existen, esta "pila" se fragmentará, debido a que será necesario mantener parte de la misma, a la vez que crear otra nueva: los registros de control "terminados" se marcan como memoria susceptible de ser liberada por el recolector de basura.

Como se está observando, es fundamental para la implementación de estos elementos de LISP o SCHEME (valores funcionales y de continuaciones) la suposición de un manejo dinámico de la memoria de toda clase, lo que implica en la práctica la existencia durante la ejecución del recolector de basura (o mecanismo alternativo), con el comportamiento habitualmente asociado a dicha implementación, poco adecuado para determinadas aplicaciones y que es causa de las críticas más extendidas, sobre todo desde la perspectiva de los lenguajes imperativos, en que la vinculación puede establecerse estáticamente, evitando la necesidad de la recolección de basura.

En este esquema, sin embargo, la recursividad final se puede fácilmente compilar en un código que no consuma memoria, como muestra Steele [1977b], y ello debido exactamente a la separación dinámica entre la información relativa al control y la información relativa al ambiente. En realidad, se obtiene como caso particular de un resultado general mucho más interesante, no solo aplicable a llamadas a funciones que sean directamente recursivas. Este caso general se refiere a las llamadas realizadas en "posición final", es decir, aquellas cuyo resultado es a su vez el resultado de la función que realiza la llamada.

Supongamos que una función f llama a otra función g en posición final. En primer lugar, no es necesario acumular nueva información de control (una dirección de retorno) para g , pues cuando esta termine, f no ha de hacer otra cosa que retornar a su vez, por lo que la dirección de retorno del proceso completo es la que se haya asociado a la llamada a f . Por ello, la memoria necesaria para la información de control no crece a lo largo de una serie de

llamadas finales consecutivas. En segundo lugar, debe darse una actualización del ambiente; si g no está definida localmente en f , el ambiente de f simplemente puede perderse (pues al ser la llamada final, es seguro que nunca más se hará referencia a un objeto local de f), por lo que solo hay que crear un nuevo ambiente para g , y actualizar el apuntador al ambiente actual con dicho valor; el ambiente de f quedará así sin ninguna referencia, y será liberado por el recolector de basura. Este comportamiento incluye como caso particular el de una llamada final recursiva directa de la misma función f : el ambiente para la activación que efectúa la llamada se libera y se crea uno nuevo para la activación que entra. En cualquier caso, no existe un crecimiento de la memoria necesaria para el ambiente.

Sin embargo, si g está definida localmente en f , el ambiente de esta última debe mantenerse en general, pues g puede hacer referencia a los objetos locales de f . Por lo tanto, en este caso, sí se aprecia un crecimiento de la memoria necesaria para la ejecución del proceso, pero puede resolverse el problema si se piensa con más cuidado el caso precedente. En efecto, hemos dicho que si g no está definida localmente en f , entonces g no podrá referirse a ningún objeto local de f , pero esta no es toda la cuestión: realmente, no puede referirse a ningún objeto local de los sucesivos bloques superiores léxicamente a f hasta el propio nivel léxico de g . Por lo tanto, todos los ambientes que van desde el propio ambiente de f hasta el que tiene el mismo nivel de anidamiento que g pueden perderse, y este caso provoca entonces una disminución de la memoria necesaria para los ambientes.

Realmente, este hecho se obtiene simplemente como consecuencia del propio cálculo del enlace de acceso que une cada ambiente con el de su predecesor léxico: el enlace de acceso para el ambiente de g señalará a un ambiente tanto más abajo en la "pila de ambientes" cuanto mayor distancia léxica exista entre f y g , por lo que toda la cadena de ambientes intermedia quedará sin referencia y será liberada por el recolector de basura.

Ante el caso más general de un proceso recursivo final expresado mediante una recursividad mutua (no directa), se obtiene que los ambientes aparecen y desaparecen como en un anillo, por lo que al fin y al cabo, tampoco existirá un crecimiento de la memoria necesaria para los ambientes. Un ejemplo aclarará definitivamente esta cuestión: el Programa 2.4.2 presenta un esquema de recursividad mutua en tres niveles, cada uno anidado en el anterior.

```

(define func
  (lambda (n)
    (define fenc
      (lambda (x y)
        (define finc
          (lambda (z)
            (define fenc
              (lambda (v)
                ...
                (fenc ...) ) ; llamada final de fenc a fenc
              ...
            (fenc ...) ) ; llamada final de finc a fenc
          ...
        (finc ...) ) ; llamada final de fenc a finc
      ...
    (fenc ...) ; activación del proceso recursivo
  ...)) ; fin de func

```

Programa 2.4.2 Un programa con recursividad final no directa en SCHEME

El comportamiento durante las sucesivas llamadas finales de los ambientes se refleja en la siguiente figura, donde *ea* es un acrónimo de “enlace de acceso”.

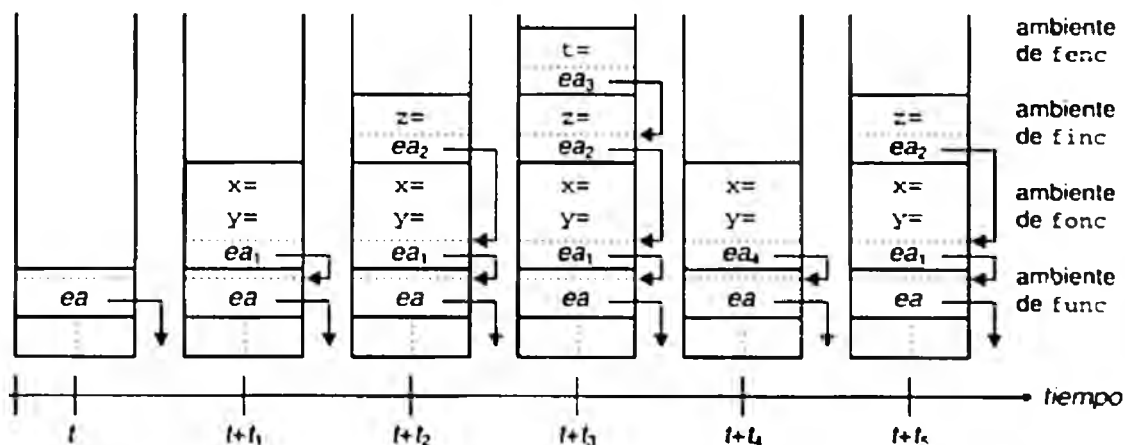


Figura 2.4.2 Instantáneas sucesivas de los ambientes creados y liberados durante la ejecución del Programa 2.4.2

Queda claro que, a pesar de producirse un crecimiento “local” de la memoria en algunos instantes, el proceso completo se ejecuta dentro de unos límites constantes.

En definitiva, un proceso recursivo expresado mediante llamadas finales a funciones se ejecuta en un espacio constante. Más aún, en algunos casos, como señala Steele [1977b], ni siquiera es necesaria la creación de un nuevo ambiente, sino que puede reutilizarse el actual; el caso más claro es el de una llamada final

directamente recursiva, pero puede darse en otras ocasiones. Entonces, el esquema mostrado conduce conceptualmente a la transformación de procedimientos recursivos finales mostrada en la sección 2.3. En efecto, la no necesidad de acumular información de control equivale a sustituir la llamada por un mero salto incondicional (*goto*), mientras que la reutilización de la memoria del ambiente se corresponde con la actualización de los parámetros. Por supuesto, la filosofía de transformación de programas solo puede aplicarse en las llamadas directamente recursivas, y bajo condiciones muy estrictas, mientras que esta nueva perspectiva produce una solución mucho más general.

Steele [1977a] propone un estilo de programación basado en un uso más intensivo de los subprogramas, como medio para alcanzar mayores cotas de estructuración y abstracción. Tal estilo requiere, para ser efectivo, dos características de los lenguajes en que se aplique: primero, que las llamadas a subprogramas no sean procesos "caros" (como parece que ocurría en la época del documento mencionado), y segundo, que la recursividad final sea convertida en iteración. De ese modo, las construcciones iterativas habituales de los lenguajes imperativos podrían sustituirse con subprogramas recursivos finales.

Es conveniente finalizar este apartado mencionando que Steele va más allá en su propósito de mejorar la calidad del proceso de llamada a una función, tratando de que sea equivalente a una instrucción *goto*, pero no solamente en el caso recursivo final, sino en cualquier posición. Steele [1977c] presenta una técnica conocida desde entonces como *continuation-passing style* (CPS) en que una llamada a función se entiende como un *goto* que pasa parámetros. Esto choca a primera vista con la necesidad de que se genere una dirección de retorno en cada llamada a un subprograma. En realidad, la dirección de retorno es en efecto generada, pero en vez de ser acumulada en cierto lugar (la pila de control) se ofrece como parámetro adicional a la función que es invocada. En el caso de una llamada recursiva final, la continuación que se pasa a la función que es llamada es exactamente la misma que se pasó a la función que invoca.

RABBIT [Steele 1978] fue el primer compilador de SCHEME que aplicó estas ideas. Kranz *et al.* [1986], para la construcción de otro compilador del mismo lenguaje llamado ORBIT, extendieron CPS para incluir la asignación. Es importante entender CPS, incluyendo la *forma de asignación*, como un lenguaje intermedio. En él, todas las llamadas a funciones son finales, no hay distinción

entre las variables originales del programa fuente y las "temporales" que pueden ser necesarias (y que han aparecido explícitamente por efecto de la transformación a CPS), y lo que es muy importante, prácticamente todas las optimaciones habituales de los compiladores pueden aplicarse sin dificultad.

Sin embargo, CPS puede hacer un uso extensivo de los aspectos que imposibilitan una vinculación por medio de una pila, es decir, valores funcionales y de continuación. Por ello, tal como Kranz *et al.* [1986] señalan, el compilador debe esforzarse para determinar cuando un cierre léxico puede situarse en la pila o debe "vivir" en el montículo. La generación de código a partir de CPS conduce, evidentemente, al comportamiento habitual en que llamadas no finales consumen memoria, mientras que los procesos recursivos finales no lo hacen.

2.4.2 Recursividad final y vinculación por medio de una pila

La solución adoptada por los compiladores de SCHEME mencionados antes sigue basándose en una clara diferenciación entre la información de control y la información relativa a los datos. El intento de integrar ambas clases de información en el modo en que se realiza habitualmente en los lenguajes imperativos, en el concepto de *registro de activación* situado en una estructura en forma de pila, conduce casi inevitablemente a la pérdida de la propiedad de recursividad final (además de la prohibición de los valores funcionales o *upward funargs*). Por tanto, la conclusión que obtienen varios autores (Friedman, Wand y Haynes [1992: 368-370], Pitman y Peters [1992: 343-344], entre otros), es que al menos parece *a priori* difícil entender cómo la vinculación de los datos locales de los ambientes en una pila puede conciliarse con el comportamiento iterativo de los procesos recursivos finales.

Sin embargo, Hanson [1990] ofrece una solución a este problema, solución que será ya uno de los puntos de partida definitivos de nuestro trabajo. Hanson aplica su solución en un compilador para el lenguaje SCHEME [Rozas 1984; Adams y Hanson 1993; Hanson 1993], pero pretende alejarse de la habitual implementación de este lenguaje, caracterizada como se ha mostrado en el apartado precedente, y acercarse a la implementación más "tradicional" de lenguajes de la familia de ALGOL (probablemente, recogiendo el espíritu

existente entre los participantes de las últimas revisiones del lenguaje, como señalamos al principio de la sección).

Hanson intenta colocar la información de control y la información de ambiente en una pila (en los casos en que sea posible, evidentemente, es decir, en ausencia de valores funcionales y de continuaciones), en la misma línea que la implementación de un lenguaje como ALGOL gestiona esa información mediante la estructura conocida como registro de activación en la pila de activaciones (como se recogió en el apartado 2.2.1), y aún así conseguir un comportamiento iterativo para la recursividad final. La idea de Hanson es que puede hacerse una modificación del concepto de registro de activación en particular, tal como tradicionalmente se entiende (Figura 2.2.2), y de la estructura de la pila de activaciones, en general, para alcanzar tal objetivo.

En la implementación tradicional de la estrategia de vinculación en la pila, toda llamada a un subprograma acumula un nuevo registro de activación, compuesto por información de control (básicamente, la dirección de retorno) e información de ambiente (la memoria de los datos). Sin embargo, se ha visto que una llamada final no precisa acumular nueva información de control, y la información de ambiente puede eliminarse en algunos casos, dependiendo de la profundidad de anidamiento relativa entre la función que hace la llamada y la función que la recibe. Entonces, en consonancia con la distinción que se hace en la implementación habitual de LISP de las dos clases de información a mantener, Hanson diferencia explícitamente dos clases de registros en la pila de activaciones:

- registros de control, que contienen direcciones de retorno, y que están unidos mediante el enlace de control;
- registros de ambiente, que contienen los datos locales, y que están unidos mediante el enlace de acceso.

Una configuración genérica de la pila de activaciones consiste en una secuencia de unidades que constan cada una de ellas, vistas desde la base de la pila, de un registro de control y posiblemente varios (al menos uno) registros de ambiente (Figura 2.4.3). La implementación tradicional, bajo esta perspectiva, consiste en que cada unidad de la pila (registro de activación) contiene exactamente un registro de control y un registro de ambiente (Figura 2.2.2).

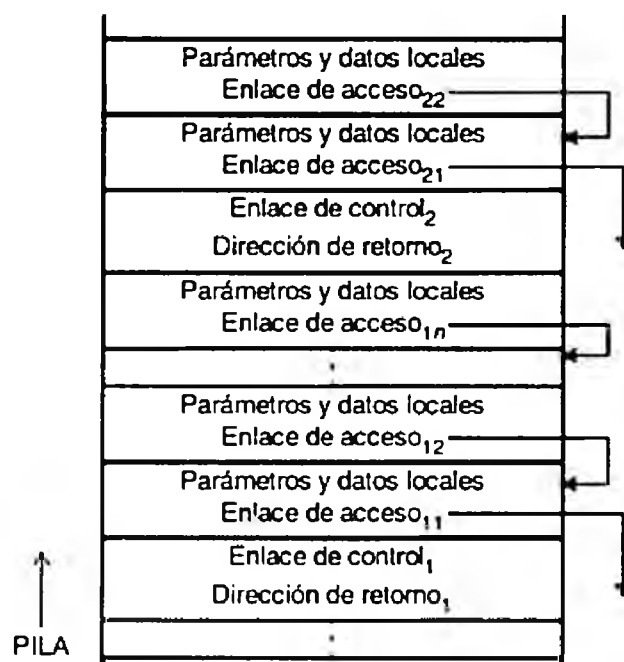


Figura 2.4.3 Estructura genérica de la pila de activaciones según Hanson [1990]

Una llamada en posición no final (denominada *subproblema* por Hanson) se procesa como habitualmente: se acumulará un nuevo registro de control con una dirección de retorno y un nuevo registro de ambiente para la función entrante. Sin embargo, ante una llamada final (denominada *reducción*) no se acumula ningún registro de control, pues la dirección de retorno será la de la función que invoca (que a su vez puede ser la de otra función que invocó a esta). Si pensamos, en un primer momento, que la llamada final acumula un registro de ambiente, se obtendría en efecto una estructura como la de la Figura 2.4.3, en que cada registro de control está seguido por uno o más registros de ambiente. En la cumbre de la pila siempre habrá un registro de esta última clase. Entonces, la secuencia de retorno consistiría simplemente en eliminar todos los registros de ambiente que estén desde la cumbre de la pila hacia abajo, hasta encontrar el primer registro de control, que también se desapila, saltándose a la dirección de retorno en él contenida.

El asunto más complejo concierne a los registros de ambiente; obviamente, si se quiere conseguir el comportamiento en espacio constante, en algún momento deberán desaparecer algunos registros de ambiente de la pila. Se puede demostrar fácilmente, como hace Hanson, que si dos registros de ambiente se encuentran adyacentes, entonces el enlace de acceso del superior

señala al inferior (cuestión ya recogida en la Figura 2.4.3). Entonces, recordando lo obtenido en el apartado precedente, las llamadas finales se procesan dependiendo de la distancia léxica relativa entre la función que invoca y la que es llamada.

- Primero, si la función llamada es local en la que llama, el registro de ambiente de esta última ha de mantenerse: se acumula un nuevo registro de ambiente, con el enlace de acceso señalando al que estaba en la cumbre de la pila.
- Segundo, si la función llamada tiene la misma profundidad de anidamiento que la que invoca (lo que incluye el caso particular de una llamada directamente recursiva), el registro de ambiente que existe en la pila puede eliminarse (una vez, eso sí, que se han calculado los posibles parámetros actuales de la llamada), y acumularse después el nuevo registro de ambiente: en este caso, la pila no crece.
- Finalmente, si la función llamada tiene menor profundidad de anidamiento que la que llama, encontramos que no son necesarios los registros de ambiente que están entre el todavía actual y el correspondiente al superior léxico de la función llamada (este excluido); por lo tanto, podrán eliminarse, antes de acumular el nuevo registro de ambiente, todos los registros de esta clase en ese rango, salvo que exista un registro de control entre ellos, en cuyo caso la eliminación debe detenerse.

En definitiva, Hanson expresa la secuencia de llamada para una reducción (llamada final) en los siguientes términos:

- Se calculan el enlace de acceso y los parámetros del nuevo ambiente, salvaguardándolos en algún lugar diferente de la pila.
- Si la cumbre de la pila es un registro de ambiente correspondiente a una activación de profundidad mayor que el superior léxico de la función llamada, se desapila. Se repite el proceso hasta encontrar o bien un registro de control, o bien el registro de ambiente del superior léxico.
- Se acumula en la pila el enlace de acceso y los parámetros calculados en el paso primero.
- Se salta al punto de comienzo de la función llamada.

Mostramos a continuación un ejemplo del comportamiento de la estrategia de Hanson. Retomamos para ello el Programa 2.4.2, en que existía una recursividad no directa en tres niveles, suponiendo que la primera llamada a la función *fonc* (dentro de la función *func*) no es final. La composición de la pila de activaciones se presenta en la figura siguiente.

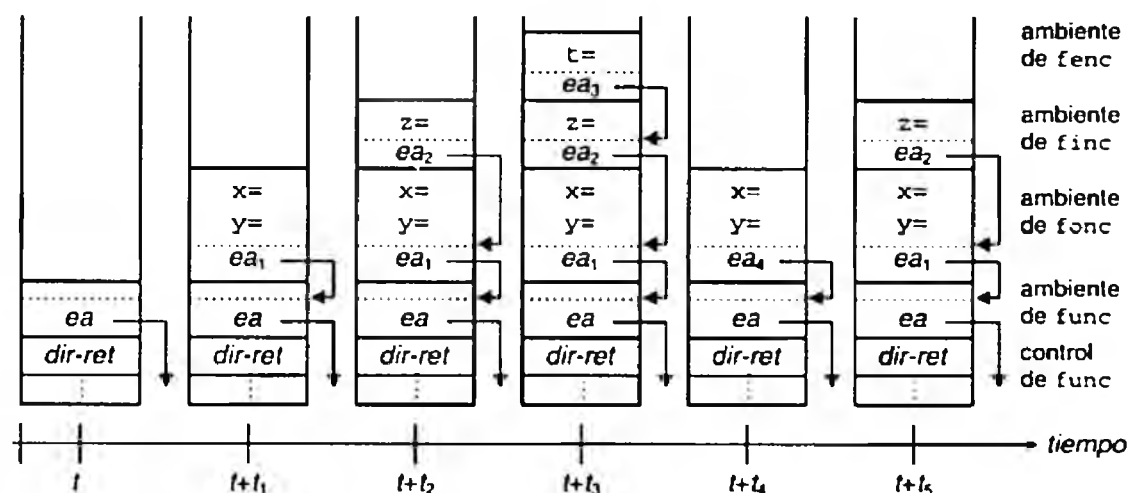


Figura 2.4.4 Instantáneas sucesivas de la pila de activaciones durante la ejecución del Programa 2.4.2, según la estrategia de vinculación en una pila de Hanson [1990]

Es evidente el comportamiento "en anillo" de los registros de ambiente, tal como ocurría con estrategia de vinculación en el montículo original del lenguaje LISP o SCHEME examinada en el apartado anterior: la similitud con la Figura 2.4.2 es clara.

En definitiva, Hanson [1990] demuestra que puede conciliarse una filosofía de vinculación de ambientes en una pila con la optimización de la recursividad final. El problema ahora radica en ajustar los detalles para obtener una implementación eficiente de la estrategia propuesta. Hanson observa que no solo han de distinguirse las dos clases de registros existentes en la pila, sino que deben ejecutarse en el caso más general bucles de eliminación de registros de la pila durante la secuencia de llamada de las reducciones y durante la secuencia de retorno, lo que ralentizaría excesivamente el programa.

Para evitar este problema, Hanson propone una implementación basada en el uso del enlace de control, pero a costa de mantener un registro global (como el que marca la cumbre de la pila) señalando en todo momento la

situación del registro de control más próximo a la cumbre de la pila. Entonces, la eliminación de registros de control y/o de ambiente que debe producirse durante las reducciones o la secuencia de retorno se consigue simplemente manejando el apuntador global de control y el apuntador a la cumbre de la pila. Concretamente, en una reducción se procede de la siguiente manera:

- Se calculan el enlace de acceso y los parámetros del nuevo ambiente, salvaguardándolos en algún lugar diferente de la pila.
- Se comparan el enlace de acceso calculado y el apuntador global de control; el apuntador de cumbre de la pila se coloca con el valor más cercano a este último.
- Se acumula en la pila el enlace de acceso y los parámetros calculados en el paso primero.
- Se salta al punto de comienzo de la función llamada.

En el retorno, se realizan las siguientes operaciones:

- Se copia el apuntador global de control sobre el apuntador de cumbre de la pila, con lo que este ya señala al registro de control.
- Se actualiza el apuntador global de control desapilando el valor previo guardado en ese registro de control.
- Se desapila la dirección de retorno y se salta a ella.

Con ello se consigue una implementación eficiente, que Hanson aún mejora ligeramente incluyendo análisis estático del programa fuente.

Como resumen de esta sección, insistimos en que el lenguaje SCHEME aporta una contestación positiva a la cuestión de si es posible mejorar el comportamiento de los programas recursivos. De hecho, los programas recursivos más simples (los recursivos finales) tienen en SCHEME exactamente la misma tasa de eficiencia (salvo por pequeños detalles) que los programas iterativos más simples en un lenguaje como PASCAL: se ejecutan en un espacio fijo y son procesos de "ida", no de "ida y vuelta" como lo es un proceso recursivo no final. Esta clase de programas es relativamente extensa en sus posibles aplicaciones, pero además es el objetivo último de la metodología de derivación y transformación de programas, tal como se concluyó en la sección 2.3, por lo que se comprende la transcendencia del resultado. Con todo ello, estamos preparados para presentar nuestro trabajo.

2.5 PLANTEAMIENTO DEL TRABAJO E HIPÓTESIS

La motivación del trabajo es la siguiente: visto que existe un lenguaje que aún a eficiencia y recursividad en su modo más simple, ¿ello resulta solo gracias a las particularidades de tal lenguaje, o es posible por contra aplicar las mismas ideas a un lenguaje radicalmente distinto, concretamente, imperativo? La solución de Hanson [1990] indica que es posible la obtención de tal grado de eficiencia con una estrategia de vinculación por medio de una pila, como la que observan todos los lenguajes imperativos, pero sigue abierta la cuestión de si las características de SCHEME como lenguaje son las que dan posibilidad al método, o dicho de otro modo, si elementos diferentes de otro lenguaje impedirían la aplicación del mismo.

La importancia de la recursividad final en el conjunto de problemas cuya solución es recursiva ha quedado clara en las secciones anteriores. Por lo tanto, disponer de un lenguaje de programación que sea capaz de ejecutar programas recursivos finales mediante un comportamiento iterativo es una gran ayuda desde varios puntos de vista, y en ello se insiste al presentar las características más notables del lenguaje SCHEME. Sin embargo, aun teniendo este lenguaje unas características admirables en muchos aspectos, lo cierto es que no es utilizado de forma mayoritaria más que en ciertos ámbitos muy restringidos. Sin ninguna duda, los lenguajes conocidos como "imperativos" siguen siendo los más habituales.

En estos últimos años, han cobrado gran importancia los lenguajes "orientados a objetos", pero se ha producido un fenómeno curioso a nuestro entender: las implementaciones que han tenido más éxito no han sido aquellas que siguen la filosofía de la orientación a objetos de una forma rigurosa, sino más bien "superconjuntos" de lenguajes imperativos tradicionales a los que se incorporan extensiones para el manejo de los elementos del nuevo paradigma de programación. Estamos hablando, evidentemente, de C++ [Stroustrup 1984, 1986], del sistema que Borland Internacional añade a su implementación del lenguaje PASCAL [Borland 1992b], y otros casos similares.

La elección del lenguaje concreto para el estudio es entonces sencilla. El lenguaje PASCAL es desde hace muchos años el paradigma de esta familia de lenguajes, tomando el relevo del lenguaje ALGOL, y su popularidad aún no ha decaído. El lenguaje C, por contra, es posiblemente el más utilizado hoy en día,

pero adolece de una serie de características que harían que el estudio no fuera suficientemente general: la estructura de bloques y los parámetros con otros mecanismos de transferencia (referencia, subprogramas formales), son aspectos, entre otros, que evidentemente tendrán una importancia fundamental en la optimación, y que quedarían sin examinar si se eligiera el lenguaje C. Sin embargo, pensamos que otras características presentes en lenguajes como MODULA o ADA y ausentes de PASCAL no serán, posiblemente, tan determinantes como las reseñadas. Por lo tanto, el trabajo se concretará en el lenguaje PASCAL, teniendo en cuenta además que existe un aspecto en el que sí sigue siendo el lenguaje por excelencia, aspecto que es uno de los primeros que dan interés a la optimación que se está planteando: la docencia de la programación, como quedó claro en la revisión de diversos textos realizada al inicio de la sección 2.2.

Consideramos en un primer momento la posibilidad de aplicar una estrategia de transformación de programas para la conversión de la recursividad final. Dijimos en la sección 2.3 que la literatura de este campo consideraba este proceso como resuelto, y mayormente trivial. Una búsqueda más concreta en el lenguaje PASCAL nos llevó a dos resultados muy significativos. Colussi [1984] plantea la solución de esquemas recursivos más generales en PASCAL, incluyendo la recursividad final como caso particular; sin embargo, para ello debe llevar a cabo restricciones muy drásticas en los programas a transformar. Concretamente, no puede manejar funciones, solo admite parámetros por valor y ha de prescindir de sentencias `for`.

Más sorprendentes fueron los resultados de Rohl [1984: 166-189], en un texto también específicamente en PASCAL dedicado en exclusiva a la recursividad. En ellos, se reconoce que la conversión de la recursividad final consistente en la sustitución de la llamada por una actualización de los parámetros y un salto incondicional al comienzo del procedimiento, con la posterior conversión en un ciclo `while`, no puede aplicarse a funciones recursivas, ni a procedimientos con parámetros por referencia que cambien en la llamada, es decir, básicamente los resultados de Colussi [1984]. Lo sorprendente es que, aun quitando esos casos, la transformación tampoco se aplica más que a esquemas simples de recursividad final de procedimientos. Concretamente, no se puede realizar la obtención del ciclo `while` para procedimientos recursivos en que existen varios casos base con condiciones de

tipo *elsif* (llamado *protasis problem*), o cuando existen varios casos recursivos del mismo tipo. La solución de Rohl [1984] incluye la adición de una variable centinela, pero solo es aplicada a los ejemplos que se muestran, y no hay indicios de una posible generalización mecanizable. Por otro lado, la recursividad final no directa (mutua) ni siquiera se menciona en estos estudios.

Un estudio más exhaustivo de gran número de esquemas recursivos finales nos permitió ver que efectivamente esta estrategia de transformación de programas para la recursividad final, entendida como resuelta y trivial, no lo era tanto al considerar todos los aspectos de un lenguaje "real" como PASCAL (iremos observando las razones de esta afirmación a lo largo del cuerpo de la tesis, entendiéndolo por tanto, como un pequeño objetivo aledaño al propósito principal). Se entiende resuelta porque la idea de sustituir la llamada por una actualización de los parámetros y un salto al comienzo del procedimiento es en efecto general (salvo para las restricciones mencionadas); sin embargo, la conversión en un programa iterativo (no basado en la sentencia *goto*) no solo no es trivial, sino que en algunos casos intrincados puede ser difícil e incluso imposible, o al menos propensa a errores. De hecho, no dejan de encontrarse en la literatura reseñas de la eliminación "manual" de esta clase de recursividad, pero nunca se llevan más allá del ejemplo (citamos, como más recientes, los trabajos de Pittman y Peters [1992: 343-345], y Swan [1994]).

Por ello, la estrategia que mejor parece funcionar es la que puede aplicar un compilador como una optimización: el problema anterior no existe, pues los saltos incondicionales quedan perfectamente ocultos al programador (solo aparecen en el código objeto). Así que al fin y al cabo, la filosofía que usa el lenguaje SCHEME, resolviendo la recursividad final a través de la propia implementación del lenguaje y como resultado de las características del modelo dinámico asociado al mismo, se muestra la más adecuada para nuestro propósito.

El objetivo de esta tesis es, entonces, la modificación del modelo dinámico del lenguaje PASCAL que permita la ejecución de programas recursivos finales sin consumo de la pila de activaciones. Tal modificación consistirá en plantear un modelo dinámico alternativo, en el que se hará un especial tratamiento de las activaciones finales, como sintaxis que generaliza la recursividad final, y que permitirá, por ejemplo, la consecución de las mismas

cualidades para subprogramas indirectamente recursivos en cualquier grado, si las llamadas entre ellos son finales.

La solución propuesta para el problema habrá de compararse, como hemos dicho, con la otra estrategia posible: la aplicación de transformaciones. Aunque ya *a priori* se ha encontrado que tal idea solo puede llevarse a cabo en un exiguo rango de programas finales, habrá de mostrarse cuán exiguo es, y las causas de tal circunstancia. Esto puede ser de interés a la hora de plantear la conversión de otros esquemas recursivos no finales.

Después deberá estudiarse la influencia de los diversos elementos del lenguaje en la posibilidad de practicar el tratamiento particular de tales llamadas finales. Finalmente, adoptando el punto de vista del diseñador de un compilador de PASCAL (más aún, lo más parecido posible a la construcción clásica de dichos compiladores), habrán de analizarse las implicaciones que tiene para tal tarea el uso de este modelo dinámico alternativo, evaluando tanto el comportamiento de los programas optimados como los requisitos estáticos para la consecución de los mismos. Los resultados teóricos han de conjugarse con la construcción efectiva de un compilador de PASCAL que muestre el comportamiento requerido.

El punto de partida es, obviamente, el trabajo de Hanson [1990] examinado en el apartado 2.4.2. Comentamos allí que, en principio, Hanson aplica su estrategia al lenguaje SCHEME en concreto, aunque no hay duda de que, como idea general, sin entrar en demasiados detalles, se podría aplicar a otros lenguajes. Pero no está claro, a nuestro entender, hasta qué punto las peculiaridades de SCHEME, como lenguaje que ciertamente tiene algunas características distintas de las que se encuentran en los lenguajes imperativos como PASCAL, influyen de manera determinante en la posibilidad de tal estrategia de vinculación en la pila. A este respecto, Hanson tampoco aclara algunos detalles de su implementación, principalmente en lo referente al mecanismo de transmisión de parámetros, lo cual cual ser un detalle bastante importante a tener en cuenta.

Por otro lado, no pretendemos una mera aplicación de sus resultados a PASCAL, por dos razones. Primera: la estrategia de Hanson presenta, como se ha comentado, ciertos aspectos negativos en cuanto a sus necesidades dinámicas, y solo algunas de ellas pueden ser relajadas gracias a un notable esfuerzo de

análisis estático (sin embargo, no queremos dejar de recalcar que estos aspectos negativos no ensombrecen en modo alguno la importancia de sus resultados); ambos aspectos están un tanto alejados de lo que suele ser habitual en la compilación e implementación de PASCAL. Segunda, relacionada con esto último: en efecto, la implementación de PASCAL sigue tradicionalmente unas pautas establecidas desde hace años, de la que solo se desvían en pequeños detalles las implementaciones concretas que se conocen (lo que contrasta con SCHEME, que es un lenguaje más "abierto" en ese aspecto). Entonces, podría suponer un cambio poco atractivo para los diseñadores de compiladores ya existentes la adopción de una estrategia tan novedosa como la de Hanson para conseguir el objetivo de la optimación de la recursividad final.

Por ello, planteamos como hipótesis de trabajo la adopción del modelo dinámico habitual en un lenguaje imperativo y particularmente en PASCAL, sin ninguna modificación del mismo al estilo de la que adopta Hanson al separar la información de control y la información de ambiente. Ello nos permitirá detectar hasta qué punto esta elección influye en la posibilidad de la optimación: es de esperar que aparezcan dificultades, y nuestra tarea será evaluar el impacto de las mismas. Por otro lado, de tener éxito, los resultados que obtengamos, si no requieren un excesivo esfuerzo estático ni suponen un empeoramiento de alguna propiedad dinámica, podrían ser de inmediata aplicación a los compiladores existentes para el lenguaje PASCAL. Los detalles concretos de esta hipótesis se recogen en el siguiente apartado, que cierra este capítulo introductorio.

2.5.1 Modelo dinámico estándar del lenguaje PASCAL

La referencia del lenguaje PASCAL es la definición estándar ISO del comité IEC JTC1/SC22, tal como se recoge en la revisión de la obra original de Jensen y Wirth [1975], llevada a cabo en 1991 por A.B. Mickel y J.F. Miner, y a la que nos referiremos como "[Jensen y Wirth 1991]" en múltiples ocasiones.

El modelo dinámico del lenguaje se basa, a grandes rasgos, en las ideas que aparecen en el apartado 2.2.1. En cuanto a los detalles, pueden existir ligeras variaciones (a lo largo de diferentes textos e implementaciones) en aspectos puntuales de la estructura del registro de activación y otros factores,

pero la filosofía general es común. Se elegirá una estructura particular a la que se aplicarán los estudios necesarios. Más adelante, sin embargo, deberá examinarse la influencia de esas posibles pequeñas variaciones en los resultados que se obtengan.

Para simplificar, nos referiremos habitualmente a la pila de activaciones como *pila de control* (y más simplemente, *pila* en las figuras) a pesar de lo no totalmente correcto de esta denominación. La organización del registro de activación elegida sigue, con pequeñas modificaciones, la descrita por Aho, Sethi y Ullman [1990: 408-411, 416-423, 429-437], Fischer y LeBlanc [1988: 287-308] o cualquier otro texto de compiladores. Esa estructura es la que aparece en la figura siguiente.

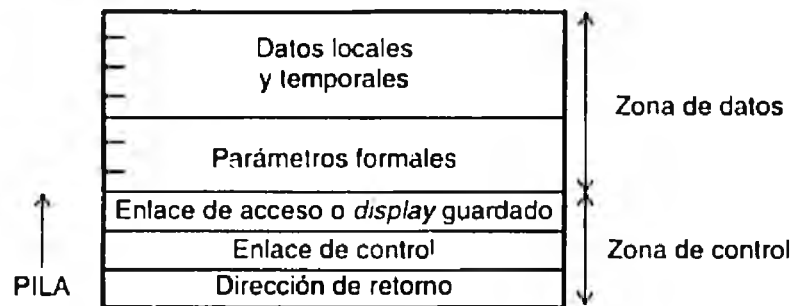


Figura 2.5.1 Estructura del registro de activación de un procedimiento en PASCAL.

En principio, para sentar las hipótesis de trabajo, nos restringiremos al caso de los subprogramas de tipo procedimiento de PASCAL, que son más simples, dejando el caso más complejo de las funciones para más adelante, ya dentro del cuerpo de la tesis (*vide* capítulo 5). La diferencia más notable con la Figura 2.2.2 consiste en incluir en la llamada *zona de control* del registro de activación el enlace de acceso o registro de *display* guardado (indicadores necesarios para el acceso a los datos no locales; en general, estudiaremos ambas estrategias en el modelo dinámico alternativo, aspecto que, incidentalmente, tampoco es contemplado por Hanson [1990]). La razón de esto es que, a efectos prácticos, las operaciones de las secuencias de llamada y retorno de subprogramas pueden dividirse mejor entre aquellas que gestionan los parámetros, por un lado, y aquellas que gestionan el resto de elementos del registro de activación, por otro, siendo estos últimos los que engloban la ahora llamada zona de control.

Las diferencias respecto a la estructura original de Aho, Sethi y Ullman [1990] son las siguientes:

- No existe una zona para el valor de retorno (solo existirá en los registros de activación de las funciones).
- La zona de parámetros formales se coloca por encima de la zona de control; la razón de esta modificación es simplemente mantener estos elementos en una posición fija dentro de cualquier registro de activación, independientemente del procedimiento. Las únicas razones aparentes para colocar los parámetros por debajo de la información de control parecen ser, por un lado, la posibilidad de procedimientos con un número variable de parámetros [Aho, Sethi y Ullman 1990: 421], y por otro, el modo de valor-resultado para el paso de parámetros. Puesto que ambas posibilidades están ausentes en PASCAL, al situar la información de control en posiciones fijas dentro del registro de activación resulta una mayor sencillez. De todos modos, esta será una de las variaciones más significativa que se considerará para generalizar los resultados.
- En principio, se utilizará la técnica de la cadena estática para el acceso a los datos no locales, pero se estudiará también la posibilidad de usar la técnica de *display*.
- No se considera necesario almacenar ninguna información sobre el estado de la máquina, simplemente la dirección de la instrucción de retorno.

Se supone que la zona de control del registro de activación ocupa un tamaño fijo para todo procedimiento, y que entonces existen dos valores globales (normalmente registros de la máquina) llamados *BasePila* y *TopePila*, que señalan en cada instante la situación del registro de activación actual en la pila de activaciones; concretamente, *BasePila* es la dirección a partir de la cual está la zona de datos del registro de activación, y *TopePila* es la primera dirección libre por encima del registro de activación, como se ve en la Figura 2.5.2.

Los enlaces de control y de acceso y, en su caso, los registros de *display*, tendrán valores previos de *BasePila* para los registros de activación a los que apunten. De este modo, es posible acceder mediante *BasePila* tanto a los datos (por desplazamientos relativos positivos) como a los elementos de control

(por desplazamientos negativos), reservándose *TopePila* para la activación de otros procedimientos o funciones.

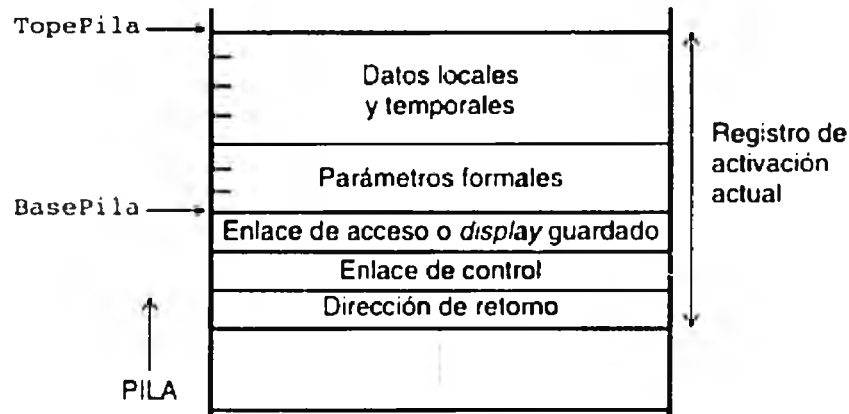


Figura 2.5.2 Modo de acceso a la información local del registro de activación

Esta disposición sigue básicamente a Ammann [1977] (específicamente para uno de los primeros compiladores del lenguaje PASCAL y muchos de sus descendientes directos), Rees y Robson [1988: 108-111] (también específicamente para el subconjunto de PASCAL conocido como PASCAL-S [Wirth 1981]), y Fischer y LeBlanc [1988]. La diferencia con lo expuesto en Aho, Sethi y Ullman [1990] radica únicamente en la utilización del apuntador adicional *TopePila*, lo cual está motivado por un ánimo de simplificación, pues la posición que señala puede ser conocida estáticamente en todo instante. Realmente, es obligada la inclusión de este apuntador si los registros de activación de un subprograma pueden tener tamaño variable, por ejemplo, cuando parámetros formales o datos locales son *arrays* cuyo tamaño exacto se establece dinámicamente [Aho, Sethi y Ullman 1990: 412-422]; como esta posibilidad puede darse en ciertas implementaciones del lenguaje (las llamadas de "nivel 1" [Jensen y Wirth 1991: 200]), se decidió incluir este apuntador.

El uso de *TopePila* tiene también que ver con la filosofía de manejo de las localidades temporales; de acuerdo a la que se ha elegido, se supone que todos los registros de activación de un mismo procedimiento tienen siempre el mismo tamaño total, que incluye todo el espacio temporal que sea necesario a lo largo de su activación, cualquiera que esta sea. Otras implementaciones pueden, sin embargo, hacer que el tamaño del registro de activación varíe a lo

largo de una activación, dependiendo de las necesidades de espacio temporal en cada momento. En ambas alternativas, sin embargo, puede determinarse durante la compilación la primera posición libre por encima del registro de activación, pero la complejidad de ello puede llegar a ser importante, por lo que la utilización de un registro especial para este propósito simplifica notablemente el problema.

El otro aspecto que integra el modelo dinámico del lenguaje es la descripción de los procesos que se ejecutan para la activación y la terminación de los subprogramas, procesos que se conocen, como sabemos, con los nombres de *secuencia de llamada* y *secuencia de retorno*, respectivamente. En una llamada que efectúa un subprograma a otro, denominaremos *subprograma llamador* al que invoca o realiza la llamada, y *subprograma receptor* al que es llamado².

La secuencia de llamada para procedimientos correspondiente al esquema hasta ahora presentado incluye el conjunto de operaciones que se recoge en el Algoritmo 2.5.1 (donde se distinguen las operaciones alternativas según se utilice el enlace de acceso o el *display* para el acceso a los datos no locales). Se ha variado un poco el orden de las operaciones respecto a la secuencia original de Aho, Sethi y Ullman [1990], principalmente para facilitar el cálculo de la dirección de retorno. El conjunto de operaciones contempladas en el Algoritmo 2.5.1 se denominará, a partir de ahora, *secuencia de llamada estándar de procedimientos en PASCAL*; a veces nos referiremos a ella sin aludir explícitamente a su situación (Algoritmo 2.5.1).

La secuencia de retorno que debe ejecutarse cuando el todavía receptor de la llamada finaliza su tarea será la que se presenta en el Algoritmo 2.5.2. En el caso de los procedimientos, el llamador puede continuar su código desde el paso R5 sin ninguna operación adicional (en el caso de las funciones, sin embargo, será necesario recoger el valor de retorno de la pila, como veremos en el capítulo 5). Nos referiremos al Algoritmo 2.5.2 a lo largo del texto como *secuencia de retorno estándar de procedimientos en PASCAL*.

La estructura del registro de activación mostrada en la Figura 2.5.1, el uso de dos registros específicos para indicar la situación del registro de activación

² Quizás los términos "llamador" y "receptor" puedan evocar alguna otra idea en el lector. Su elección ha venido dada por la necesidad de contar con dos palabras sencillas para referirse al concepto, pues va a aparecer constantemente a lo largo del trabajo, en inglés, son habituales las denominaciones *caller* y *callee*.

- L1: el llamador evalúa los parámetros actuales, y coloca los valores obtenidos por encima de su registro de activación (esto es, la dirección *TopePila* más el tamaño del nuevo registro de activación);
- L2: el llamador calcula el enlace de control (simplemente es *BasePila*) y lo coloca por encima de su registro de activación (de nuevo, usando *TopePila*);
- L3: el llamador calcula el enlace de acceso (que es *BasePila* o se obtiene siguiendo la cadena de enlaces de acceso hacia la base de la pila) o toma el registro de *display* correspondiente a la profundidad de anidamiento del subprograma receptor, dejando en cualquier caso el valor por encima de su registro de activación, mediante *TopePila*;
- L4: el llamador coloca la dirección de retorno (dos o tres instrucciones más que la actual, según la siguiente operación) por encima de su registro de activación, también usando *TopePila*;
- L5: el llamador actualiza *BasePila* a su nuevo valor, que es *TopePila* más el tamaño de la zona de control; si se usa el *display*, además se hace que el registro de *display* correspondiente a la profundidad de anidamiento del subprograma receptor tenga el valor dado a *BasePila*;
- L6: el llamador salta (jump) a la primera instrucción del código del receptor;
- L7: el receptor actualiza *TopePila* a su nuevo valor, que es *BasePila* más el tamaño de la zona de datos;
- L8: el receptor asigna valores iniciales a sus datos locales.

Algoritmo 2.5.1 Secuencia de llamada estándar de un procedimiento en PASCAL

del subprograma en ejecución (el de la cumbre de la pila), tal como se refleja en la Figura 2.5.2, y las secuencias de llamada y retorno de procedimientos (Algoritmo 2.5.1 y Algoritmo 2.5.2, respectivamente), conforman el que denominaremos *modelo dinámico estándar para procedimientos en PASCAL* (por ello, las secuencias mostradas también se califican como "estándar" a lo largo del trabajo).

- R1: el receptor, si es el caso, restablece la situación del *display* previa a la llamada, almacenando en el registro de *display* correspondiente a su nivel de anidamiento el valor guardado en su registro de activación; si se utiliza el enlace de acceso, no debe hacerse ninguna operación;
- R2: el receptor recupera el valor anterior de *TopePila*, que es *BasePila* menos el tamaño de la zona de control;
- R3: el receptor recupera el valor anterior de *BasePila*, que es el enlace de control de su registro de activación; para ello, usa el valor de *TopePila*;
- R4: el receptor salta (jump) a la instrucción de retorno del llamador, que se obtiene también a partir del registro *TopePila*;
- R5: el llamador continúa su ejecución.

Algoritmo 2.5.2 Secuencia de retorno estándar de un procedimiento en PASCAL

Nuestro objetivo a partir de ahora es modificar ciertos aspectos de este modelo dinámico, para obtener un *modelo dinámico optimado para procedimientos*, cuya característica distintiva sea que no se produzca un crecimiento de la pila de control en un proceso expresado mediante recursividad final. Los aspectos a modificar son exclusivamente las secuencias de llamada y de retorno, mientras que se mantendrán invariables la estructura del registro de activación y el uso de los registros que señalan la cumbre de la pila. La razón ya se ha comentado: la única solución conocida hasta el momento del problema, la de Hanson [1990], tiene el inconveniente de desviarse notablemente de estas hipótesis, lo que la hace de difícil aplicación para un lenguaje cuya organización dinámica ya ha sido asentada definitivamente hace tiempo. Las ideas básicas de partida, así como el estudio completo necesario, ya son parte del cuerpo de la tesis, en el siguiente capítulo.

3. OPTIMACIÓN DE ACTIVACIONES FINALES RECURSIVAS DE PROCEDIMIENTOS

3.1 INTRODUCCIÓN

El objetivo de este capítulo es obtener una optimación de las activaciones finales recursivas de procedimientos en el lenguaje de programación PASCAL. La idea básica es que los procesos expresados sintácticamente mediante recursividad final pueden teóricamente ejecutarse sin crecimiento de la pila (es decir, convertirse en procesos iterativos), mientras que el modelo dinámico estándar del lenguaje provoca un crecimiento de la pila en cada llamada a un subprograma, independientemente de si es final o no. Examinaremos en primer lugar, en el siguiente apartado, un par de ejemplos con los que clarificar las ideas acerca de cual debe ser el comportamiento del programa para que no crezca el consumo de la memoria en las llamadas finales. En la otra sección del capítulo se expondrá la solución adoptada, junto con el estudio de todos los factores que intervienen en el problema en el lenguaje PASCAL. Finalmente, se compararán los resultados con otras estrategias conocidas.

3.1.1 Posibles optimaciones en PASCAL

El punto de partida del estudio incluye, como hemos mencionado en el capítulo anterior, las ideas básicas de conversión de la recursividad final en iteración siguiendo las técnicas habituales de transformación de programas, tal como se encuentran en Bird [1977a], Rohl [1984: 166-167], y más recientemente, en Sethi [1992: 146-150] y Pittman y Peters [1992: 343-345]. Tomemos un ejemplo muy sencillo.

```

program Prueba ...
--
  procedure p (x: integer);
  begin
    if x <> 0 then p (x-1)      ( activación recursiva final )
    else writeln (x)
    end;
--
begin
--
  p (3)
--
end.

```

Programa 3.1.1 Un procedimiento recursivo final en PASCAL

La llamada recursiva del procedimiento *p* señalada con el comentario en el programa anterior es final porque una vez que termina, la activación que la produjo no ha de hacer otra cosa que terminar a su vez. La situación de la pila de control a través de las sucesivas llamadas queda reflejada en la Figura 3.1.1.

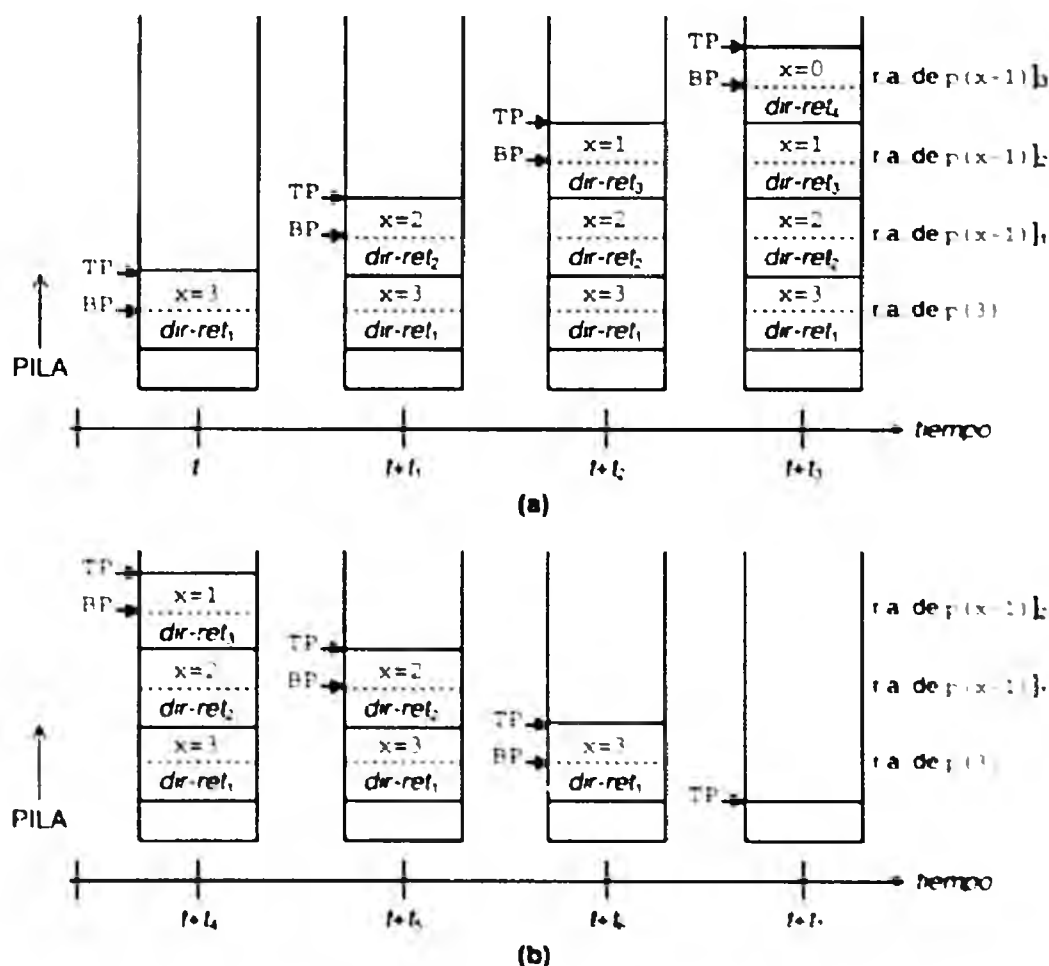


Figura 3.1.1 Instantáneas sucesivas de la pila de control durante la ejecución del Programa 3.1.1 (a) en las llamadas, y (b) en los retornos

Se muestran solamente la información específica de control y el espacio para el parámetro; BP y TP son abreviaturas de BasePila y TopePila, respectivamente. Tras la situación reflejada en la última instantánea de la Figura 3.1.1a, se ejecuta la sentencia no recursiva y se entra en la secuencia de retornos de todas las activaciones, reflejadas en la Figura 3.1.1b.

Ahora consideremos el programa resultante de aplicar la eliminación de la recursividad final según las técnicas conocidas de transformación de programas, tal como hace por ejemplo, Rohl [1984: 167]. Para ello, se sustituye la llamada recursiva por una actualización del parámetro y un salto incondicional al comienzo del código del procedimiento.

```

program PruebaNoRecursivo ...
  procedure p (x: integer);
  label 1;
  begin
  1:  if x <> 0 then
      begin
        x := x-1;      ( actualización del parámetro )
        goto 1;        ( salto al comienzo )
      end
    else writeln (x)
    end;
  begin
    p (1)
  end.

```

Programa 3.1.2 Procedimiento no recursivo equivalente al del Programa 3.1.1

El comportamiento de este programa no recursivo en los diferentes saltos (que equivalen a los momentos en que se efectuaban las llamadas recursivas) se muestra en la siguiente figura.

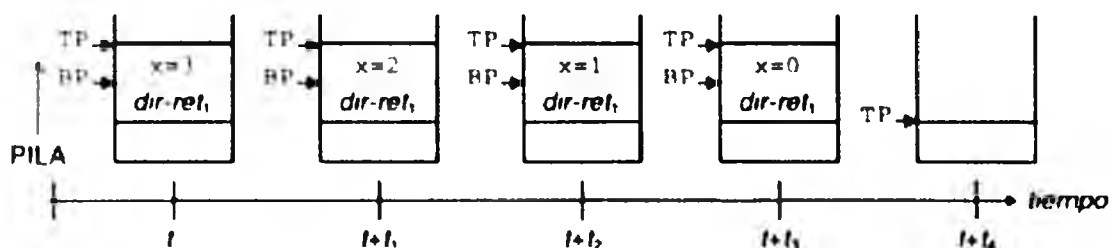


Figura 3.1.2 Instantáneas sucesivas de la pila de control durante la ejecución del Programa 3.1.2

Efectivamente, el comportamiento es iterativo: no hay consumo de memoria a lo largo del proceso.

Ahora pensemos, sin embargo, en cómo se podría generar código para el Programa 3.1.1, pero de manera que el comportamiento fuera el reflejado en la Figura 3.1.2. Tengamos en cuenta dos consecuencias del hecho de que la llamada sea final: primero, el ambiente de cada activación se vuelve innecesario una vez que se efectúa la llamada siguiente, y segundo, la información de control es en cierto modo redundante (salvo para la primera activación del proceso), pues una vez alcanzada la activación que termina el proceso recursivo, lo único que se hace es recoger sucesivamente una dirección de retorno y saltar a ella. Por lo tanto, en cuanto al primer aspecto, no sería necesario acumular espacio para el parámetro en cada activación recursiva final, sino reutilizar el espacio dedicado al mismo en la primera llamada que inició el proceso recursivo. Y en cuanto al segundo aspecto, si se mantiene la dirección de retorno de esa primera activación, se evitarían todas las secuencias de retorno reflejadas en la Figura 3.1.1b.

Ello podría conseguirse si en lugar del proceso habitual de llamada, que implica la acumulación de un nuevo registro de activación en la pila de control, se realiza un proceso alternativo, en que no se crea un nuevo registro, sino que se mantiene en la cumbre de la pila la estructura completa del registro de activación del llamador, evitando modificar ciertos componentes (concretamente la dirección de retorno) y actualizando otros (los parámetros formales).

La secuencia de llamada estándar (Algoritmo 2.5.1) puede modificarse para conseguir este objetivo. Primero, el proceso de paso de parámetros puede realizarse sobre la dirección del parámetro formal del propio registro de activación, no como una acumulación de nueva información en la pila. Segundo, tampoco hay que acumular una dirección de retorno. Tercero, los registros *BasePila* y *TopePila* se mantienen sin modificación, porque el registro de activación que debe quedar en la pila tiene la misma estructura que el que se pierde. Por lo tanto, una posible *secuencia de llamada optimizada* solo precisa de una actualización de los parámetros (alternativa del paso L1 del Algoritmo 2.5.1 de llamada estándar) y un salto al comienzo del código (paso L6 de la misma secuencia), tal como efectivamente pone de relieve el Programa 3.1.2.

El resto de operaciones del Algoritmo 2.5.1 son, en principio, innecesarias, a la espera del estudio más riguroso que considere todos los aspectos involucrados y que aquí no se han tenido en cuenta. Con esta *secuencia de llamada optimada* se consigue el comportamiento en espacio constante, exactamente como el de la Figura 3.1.2. Además, el hecho de que no se precisen gran parte de las operaciones de la secuencia de llamada estándar y el hecho de que resulten eliminadas todas las secuencias de retorno hacen que la ejecución sea, en principio, más rápida, consumiendo prácticamente el mismo tiempo que el propio Programa 3.1.2, lo que es otra ventaja añadida de la optimación que habrá que investigar.

La profundización en los demás aspectos involucrados y el estudio de otros elementos del lenguaje PASCAL que complican la situación, se hará en las siguientes secciones, donde encontraremos la definición de "posición final", las condiciones necesarias y suficientes para realizar la optimación y la secuencia optimada de llamada definitiva.

Antes de ello, consideremos otro programa sencillo en PASCAL. Este programa no es recursivo, en el sentido literal de la palabra, pero contiene una activación de procedimiento que es final.

```

program OtraPrueba
  procedure q (y, z: integer);
  procedure p (x: integer);
  begin
    if x <= 0 then
      q (x-1, x+1)      ( activación no recursiva final )
    else writeln (x)
    end;
  begin
    p (5)
  end;
end;

```

Programa 3.1.3 Un procedimiento con una activación no recursiva final en PASCAL

La situación de la pila de control durante la ejecución de las llamadas contenidas en el Programa 3.1.3 se refleja en la figura que aparece a continuación.

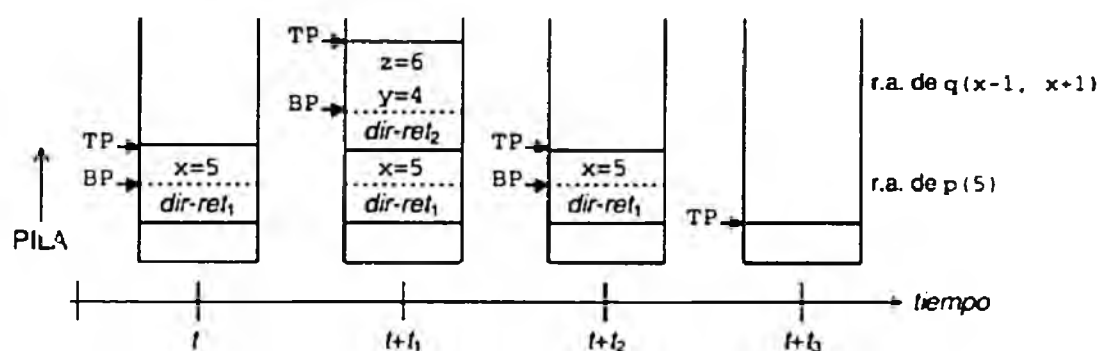


Figura 3.1.3 Instantáneas sucesivas de la pila de control durante la ejecución del Programa 3.1.3

Se puede seguir un razonamiento similar al usado en el caso anterior, pero ahora con ciertas condiciones. Si el procedimiento q no está definido localmente en p , los nombres locales en p no son visibles para él, por lo que no podrá acceder a los datos del registro de activación de p ; como este procedimiento tampoco volverá a usar dichos datos, pues la activación de q es final, entonces toda la parte de datos del registro de activación de p es superflua: una vez que se activa q ya no se necesita. Por otra parte, debido a la misma característica final de la activación de q , el proceso de retorno de esta podría hacerse a la dirección de retorno de la activación de p directamente.

Por lo tanto, ante una activación final no recursiva de un procedimiento, en vez de acumular un nuevo registro de activación, habría que conseguir colocarlo en lugar del registro de activación del llamador, de manera que la dirección de retorno fuera la de este último, y se utilizasen sus objetos locales de la forma adecuada para actualizar los objetos locales del procedimiento receptor de la llamada. Un comportamiento, en definitiva, como el reflejado en la Figura 3.1.4, y que, salvo en las posibles diferencias de tamaño entre los registros de activación del llamador y del receptor, puede calificarse en términos genéricos como de nulo crecimiento neto de la pila de control.

Para conseguir este comportamiento, parece claro que va a haber que realizar en la secuencia de llamada alternativa operaciones distintas que en la secuencia de llamada optimada apuntada antes para la activación recursiva del Programa 3.1.1. El paso de parámetros tendrá que usar una estrategia diferente, pues la estructura de la zona de parámetros del llamador puede ser distinta que en el receptor. Lo que sí se mantiene invariable, como antes, es la dirección de retorno, pero el registro $TopePila$ debe modificarse, como queda claro en la

Figura 3.1.4. Por lo tanto, las operaciones de llamada alternativas en este caso no pueden ser las mismas, pero puede darse una optimización que elimine información no necesaria y evite una secuencia de retorno.

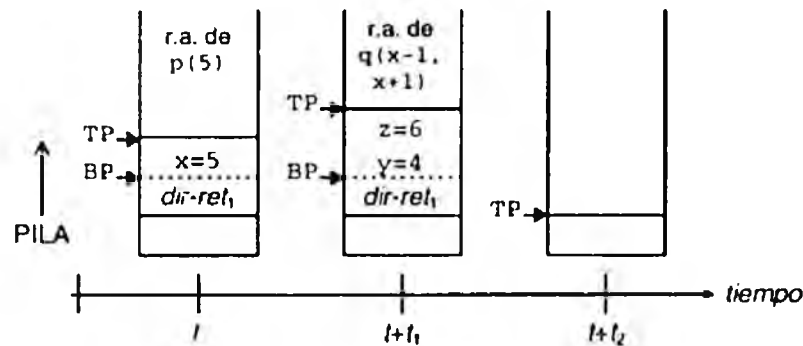


Figura 3.1.4 Instantáneas sucesivas de la pila de control durante la ejecución del Programa 3.1.3 después de la optimización de la llamada

Este caso no sería demasiado interesante de cara al estudio de los procesos recursivos si nos olvidáramos de una manera más o menos habitual de expresar los mismos: la *recursividad mutua*. En efecto, supongamos que el procedimiento q contiene a su vez una llamada final al procedimiento p: los procedimientos son entonces mutuamente recursivos.

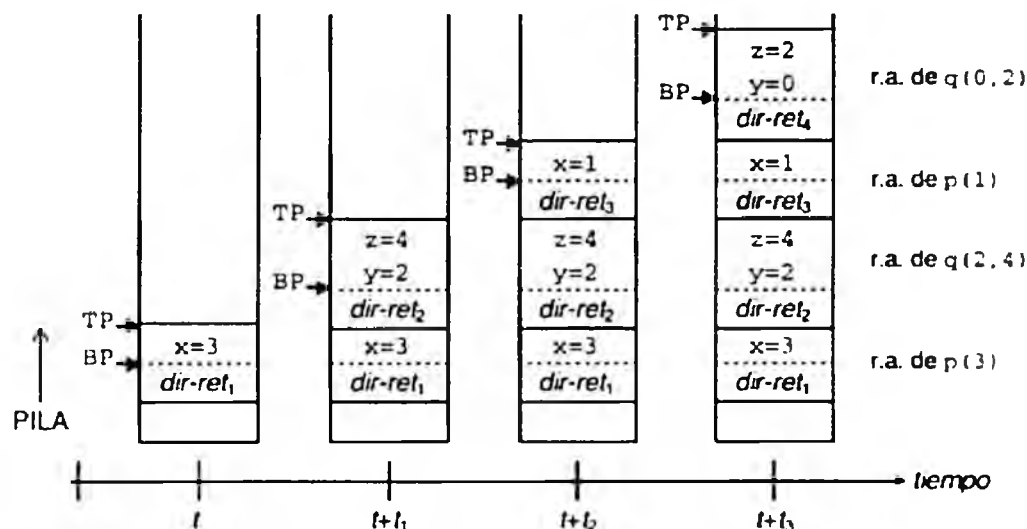
```

program OtraPrueba _
  procedure q (y, z: integer); forward;
  procedure p (x: integer);
  begin
    if x <> 0 then
      q (x-1, x+1)      ( activación no recursiva final )
    else writeln (x)
    end;
  procedure q;
  begin
    if y <> 0 then
      p (y-1)           ( activación no recursiva final )
    else writeln (z)
    end;
  begin
    p (2)
  end
end

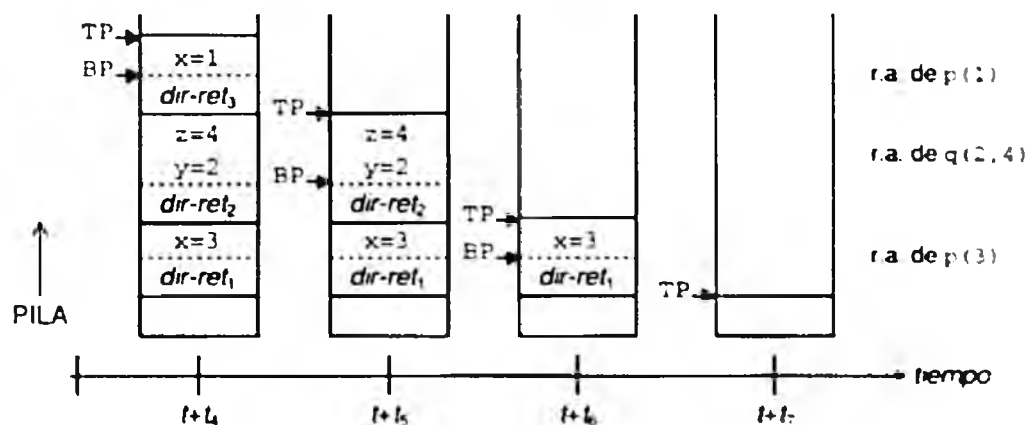
```

Programa 3.1.4 Dos procedimientos mutuamente recursivos en PASCAL

Veamos la ejecución del nuevo programa siguiendo el modelo dinámico estándar.



(a)



(b)

Figura 3.1.5 Instantáneas sucesivas de la pila de control durante la ejecución del Programa 3.1.4 (a) en las llamadas, y (b) en los retornos

Si se procede con la optimización mencionada, el comportamiento se convierte en el reflejado en la Figura 3.1.6.

La importancia de esta segunda clase de optimización se hace evidente. Todo el proceso recursivo expresado mediante recursividad mutua se ejecuta en espacio constante, sin crecimiento de la pila. Con ello, no solo la recursividad

“directa” sino cualquier proceso recursivo, en el sentido genérico del término, llevaría a un comportamiento iterativo. Por supuesto, siempre que la recursividad se exprese sintácticamente en las condiciones de posición final.

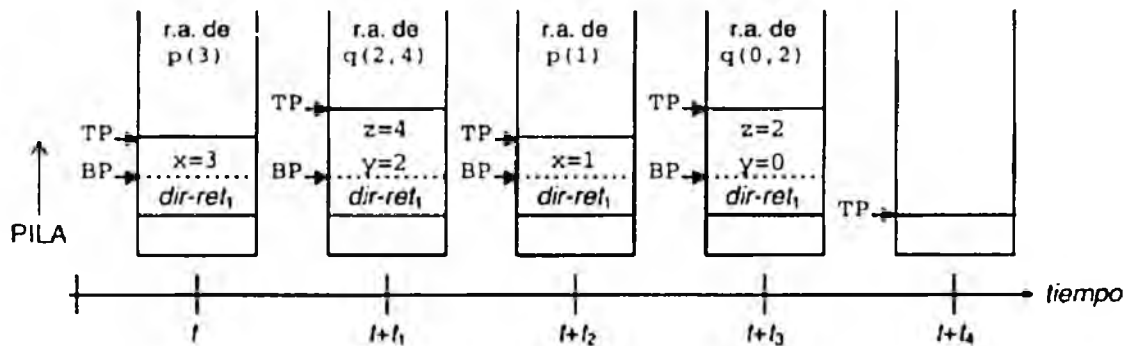


Figura 3.1.6 Instantáneas sucesivas de la pila de control durante la ejecución del Programa 3.1.4 cuando se ha efectuado la optimización

Realmente, en SCHEME no existe la distinción que hemos planteado nosotros: una llamada final de procedimiento, participe o no participe en un proceso recursivo en el sentido estricto del término, se ejecuta sin acumular ninguna información de control en la pila. Sin embargo, vistas las diferencias que existen entre las dos clases de activaciones en el lenguaje PASCAL, conviene establecer una distinción en su estudio:

- llamaremos activaciones *recursivas* a aquellas en que se invoca el mismo procedimiento que está efectuando la llamada;
- llamamos activaciones *no recursivas* a las que invocan un procedimiento distinto del que efectúa la llamada.

Quizá convendría más utilizar los términos “recursivas directas” y “recursivas no directas”, respectivamente, pero la nomenclatura puede llegar a ser engorrosa. Y decimos esto porque, efectivamente, una activación no recursiva según nuestra denominación puede llegar a producir, como hemos visto, recursividad en el sentido genérico de esta palabra, en el caso de que el proceso se exprese mediante recursividad mutua o indirecta.

Entonces, en la activación denominada “recursiva”, la optimización solo ha de actualizar ciertos valores del registro de activación de la cumbre de la pila para que sea el registro de la nueva activación: toda su estructura es

exactamente la misma. Por contra, en la activación que se denominó “no recursiva”, la estructura del nuevo registro de activación será distinta (en general) a la del registro al que reemplaza, por lo que las operaciones a ejecutar parecen, *a priori*, más complejas que en el caso anterior. El resto del presente capítulo se dedica, por ello, a las activaciones finales recursivas, como ahora las llamamos, dejando la optimización de las activaciones finales no recursivas para el siguiente capítulo.

3.2 MODELO DINÁMICO OPTIMADO: ACTIVACIONES FINALES RECURSIVAS DE PROCEDIMIENTOS

El lenguaje PASCAL cuenta con algunos aspectos que dificultarán e incluso impedirán la optimización de activaciones finales recursivas de procedimientos. En términos de ellos se obtendrán en este apartado las condiciones necesarias y suficientes para la optimización y la secuencia optimada de llamada. En primer lugar, se define la primera de esas condiciones, el concepto de “posición final” en PASCAL. Después se exponen las secuencias de llamada y de retorno que realizan la optimización de las activaciones finales recursivas. En el resto de apartados de la sección se consideran los elementos del lenguaje que obligarán a plantear restricciones adicionales para llevar a cabo la optimización.

3.2.1 Definición de posición final en PASCAL

La activación de un procedimiento es, en PASCAL, una clase de sentencia [Jensen y Wirth 1991: 170]. Definiremos entonces cuando una sentencia está en posición final. Un bloque se compone en PASCAL de una parte de declaraciones y una sentencia compuesta, cuya sintaxis es la siguiente [op. cit.: 172]:

$$\langle \text{comp-stment} \rangle \rightarrow \text{begin } \langle \text{stment-list} \rangle \text{ end}$$

$$\langle \text{stment-list} \rangle \rightarrow \langle \text{stment} \rangle \langle \text{resto-stment-list} \rangle$$

$$\begin{aligned} \langle \text{resto-stment-list} \rangle &\rightarrow ; \langle \text{stment-list} \rangle \\ &| \lambda \end{aligned}$$

Representaremos en forma de árbol sintáctico abstracto la anterior construcción, teniéndose:

$$(\text{SEQ } \langle \text{stment} \rangle_1 \langle \text{stment} \rangle_2 \dots \langle \text{stment} \rangle_n)$$

Entonces, la última sentencia de esta estructura está en posición final, debido al carácter secuencial de la sentencia compuesta [ibid.]: tras ejecutarse $\langle \text{stment} \rangle_n$, la activación del procedimiento termina, por lo que ciertamente dicha sentencia está en posición final; las sentencias anteriores están obviamente en posición no final.

Ahora estudiaremos qué sentencias pueden ocupar ese lugar final en la sentencia compuesta de un bloque. La sintaxis de la parte de sentencias de un bloque es la siguiente:

$$\begin{aligned} \langle \text{stment} \rangle &\rightarrow \langle \text{unlab-stment} \rangle \\ &| \text{label} : \langle \text{unlab-stment} \rangle \\ \langle \text{unlab-stment} \rangle &\rightarrow \langle \text{comp-stment} \rangle \\ &| \text{id } \langle \text{var-ref} \rangle := \langle \text{expr} \rangle \\ &| \text{id} \\ &| \text{id } (\langle \text{param-list} \rangle) \\ &| \text{goto label} \\ &| \text{if } \langle \text{expr} \rangle \text{ then } \langle \text{stment} \rangle \\ &| \text{if } \langle \text{expr} \rangle \text{ then } \langle \text{stment} \rangle \text{ else } \langle \text{stment} \rangle \\ &| \text{case } \langle \text{expr} \rangle \text{ of } \langle \text{case-list} \rangle \text{ end} \\ &| \text{while } \langle \text{expr} \rangle \text{ do } \langle \text{stment} \rangle \\ &| \text{repeat } \langle \text{stment-list} \rangle \text{ until } \langle \text{expr} \rangle \\ &| \text{for id} := \langle \text{expr} \rangle \text{ for-dir } \langle \text{expr} \rangle \text{ do } \langle \text{stment} \rangle \\ &| \text{with } \langle \text{record-var-list} \rangle \text{ do } \langle \text{stment} \rangle \\ &| \lambda \\ \langle \text{case-list} \rangle &\rightarrow \langle \text{case-label-list} \rangle ; \langle \text{stment} \rangle \langle \text{resto-case-list} \rangle \\ \langle \text{resto-case-list} \rangle &\rightarrow ; \langle \text{case-list} \rangle \\ &| \lambda \end{aligned}$$

Expresamos ahora en forma de árboles sintácticos las construcciones de entre las anteriores que pueden incluir otras sentencias, lo que señalamos con un subrayado:

```
(SEQ <stment>1 <stment>2 ... <stment>n)
(LABEL label <unlab-stment>)
(IF-THEN <expr> <then-stment>)
(IF-ELSE <expr> <then-stment> <else-stment>)
(CASE <expr> ((<case-list>1 <case-stment>1) ... ((<case-list>n <case-stment>n))
(WHILE <expr> <while-stment>)
(REPEAT <expr> <repeat-stment>)
(FOR id <init-expr> <final-expr> <direction> <for-stment>)
(WITH <record-var-list> <with-stment>)
```

Estudiamos caso por caso cuáles de las anteriores sentencias están en posición final:

- si una *sentencia compuesta* está en posición final, ya sabemos que la última sentencia de la lista está también en posición final, mientras que no lo está ninguna de las anteriores;
- si una *sentencia con etiqueta* está en posición final, la sentencia sin etiqueta que contiene también lo está, pues la presencia de la etiqueta es una cuestión puramente sintáctica: la ejecución de la sentencia con etiqueta consiste simplemente en la ejecución de la sentencia;
- si una *sentencia condicional* (es decir, la sentencia *if* o la sentencia *case* [Jensen y Wirth 1991: 172-173]) está en posición final, cualquiera de las sentencias que participan en la construcción (lo que llamaremos *rama* de la sentencia condicional) está en posición final: la ejecución de la sentencia condicional (de la clase que sea) consiste en la evaluación de cierta expresión y la ejecución de la sentencia de una de las ramas de la construcción correspondiente, sin ninguna operación adicional;
- las sentencias que participan en las *sentencias repetitivas* (es decir, la sentencia *while*, la sentencia *repeat* y la sentencia *for* [Jensen y Wirth 1991: 173-176]) están todas ellas en posición no final: en cada caso, tras la ejecución de la subsentencia, al menos ha de evaluarse una expresión antes de finalizar la sentencia repetitiva completa; por tanto, las construcciones para expresar iteración en PASCAL no mantienen la característica de posición final;

- si una sentencia *with* [Jensen y Wirth 1991: 176-177]) está en posición final, también lo está la sentencia que contiene: esta construcción es un mero artificio sintáctico sin influencia en el control del flujo del programa.

Siguiendo a Friedman, Wand y Haynes [1992: 251-257], si representamos con la letra T las posiciones en que una sentencia es final, y con la letra H aquellas que no son finales, tenemos la síntesis de lo anterior:

```
(SEQ H ... H T)
(LABEL label T)
(IF-THEN <expr> T)
(IF-ELSE <expr> T T)
(CASE <expr> (<case-list>1 T) ... (<case-list>n T))
(WHILE <expr> H)
(REPEAT <expr> H)
(FOR id <init-expr> <final-expr> <direction> H)
(WITH <record-var-list> T)
```

Tenemos entonces completa la definición de posición final para una sentencia en el lenguaje PASCAL. Denominamos *activación final recursiva de procedimiento* a una sentencia de activación de un procedimiento que sea recursiva (es decir, que se sitúe en la parte de sentencias del bloque del propio procedimiento que se llama) y que esté en posición final. Una *activación final no recursiva de procedimiento* será una sentencia de activación de un procedimiento distinto del que contiene en su parte de sentencias dicha activación, con tal que esté en posición final. El hecho de que solo interesan a la optimización que estudiamos esta clase de sentencias queda definido de la siguiente manera (para el caso recursivo, dejando el otro caso para el próximo capítulo):

Restricción 3.2.1. Si la sentencia de activación recursiva de un procedimiento no está en posición final en la parte de sentencias de su propio bloque, la optimización no puede llevarse a cabo. Una sentencia está en posición final si es la última que aparece en la sentencia compuesta del bloque de un subprograma, o si, recursivamente, es la sentencia de una sentencia con etiqueta en posición final, o es una rama de una sentencia condicional en posición final, o es la sentencia de una sentencia *with* en posición final, o es la última sentencia de una sentencia compuesta en posición final.

Denominamos "restricción" a este resultado porque es la primera condición necesaria que ha de cumplir una llamada a un procedimiento para poder realizar su optimación. Obsérvese, especialmente, que focalizamos nuestra atención en las activaciones de procedimientos desde otros procedimientos; el caso de una posible activación de un procedimiento desde una función se deja para el capítulo 5.

3.2.2 Optimación de la secuencia de llamada

La hipótesis de trabajo planteada en la introducción nos impone mantener la filosofía básica de máquina virtual asociada habitualmente al lenguaje PASCAL. Nuestro objetivo es modificar (lo más ligeramente posible) ese modelo dinámico estándar presentado en aquel momento: se seguirá realizando *stack-allocation*, los registros de activación tendrán la misma estructura, se usarán los apuntadores de pila *BasePila* y *TopePila* del mismo modo, continuaremos utilizando una cierta estrategia (enlace de acceso o *display*) para el acceso a los datos no locales, y así con casi todo. Nuestro nuevo modelo dinámico se diferenciará, en principio, solamente en las instrucciones que se realizan durante la secuencia de llamada de las activaciones finales recursivas de procedimientos que puedan ser optimadas (el resto de llamadas se tratará normalmente).

Observemos que una hipótesis implícita en el estudio de los dos ejemplos de la introducción, el Programa 3.1.1 y el Programa 3.1.3, es que la secuencia de retorno definida en el modelo dinámico estándar se mantiene invariable. Esta es una primera cuestión importante que debemos fijar. Supongamos que es el procedimiento *p* el que tiene o bien alguna activación final recursiva, o bien una activación final no recursiva de otro procedimiento *q*. Durante la compilación, podremos detectar sintácticamente esa situación y generar código alternativo para la llamada, o mejor dicho, para las operaciones (ya sean de llamada o de retorno) que corresponden al procedimiento *p* como llamador.

Sin embargo, sería difícil, por no decir imposible, que actuásemos sobre las operaciones que corresponden al receptor, ya sea el propio *p* u otro procedimiento, sobre todo porque ambos procedimientos pueden ser invocados en otro lugar, en posición no final, recursivamente o no recursivamente, así que

el código que se genere para ellos como receptores ha de servir para todas las posibles invocaciones. Por tanto, expresamos de forma explícita la siguiente hipótesis:

Condición 3.2.1. En el modelo dinámico optimado, las operaciones de las secuencias de llamada y de retorno que son realizadas por el receptor de la llamada han de mantenerse sin variación respecto al modelo dinámico estándar.

Usamos el término "condición" aquí para referirnos a un aspecto distinto del que se ha denominado anteriormente "restricción". Esta última es una condición necesaria de tipo sintáctico para proceder a la optimación, mientras que por "condición" entenderemos cierto requisito que ha de cumplir el modelo dinámico alternativo que se propone. Obsérvese por ejemplo que lo que venimos llamando "modelo dinámico estándar" admite en la práctica diferentes versiones en todos o casi todos sus aspectos constituyentes; del mismo modo, nosotros daremos una de las posibles secuencias de llamada que podrían llevar al comportamiento optimado. Se supone entonces que las "condiciones" que enunciaremos se aplican a todas ellas, sea cual sea la elegida en cada caso.

En la filosofía de máquina virtual elegida para el lenguaje PASCAL, la secuencia de llamada es responsabilidad primordialmente del procedimiento que invoca, así como la secuencia de retorno lo es del invocado, por lo que la Condición 3.2.1 se traduce en que la optimación afectará en todo caso a la secuencia de llamada, no variando la secuencia de retorno. Estudiaremos en este apartado esa modificación de la secuencia de llamada estándar que conduce al comportamiento optimado. Más adelante a lo largo de este trabajo deberán estudiarse esas otras versiones alternativas del modelo dinámico estándar, para comprobar si admiten la optimación en las condiciones que encontremos.

Retomamos lo discutido en la introducción, cuando descubrimos que podía no acumularse un nuevo registro de activación en la pila para una llamada recursiva final de un procedimiento, sino que en vez de ello bastaba con actualizar ciertos valores, mantener la dirección de retorno original y saltar al comienzo del código del procedimiento. Ahora tenemos que asegurar lo que debe ocurrir con estos componentes del registro de activación, y con los que no se tuvieron en cuenta (enlace de acceso o *display* guardado, enlace de control, apuntadores de pila y datos locales principalmente).

Empecemos con los valores de *BasePila* y *TopePila*. Evidentemente, *BasePila* no debe cambiar durante la secuencia de llamada, puesto que el registro de activación que queda en la pila tiene exactamente la misma estructura que el que "desaparece"; además, es lógico pensar que ese apuntador se usará para actualizar los elementos necesarios del mismo modo que se utiliza, por ejemplo, para el acceso a los datos dentro del registro de activación. Por tanto, tenemos la primera condición que ha de cumplir la secuencia de llamada optimada.

Condición 3.2.2. El valor de *BasePila* de la primera activación de la cadena de activaciones finales recursivas ha de mantenerse a lo largo de toda la cadena hasta la activación terminal.

Para que el valor de *BasePila* no cambie, según precisa la anterior condición, la operación L5 de la secuencia de llamada estándar (Algoritmo 2.5.1) no ha de realizarse, y las demás operaciones tampoco han de modificar dicho valor.

En cuanto a *TopePila*, la cosa es un poco más compleja. En principio, por las mismas razones que en el caso anterior, el valor de *TopePila* no ha de cambiar, lo que parece favorecer la optimación, por lo que formulamos la siguiente condición:

Condición 3.2.3. El valor de *TopePila* de la primera activación de la cadena de activaciones finales recursivas ha de mantenerse a lo largo de toda la cadena hasta la activación terminal.

Paradójicamente, la primera instrucción que ejecutará el código del procedimiento receptor es la actualización de ese apuntador, que es el paso L7 del Algoritmo 2.5.1. La Condición 3.2.1 obliga a que ese código no cambie, así que se ejecuta una instrucción para modificar supuestamente el valor de *TopePila* que tiene el efecto de no modificarlo realmente. Obviamente, es una lástima que esto sea así, cuando parece posible evitar una operación, así que estudiaremos una posible solución.

Para evitar el paso L7, habría que modificar el paso L6: en lugar de ejecutar un salto al punto de entrada del procedimiento, se trataría de saltar a la instrucción del paso L8 directamente. A primera vista, esto parece bastante arriesgado, pero puede que no sea tan difícil. En efecto, es lógico pensar que el compilador se diseña con una estrategia determinada para la generación del código correspondiente a procedimientos y funciones, y que las instrucciones

que corresponden al paso L7 se situarán en un punto determinado de ese código (de hecho, normalmente estarán en la parte inicial, pero puede que no sea así, lo cual no impedirá, sin embargo, nuestro argumento). Parece por tanto posible que el compilador sea capaz de conocer la posición de la instrucción siguiente a la del paso L7 del Algoritmo 2.5.1.

Por ejemplo, la mayoría de los compiladores generan código en ensamblador o lenguaje similar, en que existen etiquetas abstractas para las instrucciones; simplemente adoptando la estrategia de etiquetar el paso L8 de la secuencia de llamada correspondiente al procedimiento, y haciendo esa etiqueta "visible" como punto de entrada alternativo al procedimiento, se conseguiría el fin propuesto. Si el compilador genera código con direcciones absolutas, también puede hacerse algo parecido, pues al fin y al cabo es lógico suponer que las instrucciones del paso L7 tendrán un tamaño fijo, con lo que simplemente se trataría de saltar a la dirección de entrada del procedimiento más ese tamaño. En definitiva, parece posible evitar el paso L7 de la secuencia de llamada estándar si el compilador es capaz de conocer para todo procedimiento la situación del paso L8 en el código del mismo. Recogemos esta discusión en la condición siguiente:

Condición 3.2.4. Si el compilador es capaz de situar las instrucciones del paso L8 de la secuencia de llamada de procedimientos, entonces el salto por el que se activa el procedimiento recursivo final (paso L6) ha de efectuarse a dicho punto; en caso contrario, el salto ha de efectuarse al punto de entrada del procedimiento.

En efecto, si el compilador no puede, por las razones que sean, situar dicho paso L8, el único problema es que se va a realizar una operación cuyo efecto neto es nulo, por lo que la optimación resultará de peor calidad temporal, pero igualmente válida desde el punto de vista espacial.

Tomemos ahora la dirección de retorno. Ya hemos visto en la introducción que no debe modificarse este valor a lo largo de la cadena de activaciones finales recursivas; de ese modo, se mantiene en la pila de control la dirección de retorno de la primera activación del procedimiento recursivo, con lo que se evita ejecutar toda la cadena de secuencias de retorno sucesivas:

Condición 3.2.5. La dirección de retorno de la primera activación de la cadena de activaciones finales recursivas ha de mantenerse a lo largo de toda la cadena hasta la activación terminal.

Esto se puede conseguir simplemente eliminando el paso L4 de la secuencia de llamada estándar (Algoritmo 2.5.1), siempre que se asegure que la dirección de retorno no se pierde durante dicho proceso.

Examinemos ahora el enlace de control. Este elemento proporciona, en general, la posición del registro de activación del subprograma llamador; concretamente es el valor que tenía *BasePila* para el registro de activación de este subprograma llamador, y se utiliza para restablecer *BasePila* con ese valor cuando el subprograma receptor finalice su activación. En el caso de una cadena de activaciones finales recursivas, interesa el valor a restablecer para el subprograma que activó la cadena, de una forma similar a lo que ocurre con la dirección de retorno; por tanto:

Condición 3.2.6. El enlace de control de la primera activación de la cadena de activaciones finales recursivas ha de mantenerse a lo largo de toda la cadena hasta la activación terminal.

De nuevo, esto se consigue sin más trámite que eliminando el paso L2 de la secuencia de llamada estándar (Algoritmo 2.5.1), siempre que se asegure también que dicho valor no se modifica en la secuencia de llamada optimada.

En cuanto al enlace de acceso, también se puede obtener un resultado similar, aunque por razones un poco distintas. El enlace de acceso sirve, como se sabe, para encontrar la situación de los datos no locales: el enlace de acceso tiene el valor de *BasePila* del registro de activación más reciente del subprograma que contiene léxicamente al subprograma actual, y el seguimiento de la denominada *cadena estática* permite el acceso a los datos no locales de cualquier nivel de anidamiento. En tal caso, en una cadena de activaciones recursivas (incluso sin ser finales) el enlace de acceso tiene el mismo valor para todos los registros de activación de la cadena, apuntando a un cierto registro de activación que está por debajo de todos los de la cadena recursiva, como se aprecia en la Figura 3.2.1.

Como la optimación que proponemos consiste en mantener únicamente el último registro de activación de la cadena, el valor del enlace de acceso ya está calculado desde la activación inicial, y no es necesario ni recalcularlo ni

modificarlo (realmente, como vemos, esta optimación podría hacerse para llamadas recursivas aún si se acumula un nuevo registro de activación en la pila de control: basta con hacer que el enlace de acceso del registro de activación de la nueva llamada tenga el mismo valor que el enlace de acceso de la llamada previa). Tenemos por tanto:

Condición 3.2.7. El enlace de acceso de la primera activación de la cadena de activaciones finales recursivas ha de mantenerse a lo largo de toda la cadena hasta la activación terminal.

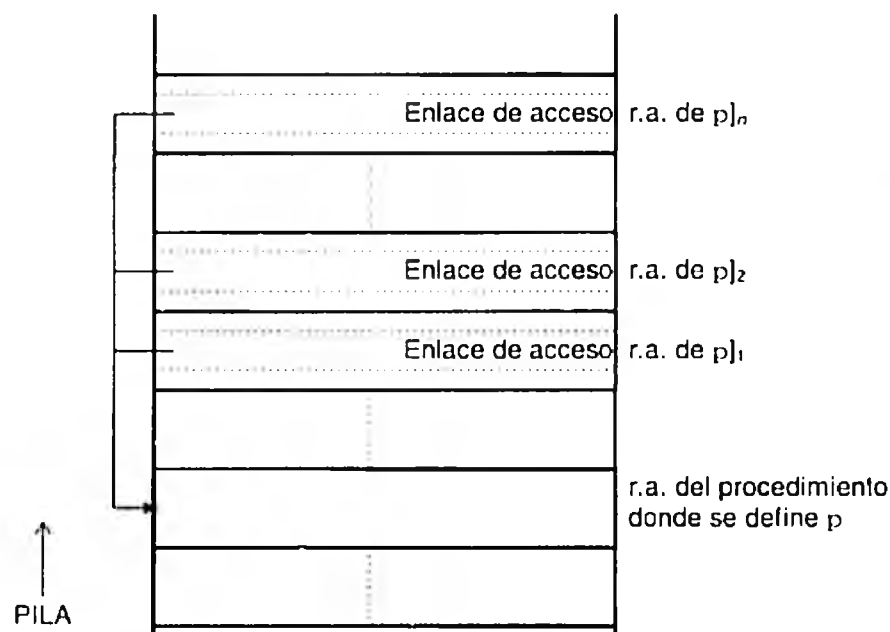


Figura 3.2.1 Enlaces de acceso en una cadena de activaciones recursivas

De igual modo que en los casos anteriores, eliminando el paso L3 del Algoritmo 2.5.1 de llamada estándar se consigue el efecto señalado, si se asegura que en el nuevo registro de activación el valor del enlace de acceso no cambia respecto al que había previamente.

Según todo ello, la versión completa de una optimación de la secuencia de llamada estándar para el lenguaje PASCAL, que llamaremos *secuencia de llamada optimada para activaciones finales recursivas de procedimiento*, es la que se recoge en el Algoritmo 3.2.1 (cuando se utiliza la estrategia del enlace de acceso para obtener los datos no locales).

- LO1: el llamador evalúa los parámetros actuales, y coloca los valores obtenidos en el lugar correspondiente a sus propios parámetros formales;
- LO2: el llamador salta (jump) a la instrucción del código del procedimiento que sigue a la actualización del valor de *TopePila*;
- LO3: el receptor (realmente, el mismo procedimiento) asigna valores iniciales a sus datos locales.

Algoritmo 3.2.1 Secuencia de llamada optimada para una activación final recursiva de procedimiento

Por supuesto, nos mantenemos a la espera de estudiar en profundidad el proceso de evaluación de los parámetros actuales, y su asignación a las localidades de los formales (paso LO1), cosa que abordaremos en los apartados a continuación.

Por lo demás, como vemos, la operación LO3 es la misma que la operación L8 de la secuencia de llamada estándar (Algoritmo 2.5.1), pues corresponde a la parte ejecutada por el receptor, y que no puede cambiar, según la Condición 3.2.1. Suponemos también, siguiendo la discusión de la Condición 3.2.4, que el compilador es capaz de conocer la dirección de la instrucción que sigue al paso L7 de la secuencia estándar. Implícitamente, como hemos concluido en las diversas condiciones previas, ha de asegurarse además que las operaciones de esta secuencia de llamada optimada mantienen los valores (y las posiciones, lógicamente) de la dirección de retorno, del enlace de control, del enlace de acceso, de *BasePila* y de *TopePila* que existían para la activación previa o llamadora; concretamente, ha de matizarse la manera en que se realizará el paso LO1 del Algoritmo 3.2.1.

La utilización de la estrategia del *display* para el acceso a los datos no locales modifica poco los resultados conseguidos, pero se necesita un razonamiento algo más complejo. Recordemos que el registro de *display* $d[n]$ proporciona el valor de *BasePila* para el registro de activación más reciente correspondiente a un subprograma con nivel de anidamiento n , y que en el registro de activación de un subprograma con nivel de anidamiento n se almacena el valor de $d[n]$ previo a la llamada, el denominado "*display guardado*" en la Figura 2.5.1.

Al proceder a la optimación, debemos mantener el hecho de que $d[n]$ proporcione el registro de activación más reciente de la cadena de activaciones recursivas, y que el "display guardado" sea el valor que tenía tal registro de *display* previamente a la primera de las activaciones de la cadena. De nuevo, la solución más interesante de cara a la optimación, como es eliminar sin más la actualización del *display*, es la que funciona correctamente. En efecto, si se asegura que:

Condición 3.2.8. El registro de *display* actual y el registro de *display* guardado de la primera activación de la cadena de activaciones finales recursivas han de mantenerse a lo largo de toda la cadena hasta la activación terminal.

por un lado, se accede correctamente a los datos locales (pues la posición relativa de los elementos del registro de activación no cambia), y por otro, se restablecerá correctamente la situación del *display* previa a la cadena de llamadas recursivas (ya que se tomará el registro de *display* guardado que existe desde la primera activación). Por lo tanto, la secuencia de llamada optimada obtenida hace un momento es válida también si se utiliza la técnica del *display*.

Claramente, este nuevo modelo dinámico optimado (ya sea usando la cadena estática o el *display*) supone una mejora considerable de la ejecución, tanto espacialmente (lo que era *a priori* el interés principal del trabajo) como temporalmente (lo cual resulta ser un efecto en cierto modo sorprendente), además de agilizar, por supuesto, la generación de código, aunque este aspecto, que estudiaremos en el capítulo 6, quedará lógicamente rebasado con creces por la necesidad de un análisis más fuerte que el habitual del programa fuente. Compararemos también allí los comportamientos dinámicos estándar y optimado desde un punto de vista más riguroso.

Ahora, una vez definido ya el modelo dinámico optimado, por lo menos en lo concerniente a la secuencia de llamada, hemos de estudiar cómo las demás características del lenguaje de programación PASCAL influyen en el mismo. Este es el objetivo de los apartados que vienen a continuación.

3.2.3 Datos locales

Consideremos un primer ejemplo sencillo de un procedimiento recursivo con datos locales:

```

program Prueba1 ...
...
  procedure p (x: integer);
  var i: integer;
  begin
    if x <= 0 then
      begin
        i := 7;
        p (x-1)      { activación recursiva final }
      end
    else writeln (i)
    end;
  begin
    p (5)
  end.

```

Programa 3.2.1 Un procedimiento recursivo final con datos locales en PASCAL.

Observemos que el razonamiento que permitió ver (apartado 3.1.1) que podían eliminarse los sucesivos registros de activación de la cadena de llamadas recursivas sigue siendo válido ahora. En efecto, al estar los datos locales de cada llamada en el registro de activación correspondiente, no es posible acceder en la activación *n*-ésima de *p* a ninguno de los datos de las activaciones previas, ya sean datos locales o parámetros formales. Como dichos datos tampoco se van a utilizar en los procesos de retorno, pues las activaciones son finales, realmente pueden eliminarse de la pila de control, y solo se precisa mantener el registro de activación actual, exactamente de la manera que habíamos concluido.

Según ello, puesto que en el modelo dinámico optimado cada registro de activación receptor (en la cadena de llamadas recursivas finales) reemplaza al llamador, los datos locales simplemente se encuentran en el mismo lugar en que se situaban en la llamada previa. Por tanto, la Condición 3.2.2 permite, sin ningún requerimiento adicional sobre el código, que el acceso a los datos locales se realice según la estrategia habitual.

A pesar de todo, resulta necesario profundizar en un cierto aspecto, que está relacionado con los valores iniciales que toman los datos locales cuando se activa un procedimiento. Obsérvese que, según la definición estándar del lenguaje PASCAL:

«When the algorithm commences, the [local] variable is totally undefined» [Jensen y Wirth 1991: 179-180]. «A variable is *undefined* if it does not have a value of its type. A variable is *totally undefined* if it is undefined and further if every component of the (structured) variable is totally undefined» [ibíd.: 161; cursivas en el original]

por lo que debe asegurarse que el código del procedimiento realiza la oportuna inicialización de los datos locales antes de cualquier otra operación con los mismos. Realmente, este es el cometido de la operación L8 de la secuencia de llamada estándar (Algoritmo 2.5.1). Por tanto, los datos locales se colocarán en dicha situación de total indefinición sea cual sea el modelo dinámico que se utilice, o dicho de otro modo, sea cual sea su posición en la pila de control. Y del mismo modo, el código del procedimiento ha de contener la operación L8 antes de ninguna instrucción referente a los datos locales, independientemente del modelo dinámico elegido.

Entonces, démonos cuenta que si se realiza una transformación del Programa 3.2.1 similar a la realizada entre el Programa 3.1.1 y el Programa 3.1.2, se obtiene un resultado que no es equivalente al inicial:

```

program PruebaNoRecursivo ...
  procedure p (x: integer);
    label 1;
    var i: integer;
    begin
      1: if x <> 0 then
          begin
            i := 7;
            x := x-1;      { actualización parámetro }
            goto 1         { salto al comienzo }
          end
        else writeln (i)
        end;
    begin
      p (5)
    end.

```

Programa 3.2.2 Procedimiento no recursivo (¡no equivalente!) correspondiente al del Programa 3.2.1

pues desde la primera activación recursiva el dato local no adquiere la característica de indefinido, con lo que la ejecución del caso terminal no supone ningún problema, mientras que en el programa original se ha de dar un error de ejecución en tal momento, de acuerdo con la norma semántica señalada del lenguaje PASCAL.

Esto tiene importancia por dos razones. En primer lugar, obliga a abandonar la filosofía transformacional (tal como aparece descrita en el capítulo 2 y tal como se ha usado en el apartado 3.1.1 para convertir el Programa 3.1.1 en el Programa 3.1.2) para la consecución del objetivo planteado. Ya dijimos en la introducción que una posible manera de abordar el problema era usando técnicas de transformación de programas, de modo que una especie de preprocesador avanzado preparase el programa, sustituyendo procedimientos recursivos finales por procedimientos iterativos. Esta idea, además de ser bastante costosa, se muestra inaplicable si un procedimiento tiene datos locales, lo cual es una restricción demasiado fuerte para que la optimización sea interesante.

Por otra parte, esta inicialización es una operación que puede llegar a ser costosa (dependiendo del tamaño de los datos locales). Debido a ello, algunos compiladores de PASCAL (por ejemplo, TURBO PASCAL [Borland 1992a], VAX PASCAL [Digital Equipment Corporation 1987]) evitan tal tarea, y por supuesto, todas las normas semánticas dinámicas acerca de la posible indefinición de los datos. Incluso en este caso el modelo dinámico optimado es equivalente al estándar, aunque ciertamente se pueden obtener comportamientos diferentes para un mismo programa (si es un programa que lleva a un error de ejecución de ser compilado siguiendo la definición estándar del lenguaje).

Tomemos el Programa 3.2.1; al ser procesado con un compilador no estándar sin optimación, el efecto de la sentencia `writeln(i)` sería la impresión de un valor numérico no significativo: al activarse por última vez el procedimiento recursivo, se crea un registro de activación para él, y aparece la memoria para el dato `i`, que, al no serle dado un valor inicial, adquiere como valor lo que hubiera en las posiciones de memoria correspondientes hasta ese momento. Si el mismo programa se compila con dicha característica no estándar, pero según el modelo dinámico optimado, el valor del dato local `i` en cada activación es también el valor que tenga la memoria correspondiente, solo que ahora es exactamente la del dato local `i` de la activación previa; como cada

activación recursiva da a i el valor 7, este es el valor inicial que tendrá i , y el que se imprimirá en la llamada terminal. Obsérvese que, en este caso, ese es justamente el comportamiento del Programa 3.2.2, por lo que se podría afirmar que los dos programas (y en general, los dos modelos dinámicos) son equivalentes, a pesar de comportarse de diferente modo, pero evidentemente debido a que no se observa una norma del lenguaje que provoca la admisión de programas con instrucciones "no significativas".

En definitiva, desde el momento en que no se sigue la norma de la definición estándar de PASCAL sobre la indefinición de los datos locales, las posibles variaciones del comportamiento de programas que serían incorrectos aplicando dicha norma no invalidan la equivalencia entre los modelos dinámicos estándar y optimado.

De todos modos, esta discusión es útil porque lleva inmediatamente a otra, como es la aplicación de la optimación a otros lenguajes imperativos de la familia de PASCAL. En el lenguaje C, por ejemplo, no existe inicializaciones de variables locales, ni se produce un error dinámico al acceder a una variable a la que lógicamente aún no se le ha asignado un valor. En otras palabras, no existe como tal un paso L8 en la secuencia de llamada. Incluso, podríamos considerar un lenguaje en que existiera inicialización de variables locales, pero que no produjera error si estas se encuentran "indefinidas".

El asunto es que para lenguajes con estas características sí parecería probable, al menos en lo que respecta a las activaciones finales recursivas, utilizar una filosofía transformacional para resolver la optimación. En efecto, las inicializaciones siempre se pueden sustituir con instrucciones propias del código del subprograma, es decir, no tienen necesariamente que incluirse como paso L8 de la secuencia de llamada, así que podría realizarse la optimación mediante una transformación del programa fuente similar a la realizada en ejemplos anteriores (Programa 3.1.2). Sin embargo, veremos que otros aspectos de la optimación de las activaciones finales recursivas y no recursivas impiden tal filosofía en cualquier lenguaje (debido principalmente a la manera en que deben actualizarse los parámetros y también a la dificultad de expresar un salto de un procedimiento a otro: C, por ejemplo, no admite saltos no locales, y de hecho, no mostramos un posible programa PASCAL equivalente al Programa 3.1.3).

3.2.4 Parámetros formales por valor

En este apartado y los tres que siguen se estudian los problemas relacionados con los diferentes modos de paso de parámetros en el lenguaje PASCAL, empezando con el paso por valor (o parámetro de lectura). En cualquier caso, trataremos de explicitar exactamente el significado del paso L1 del Algoritmo 2.5.1 de llamada estándar, y su posible modificación bajo el modelo dinámico optimado (paso LO1 del Algoritmo 3.2.1).

El caso general de un procedimiento recursivo final con más de un parámetro formal por valor no va a suponer ningún problema respecto al caso más particular considerado hasta ahora (en que existía un solo parámetro por valor), pero precisa ciertas matizaciones. En efecto, en el caso del Programa 3.1.1 simplemente había que colocar el valor del parámetro actual en la posición correspondiente del parámetro formal, dentro del registro de activación único para todas las activaciones finales recursivas del procedimiento, lo cual era equivalente a sustituir la llamada recursiva final por:

```
x := x-1;      ( actualización del parámetro )  
goto 1        ( salto al comienzo )
```

tal como se vio en el Programa 3.1.2.

Sin embargo, los casos en que hay, en general, cualquier número de parámetros formales superior a la unidad, han de manejarse con cuidado. Si tenemos el siguiente programa, donde existen dos parámetros formales:

```
program Prueba2 ...  
  procedure p (x, y: integer);  
  begin  
    p (x+y, x-y)    ( activación recursiva final )  
  end;  
end.
```

Programa 3.2.3 Un procedimiento recursivo final con dos parámetros formales por valor en PASCAL

y pretendemos que sea equivalente al que aparece a continuación (Programa 3.2.4), estaríamos obviamente actuando de un modo incorrecto.

```

program Prueba2NoRecursivo ...
  procedure p (x, y: integer);
  label 1;
  begin
  1: ...
    x := x+y;      ( actualización de los parámetros )
    y := x-y;
    goto 1         ( salto al comienzo )
  end;
end.

```

Programa 3.2.4 Procedimiento no recursivo (¡no equivalente!) correspondiente al del Programa 3.2.3

Para expresar con una construcción de alto nivel la forma correcta de evaluar los parámetros actuales y actualizar los parámetros formales, usaremos la *sentencia de asignación múltiple*, de la manera en que es utilizada por Balcázar [1993: 25-26]:

```

program Prueba2NoRecursivo ...
  procedure p (x, y: integer);
  label 1;
  begin
  1: ...
    (x, y) := (x+y, x-y); ( actualización de parámetros )
    goto 1               ( salto al comienzo )
  end;
end.

```

Programa 3.2.5 Procedimiento no recursivo equivalente al del Programa 3.2.3

La sentencia de asignación múltiple supone que primero se evalúan *todas* las expresiones de la parte derecha, y solo acabado esto se asignan los valores a las variables correspondientes de la parte izquierda. Es, por tanto, la manera correcta de expresar la actualización de los parámetros formales por valor con los parámetros actuales correspondientes. Evidentemente, si se quiere expresar el proceso en términos del propio lenguaje PASCAL, es preciso utilizar variables temporales, en general una por cada parámetro, tal como aparece en el Programa 3.2.6. Incidentalmente, vemos que esta transformación del programa fuente puede ser relativamente fácil de realizar, por lo que en principio podría seguirse tal estrategia para resolver la optimación, en lo que concierne a la actualización de los parámetros por valor.

```

program Prueba2NoRecursoivo2 _
--
  procedure p (x, y: integer);
  label 1;
  var temp1, temp2: integer;
  begin
    1: --
      temp1 := x+y;   ( actualización de parámetros )
      temp2 := x-y;
      x := temp1;
      y := temp2;
      goto 1          ( salto al comienzo )
    --
  end;
--
end.

```

Programa 3.2.6 Procedimiento no recursivo equivalente al del Programa 3.2.3 en PASCAL estricto

El paso LO1 de la secuencia de llamada optimada (Algoritmo 3.2.1), en lo relativo a los parámetros por valor, debe realizarse entonces de acuerdo con la condición expuesta, que formalizamos de la siguiente manera:

Condición 3.2.9 (primera formulación). La evaluación de los parámetros actuales por valor y su asignación a los parámetros formales debe ser de tal modo que no se produzca la lectura de un parámetro formal durante algún proceso de evaluación de un parámetro actual después de haber actualizado dicho parámetro formal.

La razón de esta formulación tan general es permitir diversas implementaciones del paso de parámetros, y a la vez, obviamente, rechazar otras que pueden producir problemas con el modelo optimado. En general, el proceso correcto será aquel que tenga una filosofía similar a lo expresado en el Programa 3.2.6. Por ejemplo, la primera idea puede ser utilizar registros para realizar la evaluación, y después llevar los valores conseguidos a las posiciones de los parámetros formales en el nuevo registro de activación; esta estrategia no ofrece absolutamente ninguna dificultad en el modelo dinámico optimado, pues simplemente las direcciones de los parámetros formales son las del propio registro de activación.

Sin embargo, otra estrategia posible podría ser la consistente en evaluar los parámetros actuales usando como posiciones "de trabajo" las direcciones definitivas de los parámetros formales en el nuevo registro de activación, por encima del registro de activación del procedimiento que hace la llamada; esta idea precisa que la máquina admita instrucciones sobre memoria del mismo

modo que sobre registros (lo cual no siempre es cierto), pero si es así puede conducir a una ejecución más rápida, al evitar la operación de movimiento del parámetro actual evaluado de registro a memoria (si bien es probable que las operaciones con memoria sean más lentas que con registros, con lo que la mejora temporal puede no ser tan clara). El problema es que esta estrategia no sigue la Condición 3.2.9, pues ocurre que la posición del parámetro formal y algún componente de un parámetro actual puede ser la misma, con lo que se precisaría un cambio de filosofía. En cualquier caso, dejaremos esta discusión para más adelante, cuando estudiemos más detenidamente la influencia del modelo dinámico optimado en las estrategias habituales de compilación y organización del programa objeto.

Aparte de ello, obsérvese que no existe ninguna otra restricción sobre la optimación. Concretamente, los parámetros actuales pueden ser expresiones cualesquiera, que incluyan objetos de cualquier tipo: constantes, variables locales, variables no locales, parámetros formales por valor y parámetros formales por referencia. En efecto, salvo las constantes obviamente, cualquier otro objeto de los enumerados se encuentra (según el modelo dinámico estándar) en el registro de activación actual o en algún registro de activación por debajo de este en la pila de control; en cuanto a los objetos que se encuentren en registros de activación por debajo del actual no hay ningún problema, y en cuanto a los objetos del propio registro de activación (que en el modelo optimado va a ser sustituido por el entrante), la Condición 3.2.9 asegura que se van a tomar los valores y evaluar los parámetros actuales completamente antes de actualizar esos valores. Por tanto, la optimación puede hacerse en todos los casos.

Todavía existe, a pesar de todo, un aspecto que debe observarse. Consideremos el programa siguiente:

```
program Prueba3 ...  
  procedure p (x, y: integer);  
  begin  
    ...  
    p (x, y)      ( activación recursiva final )  
    ...  
  end;  
  ...  
end.
```

Programa 3.2.7 Otro procedimiento recursivo final con dos parámetros formales por valor en PASCAL

Si se aplica una transformación similar a la que produjo el Programa 3.2.6 a partir del Programa 3.2.3, se obtendría:

```

program Prueba3NoRecursoivo ...
...
  procedure p (x, y: integer);
  label 1;
  var temp1, temp2: integer;
  begin
  1: ...
    temp1 := x;      { actualización de parámetros }
    temp2 := y;
    x := temp1;
    y := temp2;
    goto 1           { salto al comienzo }
  ...
  end;
...
end.

```

Programa 3.2.8 Procedimiento no recursivo equivalente al del Programa 3.2.7

Claramente, la actualización de los parámetros es redundante. Esto ocurrirá en general cuando el parámetro actual sea una expresión formada exclusivamente por el propio parámetro formal situado en la posición correspondiente del procedimiento recursivo. Reflejamos este aspecto en una nueva condición:

Condición 3.2.10. Si el n -ésimo parámetro actual por valor del procedimiento recursivo final es una expresión formada exclusivamente por el n -ésimo parámetro formal del procedimiento, entonces no es preciso actualizar dicho parámetro.

Con ello, se incorpora un aspecto más de mejora en la optimación planteada, a añadir a los ya conseguidos hasta el momento.

En realidad, lo anterior es un caso particular de una optimación más amplia, que se obtiene considerando la implementación de la asignación múltiple. Hemos visto que, en general, transformar una sentencia que expresa el tipo de procesamiento que debe hacerse en el paso LO1 de actualización de los parámetros (Algoritmo 3.2.1), tal como:

$$\langle x, y \rangle := \langle x+y, x-y \rangle;$$

requiere la utilización de tantas variables temporales como variables intervienen en la asignación múltiple:


```
temp1 := x+y;  
temp2 := x-y;  
x := temp1;  
y := temp2;
```

Sin embargo, consideremos:

$$\langle x, y \rangle := \langle x+1, y-1 \rangle;$$

Claramente, no es preciso el uso de ninguna variable temporal:

```
x := x+1;  
y := y-1;
```

Y si tenemos en cuenta este otro caso:

$$\langle x, y \rangle := \langle x+1, x+y-1 \rangle;$$

tampoco es necesario el uso de temporales, pero importa el orden en que establecemos las asignaciones simples:

```
y := x+y-1;  
x := x+1;
```

Puede llegarse a una generalización de estas situaciones, y realmente contamos con un algoritmo sencillo que detecta si es preciso el uso de alguna o algunas variables temporales, y determina el orden correcto de asignaciones simples que evita su uso. El algoritmo es sencillo cuando se considera el problema teóricamente, es decir, cuando se supone que únicamente aparecen nombres de variables completas en las partes derechas de las asignaciones, que son los que se examinan para establecer las dependencias entre las expresiones. La incorporación de elementos de datos de tipos estructurados (*arrays*, *records*, etc.) y de activaciones de funciones complica sobremanera la aplicación de este algoritmo, y obviamente la complejidad de análisis necesaria en el compilador no justifica su inclusión. Por contra, el caso particular en que una expresión de la parte derecha de la asignación consta únicamente de la variable de la posición correspondiente de la parte izquierda es de muy fácil detección, y por ello será el que incorporemos al optimador, tal como establece la Condición 3.2.10 y condiciones similares que encontraremos en los próximos apartados.

3.2.5 Parámetros por referencia

El otro modo de paso de parámetros en el lenguaje PASCAL es el *paso por referencia*, que consiste, como se sabe, en asociar al parámetro formal la *dirección* del parámetro actual, de manera que un efecto lateral sobre el parámetro formal es realmente un efecto lateral sobre el parámetro actual. Más concretamente, el modelo dinámico estándar del lenguaje PASCAL establece una diferencia en cuanto a la forma de acceso entre nombres de variables asociadas "directamente" a memoria (datos locales de cualquier subprograma, incluido el principal, y parámetros formales por valor) y nombres de variables asociadas "indirectamente" a memoria (que son los parámetros formales por referencia):

- Una referencia de lado derecho (en una expresión, básicamente) a una variable "directa" implica un acceso al contenido de la memoria asociada a dicha variable; esto es así incluso cuando la variable participa en una expresión que sea parámetro actual por valor de algún subprograma. Sin embargo, una referencia del mismo tipo a un parámetro formal por referencia significa acceder al contenido de la memoria cuya dirección es el contenido de dicho parámetro.
- Una referencia de lado izquierdo (en una asignación o como parámetro actual por referencia) a una variable "directa" implica calcular la dirección de la memoria asociada a dicha variable. Por contra, una referencia de lado izquierdo a un parámetro formal por referencia se resuelve tomando exactamente la dirección que contiene como valor dicho parámetro formal.

Usaremos la expresión $\langle dir\ x \rangle$, si x es un parámetro formal por referencia, para referirnos a la dirección a la que señala x , e $\langle indir\ x \rangle$ para denotar el contenido de esa dirección. Así, el hecho de asociar el parámetro formal x con un parámetro actual que sea una variable "directa" n podría quedar reflejado por:

$$\langle dir\ x \rangle := \text{dirección de } n;$$

mientras que si el parámetro formal x se asocia con un parámetro actual que sea otro parámetro formal por referencia y hay que ejecutar:

$$\langle dir\ x \rangle := \langle dir\ y \rangle;$$

finalmente, asignar un valor a la variable "directa" que señala x puede escribirse como:

$x := 27;$ o también $\langle \text{indir } x \rangle := 27;$

Esta notación será conveniente en un momento.

El modelo dinámico optimado ha de mantener las características comentadas, pero veremos cómo se modifica lo relativo a la actualización de los parámetros formales por referencia en las activaciones finales recursivas. Examinemos un primer programa con un procedimiento recursivo final que tiene un parámetro por referencia.

```

program Prueba4
var n: integer;

procedure p (var x: integer);
begin
  if x <> 0 then
  begin
    x := x-1;
    p (x)          ( activación recursiva final )
  end
  else writeln (x)
  end;
begin
  p (n)
end.

```

Programa 3.2.9 Procedimiento recursivo final con un parámetro por referencia en PASCAL

En el modelo dinámico estándar, la primera activación de p hace que se asocie al parámetro formal x la dirección de la variable n ; como el parámetro formal en la llamada recursiva es siempre x , este parámetro siempre va a tener la dirección de n , como se observa en la Figura 3.2.2.

En este caso, vemos que la optimación puede llevarse a cabo, pues la dirección del dato al que "apunta" el parámetro formal se mantiene *por debajo* de todos los registros de activación de la cadena de llamadas recursivas finales; por lo tanto, aglutinar estos registros de activación en uno solo no implica ninguna pérdida de información durante la cadena de llamadas, y el resto del argumento es similar al de los casos estudiados previamente. Por tanto, es posible mantener un único registro de activación en la pila de control, convenientemente actualizado durante cada llamada, en el modo en que aparece reflejado en la Figura 3.2.3.

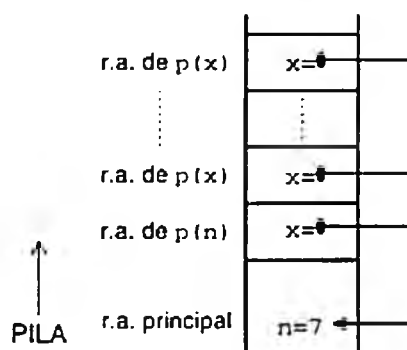


Figura 3.2.2 Instantánea de la pila de control durante una cadena de activaciones finales recursivas del procedimiento del Programa 3.2.9 (según el modelo dinámico estándar)

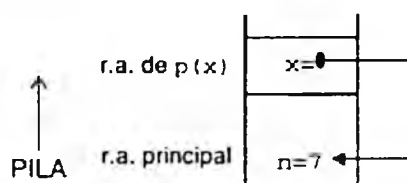


Figura 3.2.3 Instantánea de la pila de control durante una cadena de activaciones finales recursivas del procedimiento del Programa 3.2.9 (según el modelo dinámico optimado)

Para ello, al encontrar una llamada recursiva final $p(n)$, donde n es el parámetro actual por referencia, basta con generar código para obtener la dirección que n representa (si es un dato "directo", su posición, y si es otro parámetro formal por referencia, su propio valor, como hemos dicho) y colocar esta dirección en el lugar correspondiente a x en el registro de activación actual.

En el Programa 3.2.9 concretamente, hemos notado que el valor del parámetro formal x siempre es el mismo, pues es el parámetro actual de la llamada recursiva. Siguiendo el algoritmo de actualización anterior, esta característica sigue siendo cierta, pues se calcula como parámetro actual el valor de x , es decir, la dirección de n , tal como se expresa en el Programa 3.2.10. Con la notación definida al comienzo del apartado, expresamos con precisión el proceso de actualización. Y ciertamente, no hay ninguna dificultad en generar código que realice estas operaciones, pero el hecho es que es imposible expresar las mismas en el propio lenguaje PASCAL. Por lo tanto, es en estos momentos cuando tenemos que rechazar definitivamente la transformación de programas

como filosofía alternativa de solución de la optimación de las activaciones finales recursivas de procedimientos, pues impediría la aplicación a procedimientos que tuvieran parámetros por referencia.

```

program Prueba4NoRecursivo ...
var  n: integer;
...
  procedure p (var x: integer);
  label 1;
  begin
  1:  if x <> 0 then
      begin
        x := x-1;
        <dir x> := <dir x>;      { actualización }
        goto 1                  { salto al comienzo }
      end
    else writeln (x)
  end;
...
begin
...
  p (n)
...
end.

```

Programa 3.2.10 Procedimiento no recursivo equivalente al del Programa 3.2.9

Por supuesto, la actualización del parámetro formal reflejada en este programa es claramente redundante. Esto es similar al resultado encontrado con los parámetros por valor, y que permitió la optimación recogida en la Condición 3.2.10. Ahora podemos enunciar un resultado parecido:

Condición 3.2.11. Si el enésimo parámetro actual por referencia del procedimiento recursivo final es exactamente el enésimo parámetro formal del procedimiento, entonces no es preciso actualizar dicho parámetro.

lo cual también redunda en un interés añadido de la optimación.

Antes de dar este estudio como general, examinemos otro caso, en el que existen dos parámetros por referencia (Programa 3.2.11). Si la activación final recursiva fuera $p(x, z)$ entonces no habría ningún problema en aplicar la optimación planteada para el caso previo: como los parámetros actuales son los parámetros formales (en el mismo orden), no es preciso actualizar sus valores, pues contienen direcciones que no varían entre una llamada y otra.

```

program Prueba5 ...
var n, m: integer;
...
procedure p (var x, z: integer);
begin
    if x <> 0 then
        begin
            x := x-1;
            z := z+1;
            p (z, x)      ( activación recursiva final )
        end
    else writeln (x, z)
    end;
...
begin
    ...
    p (n, m)
    ...
end.

```

Programa 3.2.11 Procedimiento recursivo final con dos parámetros por referencia en PASCAL

Sin embargo, al tener la llamada $p(z, x)$, la cosa cambia. Ahora, en la nueva activación debe asociarse a x la dirección a la que apuntaba z , y a z la dirección a la que apuntaba x . Como en el caso de varios parámetros por valor, esto precisa que la evaluación de los parámetros actuales se haga completamente antes de colocar los nuevos valores en las posiciones correspondientes de los parámetros formales, lo cual se puede representar en el siguiente programa equivalente.

```

program Prueba5NoRecurativo ...
var n, m: integer;
...
procedure p (var x, z: integer);
label l;
begin
    l: if x <> 0 then
        begin
            x := x-1;
            z := z+1;
            (dir x, dir z) ← (dir z, dir x);  ( actualización )
            goto l                          ( salto al comienzo )
        end
    else writeln (x, z)
    end;
...
begin
    ...
    p (n, m)
    ...
end.

```

Programa 3.2.12 Procedimiento no recursivo equivalente al del Programa 3.2.11

Entonces, podemos reformular la Condición 3.2.9 de la siguiente manera:

Condición 3.2.9 (segunda formulación). La evaluación de los parámetros actuales por valor y por referencia, y su asignación a los parámetros formales debe ser de tal modo que no se produzca la lectura de un parámetro formal durante algún proceso de evaluación de un parámetro actual después de haber actualizado dicho parámetro formal.

Por lo tanto, en el modelo optimado se ha de seguir la misma estrategia para actualizar los parámetros formales, sin distinguir entre modo por valor y modo por referencia (salvo, lógicamente, en lo relativo al cálculo del valor del parámetro actual y su vinculación al parámetro formal según la definición del lenguaje).

Hemos de estudiar ahora la condición fundamental para poder llevar a cabo la optimación de un procedimiento recursivo final con parámetros formales por referencia: como ya hemos dicho al principio, la zona de memoria a la que apunte el parámetro formal por referencia que se va a calcular en una nueva activación debe mantenerse *por debajo* del registro de activación actual, o en el modelo dinámico estándar, por debajo de todos los registros de activación de la cadena de llamadas recursivas. Observemos que tal condición se verifica en el Programa 3.2.11 sin problemas, pues como ya se señala anteriormente, si el parámetro actual es un parámetro formal por referencia, la dirección a calcular es la indicada por dicho parámetro actual, no la propia dirección del parámetro formal en el registro de activación.

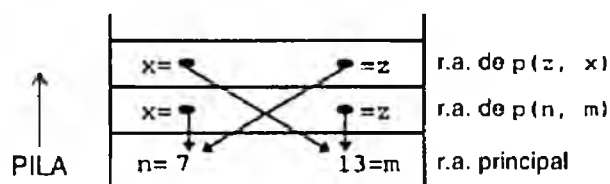


Figura 3.2.4

Instantánea de la pila de control durante una cadena de dos activaciones finales recursivas del procedimiento del Programa 3.2.11, según el modelo dinámico estándar

Ahora bien, ¿puede dejar de cumplirse tal condición de alguna manera? Es decir, ¿se puede escribir un procedimiento con un parámetro por referencia, y hacer que el parámetro señale a algún lugar *dentro* de los registros de activación de la cadena de llamadas recursivas? Desafortunadamente, es posible, y sirva como ejemplo el siguiente procedimiento.

```

program Prueba6 ...
var n: integer;

...

procedure p (var x: integer);
var k: integer;
begin
    k := x;
    if x <> 0 then
        begin
            k := k-1;
            p (k)          ( activación recursiva final )
        end
    else writeln (k)
    end;
end;

...
begin
    ...
    p (n)
    ...
end.

```

Programa 3.2.13 Procedimiento recursivo final con un dato local como parámetro actual por referencia en PASCAL.

Al ser activado el procedimiento *p* con un parámetro actual que es un dato local, el parámetro formal por referencia *x* quedará señalando en esa nueva activación a la memoria del dato local *k* en el registro de activación previo.

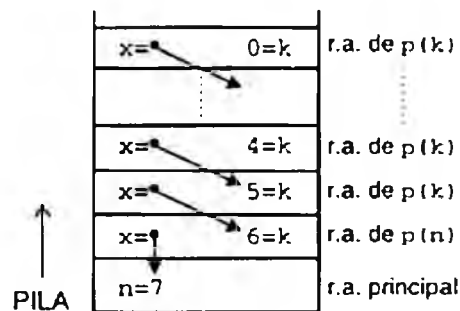


Figura 3.2.5 Instantánea de la pila de control durante una cadena de activaciones finales recursivas del procedimiento del Programa 3.2.13, según el modelo dinámico estándar

Desde luego, en estos momentos deja de ser válida toda la argumentación realizada al comienzo del capítulo y que permitió idear el modelo dinámico optimado. Ahora, no puede eliminarse el registro de la activación que invoca, pues contiene un dato al que se hace referencia desde el receptor de la llamada recursiva.

En general, se puede formular una de las conclusiones fundamentales de este trabajo: si la activación final recursiva de un procedimiento *tiene como parámetro actual por referencia un objeto local del propio procedimiento* (es decir, un objeto cuya memoria se sitúa en el propio registro de activación), entonces no es posible realizar la optimación. Ahora bien, "objeto local" del procedimiento parece ser un dato local (incluido en la parte de declaración de variables del procedimiento) o un parámetro formal por valor. Estos casos son claros, si se entienden como variables completas. Sin embargo, puede darse casos en que la memoria referida por el parámetro actual variable, aunque "provenga" de un objeto local, no se encuentre en el registro de activación del procedimiento. Nos estamos refiriendo, exactamente, al caso en que el parámetro actual variable refiere a memoria dinámica, que se sitúa en el montículo.

Supongamos que *loc* es un dato local, de tipo apuntador. El objeto *loc*[^] representa un dato que se sitúa en el montículo (suponiendo que se haya realizado la correspondiente reserva de memoria). Al colocar *loc*[^] como parámetro actual variable, se "transmite" realmente la dirección del montículo en que se sitúa el dato al que se refiere. Por lo tanto, no existe ningún problema en este caso para practicar la optimación, a pesar de que el parámetro actual variable se obtenga a partir de un objeto local. Esta situación puede generalizarse: si el parámetro actual por referencia es una variable identificada, entonces es posible la optimación, aunque el objeto a partir del cual se obtenga la variable identificada sea local en el procedimiento (Jensen y Wirth [1991: 163-164] denominan "variable identificada" a aquella que se obtiene a partir del valor de una variable apuntador). Esta conclusión puede obtenerse, por otro lado, gracias al hecho de que en PASCAL las variables identificadas siempre refieren a memoria dinámica, esto es, situada en el montículo; y ello a su vez por la imposibilidad de que una variable apuntador señale a memoria estática, es decir, situada en la pila.

La expresión definitiva de las conclusiones de los párrafos precedentes es la siguiente:

Restricción 3.2.2. Si la activación final recursiva de un procedimiento tiene como parámetro actual por referencia una variable, que no sea variable identificada, correspondiente a un dato local o un parámetro formal por valor del propio procedimiento, entonces no es posible realizar la optimación.

Sin embargo, puede colocarse como parámetro actual por referencia cualquier otro objeto: si es un parámetro formal por referencia del propio procedimiento, hemos visto con los programas previos que no existe ningún problema mientras estos parámetros contengan direcciones de objetos que se encuentran por debajo del registro de la activación que invoca, lo cual queda garantizado ahora por la Restricción 3.2.2; si se trata de un objeto no local al procedimiento (bien dato local o parámetro formal por valor o parámetro formal por referencia de un procedimiento de anidamiento superior), su memoria se sitúa estrictamente por debajo del bloque de registros de activación de la cadena de llamadas recursivas finales, así que la optimación sigue siendo posible.

El efecto de la Restricción 3.2.2 debe ser estudiado desde diferentes perspectivas. Primero, ha de valorarse su importancia en términos de la frecuencia en que tal construcción aparece en los programas. Concretamente, parece una técnica que cuando menos se puede calificar de "extraña" la de ofrecer como parámetro actual por referencia a un procedimiento recursivo un dato local o un parámetro formal por valor del propio procedimiento; pensamos, por lo tanto, a falta de un estudio más profundo, que el número de situaciones en que tal restricción impide la optimación debe ser bastante más reducido que el de situaciones en que esta sí es posible.

En segundo lugar, debe observarse que la Restricción 3.2.2 invalida la optimación de la llamada recursiva final que posea tales características, pero no impide la posible optimación de otras activaciones finales recursivas del mismo procedimiento que no estén sujetas a tal condición. Para verlo, estudiemos el siguiente ejemplo (Programa 3.2.14). Como se observa, existe una activación final recursiva optimable (la que tiene como parámetro actual el parámetro formal por referencia x) y otra no optimable (la que tiene como parámetro actual por referencia el dato local k). Si en la llamada origen del proceso recursivo el valor de n es 3, se tendría la siguiente configuración en la pila de control (según el modelo dinámico normal) tras un cierto número de activaciones (Figura 3.2.6).

Observamos que la activación final recursiva $p(x)$ podrían optimarse en las mismas condiciones que los casos estudiados anteriormente, y los sucesivos registros de activación de dichas llamadas podrían ocupar el lugar del último registro de activación de $p(k)$ que aparece en la Figura 3.2.6.

```

program Prueba7 ...
var n: integer;

...
procedure p (var x: integer);
var k: integer;
begin
  if x <= 0 and x > -2 then
  begin
    x := x-1;
    p (x)          ( activación recursiva final )
  end
  else
  begin
    k := x-1;
    p (k)          ( activación recursiva final )
  end
end;

...
begin
  ...
  p (n)
  ...
end.

```

Programa 3.2.14 Procedimiento recursivo final con un dato local como parámetro actual por referencia en PASCAL.

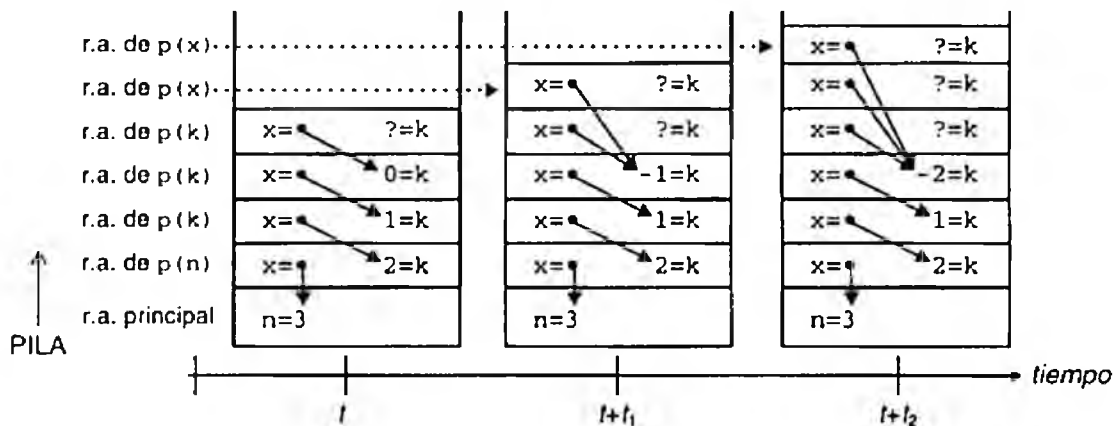


Figura 3.2.6 Instantáneas de la pila de control durante una cadena de activaciones finales recursivas del procedimiento del Programa 3.2.14, según el modelo dinámico estándar

El argumento es completamente general, pues en cuanto aparece una llamada recursiva final que no esté sujeta a la Restricción 3.2.2 vuelve a aplicarse todo lo demostrado en los apartados previos: manteniendo la información del registro de activación llamador y actualizando los valores de los datos locales y parámetros formales, la ejecución puede continuar normalmente. Por tanto, el Programa 3.2.14 es equivalente al siguiente, en que una de las llamadas sigue siendo recursiva, pero no la otra.

```

program Prueba7MitadRecursoivo --
var n: integer;

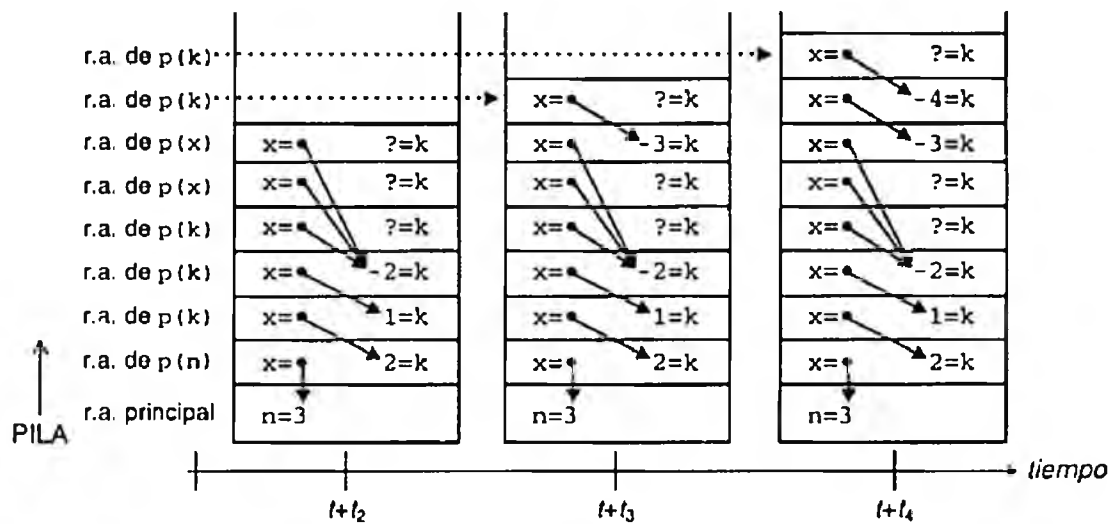
--
procedure p (var x: integer);
label 1;
var k: integer;
begin
  1: if x <= 0 and x > -2 then
    begin
      x := x-1;
      goto 1 { activación recursiva final optimada }
    end
  else
    begin
      k := x-1;
      p (k) { activación recursiva final no optimada }
    end
  end;
--
begin
  --
  p (n)
  --
end.

```

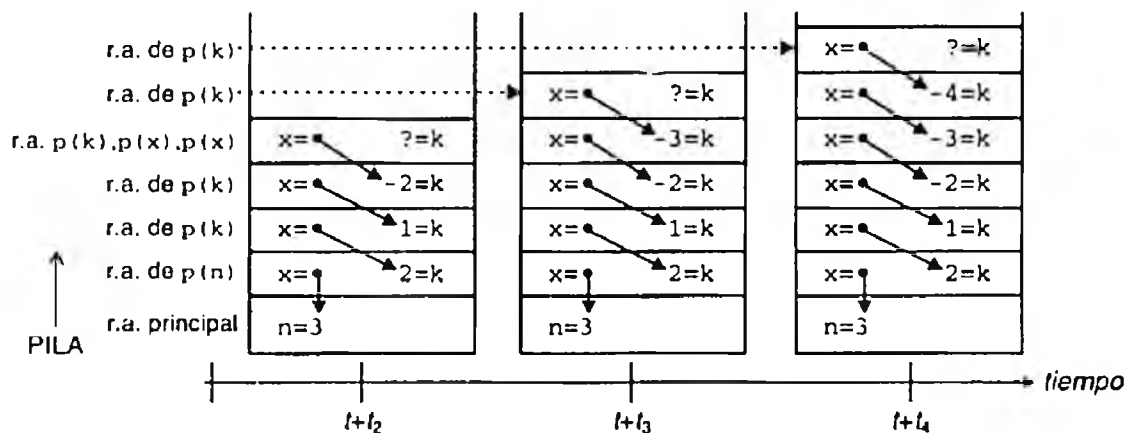
Programa 3.2.15 Procedimiento recursivo final (con una activación final optimada) equivalente al del Programa 3.2.14

Incluso esto es cierto aunque se vuelva a producir una nueva activación que sí verifica la Restricción 3.2.2: simplemente se vuelven a acumular nuevos registros de activación en la pila, como en el modelo dinámico estándar. Veamos que eso es lo que ocurre justamente en el Programa 3.2.14, si continuamos la cadena de activaciones recursivas, tal como queda reflejado en la Figura 3.2.7 que aparece más adelante: los registros de activación de las llamadas optimables $p(x)$ pueden suplantarse el registro de activación de la llamada $p(k)$ previa, y cuando vuelve a darse la activación $p(k)$ no optimable, se continúa con la acumulación de registros de activación, igual que en el modelo estándar. Por lo tanto, el Programa 3.2.15 expresa correctamente la optimación.

Un tercer aspecto ha de ser considerado. Por lo visto con los anteriores argumentos, el único problema para efectuar la optimación cuando la llamada recursiva final contiene como parámetro actual por referencia algún objeto local del procedimiento recursivo es la situación de la memoria a la que se va a asociar el parámetro formal por referencia: está en el registro de activación que tendría que eliminarse (o más bien, actualizarse) de la pila de control. Los otros elementos del registro de activación podrían perfectamente ser modificados de acuerdo a la secuencia de llamada optimada, y conseguir que tampoco en este caso se diera un crecimiento de la pila de control.



(a)



(b)

Figura 3.2.7

Más instantáneas de la pila de control durante una cadena de activaciones finales recursivas del procedimiento del Programa 3.2.14, (a) según el modelo dinámico estándar, y (b) según el modelo dinámico optimizado

Para ello, deberíamos conseguir que el valor del objeto local que se coloca como parámetro actual por referencia no se perdiese al hacer la actualización en la secuencia de llamada optimada. Eguíluz [1994] encontró que esto puede cumplirse si se crea una zona de memoria temporal en el registro de activación, de modo que al encontrar la llamada recursiva con las características que estamos considerando se haga lo siguiente:

- se copia el valor del parámetro actual por referencia en la zona temporal creada a tal efecto;
- se hace que el parámetro formal por referencia que se asociaría al anterior parámetro actual (siguiendo el modelo estándar) señale en lugar de ello a esa zona temporal.

Todo ello se realiza durante el paso LO1 del Algoritmo 3.2.1 de llamada optimada, lógicamente bajo las premisas dadas en la Condición 3.2.9, es decir, habiendo realizado previamente el cálculo de los valores de todos los parámetros actuales, y como parte entonces de las instrucciones de actualización de los parámetros formales correspondientes.

El comportamiento propuesto puede quedar mejor reflejado en el programa siguiente.

```

program Prueba7NoRecursoivo _
var n: integer;

procedure p (var x: integer);
label 1;
var k: integer;
    kt: integer;    ( variable temporal para duplicar k )
begin
1:  if x <= 0 and x > -2 then
    begin
        x := x-1;
        goto 1      ( salto al comienzo )
    end
    else
    begin
        k := x-1;
        kt := k;      ( copia del valor de k en kt )
        (dir x) := (dirección de kt);    ( x apunta a kt )
        goto 1      ( salto al comienzo )
    end
end;

begin
    p (n)
end.

```

Programa 3.2.16 Procedimiento no recursivo (con dos activaciones finales optimadas) equivalente al del Programa 3.2.14

La Figura 3.2.8 muestra la situación real de la pila de control con esta modificación del modelo dinámico optimado. En definitiva, se puede conseguir aún que el programa optimado se comporte de manera iterativa, aunque se dé como parámetro actual por referencia un objeto local del procedimiento recursivo.

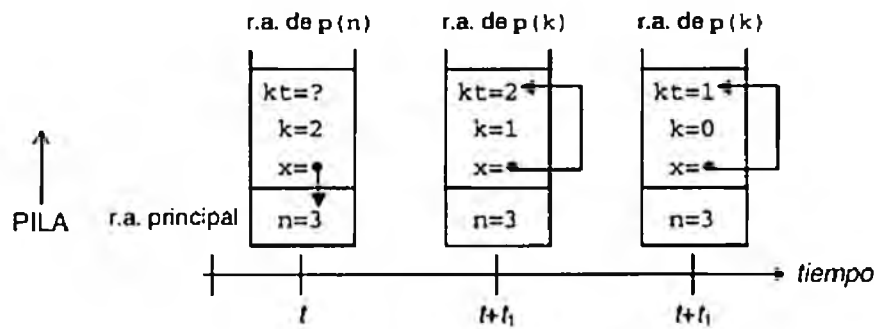


Figura 3.2.8 Instantáneas de la pila de control durante una cadena de activaciones finales recursivas del procedimiento del Programa 3.2.14, según el modelo dinámico optimado modificado

Sin embargo, esto plantea dos problemas que pueden ser graves. Primero, el esfuerzo que ha de hacer el compilador para detectar esta situación y para generar el código correspondiente; aunque trataremos este tema más adelante, ya podemos avanzar que no va a existir mucha más complejidad que en otras situaciones en que es posible la optimación. Y segundo, la complejidad del propio código generado; esta cuestión sí puede plantear un problema grave, pues observemos que cada llamada optimada según este algoritmo precisa de un movimiento de datos de una zona de memoria a otra; en el caso de datos básicos (enteros, reales, caracteres, etc.) ello no comporta más que una operación sencilla, pero la cosa cambia si se trata de un tipo estructurado, tal como un *array* o un *record*: la operación de movimiento puede conllevar entonces una ralentización considerable y significativa de la secuencia de llamada, y del comportamiento del programa en general.

Nos enfrentamos en esta situación, por tanto, a una disyuntiva, en la cual podemos hacer que un programa recursivo se ejecute en espacio constante, pero al precio de un posiblemente peor comportamiento temporal. Dejaremos pendiente esta cuestión para el momento en que estudiemos más en profundidad el comportamiento temporal de los programas según el modelo dinámico optimado, en el capítulo 6.

3.2.6 Parámetros procedimentales y funcionales

El uso de esta clase de parámetros en el lenguaje PASCAL es más bien esporádico, pero hemos de incluirlos en nuestro estudio si queremos dar la mayor generalidad al mismo.

El problema principal en relación con el modelo dinámico del lenguaje a la hora de implementar estos parámetros es la manera de asociar al parámetro actual el ambiente léxico que le corresponde, para ser usado en el momento en que se efectúe la llamada al parámetro formal procedimental o funcional. Según Aho, Sethi y Ullman [1990: 432-433], la manera más sencilla de conseguir esto es a través de la estrategia que usa el enlace de acceso: cuando aparece el parámetro actual procedimental o funcional, se calcula la dirección del punto de entrada al código del mismo, y el enlace de acceso que le correspondería como si fuera activado en ese momento; estos dos valores se asocian al parámetro formal procedimental o funcional; cuando este es activado, se realiza una secuencia de llamada habitual, colocando como enlace de acceso y como dirección de salto los valores que están asociados al parámetro formal. Por ejemplo, sea el siguiente esquema de programa.

```

program Ejemplo ...
...
  procedure p (...; procedure x (...); ...);
  ...
    x (...);    { activación del parámetro procedimental }
    ...        { puede darse en cualquier lugar dentro de p }
  end;
  ...
begin
  ...          { p es activado en algún lugar }
  p (q);       { q es un procedimiento definido en otro sitio }
  ...
end;
...
end.

```

Programa 3.2.17 Un programa PASCAL que incluye un procedimiento con un parámetro formal procedimental

Se tendrían las situaciones de la pila de control que se recogen a continuación, después de la activación del procedimiento *p* que tiene el parámetro formal procedimental (Figura 3.2.9a), y después de la activación del propio parámetro *x* (Figura 3.2.9b).

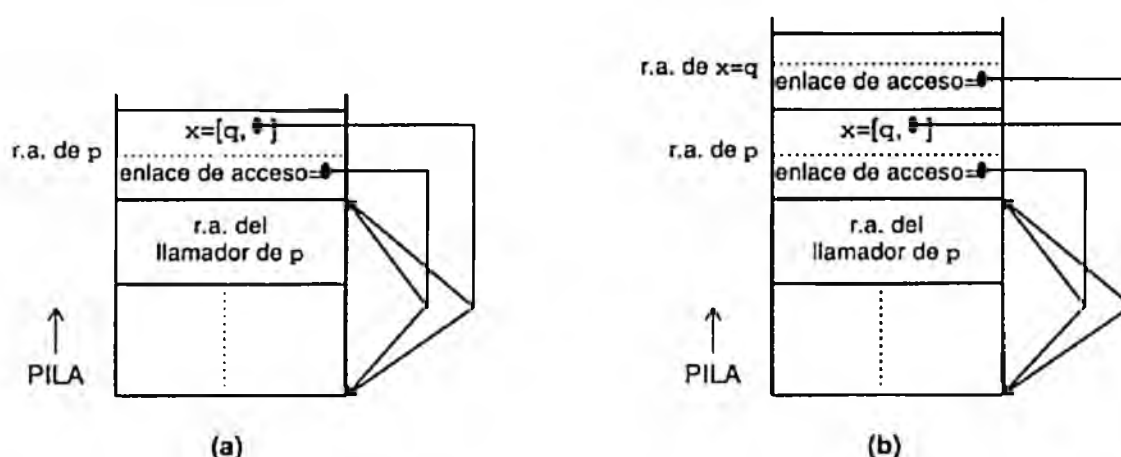


Figura 3.2.9 Instantáneas de la pila de control durante la ejecución del Programa 3.2.17 (a) después de la activación del procedimiento que incluye el parámetro formal procedimental, y (b) después de la activación del propio parámetro formal procedimental

La Figura 3.2.9a muestra que el parámetro formal procedimental consta de dos componentes: por un lado, el punto de entrada al procedimiento que se ha dado como parámetro actual (representado en la figura con el nombre q del procedimiento), y por otro, el enlace de acceso para el mismo, calculado como si el parámetro actual (esto es, el procedimiento q) hubiera sido activado en ese momento. En la Figura 3.2.9b se observa la situación de la pila cuando el parámetro formal procedimental es activado: se acumula en la pila un registro de activación para x , de manera que el enlace de acceso es el que se calculó previamente.

El proceso de asociación del parámetro actual q al parámetro formal procedimental x que se realizará durante la llamada $p(q)$ lo denotamos de la siguiente forma:

$$\begin{aligned} \langle ea \ x \rangle &:= \text{enlace de acceso calculado para } q; \\ \langle cod \ x \rangle &:= \text{punto de entrada al código de } q; \end{aligned}$$

Por otra parte, si la llamada fuera $p(y)$, donde y es otro parámetro formal procedimental, entonces debería hacerse:

$$\langle ea \ x \rangle := \langle ea \ y \rangle; \quad \langle cod \ x \rangle := \langle cod \ y \rangle;$$

es decir, se trata simplemente de copiar el propio contenido del parámetro actual sobre el parámetro formal, de modo muy similar a lo que ocurre con los parámetros por referencia estudiados en el apartado anterior.

Un par de aspectos merecen ser resaltados. Primero, el registro de activación que se crea cuando el parámetro formal procedimental se activa es realmente un registro de activación para el procedimiento dado como parámetro actual al que fue vinculado; para ello, la sintaxis del lenguaje PASCAL obliga a que la declaración del parámetro formal procedimental sea completa [Jensen y Wirth 1991: 186], de manera que la generación de código pueda establecer la estructura del registro de activación que debe acumularse en la pila de control sin necesidad de saber exactamente cuál va a ser el procedimiento real al que va a corresponder dicho registro de activación. Incidentalmente, esta necesidad ya fue notada por Welsh, Sneeringer y Hoare [1977], lo que motivó una de las primeras revisiones de la definición del lenguaje.

El otro aspecto atañe más directamente a nuestro estudio, y hace referencia a los enlaces de acceso involucrados en el comportamiento dinámico expuesto. En la Figura 3.2.9, el enlace de acceso calculado para el parámetro formal procedimental puede señalar a cualquier registro de activación de la cadena estática que va desde el llamador de p (inclusive) hasta el bloque principal (también incluido, obviamente). Veremos entonces lo que ocurre cuando el procedimiento p es recursivo final, es decir, cuando el llamador de p es el propio p , y la medida en que la situación afecta al modelo dinámico optimado.

Sea entonces p un procedimiento recursivo final con un parámetro procedimental o funcional (los casos con más parámetros de esta clase se resuelven de idéntica manera); supongamos también que el resto de elementos del procedimiento p son tales que admiten la optimación como hasta ahora la conocemos (por ejemplo, que el resto de parámetros son por valor). La primera activación de p ha de hacerse externamente a él, con un parámetro actual procedimental o funcional también externo a su bloque, por lo que el enlace de acceso calculado para el parámetro formal procedimental o funcional cae estrictamente fuera de la cadena de activaciones finales recursivas. Ahora bien, ¿podría ocurrir de algún modo que en cierta llamada de esa cadena el enlace de acceso calculado señalase a alguno de los registros de activación de la propia cadena? De ser así, la optimación quedaría invalidada, pues no podría perderse el registro de activación al que se establece la referencia. Y efectivamente, tal

situación es posible, y veremos que con grandes similitudes a lo encontrado en el apartado anterior en relación con los parámetros por referencia.

Examinemos un primer programa sencillo, con un parámetro procedimental en este caso. Para simplificar, se asume que tanto el procedimiento recursivo final como el procedimiento que se usa como parámetro actual procedimental están definidos como objetos locales del programa principal.

```

program Prueba8 ...
var n: integer;

...
procedure p (procedure x; y : integer);
begin
  if y <> 0 then
    p (x, y-1)      { activación recursiva final }
  else x            { activación del parámetro procedimental }
end;

...
procedure q;
begin
  writeln ('Hola')
end;

...
begin
  ...
  p (q, n)
  ...
end.

```

Programa 3.2.18 Procedimiento recursivo final con un parámetro procedimental en PASCAL

Los elementos relevantes de la pila de control durante las sucesivas activaciones recursivas de p están reflejados en la siguiente figura.

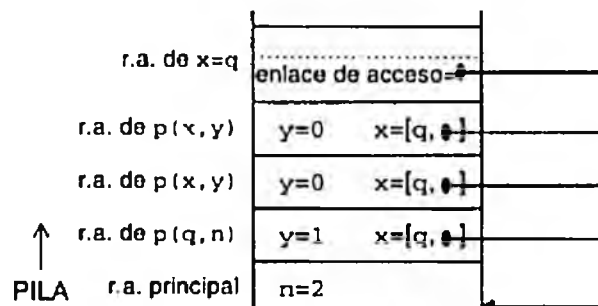


Figura 3.2.10 Instantánea de la pila de control durante una cadena de activaciones finales recursivas del procedimiento del Programa 3.2.18 (según el modelo dinámico estándar)

Con ello, se puede seguir un razonamiento similar al utilizado con los parámetros por referencia: puesto que todas las referencias que se realizan quedan estrictamente por debajo del grupo de registros de activación de la cadena de llamadas recursivas, nada impide sustituirlos por un solo registro de activación en el que se vayan realizando las oportunas actualizaciones.

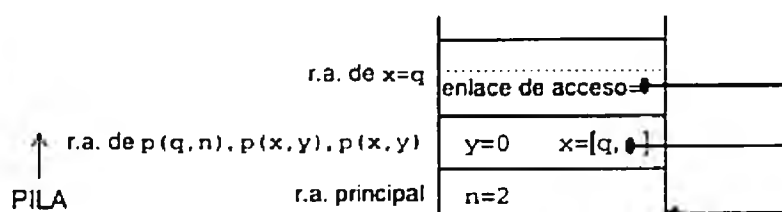


Figura 3.2.11 Instantánea de la pila de control durante una cadena de activaciones finales recursivas del procedimiento del Programa 3.2.18 (según el modelo dinámico optimado)

Se puede ofrecer una versión no recursiva del Programa 3.2.18, usando evidentemente la notación especial dada al principio de este apartado:

```

program Prueba8NoRecursivo ...
var n: integer;

procedure p (procedure x; y : integer);
label 1;
begin
1: if y <> 0 then
    begin
        y := y-1;           ( actualización de y )
        <ra x> := <ra x>;
        <ra x> := <ra x>;    ( actualización de x )
        goto 1
    end
    else x ( activación del parámetro procedimental )
end;

procedure q;
begin
    writeln ('Hola')
end;

begin
    p (q, n)
end.
```

Programa 3.2.19 Procedimiento no recursivo equivalente al del Programa 3.2.18

Por supuesto, es posible aplicar la idea encontrada ya varias veces en apartados anteriores: al ser el parámetro actual procedimental de las llamadas recursivas finales el propio parámetro formal, no es preciso siquiera actualizar su valor, sino que es invariablemente el mismo a lo largo de todas ellas, con lo que puede darse una nueva condición que completa la Condición 3.2.10 y la Condición 3.2.11:

Condición 3.2.12. Si el enésimo parámetro actual procedimental o funcional del procedimiento recursivo final es exactamente el enésimo parámetro formal del procedimiento, entonces no es preciso actualizar dicho parámetro.

La analogía con el caso de los parámetros por referencia es, de momento, total, y lo sigue siendo cuando se consideran varios parámetros procedimentales, que se "cruzan" en la llamada recursiva (de forma similar a lo expuesto en el Programa 3.2.11 y en el Programa 3.2.12).

```

program Prueba9 ...
var n: integer;

...
procedure p (procedure x; procedure z; y : integer);
begin
  if y <> 0 then
    p (z, x, y-1)           { activación recursiva final }
  else
    begin
      x; z { activación de los parámetros procedimentales }
    end
  end;
...
begin
  ...
  p (q, r, n)
  ...
end.

```

Programa 3.2.20 Procedimiento recursivo final con dos parámetros procedimentales en PASCAL

Simplemente hay que darse cuenta que, al actualizar un parámetro formal procedimental con un objeto que es a su vez parámetro formal procedimental, deben copiarse los valores asociados a tal objeto según el modelo dinámico; en el ejemplo, han de llevarse a x los valores asociados a z en la activación previa (o sea, el punto de entrada al procedimiento "real" y el enlace de acceso calculado para él), y al contrario, pero sin necesidad de volver a calcular ningún

nuevo enlace de acceso o nuevo punto de entrada. Lo podemos expresar, en nuestra notación, claro, de la siguiente forma:

```

program Prueba9NoRecursoivo ...
var n: integer;

...
procedure p (procedure x; procedure z; y : integer);
label 1;
var tmp1, tmp2: procedure;      { procedimentales "temporales" }
    tmp3: integer;              { el otro temporal }
begin
1:  if y <> 0 then
    begin
        <ra tmp1> := <ra z>;      <cod tmp1> := <cod z>;
        <ra tmp2> := <ra x>;      <cod tmp2> := <cod x>;
        tmp3 := y-1;
        <ra x> := <ra tmp1>;      <cod x> := <cod tmp1>;
        <ra z> := <ra tmp2>;      <cod z> := <cod tmp2>;
        y := tmp3;                { actualización de parámetros }
        goto 1
    end
end;

...
begin
...
    p (q, r, n)
...
end.

```

Programa 3.2.21 Procedimiento no recursivo equivalente al del Programa 3.2.20

Entonces, podemos generalizar la Condición 3.2.9 de la siguiente manera:

Condición 3.2.9 (tercera formulación y definitiva). La evaluación de los parámetros actuales por valor, por referencia, procedimentales y funcionales, y su asignación a los parámetros formales debe ser de tal modo que no se produzca la lectura de un parámetro formal durante algún proceso de evaluación de un parámetro actual después de haber actualizado dicho parámetro formal.

Con ello, queda representado el hecho de que, durante la optimación de procedimientos similares al del Programa 3.2.20, la actualización de los parámetros formales ha de hacerse con cuidado de preservar los valores previos de dichos parámetros, tal como siempre hasta ahora. Y también podemos reafirmar la imposibilidad de adoptar la estrategia de la transformación de programas para resolver la optimación, pues no es posible de ningún modo expresar las operaciones de actualización de los parámetros formales mediante el propio lenguaje PASCAL. Con ello, queda prácticamente zanjada una de las

hipótesis básicas del presente trabajo (sección 2.5), a pesar de que incluso aparecerán más argumentos a favor de ella.

Estudiemos ahora una modificación del procedimiento del Programa 3.2.18, en la que el parámetro actual procedimental de la llamada recursiva es otro procedimiento. Si este procedimiento fuera externo al bloque del procedimiento recursivo, al calcular su enlace de acceso se señalaría a algún registro de activación por debajo de los registros de activación de la cadena de llamadas recursivas; por lo tanto, la optimación seguiría siendo posible, en los mismos términos que antes. Sin embargo, si el parámetro actual procedimental de la llamada recursiva está definido dentro del bloque del procedimiento recursivo, entonces se va a establecer (a través del enlace de acceso) una referencia a uno de los registros de activación de la cadena de llamadas recursivas, con lo que no podrá perderse ese valor en las sucesivas activaciones recursivas, y la optimación no será posible. Más concretamente, tomemos el siguiente programa.

```

program Prueba10 _
  ***
  procedure p (procedure x; y : integer);
  var   n: integer;
  ***
  procedure q;
  begin
    writeln (n)
  end;
  ***
  begin
    n := y-1;
    if n = 0 then
      x   ( activación del parámetro procedimental )
    else if n = 3 then
      p (q, n)           ( activación (1) recursiva final )
    else
      p (x, n)           ( activación (2) recursiva final )
    end;
  ***
  begin
    ***
    p (r, 6)           ( r es un procedimiento no local a p )
    ***
  end.

```

Programa 3.2.22 Otro procedimiento recursivo final con un parámetro procedimental en PASCAL

Al llegar al punto en que n vale 3, se ha de hacer una nueva llamada recursiva con el procedimiento q como parámetro actual; según el modelo dinámico estándar, se acumula un nuevo registro de activación en que el

parámetro formal procedimental x se ha asociado a q : el enlace de acceso a calcular para x es el que tendría q si hubiese sido llamado en estos momentos; como q es local en p , su enlace de acceso apuntaría al registro de activación de p más reciente, o sea, al propio registro que está haciendo la llamada.

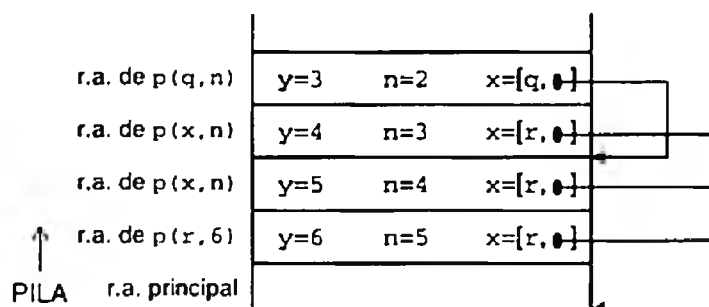


Figura 3.2.12 Instantánea de la pila de control durante una cadena de activaciones finales recursivas del procedimiento del Programa 3.2.22 (según el modelo dinámico estándar)

Al haberse establecido una referencia a uno de los registros de activación de la cadena de llamadas recursivas finales, la llamada $p(q, n)$ no puede optimarse, y debe ser procesada siguiendo la secuencia de llamada estándar. El resultado es análogo al encontrado con los parámetros por referencia: si el parámetro actual procedimental (o funcional) de una llamada recursiva final de un cierto procedimiento es un procedimiento (o función) local al llamador, entonces no es posible la optimación. Antes de formular esta nueva restricción, estudiemos más profundamente el problema, siguiendo las ideas ya conocidas para los parámetros por referencia, que se han mostrado igual de útiles en este caso.

Observemos primero lo que ocurre, con el programa anterior, si se siguen las llamadas recursivas hasta el caso no recursivo (*vide* Figura 3.2.13). Por un lado, ha de comprenderse bien el posible efecto de los parámetros procedimentales (o funcionales). Véase que, al guardarse con la referencia al procedimiento q su enlace de acceso, cuando luego llega la activación de este procedimiento (a través de la activación del parámetro formal procedimental), se trabaja con los valores de los datos tal como estaban en el momento de establecerse la referencia. El Programa 3.2.22 se ha escrito justamente para mostrar esta situación. La ejecución de la sentencia:

```
writeln (n)
```

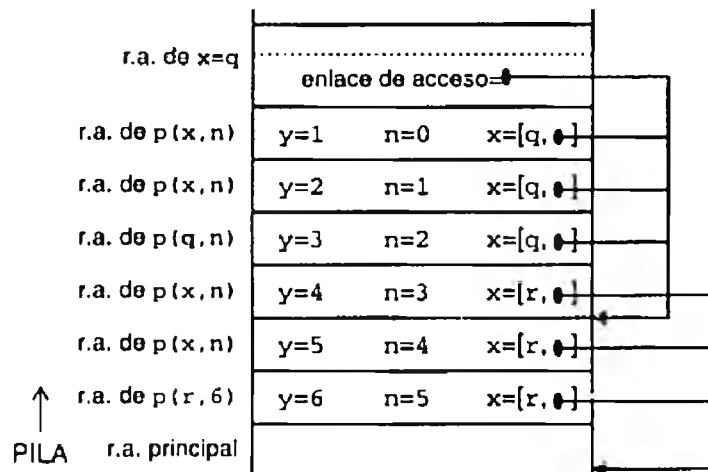



Figura 3.2.13 Otra instantánea de la pila de control durante una cadena de activaciones finales recursivas del procedimiento del Programa 3.2.22 (según el modelo dinámico estándar)

muestra (sorprendentemente) el valor 3, porque n no es local en q , así que se busca siguiendo su enlace de acceso una vez (distancia léxica entre la definición de n y su referencia), con lo que se llega al registro de activación de p en que se estableció el enlace de acceso para q , donde el valor de n es, efectivamente, 3.

Por otro lado, la Figura 3.2.13 muestra claramente que podrían optimarse todas las llamadas recursivas finales salvo aquella en que se establece el enlace de acceso a uno de los registros de activación de la cadena de llamadas recursivas; sintácticamente, esta es justamente la única llamada en que el parámetro actual procedimental es un procedimiento local al recursivo. Por lo tanto, exactamente como ocurría con los parámetros por referencia, aunque existan llamadas recursivas finales que no puedan optimarse, las demás, si cumplen los requisitos, pueden optimarse siguiendo la mecánica habitual.

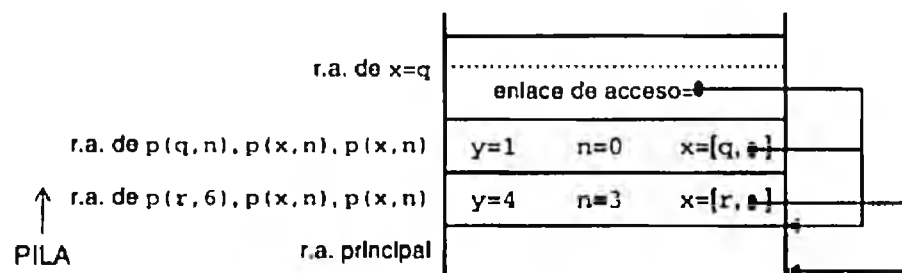


Figura 3.2.14 Instantánea de la pila de control durante una cadena de activaciones finales recursivas del procedimiento del Programa 3.2.22 (según el modelo dinámico optimado)

Para ver este aspecto aún más claramente, se modifica ligeramente el Programa 3.2.22, de manera que existan varias llamadas no optimables en la cadena de activaciones finales recursivas.

```

program Prueball ...
-
  procedure p (procedure x; y : integer);
  var   n: integer;
  -
    procedure q;
    begin
      writeln (n)
    end;
  -
  begin
    n := y-1;
    if n = 0 then
      x      ( activación del parámetro procedimental )
    else if n <= 3 then
      p (q, n)          ( activación (1) recursiva final )
    else
      p (x, n)          ( activación (2) recursiva final )
    end;
  -
begin
  -
  p (r, 6)      ( r es un procedimiento no local a p )
  -
end.

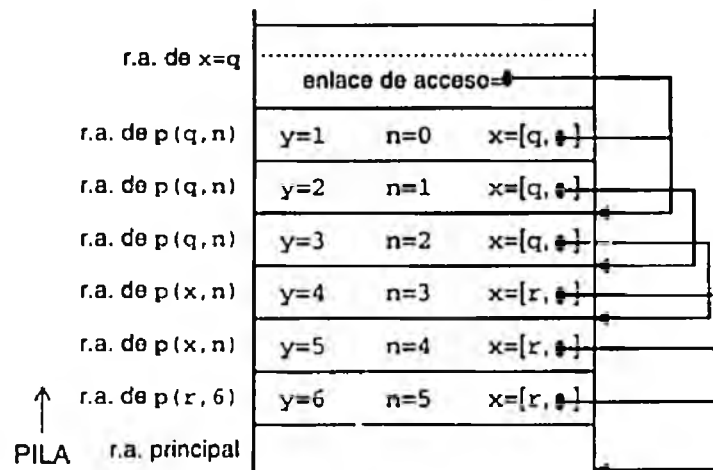
```

Programa 3.2.23 Procedimiento recursivo final con un parámetro procedimental en PASCAL (modificado a partir del Programa 3.2.22)

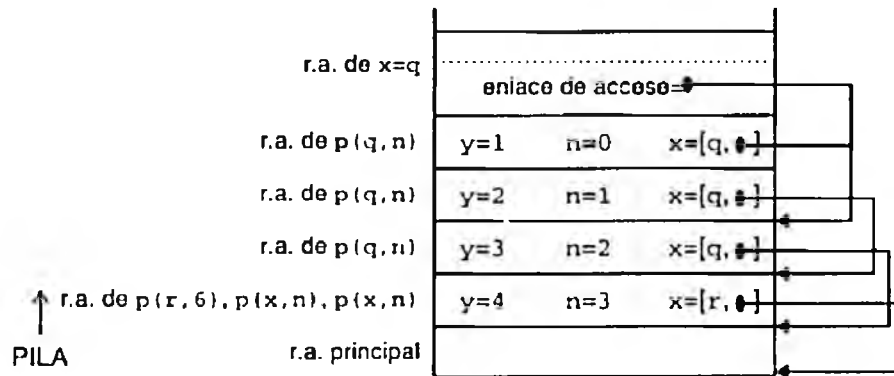
La situación de la pila de control tras la cadena de activaciones según uno y otro modelo está representada en la Figura 3.2.15. Se concluye que sigue siendo válida la estrategia de optimar (mediante el Algoritmo 3.2.1) las activaciones finales recursivas que sintácticamente contienen como parámetro actual procedimental un procedimiento no local al recursivo o bien un parámetro formal procedimental, y aplicar la secuencia de llamada estándar (Algoritmo 2.5.1) a las que tienen un procedimiento local al recursivo.

En definitiva, podemos formular una restricción similar a la Restricción 3.2.2 detectada en relación con los parámetros por referencia:

Restricción 3.2.3. Si la activación final recursiva de un procedimiento tiene como parámetro actual procedimental (o funcional) un procedimiento (o función) local al propio procedimiento, entonces no es posible realizar la optimación.



(a)



(b)

Figura 3.2.15 Instantáneas de la pila de control durante una cadena de activaciones finales recursivas del procedimiento del Programa 3.2.23, (a) según el modelo dinámico estándar, y (b) según el modelo dinámico optimado

El efecto de esta segunda restricción de la optimación planteada ha de estudiarse bajo los mismos aspectos que la primera. Uno de ellos tiene que ver con la frecuencia en que aparece este problema, que (debido a la "rareza" del uso de esta clase de parámetros en un lenguaje imperativo) es indudablemente muy pequeña; por lo tanto, consideramos que este aspecto tiene una trascendencia relativamente baja en este caso, pero así y todo lo estudiaremos más adelante con mayor profundidad.

Es interesante también darse cuenta que no es posible plantear una posible reforma de la optimación para tener un comportamiento de espacio constante, similar al ofrecido al estudiar los parámetros por referencia, al final

del apartado anterior. En efecto, la referencia en aquel caso era a un componente de clase "dato" del registro de activación (un dato local o un parámetro por valor); simplemente guardando una copia de este componente en el registro de activación y haciendo que el parámetro formal apuntase a él, se podía todavía hacer la optimación y asegurar el nulo crecimiento neto de la pila de control. Sin embargo, ahora la referencia se establece al enlace de acceso, lo que quiere decir que se establece potencialmente a prácticamente todos los elementos del registro de activación: no solo a los datos presentes en tal registro de activación sino también a otros datos situados en registros de activación accesibles mediante la cadena estática. Por lo tanto, no es posible en principio imaginar una estrategia que asegure un comportamiento en espacio constante, que es el objetivo último de la optimación que estamos estudiando.

Finalizamos el estudio de esta clase de parámetros con un comentario acerca del uso del *display* en lugar del enlace estático para el acceso a los datos no locales. Es ampliamente reconocido [Fischer y LeBlanc 1988: 310-314] que esta variante del modelo dinámico estándar impone una complejidad muy alta en el manejo de los parámetros procedimentales y funcionales (sobre todo, si se combina con los saltos no locales, que estudiaremos en un apartado posterior). Básicamente, la acumulación de un nuevo registro de activación supone guardar un componente de la matriz de *display*, cuando los parámetros son por valor o por referencia; sin embargo, si el lenguaje cuenta con esta otra clase de parámetros, la única estrategia sencilla conocida implica la acumulación en el nuevo registro de activación de toda la matriz de *display*, lo cual obviamente obliga a un comportamiento dinámico bastante limitado, tanto en términos espaciales como temporales. Por esta razón, no hemos hecho el estudio para esta modalidad del modelo dinámico, aunque es plausible pensar que no van a existir demasiadas diferencias con el modelo elegido.

3.2.7 Activaciones de otros procedimientos o funciones

Se podría argüir que quizás otras activaciones de procedimientos o funciones en el cuerpo del procedimiento que contiene las llamadas recursivas finales impidiesen la optimación. Sin embargo, no hay ningún problema a este respecto.

Supongamos un procedimiento con llamadas recursivas finales, de las cuales se han ejecutado cierto número, de manera que la pila de control, según el modelo dinámico estándar, contiene a partir de la cumbre ese mismo número de registros de activación (más el correspondiente a la primera llamada). Supongamos que durante la nueva activación del procedimiento, existen una serie de activaciones no finales; estas pueden ser tan complejas como se quiera, es decir, pueden por ejemplo dar lugar a otras activaciones anidadas antes de devolver el control al procedimiento recursivo, pueden ser activaciones recursivas (del propio procedimiento, pero no finales), incluso pueden ser activaciones recursivas indirectas (a través de otro u otros procedimientos o funciones).

En los casos que no están sujetos a la Restricción 3.2.2 y a la Restricción 3.2.3, solo el más reciente registro de activación de la cadena de activaciones finales recursivas es "visible" por cualquier otro subprograma que se active a partir de él: los objetos locales del procedimiento se encuentran, si son buscados, en ese registro de activación y los enlaces de acceso se van a colocar en todo caso señalando a ese registro de activación; por lo tanto, la presencia de los registros de activación previos es superflua, tal como se vio en el estudio inicial. En los casos sujetos a las restricciones anteriores, es justamente cuando las posibles referencias a registros de activación que no sean el más reciente obligan a mantener tales registros de activación, como hemos visto, pero los demás siguen siendo completamente inaccesibles a otras activaciones, luego su presencia también es superflua.

En definitiva, en la medida que el comportamiento del programa, en lo relativo a la pila de control, está determinado o bien por el registro de activación más reciente de la cadena de llamadas recursivas finales (y todos los accesibles desde este a través de la cadena estática), o bien por el registro de activación más reciente y todos aquellos a los que se hayan establecido referencias de las clases contempladas por la Restricción 3.2.2 y la Restricción 3.2.3 (y los accesibles a través de ellos por la cadena estática), entonces podemos concluir que la activación de otros procedimientos o funciones no va a invalidar la optimación, consistente justamente en la eliminación de los registros de activación sobre los cuales no se establece ninguna referencia, tal como se ha visto en los apartados previos.

3.2.8 Saltos no locales

Existe un último elemento del lenguaje PASCAL que tiene una influencia directa en el comportamiento dinámico en lo que a la pila de control se refiere: los saltos incondicionales, que se realizan mediante la sentencia *goto*. Existen dos clases de saltos:

«The statement that is prefixed by a label and each *goto* statement that refers to that label must satisfy one of the following two rules:

(a) The statement either must contain the *goto* statement or else must be one of the statements in a statement sequence [...] that contains the *goto* statement.

(b) The statement must be one of the statements in the statement sequence of the compound statement of the statement part of the block where the label is declared, and the *goto* statement must be contained in the procedure and function declaration part of that block» [Jensen y Wirth 1991: 171]

La primera clase corresponde a *saltos locales*, que no tienen ninguna relevancia sobre la pila de control, mientras que la otra clase, los *saltos no locales*, provocan, en la implementación habitual, una modificación drástica de aquella. En efecto, debe restaurarse el ambiente del subprograma al que se dirige el salto no local, lo que implica eliminar registros de activación de la pila [Fischer y LeBlanc 1988: 452-454; MacLennan 1987: 244-246].

Más concretamente, debe seguirse la cadena estática tantas veces como distancia de anidamiento léxico haya entre la sentencia *goto* y la etiqueta de salto, perdiéndose en la práctica todos los registros de activación entre esos dos puntos, es decir, finalizando todas las activaciones que hayan sido lanzadas a partir de la activación receptora del salto. El registro *BasePila* queda así establecido, y *TopePila* se calcula a partir de *BasePila* gracias al tamaño del registro de activación del receptor del salto, conocido para el compilador.

No hay dificultad en asegurar que los saltos no locales se van a ejecutar sin problema en el modelo dinámico optimado, justamente por la Condición 3.2.2 y la Condición 3.2.7, puesto que todas las activaciones de una cadena de llamadas recursivas de un procedimiento llevan su enlace de acceso al mismo procedimiento (Figura 3.2.1); por lo tanto, la presencia de un único registro de activación en lugar de todos los correspondientes a la cadena de llamadas recursivas mantiene la cadena estática, igual que se mantiene del

mismo modo el valor de `BasePila` que en todo caso debiera utilizarse para ejecutar el salto no local.

Finalmente, es conocido que la implementación de los saltos no locales cuando el acceso a los datos no locales se basa en la estrategia del *display* es extremadamente costosa, por lo que no la consideramos aquí. Realmente, el problema de los saltos incondicionales es otro de los aspectos que causa más desviaciones de la definición estándar del lenguaje PASCAL. Algunos compiladores examinados (de nuevo, el ejemplo es TURBO PASCAL [Borland 1992a]) no admiten saltos no locales, permitiendo, por contra, cualquier salto local, sin observar la restricción que prescribe el lenguaje. Por supuesto, esto parece ser debido a la complejidad que se requiere en el compilador para asegurar el cumplimiento de tales normas. En cualquier caso, es improbable que estos comportamientos alternativos invaliden la optimación, al menos en lo que a los saltos se refiere.

3.3 CONSIDERACIONES ADICIONALES

El estudio del caso particular de una llamada final directamente recursiva nos ha permitido obtener un modelo dinámico alternativo, en el que, bajo ciertas restricciones, no se consume memoria de la pila de activaciones: el proceso se ejecuta en un espacio constante. Por otro lado, la filosofía de reutilización del registro de activación del procedimiento llamador, que es la base del método, permite que la secuencia de llamada se reduzca a una actualización de parámetros y un simple salto al comienzo del código del procedimiento recursivo, con lo que se mejorará, como efecto adicional en cierto modo inesperado, el tiempo de ejecución del proceso.

Es cierto que la aplicación de esta secuencia de llamada optimada está sujeta a restricciones. Sin embargo, las situaciones en que puede aplicarse superan con creces el exiguo rango de posibilidades que propiciaba la filosofía de transformación de programas, debido a la falta de expresividad de PASCAL para expresar su propia semántica, tal como se ha observado en diversos ejemplos relativos a los parámetros que no se pasan por valor. El estudio más pormenorizado del alcance de las restricciones se realizará en el capítulo 7.

Ahora es más interesante comparar los resultados obtenidos con la estrategia precursora de nuestro trabajo, inventada por Hanson [1990], y recogida en el apartado 2.4.2. En principio, cabría decir que en el caso particular que estamos considerando, obtenemos exactamente el mismo comportamiento. Sin embargo, Hanson no menciona en ningún momento la existencia de ninguna clase de restricciones, así que este debe ser el tema a analizar.

La Restricción 3.2.2 plantea la imposibilidad de llevar a cabo la optimación si un objeto local (vinculado en la pila de activaciones) es colocado como parámetro actual por referencia. Ocurre que SCHEME (lenguaje sobre el que Hanson desarrolla su técnica) no posee este modo de transmisión de parámetros, por lo que es comprensible que no haya sido tenida en cuenta. Pero lo importante es que tal restricción surge también inexorablemente al aplicar su técnica al lenguaje PASCAL, esto es, se trata en cierto modo de una restricción inherente a las características del lenguaje (dado que se implementa con vinculación en una pila), más que un efecto de la solución particular adoptada. En efecto, al establecerse una referencia al registro de ambiente de la activación llamadora, no puede aplicarse la norma obtenida por Hanson de eliminar todos los registros de ambiente hasta el del superior léxico. por lo que la pila crece, exactamente como en nuestra solución.

La similitud con nuestros resultados no acaba ahí. Hanson se vería obligado a tratar una llamada que cumpliera la Restricción 3.2.2 como una llamada no final, según el modelo estándar (como nosotros), pues no puede acumular solamente un registro de ambiente para esa llamada. En principio, debemos darnos cuenta que la información de control para esa llamada que es final pero no optimable no sería nueva: sería, como en los otros casos, la ya existente en la pila. Sin embargo, si acumulase por ello solo un registro de ambiente, incumpliría la norma de que dos registros de ambiente adyacentes están unidos por el enlace de acceso (el del superior apunta justamente al inferior, como se recoge en la Figura 2.4.3). Esta norma es condición necesaria de su método de eliminación de registros de ambiente en la secuencia de llamada para las reducciones (lo que nosotros llamamos secuencia de llamada optimada), por lo que se concluye que ha de procederse como en un subproblema (llamada no final), acumulando el registro de control y el registro de ambiente correspondientes a la activación entrante. En definitiva, el

comportamiento de las dos soluciones será en todos los aspectos, espacial y temporalmente, idéntico.

La solución aportada por Eguiluz [1994] para evitar la Restricción 3.2.2 en nuestra técnica también puede aplicarse sin problemas a la estrategia de Hanson. Recordemos que consiste en crear un espacio temporal en el propio registro de activación que quede en la pila para que señale allí el parámetro formal por referencia que impedía la optimación, en lugar de mantener ese espacio en el registro de activación que no podía antes desaparecer. Las posibles "contraindicaciones" de esta solución (si debe aplicarse a un objeto de gran tamaño) siguen siendo su principal inconveniente, pero es desde luego una opción a considerar en casos de datos simples.

Pasemos al otro impedimento de la optimación, que tiene un análisis algo más complejo. La Restricción 3.2.3 prohíbe la optimación si un subprograma local es colocado como parámetro actual procedimental o funcional. En principio, Hanson no considera tampoco este problema, y ello sí es sorprendente en este caso, pues resulta que el lenguaje SCHEME sí admite esta clase de parámetros: en realidad, se permite cualquier dato como parámetro, y un procedimiento en SCHEME entra en el conjunto denominado "cualquier dato". Como se explicó en el apartado 2.4.1, los parámetros procedimentales se representan en SCHEME como en PASCAL (cf. apartado 3.2.6): con un *cierre léxico*, que es la combinación del punto de entrada al código del procedimiento y el enlace de acceso que señala a su ambiente. Entonces, el hecho es que la Restricción 3.2.3 debe aplicarse a la estrategia de Hanson de igual modo que a la nuestra, pues no es posible eliminar el ambiente de la activación que realiza la llamada, al generarse un enlace de acceso (como parte del cierre léxico) que señala directamente a ese ambiente. Y de igual modo que en el caso anterior, ello llevaría a la necesidad de tratar la llamada como no final, con la acumulación no solo de un nuevo registro de ambiente sino del registro de control previo correspondiente.

Por lo tanto, resultó, insistimos, sorprendente para nosotros que Hanson no mencionara esta cuestión. Ahora bien, podríamos preguntarnos si, en cualquier caso, esta restricción de la recursividad final existe en SCHEME, independientemente de que se siga la estrategia de Hanson para su implementación. Aquí estamos arañando en un problema que parece no del todo aclarado en la definición del propio lenguaje SCHEME. De hecho,

Kamin [1990: 442] llega a unas conclusiones parecidas, considerando justamente un ejemplo en el que se aplica la Restricción 3.2.3. Kamin expone el siguiente programa.

```
(set fac1 (lambda (x f)
  (if (<= x 1) (f 1)
      (fac1 (- x 1) (lambda (y) (f (* x y)))))))
```

Programa 3.3.1 Una función recursiva final en SCHEME (tomada de Kamin [1990: 442])

La función "anónima" `(lambda (y) (f (* x y)))` es una función definida localmente en `fac1`, que es pasada como parámetro actual funcional (*downward funarg*) a la propia función recursiva final `fac1`. Kamin explica que la implementación habitual de SCHEME lleva los cierres léxicos al montículo, a pesar de que en este caso no sería necesario, pues se trata de un parámetro funcional, y se puede mantener en la pila de activaciones, exactamente como en PASCAL. El resultado es que, efectivamente, la pila no crece, pero sí lo hace el montículo, así que el comportamiento conjunto es de crecimiento constante de la memoria necesaria para la ejecución del proceso. De hecho, puede escribirse una variación del Programa 3.3.1, para que la recursividad no se agote, y se obtiene una terminación anormal del programa debida al agotamiento de la memoria dinámica (el montículo).

Friedman, Wand y Haynes [1992: 243-247] clarifican de una forma elegante esta cuestión, distinguiendo los conceptos de "comportamiento iterativo" y "comportamiento iterativo de control". Un proceso tiene *comportamiento iterativo* a secas si sus necesidades totales de memoria están limitadas; sin embargo, un *comportamiento iterativo de control* significa que solo han de estar acotadas las necesidades de memoria para almacenar la información de control. Entonces, podemos ver que muchos procesos recursivos finales en SCHEME tienen comportamiento iterativo, pues ni el control ni el ambiente crecen durante el proceso. Sin embargo, existen otros procesos, y el Programa 3.3.1 es uno de los ejemplos paradigmáticos, que no tienen comportamiento iterativo, sino solo comportamiento iterativo de control: la memoria para almacenar ambientes crece conforme avanza el proceso. Y cuando la definición del lenguaje SCHEME expresa la necesidad de que su implementación sea propiamente recursiva final está requiriendo que los

procesos sintácticamente recursivos finales se ejecuten con un comportamiento iterativo de control.

Esta clarificación nos ayuda también a entender nuestros resultados aplicando la recursividad final al lenguaje PASCAL. Realmente, al partir de la hipótesis de una implementación tradicional del lenguaje (vinculación exclusivamente en la pila), estamos abocados a conseguir como mucho solamente la optimación de los procesos que en SCHEME tendrían un comportamiento iterativo, mientras que aquellos procesos que en este lenguaje mostrasen un comportamiento iterativo de control, aunque no comportamiento iterativo a secas, no podrán, con toda seguridad, optimarse en PASCAL. La situación reflejada en la Restricción 3.2.3 es una condición suficiente de no comportamiento iterativo: en SCHEME aún puede decirse que existirá al menos comportamiento iterativo de control, pero en PASCAL solo puede darse el comportamiento recursivo.

De todos modos, nos queda el consuelo de que programas como el Programa 3.3.1 pueden producir en SCHEME la eventual terminación anormal por un agotamiento de la memoria: en nuestra solución para PASCAL eso se producirá quizá antes (pues también crece la información de control), pero el efecto conjunto es el mismo. Por lo tanto, la Restricción 3.2.3 debe verse, a nuestro parecer, no como un inconveniente del método o solución elegida, sino, como ocurría con la Restricción 3.2.2, como un aspecto inherente a las características y propiedades de los parámetros funcionales en los lenguajes de programación. En definitiva, concluimos que no existen en la optimación de llamadas finales recursivas a procedimientos en PASCAL más restricciones que las que existen en SCHEME, por lo que la estrategia elegida puede tenerse por adecuada. Por supuesto, otra cosa distinta es la optimación de las que hemos denominado llamadas finales no recursivas, que pasamos a estudiar en el capítulo a continuación.

4. OPTIMACIÓN DE ACTIVACIONES FINALES NO RECURSIVAS DE PROCEDIMIENTOS

4.1 INTRODUCCIÓN

Este capítulo se dedica al estudio de la otra gama de optimaciones de llamadas a procedimientos en PASCAL, las que en el capítulo previo se denominaron *activaciones finales no recursivas*. Las ideas básicas de la optimación fueron introducidas también allí (apartado 3.1.1), y serán el punto de partida del estudio que se hará ahora. La estructura completa del capítulo es muy similar al anterior. Primero se estudiarán las condiciones sintácticas necesarias de posición final y nivel de anidamiento del procedimiento llamado. Después, se darán las secuencias definitivas de llamada y retorno. A la luz de ellas, examinaremos los elementos del lenguaje PASCAL que pueden tener influencia en la optimación, y encontraremos restricciones casi idénticas a las descubiertas en la optimación de activaciones recursivas finales. Finalmente, se compararán los resultados obtenidos con la posible filosofía transformacional y con las otras técnicas más próximas en espíritu a la que planteamos.

4.2 MODELO DINÁMICO OPTIMADO: ACTIVACIONES FINALES NO RECURSIVAS DE PROCEDIMIENTOS

Conviene empezar recordando las ideas básicas de la optimación de esta clase de activaciones finales, dadas en la introducción del capítulo anterior. Cuando un procedimiento p activaba en posición final a otro procedimiento

distinto q , el registro de activación de q podía ocupar el lugar del registro de activación del llamador p . Para ello, debía salvaguardarse cierta información de control del registro de activación de p (nosotros hablábamos de la dirección de retorno, pero puede que haya otros elementos) para que siguiera formando parte del registro de activación de q ; también concluimos que habría que prestar atención a los objetos locales del registro de activación del llamador, por si interviniesen en los parámetros actuales de la llamada. En definitiva, la secuencia de llamada optimada, a grandes rasgos, debía asegurar temporalmente ciertos componentes del registro de activación del llamador, y crear en su lugar el registro de activación para el receptor.

4.2.1 Posición final y nivel de anidamiento

La característica de posición final es, como en el caso de las activaciones recursivas, una condición sintáctica necesaria para la optimación. En el apartado 3.2.1 se definió ya lo que se iba a entender por activación final no recursiva: aquella que se realizará sobre un procedimiento distinto del llamador y que estuviera en posición final. La siguiente restricción, similar a la Restricción 3.2.1, da cuenta de que esta clase de sentencias son las que interesa a este nuevo tipo de optimación:

Restricción 4.2.1. Si la sentencia de activación no recursiva de un procedimiento no está en posición final en la parte de sentencias del bloque de otro procedimiento, entonces la optimación no puede llevarse a cabo.

La manera de verificar si una activación está en posición final es, por supuesto, la misma que se definió en el capítulo anterior, en el apartado referido antes.

El que sí es un aspecto importante ahora es el que descubrimos al presentar la idea básica de la optimación, en el apartado 3.1.1. Recordemos que no era posible aplicarla si el procedimiento llamado estaba definido localmente en el procedimiento llamador. La razón es clara: para el procedimiento activado son visibles los objetos locales del procedimiento que lo invocó, cuya memoria está justamente en el registro de activación que se eliminaría de la pila de control si se aplicase la optimación. Tenemos entonces otro de los resultados cruciales de nuestro estudio:

Restricción 4.2.2. Si una activación final no recursiva de procedimiento es de un procedimiento definido en la parte de declaración de procedimientos y funciones del procedimiento que realiza la llamada, entonces la optimación no puede llevarse a cabo.

Veamos que el resto de posibles activaciones, en lo que respecta al nivel de anidamiento, sí son optimables.

Sea, como antes, p el procedimiento llamador y q el receptor. Si no están sujetos a la Restricción 4.2.2 quiere decir que q tiene un nivel de anidamiento igual o menor que p ; en cualquier caso, siguiendo el modelo dinámico estándar, el registro de activación de q que se cree en la pila de control tendrá su enlace de acceso apuntando a algún registro de activación estrictamente *por debajo* del registro de activación de p .

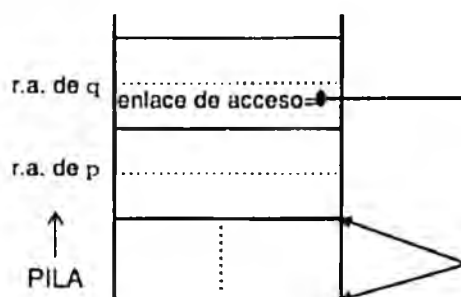


Figura 4.2.1 Enlace de acceso de un procedimiento activado no recursivamente sujeto a la Restricción 4.2.2

Por tanto, nunca serán visibles para q los objetos locales de p , y en lo que ellos respecta, la eliminación del registro de activación de p es perfectamente posible.

La Restricción 4.2.2 excluye implícitamente la optimación de una llamada final en la parte de sentencias del bloque principal; esto es lógico, pues en ese momento existe un solo registro de activación en la pila, que lógicamente nunca ha de desaparecer. Sin embargo, sí se admite la optimación de otro caso de activación final de procedimiento que hay que mencionar explícitamente: la de un parámetro formal procedimental. Esa es la razón por la que se ha usado la expresión "un procedimiento definido en la parte de declaración de procedimientos y funciones del procedimiento que realiza la llamada": un parámetro formal procedimental local en un procedimiento puede ser activado finalmente por este, y aun así la optimación es posible, según demostraremos.

Conviene recordar que la implementación habitual de los parámetros procedimentales, como vimos en el capítulo anterior, consiste en colocar en el registro de activación el "cierre léxico", formado por dos valores: el punto de entrada al procedimiento "real" asociado a dicho parámetro formal, y el enlace de acceso que correspondería a dicho procedimiento si hubiese sido llamado en lugar de ser transmitido como parámetro actual. Podemos retomar concretamente el Programa 3.2.17, suponiendo que la activación del parámetro formal procedimental x se produce en posición final; la Figura 3.2.9b muestra que el enlace de acceso de la activación de x nunca puede llevar al registro de activación del procedimiento llamador de x . Por tanto, la optimización puede efectuarse del mismo modo que en los casos estudiados. Además, ello es independiente del nivel de anidamiento del procedimiento "real" que fue pasado como parámetro actual procedimental; exactamente, en el Programa 3.2.17 el procedimiento α puede ser incluso un procedimiento definido localmente en p , pues el resultado obtenido sigue cumpliéndose.

4.2.2 Optimación de la secuencia de llamada

La primera hipótesis del modelo optimado ya la conocemos del estudio del otro caso de optimación, y realmente es una condición general para todas las clases de optimaciones que se proponen; la repetimos aquí con el ánimo de mantener la completitud en la descripción de todas ellas:

Condición 4.2.1. En el modelo dinámico optimado, las operaciones de las secuencias de llamada y de retorno que son realizadas por el receptor de la llamada han de mantenerse sin variación respecto al modelo dinámico estándar.

La modificación de la secuencia de llamada de procedimientos (Algoritmo 2.5.1) del modelo dinámico estándar para las activaciones finales no recursivas precisa ahora del conocimiento de cómo se ven afectados el resto de componentes del registro de activación. Con la idea básica de que el registro de activación del receptor ocupará el lugar del registro de activación del llamador, haremos ese estudio antes de ofrecer la secuencia de llamada optimada definitiva. A simple vista, el requisito a cumplir es que debe preservarse toda la información (tanto de datos como de control) del registro de activación del

procedimiento llamador hasta que sea seguro que ya no es necesaria, o bien hasta que se haya colocado en una posición definitiva para la activación receptora.

Examinamos en primer lugar los valores de los registros *BasePila* y *TopePila*. Dada la estructura del registro de activación elegida para el modelo dinámico en el apartado 2.5.1 (Figura 2.5.2), está claro que el valor de *BasePila* ha de mantenerse intacto, mientras que *TopePila* debe ser actualizado, puesto que el nuevo registro de activación para el procedimiento receptor puede tener un tamaño diferente del registro de activación del llamador. Por tanto, tenemos:

Condición 4.2.2. El valor de *BasePila* de la activación llamadora ha de mantenerse en la nueva activación final no recursiva.

Este resultado depende directamente de la estructura particular elegida para el registro de activación, en que la zona de control se sitúa en la base del registro para cualquier subprograma; si se elige otra estructura, como por ejemplo la descrita por Aho, Sethi y Ullman [1990], puede que esta condición no pueda considerarse como tal para el modelo dinámico optimado. Este problema lo estudiaremos más detalladamente en el capítulo 6.

En cuanto a *TopePila*, dado que su actualización es competencia del procedimiento receptor de la llamada (paso L7 del Algoritmo 2.5.1 de llamada estándar), el valor con el que debe llegar a ese punto depende de la instrucción con que se realice tal paso de la llamada. En la secuencia de llamada elegida, su cálculo se realiza a partir del valor de *BasePila*, sumando a este registro el tamaño de la zona de datos del procedimiento receptor; por lo tanto, simplemente no es necesaria ninguna modificación de *TopePila*, lo que elimina una instrucción en la secuencia de llamada optimada. Sin embargo, podría ser que se calculase de alguna otra manera (por ejemplo, a partir del valor de *TopePila* del llamador), con lo cual sería necesario modificar el registro de la forma conveniente para que la actualización fuera correcta:

Condición 4.2.3. Si la actualización de *TopePila* en la secuencia de llamada estándar (paso L7) depende del valor del propio registro, la activación llamadora ha de modificar dicho registro para que la actualización sea correcta.

Por otro lado, hemos establecido implícitamente el punto al que debe efectuarse el salto en la activación final no recursiva:

Condición 4.2.4. El salto por el que se activa el procedimiento final no recursivo ha de efectuarse al punto de entrada del procedimiento.

En definitiva, el procedimiento receptor ejecuta su parte de secuencia de llamada normalmente, tal como se establece en la Condición 4.2.1.

La dirección de retorno, como ya sabemos, no ha de cambiar, así que tenemos el siguiente requisito que ha de cumplirse:

Condición 4.2.5. La dirección de retorno del procedimiento que realiza la llamada no recursiva final ha de mantenerse en esta última.

Con el enlace de control ocurre otro tanto, por razones similares a las ofrecidas en el caso de las activaciones recursivas:

Condición 4.2.6. El enlace de control del procedimiento que realiza la llamada no recursiva final ha de mantenerse en esta última.

Ambos requisitos, por tanto, son también los que existen en la optimación estudiada en el capítulo anterior.

Donde sí van a existir diferencias es en los elementos que permiten el acceso a los datos no locales, es decir, el enlace estático o el *display*. Obviamente, en el nuevo registro de activación estos componentes tendrán valores diferentes, puesto que es un procedimiento distinto el que se activa, por lo que habrá que proceder a calcularlos, en principio siguiendo la estrategia habitual prevista en el modelo dinámico estándar.

En el caso del enlace de acceso, como se sabe [Aho, Sethi y Ullman 1990: 431-432], el compilador debe establecer la distancia léxica entre el nivel de anidamiento de la llamada, n_{act} , y el nivel de anidamiento de la definición del procedimiento llamado, n_{def} . Solo puede ocurrir que $n_{def} < n_{act}$ (el otro caso posible, $n_{def} = n_{act}$, es justamente el que evita la Restricción 4.2.2). Entonces, el nuevo enlace de acceso se calcula siguiendo $n_{act} - n_{def}$ veces la cadena estática. Este proceso sigue siendo válido, pero hay que asegurar que el enlace de acceso del procedimiento llamador no se pierde antes de calcular el enlace de acceso del procedimiento llamado:

Condición 4.2.7 (primera formulación). El enlace de acceso del procedimiento que realiza la llamada no recursiva final ha de mantenerse al menos hasta que se haya calculado el enlace de acceso del nuevo registro de activación.

Dicho de otro modo, en la secuencia de llamada optimada se ha de tener cuidado en no perder el valor del enlace de acceso del procedimiento que invoca mientras se crean los componentes del registro de activación del procedimiento llamado. Esta condición no tiene por qué ser difícil de cumplir, como veremos; a pesar de que parece más compleja que la Condición 4.2.5 y la Condición 4.2.6, realmente es una condición parecida: del mismo modo que la dirección de retorno y el enlace de control del antiguo registro de activación han de salvaguardarse de alguna forma antes de ser situados en el nuevo registro de activación, el enlace de acceso antiguo ha de ser usado para calcular el nuevo valor y entonces esperar a situarlo en su nueva localidad. Con la estructura elegida para el registro de activación en este estudio, la Condición 4.2.5, la Condición 4.2.6 y la Condición 4.2.7 se verifican de manera trivial, pues los elementos de la zona de control del nuevo registro de activación ocupan exactamente las mismas posiciones que en el registro de activación del llamador; sin embargo, otras estructuras alternativas pueden tener problemas a este respecto, y por esa razón hemos enunciado las condiciones mencionadas.

Además, conviene tener en cuenta un caso particular que puede ser interesante, por la frecuencia con que aparece. Es el caso en que el nivel de anidamiento del procedimiento llamador es el mismo que el del procedimiento receptor. Entonces, $n_{act} - n_{del}$ vale 1, por lo que el enlace de acceso para el receptor es exactamente el mismo que tenía el receptor, así que no es necesario calcularlo (obsérvese de paso la analogía con las activaciones recursivas, que están sujetas obviamente a esta misma característica). Se puede entonces reformular la Condición 4.2.7 de la siguiente manera:

Condición 4.2.7 (segunda formulación). Si el procedimiento llamado en posición final tiene el mismo nivel de anidamiento que el procedimiento llamador, entonces el enlace de acceso del llamador ha de mantenerse en el receptor. Si el procedimiento llamado en posición final tiene menor nivel de anidamiento que el procedimiento llamador, entonces el enlace de acceso del llamador ha de mantenerse al menos hasta que se haya calculado el enlace de acceso para el receptor.

Realmente, esta es una optimación que podría llevar a cabo sin ninguna dificultad cualquier compilador convencional (independientemente de la posición de la llamada); como hemos dicho, que un procedimiento active a otro

con el mismo nivel de anidamiento es una situación bastante frecuente, así que es interesante para el código objeto evitar una operación que es innecesaria.

El uso del *display* en lugar del enlace de acceso sí requiere un tratamiento un tanto complejo, pero el estudio nos va a servir al menos para apreciar mejor el efecto de la optimación de las llamadas recursivas si se usa esta estrategia. Para comprender el problema, supongamos que el procedimiento p activa al procedimiento q en posición final; las operaciones que se realizarían en lo que respecta al *display* desde la activación de p hasta su retorno son las siguientes (siguiendo las secuencias de llamada y de retorno estándares, Algoritmo 2.5.1 y Algoritmo 2.5.2, respectivamente):

- cuando p es llamado y se acumula su registro de activación en la pila, se almacena en él el registro de *display* $d[n_p]$ antiguo, y se da a este el nuevo valor, donde n_p es el nivel de anidamiento de la definición de p ;
- ahora p llama a q , con lo que se realiza la misma operación con $d[n_q]$, donde n_q es el nivel de anidamiento de la definición de q ;
- en este momento q retorna, recuperando el registro $d[n_q]$ guardado;
- y también p retorna, recuperando a su vez el registro $d[n_p]$ guardado.

En el modelo optimado, sin embargo, hay una sola operación de retorno, la de q , aunque se haga a la dirección de retorno de p ; la Condición 4.2.1 obliga a mantener la actualización del *display* en la secuencia de retorno, pero se hará solo la que corresponde a q , así que nos queda la de p . Parece que hemos de llevar a cabo el restablecimiento del *display* para el procedimiento llamador en algún otro instante. ¿Qué tal durante la secuencia de llamada de q ? Al fin y al cabo es el único momento que nos queda: antes no puede ser de ninguna manera, y después, una vez que se activa q , se pierde esa información del registro de activación de p . (Compárese esto, antes de nada, con la conclusión obtenida en el caso de las activaciones recursivas finales, la Condición 3.2.8; allí se mantenían sin variación los valores de $d[n_p]$ actual y $d[n_p]$ guardado, porque cada nueva activación según el modelo estándar significaba simplemente una nueva referencia entre ellos mismos. De este modo, comprendemos que la optimación es más drástica en las activaciones recursivas, apoyando la distinción hecha desde el primer momento respecto a la clase de llamadas que ahora consideramos.)

Es posible realizar el restablecimiento prematuro del registro de *display* correspondiente al procedimiento llamador, y ello de nuevo gracias a la Restricción 4.2.2. Sea, como antes, n_p el nivel de anidamiento de la definición de p . Durante la activación de este procedimiento, el registro de *display* $d[n_p]$ señala al registro de activación de p , donde se acumula el valor de $d[n_p]$ previo a la activación de p . Este valor o bien no es significativo, si p fue el primer procedimiento con nivel n_p que se activó, o bien señala a un registro de activación situado más abajo en la pila, perteneciente a un procedimiento con el mismo nivel de anidamiento (que puede ser el propio p u otro procedimiento distinto).

Ahora se activa el procedimiento q en posición final, con nivel de anidamiento de definición n_q . Por la Restricción 4.2.2 solo puede ser que $n_p \geq n_q$. Recordemos que esto garantiza que q no puede acceder a los datos locales de p , pero entonces tampoco a ninguno de los procedimientos con el mismo nivel de anidamiento que p . Pueden ocurrir ahora dos casos:

- $n_p = n_q$. Según el modelo dinámico estándar, como p y q tienen el mismo nivel de anidamiento, en el registro de activación de q se almacenaría el valor de $d[n_p]$, que señala al registro de activación de p (vide Figura 4.2.2a). Si en lugar de ello, al llamar a q se copia el valor de $d[n_p]$ guardado para p en el nuevo registro de activación de q , estaría resuelto el problema: por un lado, el nivel n_p es ostentado ahora por q , y los demás niveles no han cambiado, así que el acceso a los datos locales es idéntico al que se tiene según el estándar; por otro lado, cuando q retorne, dejará el registro $d[n_p]$ exactamente como quedaría en el modelo estándar tras retornar q y luego p , con el valor de $d[n_p]$ previo a la activación de este último (consúltese la Figura 4.2.2b).
- $n_p > n_q$. Según el modelo dinámico estándar, en el registro de activación de q se almacenaría el valor de $d[n_q]$, que en este caso no tiene relación con el registro de activación de p (véase la Figura 4.2.3a). Si en lugar de ello, al llamar a q se restablece previamente el valor de $d[n_p]$ guardado para p , también estaría resuelto el problema: por un lado, todos los niveles superiores a n_q son ahora inaccesibles para q , así que da igual que $d[n_p]$ señale al registro de activación de p o a otro registro de activación; los demás niveles no han cambiado, así que el acceso a los datos locales es idéntico al que se tiene según el modelo estándar; por otro lado, cuando q

retorne, modificará el registro $d[n_q]$ como en el modelo estándar, y $d[n_p]$ ya tendrá el valor previo a la activación de p (lo que está reflejado en la Figura 4.2.3b).

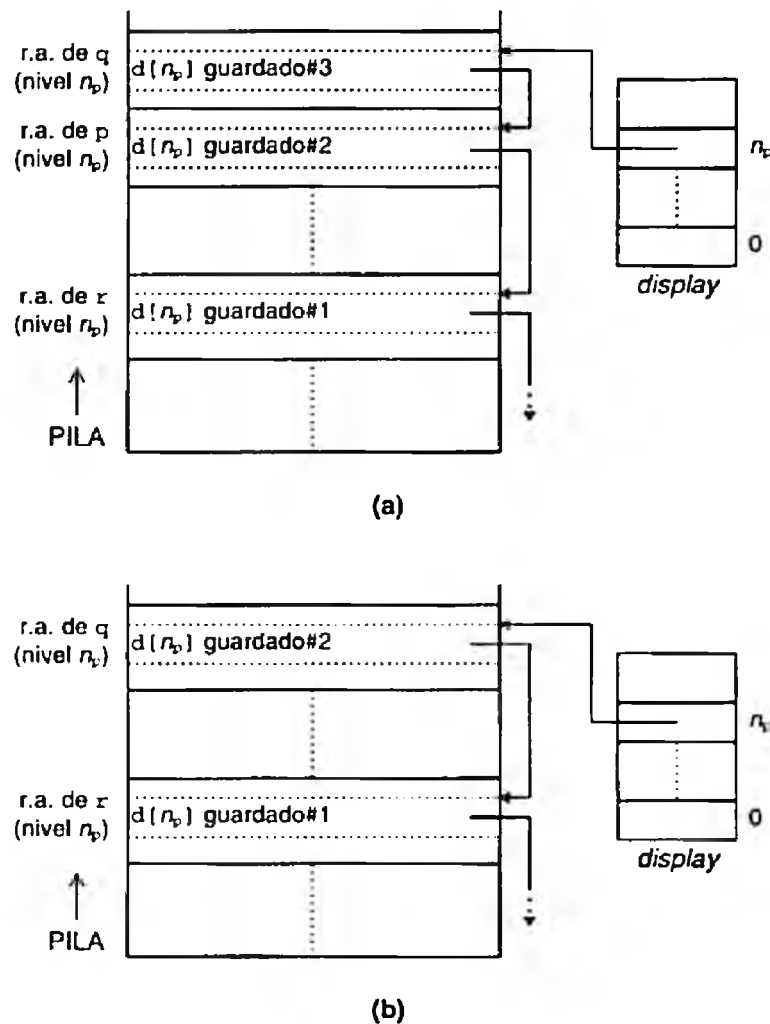


Figura 4.2.2 Situaciones de la pila y del *display* después de la activación no recursiva final de un procedimiento con el mismo nivel de anidamiento que su llamador (a) según el modelo dinámico estándar, y (b) según el modelo dinámico optimado

En definitiva, realizando el restablecimiento prematuro del registro de *display* del procedimiento llamador durante la secuencia de llamada optimada del procedimiento en posición final, se consigue mantener el comportamiento requerido del programa objeto.

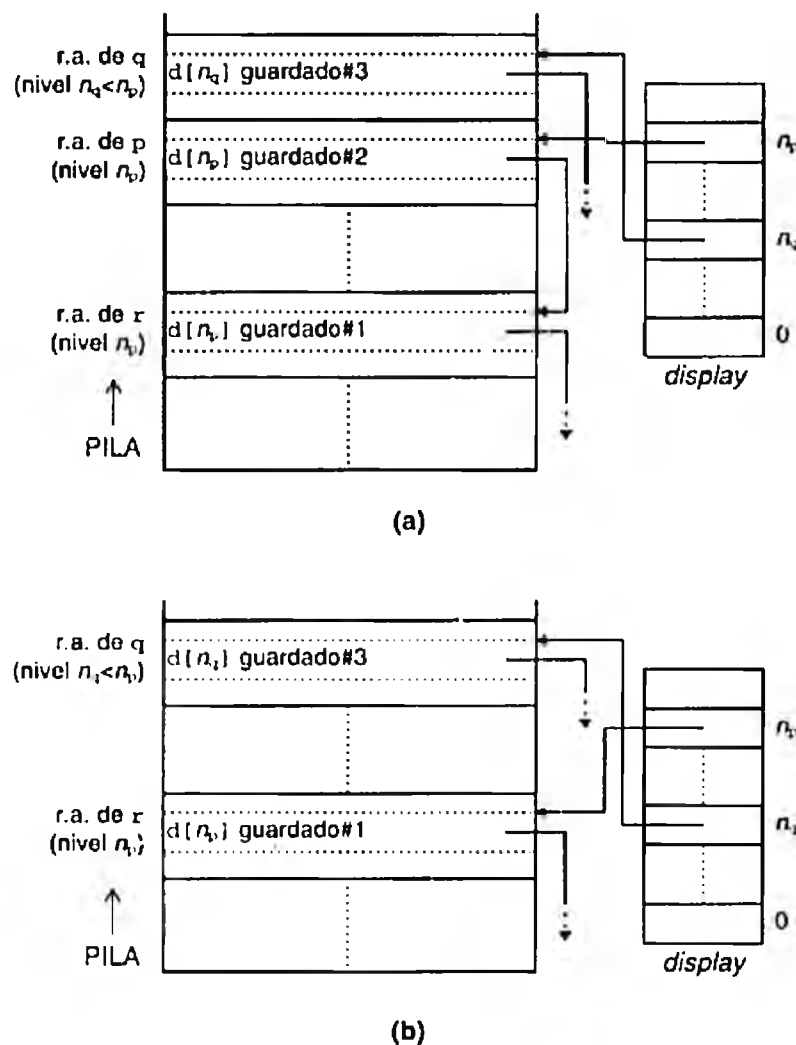


Figura 4.2.3 Situaciones de la pila y del *display* después de la activación no recursiva final de un procedimiento con nivel de anidamiento inferior a su llamador (a) según el modelo dinámico estándar y (b) según el modelo dinámico optimado

La conclusión anterior se recoge tentativamente en la forma general siguiente:

Condición 4.2.8 (primera formulación). El restablecimiento del registro de *display* correspondiente al procedimiento que realiza la llamada no recursiva final ha de realizarse durante la propia secuencia de llamada, y antes del establecimiento del nuevo valor del registro de *display* correspondiente al receptor.

Esta formulación, sin embargo, no tiene en cuenta que, según el estudio realizado, existen dos posibles operaciones alternativas de actualización del *display* en la secuencia de llamada optimada:

- una que agrupa los dos casos anteriormente descritos: se restablece el registro de *display* del procedimiento llamador, y después se calcula el registro de *display* para el procedimiento receptor;
- otra que diferencia los dos casos: cuando $n_p = n_q$, se mantiene el valor de $d[n_p]$ guardado para el nuevo registro de activación del procedimiento receptor; cuando $n_p > n_q$, se actúa como en la alternativa previa; obviamente, esta alternativa es mejor que la otra (al evitar operaciones redundantes en el caso $n_p = n_q$), pero requiere cierto análisis estático.

Se puede dar entonces la expresión alternativa:

Condición 4.2.8 (segunda formulación). Si el procedimiento llamado en posición final tiene el mismo nivel de anidamiento que el procedimiento llamador, entonces el registro de *display* guardado de la activación del llamador ha de mantenerse en la activación del receptor. Si el procedimiento llamado en posición final tiene menor nivel de anidamiento que el procedimiento llamador, entonces el restablecimiento del registro de *display* del llamador ha de realizarse durante la propia secuencia de llamada, y antes del establecimiento del nuevo valor del registro de *display* correspondiente al receptor.

Igual que en el caso del enlace de acceso, la detección del caso particular de procedimientos con el mismo nivel de anidamiento es trivial para el compilador, y se evita una operación en la secuencia de llamada.

Una vez examinados todos los componentes del registro de activación, podemos ya dar la *secuencia de llamada optimada para activaciones finales no recursivas de procedimientos*, recogida en el Algoritmo 4.2.1 más adelante. En lo que respecta al paso LO1, nos mantenemos a la espera de un estudio más en profundidad, pues es un proceso que requiere mayor complejidad que en el caso de las activaciones recursivas finales. Por lo demás, los pasos LO4 y LO5 son exactamente los mismos que los pasos L7 y L8 de la secuencia de llamada estándar (Algoritmo 2.5.1) correspondiente al receptor, tal como indica la Condición 4.2.1. Finalmente, no hemos incorporado en el Algoritmo 4.2.1 el estudio precedente sobre el enlace de acceso o el *display*: solo hay que añadir

que, si el procedimiento que recibe la llamada tiene la misma profundidad de anidamiento que el que invoca, entonces el paso LO2 no es preciso.

- LO1: el llamador evalúa los parámetros actuales, y coloca los valores obtenidos en el lugar correspondiente de su propio registro de activación, mediante desplazamientos respecto a *BasePila*;
- LO2: el llamador calcula el nuevo enlace de acceso, si se usa la cadena estática, y lo coloca en el lugar correspondiente de su propio registro de activación; si se usa el *display*, restablece el registro de *display* que le corresponde, según se haría normalmente en su propia secuencia de retorno, coloca en el lugar correspondiente de su propio registro de activación el registro de *display* a guardar para la profundidad de anidamiento del receptor, y actualiza dicho registro;
- LO3: el llamador salta (jump) a la primera instrucción del código del receptor;
- LO4: el receptor actualiza *TopePila* a su nuevo valor, que es *BasePila* más el tamaño de la zona de datos;
- LO5: el receptor asigna valores iniciales a sus datos locales.

Algoritmo 4.2.1 Secuencia de llamada optimada para una activación final no recursiva de procedimiento

Por supuesto, esta secuencia de llamada tiene un aspecto un poco más complejo que la secuencia de llamada optimada para las activaciones recursivas finales (Algoritmo 3.2.1), pero al fin y al cabo se observa que la diferencia está en la actualización del elemento destinado al acceso a los nombres no locales (enlace de acceso o *display*), y en la actualización de *TopePila*. Por lo tanto, a primera vista puede concluirse que las mejoras introducidas por esta secuencia de llamada optimada son considerables, no solo por evitar el crecimiento de la pila de control, sino también por la agilización de la llamada. Más adelante, en el capítulo 6 se evaluarán en profundidad estas características.

Además, es obvio que, si ese paso LO2 del Algoritmo 4.2.1, que establece la principal diferencia con el Algoritmo 3.2.1, puede evitarse en el caso particular reseñado, entonces tal situación merece un *status* propio a la hora de conseguir la eficiencia máxima. Por ello, podríamos definir una *activación final casi recursiva de procedimiento* como aquella que, no siendo directamente recursiva,

se realiza de un procedimiento con la misma profundidad de anidamiento que el que realiza la invocación. El término "casi recursiva" hace referencia a la gran similitud de este caso con las denominadas activaciones recursivas. La *secuencia de llamada optimada para activaciones finales casi recursivas de procedimientos* sería similar al Algoritmo 4.2.1, salvo por la inexistencia del paso LO2, por lo que no es preciso repetirlo. Esta nueva denominación será muy interesante en el mencionado capítulo 6, al evaluar el comportamiento temporal de las llamadas optimadas, pero en lo que se refiere al resto de este capítulo y al próximo seguiremos denominando activaciones finales "no recursivas", en general, a las que no son directamente recursivas, independientemente del anidamiento.

Una vez definido ya el modelo dinámico optimado hemos de estudiar cómo las demás características del lenguaje de programación PASCAL influyen en el mismo. Este es el objetivo de los apartados a continuación.

4.2.3 Datos locales, parámetros formales, otras activaciones y saltos no locales

El estudio realizado al respecto de estos elementos del lenguaje en la optimación de las llamadas recursivas finales (sección 3.2) puede aplicarse en su totalidad en las llamadas no recursivas. Considerando la misma hipótesis, que la optimación no puede realizarse si se establece una referencia de la clase que sea a alguno de los componentes del registro de activación del procedimiento llamador, se llega a los mismos resultados que en capítulo anterior.

Por un lado, el tema de los datos locales no precisa ningún razonamiento más que el que allí se hizo: al mantener sin modificación la parte de la secuencia de llamada correspondiente al receptor, que incluye la inicialización de las variables locales, el comportamiento con el modelo optimado es el adecuado. Por otro lado, en relación con los parámetros formales y actuales, sí que deben hacerse varios comentarios.

Primeramente hemos de profundizar en la manera en que se evalúan los parámetros actuales y se asocian a los parámetros formales del receptor. En general, ocurre que estos últimos ocuparán parte del espacio del registro de activación del llamador, exactamente sobre el espacio destinado a sus propios parámetros formales y datos locales. Como los tamaños de la zona de datos de

los registros de activación de llamador y receptor pueden ser diferentes, puede darse el caso que los parámetros formales del receptor acaben situados sobre memoria que previamente se había destinado, no solo a los parámetros formales, sino también a los datos locales y temporales del llamador. Evidentemente, esto obliga a poner especial cuidado en la realización del paso LO1 del Algoritmo 4.2.1 de llamada optimada. En cierto modo, lo mismo ocurría en las activaciones finales recursivas optimadas, razón por la cual se estableció la Condición 3.2.9, solo que entonces los nuevos parámetros ocupaban exactamente el espacio de los antiguos y no había ningún problema con respecto a los datos locales. En definitiva, los requisitos establecidos en aquella condición han de hacerse extensivos a todos los datos locales del procedimiento llamador:

Condición 4.2.9. La evaluación de los parámetros actuales y su asignación a los parámetros formales debe ser de tal modo que no se produzca la lectura de algún parámetro formal o dato local del procedimiento llamador después de haber asignado un parámetro formal del receptor que ocupe parte de la zona de memoria que estaba destinada al parámetro formal o dato local del llamador.

La condición expresa al fin y al cabo el hecho de que ha de preservarse toda la información necesaria de los objetos del registro de activación del procedimiento llamador antes de asignar los parámetros actuales a sus localidades definitivas, lo mismo que hacen las condiciones anteriores respecto a los elementos de control. La estrategia concreta que se elija durante el diseño del compilador puede, sin embargo, variar. Podrían, por ejemplo, calcularse los parámetros actuales sobre registros y luego llevarlos a memoria, o también podría hacerse sobre memoria temporal; ahora bien, en este último caso ha de asegurarse que esa memoria temporal se encuentra estrictamente por encima de la zona que ocuparán los parámetros formales del receptor.

Si se elige otra estructura para el registro de activación, pueden surgir dificultades considerables a este respecto. Por ejemplo, con la estructura de Aho, Sethi y Ullman [1990], caracterizada por tener la información de control por encima de los parámetros formales, el problema estudiado se extiende también a esa información, por lo que será necesario preservarla de igual modo. Es probable que ello no impida la optimación, pero a buen seguro la hará más

costosa y falta de elegancia en comparación con la solución alcanzada por nosotros. En cualquier caso, lo estudiaremos más adelante, en el capítulo 6.

Aun tenemos que asegurarnos que el principio de preservación de la información del procedimiento llamador se cumple en toda su extensión. Si suponemos que es así durante el paso LO1 de la secuencia de llamada optimada (Algoritmo 4.2.1), parecería que el resto de pasos lo cumplen de modo inmediato: en efecto, se trata del cálculo del enlace de acceso y del salto al procedimiento receptor; como la zona de control no se ve afectada por el paso LO1, dada la estructura del registro de activación, el enlace de acceso puede calcularse exactamente como en cualquier otra llamada.

Sin embargo, puede dejar de cumplirse el principio, y ello debido a un aspecto que, si bien fácil de entender, resultó de muy difícil detección en un primer momento. Imaginemos que el procedimiento activado en posición final es un parámetro formal procedimental del procedimiento que hace la llamada; sabemos que este caso no viola la Restricción 4.2.2, por lo que la optimación es posible; también sabemos que la implementación habitual consiste en almacenar como parámetro procedimental la pareja formada por el enlace de acceso calculado para el procedimiento "real" y el punto de entrada a su código. Entonces, la activación de un parámetro formal procedimental implica colocar como enlace de acceso del registro de activación entrante el valor del enlace de acceso guardado con el parámetro formal procedimental, y saltar a la dirección que también está guardada con dicho parámetro.

Ahora, tengamos en cuenta que el parámetro formal procedimental es local al procedimiento que efectúa la llamada (si no es local, no existirá problema, como veremos). ¿Dónde se sitúa la memoria de dicho parámetro? Claramente, junto con el resto de parámetros formales del procedimiento llamador. ¿Qué ocurre con esa zona después del paso LO1 de la secuencia de llamada optimada propuesta? Pues que se habrá perdido, ya que estará ocupada por los parámetros actuales (formales del receptor), salvo casualidades motivadas por un menor tamaño de los formales del receptor que de los formales del llamador.

Desde luego, la solución a este problema consiste en llevar la información del parámetro formal procedimental a un lugar seguro (memoria temporal o registro) antes de comenzar el paso LO1 de la secuencia de llamada

optimada, lo cual expresamos formalmente en la siguiente condición de la optimación:

Condición 4.2.10. Si el procedimiento que se activa en posición final es un parámetro formal procedimental del procedimiento que efectúa la llamada, entonces la información de dicho parámetro formal ha de mantenerse hasta realizar el cálculo del enlace de acceso y el salto al procedimiento receptor.

Realmente, no hay casi ninguna dificultad en esto; lo difícil fue darse cuenta del problema. Tenemos entonces que incluir este aspecto en la secuencia de llamada optimada para las activaciones finales no recursivas de procedimientos.

- LO1: si el procedimiento receptor es un parámetro formal procedimental del procedimiento que efectúa la llamada, el llamador coloca en lugar seguro el enlace de acceso y la dirección de entrada asociados a dicho parámetro formal procedimental; después, evalúa los parámetros actuales, y coloca los valores obtenidos en el lugar correspondiente de su propio registro de activación, mediante desplazamientos respecto a *BasePila*;
- LO2: el llamador calcula el nuevo enlace de acceso, si se usa la cadena estática, y lo coloca en el lugar correspondiente de su propio registro de activación; si se usa el *display*, restablece el registro de *display* que le corresponde, según se haría normalmente en su propia secuencia de retorno, coloca en el lugar correspondiente de su propio registro de activación el registro de *display* a guardar para la profundidad de anidamiento del receptor, y actualiza dicho registro;
- LO3: el llamador salta (*jump*) a la primera instrucción del código del receptor;
- LO4: el receptor actualiza *TopePila* a su nuevo valor, que es *BasePila* más el tamaño de la zona de datos;
- LO5: el receptor asigna valores iniciales a sus datos locales.

Algoritmo 4.2.2 Secuencia de llamada optimada para una activación final no recursiva de procedimiento (incluyendo el caso en que se invoca un parámetro formal procedimental)

El segundo aspecto relacionado con los parámetros formales se refiere a las restricciones adicionales encontradas respecto a los parámetros por referencia y los parámetros procedimentales o funcionales. Tales restricciones siguen siendo válidas, y las volvemos a repetir aquí:

Restricción 4.2.3. Si la llamada no recursiva final de un procedimiento tiene como parámetro actual por referencia una variable, que no sea variable identificada, correspondiente a un dato local o un parámetro formal por valor del procedimiento que hace la llamada, entonces no es posible realizar la optimización.

Restricción 4.2.4. Si la llamada no recursiva final de un procedimiento tiene como parámetro actual procedimental (o funcional) un procedimiento (o función) local al procedimiento que hace la llamada, entonces no es posible realizar la optimización.

En ambos casos, se establecen referencias a ciertos componentes del registro de activación del procedimiento llamador (en los parámetros por referencia a un dato, en los parámetros procedimentales o funcionales, al registro de activación completo a través del enlace de acceso), que impiden la desaparición de dicho registro de activación.

Respecto a otras activaciones que puedan darse a partir del procedimiento llamado en posición final, parecería que alguna podría entrar en relación con el procedimiento llamador; sin embargo, eso no es posible, debido a la Restricción 4.2.2. En efecto, el único vínculo que se podría establecer sería a través de un enlace de acceso, pero dicha restricción asegura que el enlace de acceso para el procedimiento llamado señalará estrictamente más abajo que al registro de activación de su llamador, así que no habrá manera de que otras activaciones se refieran a esa activación.

Con los saltos no locales ocurre otro tanto, pero es necesario un argumento un poco más elaborado. Recordemos del apartado 3.2.8 que los saltos no locales exigen la eliminación de la pila de control de los registros de activación comprendidos entre aquel donde se encuentra la sentencia de salto (el de la cumbre) y aquel que contiene la etiqueta. Para ello, se sigue la cadena estática tantas veces como distancia léxica exista entre ambos procedimientos. En las activaciones finales recursivas no encontramos problema, pues la presencia o no presencia de registros de activación del mismo procedimiento no

afectaba al recorrido de la cadena estática. Sin embargo, este razonamiento no sirve para el presente caso, pues el enlace de acceso del procedimiento receptor puede cambiar. Ahora bien, uno de los hechos que garantiza la Restricción 4.2.2 es que la cadena estática se mantendrá en el modelo optimado exactamente igual que se mantendría en el modelo estándar, por lo que la ejecución de un salto no local desde el procedimiento cuyo registro de activación ha quedado en la pila se realizará correctamente, esté o no esté el registro de activación del procedimiento llamador.

En definitiva, concluimos que no existen otras restricciones para la optimación de las activaciones no recursivas finales (salvo la Restricción 4.2.2) que las que se descubrieron para las llamadas recursivas en el capítulo anterior.

4.3 CONSIDERACIONES ADICIONALES

Uno de los aspectos que precisa un comentario es el de la optimación del proceso de actualización de los parámetros formales, tal como se obtuvo en las activaciones recursivas y se reflejó formalmente en la Condición 3.2.10, la Condición 3.2.11 y la Condición 3.2.12. Recordemos que consistía en no realizar la actualización si el parámetro actual era exactamente el propio parámetro formal de la posición correspondiente. A primera vista, parece que no va a ser posible aplicar ninguna optimación de similares características en las activaciones no recursivas, pues la disposición de los parámetros en los registros de activación del llamador y del receptor serán en general diferentes. Pero consideremos un ejemplo típico de recursividad mutua, el de los predicados de paridad, implementados mediante procedimientos en el Programa 4.3.1.

Antes de todo, hay que decir que este puede ser el ejemplo paradigmático de programa con un comportamiento en espacio constante con el modelo dinámico optimado, y que será objeto de otra discusión dentro de un momento. En lo que al problema planteado respecta, es obvio que en las optimaciones de las activaciones finales del programa mencionado no sería preciso actualizar el parámetro por referencia `result`: como ambos procedimientos tienen la misma estructura en su registro de activación, al menos en lo que toca a la zona de parámetros formales, la posición y el

contenido de dicho parámetro son un invariante a lo largo de toda la cadena de llamadas mutuamente recursivas.

```

program Prueba1 ...
  procedure par (x: integer; var result: boolean); forward;

  procedure impar (x: integer; var result: boolean);
  begin
    if x = 0 then result := false
    else par (x-1, result)      ( activación final no recursiva )
  end;

  procedure par;
  begin
    if x = 0 then result := true
    else impar (x-1, result)    ( activación final no recursiva )
  end;
begin
  par (25, resultado)
end.

```

Programa 4.3.1 Dos procedimientos finales mutuamente recursivos en PASCAL

Ahora bien, una generalización de lo anterior, aunque ciertamente posible, requiere, pensamos, una complejidad de análisis en el compilador bastante mayor que la necesaria para realizar el mismo proceso en las activaciones recursivas. En efecto, en este último caso el compilador solo ha de verificar que el nombre del parámetro actual de la *enésima* posición coincide con el nombre del parámetro formal de la misma posición. Sin embargo, en las activaciones no recursivas, los nombres no importan, como se comprueba simplemente renombrando el parámetro *result* del procedimiento *impar* del Programa 4.3.1 de otra manera.

Por lo tanto, habría que verificar tres aspectos: primero, que el parámetro actual sea exclusivamente un parámetro formal del procedimiento llamador (realmente, lo mismo que en las activaciones recursivas); segundo, que el modo de transferencia y el tipo del parámetro actual sean los mismos que en el procedimiento receptor; y tercero, que el desplazamiento relativo de la memoria asociada al parámetro actual en el registro de activación del llamador sea el mismo que el desplazamiento relativo del parámetro formal en el registro de activación del receptor. Como vemos, la verificación de la posible optimización de la actualización de los parámetros es técnicamente factible, pero

requiere bastantes más comprobaciones estáticas, por lo que debería estudiarse más en profundidad la conveniencia de incluirla en el optimador.

El otro asunto que merece cierta consideración es la posibilidad de una optimación de similares características a la considerada, pero basada en una transformación del programa fuente. En el estudio de la optimación de las activaciones finales recursivas se concluyó que existían diversos motivos que impedían la aplicación de tal estrategia para obtener los mismos resultados: la obligación en PASCAL de dar valores iniciales especiales a los datos locales era uno de ellos, pero más importante era la imposibilidad de expresar en el propio lenguaje PASCAL el proceso de actualización de los parámetros por referencia, procedimentales y funcionales. Evidentemente, idénticos motivos siguen siendo válidos en la optimación de las activaciones finales no recursivas, pero aún deberemos añadir otro, para afianzar más, si cabe, la idea mantenida desde el comienzo del trabajo.

El problema radica ahora en cómo expresar el propio salto al procedimiento receptor como sustitución de la activación de este. Si tomamos el Programa 4.3.1, está claro que no es posible colocar etiquetas en cada procedimiento y sentencias `goto` de uno a otro, pues no están permitidos esta clase de saltos no locales en PASCAL: siempre han de darse desde un subprograma a otro que lo incluya con una profundidad de anidamiento estrictamente menor.

¿Es posible, al menos, construir algún programa en PASCAL en que se pueda expresar la optimación de una activación final no recursiva mediante una sentencia `goto` válida? No se ha encontrado ningún ejemplo en la literatura acerca de este extremo (la eliminación de recursividad final mutua no se considera), a pesar de que, según Bird [1977a], debe ser posible, tal como vimos en el capítulo 2. Volvemos a repetir aquí sus palabras:

«If the last action of a procedure *p* before it returns is to call procedure *q*, then simply `goto` the beginning of procedure *q* instead.

»This useful rule, which is easy to justify, is discussed in Knuth [5]; it also works when $q = p$ [...]. When parameters are present they must be reassigned just before the jump is made» [Bird 1977a]

En el ejemplo de los predicados de paridad, por lo tanto, el requisito es que uno de los procedimientos mutuamente recursivos sea local al otro. Un primer intento llevaría entonces al Programa 4.3.2.

```

program Prueba2 ...
  procedure par (x: integer; var result: boolean);
    procedure impar (x: integer; var result: boolean);
    begin
      if x = 0 then result := false
      else par (x-1, result)
      end; { impar }
    begin
      if x = 0 then result := true
      else impar (x-1, result)
      end; { par }
    begin
      par (25, resultado)
    end.

```

Programa 4.3.2 Dos procedimientos finales, uno local al otro, mutuamente recursivos en PASCAL (modificado a partir del Programa 4.3.1)

Tengamos en cuenta que, según la estrategia de optimación que hemos desarrollado en los apartados anteriores, la activación de par dentro de impar del Programa 4.3.2 es optimable, pero no así la de impar dentro de par, por referirse a un procedimiento local en el llamador. Con ello se conseguiría solo que hubiera la mitad de registros de activación en la pila de control durante el proceso recursivo, pero evidentemente no sería un comportamiento en espacio constante, y sigue existiendo potencialmente el peligro de desbordamiento de la pila de control. Esto nos hace pensar en un aspecto que tendrá suma importancia sobre el impacto "real" o efectivo de la optimación en el día a día de la programación en PASCAL: es obvio que la metodología de programación en el lenguaje incidirá de manera drástica en la posibilidad o imposibilidad de la optimación. Al fin y al cabo, ocurre exactamente lo mismo en el lenguaje SCHEME, y se han desarrollado diversas técnicas para conseguir la recursividad final de las funciones (*vide* [Abelson, Sussman y Sussman 1985: 36-37; Dybvig 1987: 71-74; Springer y Friedman 1989: 115-127]), que debemos considerar como normas o recomendaciones metodológicas de uso del lenguaje. Por ello, dedicaremos a este asunto una sección más adelante, en el capítulo 7.

Volviendo al problema planteado y al Programa 4.3.2, se trataría de sustituir la llamada a par dentro de impar por una actualización de los parámetros y un salto al código de par. Aquí nos encontramos con un primer problema, pues no es posible nombrar los datos locales del bloque par desde el

bloque *impar*, al estar "sombreados" por los mismos nombres en este último. Así, deberían hacerse visibles los datos del bloque *par* antes de poder llevar a cabo la actualización de los parámetros, lo que implica renombrar los parámetros del procedimiento *impar*. Solo entonces podríamos escribir esa actualización (que no es necesaria para el parámetro *result*, porque de serlo sería imposible expresarlo en PASCAL, como sabemos), y sustituir la llamada a *par* por un salto no local al comienzo de su código, tal como se refleja en el siguiente programa.

```

program Prueba2NoRecursoivo _
...
  procedure par (x: integer; var result: boolean);
  label 1;
    procedure impar (x2: integer; var result2: boolean);
    begin
      if x2 = 0 then result2 := false
      else
        begin
          x := x2-1;
          goto 1
        end
      end; ( impar )
    begin
      1: if x = 0 then result := true
        else impar (x-1, result)
      end; ( par )
    ...
  begin
    ...
    par (25, resultado)
    ...
  end.

```

Programa 4.3.3 Transformación del Programa 4.3.2 en un programa no recursivo equivalente

Es de reseñar que, aunque la consecución de este programa alternativo parece ser similar a lo que se haría en el modelo dinámico optimado que proponemos (al aplicar la optimación solo en una de las llamadas y no en las dos de cada paso del proceso mutuamente recursivo), el efecto es radicalmente distinto, ya que de hecho el nuevo programa tiene un comportamiento en espacio constante. En efecto, al aplicarse el salto no local, desaparece de la pila el registro de activación para el procedimiento *impar*, y se sigue la computación en el contexto del procedimiento *par*, convenientemente actualizado. En el modelo dinámico optimado, sin embargo, esta operación se resuelve con la sustitución del registro de activación de *impar* por un nuevo registro de activación para el procedimiento *par*, manteniéndose el o los

correspondientes a las activaciones precedentes, por lo que el crecimiento neto de la pila de control *no* es nulo.

Este hecho sorprende sobremanera, pues no esperábamos encontrar que una optimización basada en la transformación de programas fuera de mejor calidad que la basada en un modelo dinámico alternativo. De ahí que sea necesario estudiar más profundamente este problema para mantener intactas nuestras posiciones al respecto de tal hipótesis. Por fortuna, ocurre que, siendo el Programa 4.3.3 correcto como transformación del original, no lo es la supuesta norma que conduce a la optimización. Consideremos una variación del Programa 4.3.2, en la que el procedimiento *par* llama al procedimiento *impar* en posición no final (lo cual en principio no invalidaría la solución propuesta).

```

program Prueba3 ...
...
  procedure par (x: integer; var result: boolean);
    procedure impar (x: integer; var result: boolean);
    begin
      if x = 0 then result := false
      else par (x-1, result)
      end; ( impar )
    begin
      if x = 0 then result := true
      else
        begin
          impar (x-1, result);
          writeln ('Pasa por par ', x)
        end
      end; ( par )
    end;
  begin
    ...
    par (7, resultado)
    ...
  end.

```

Programa 4.3.4 Una variación minúscula del Programa 4.3.2

Insistimos en que nada parece impedir aplicar a este programa la misma optimización que condujo al Programa 4.3.3 a partir del Programa 4.3.2. Obtendríamos entonces el Programa 4.3.5 que aparece a continuación, del que debemos decir cuanto antes que *no* es equivalente al original.

En efecto, una mera comprobación de las ejecuciones de ambos programas lo corrobora: el Programa 4.3.4 muestra el mensaje de la sentencia de escritura *writeln* un número de veces que es la mitad del valor con que se activa *par* por primera vez, mientras que el Programa 4.3.5 lo hace una o

ninguna vez (dependiendo de que el primer valor sea impar o par, respectivamente).

```

program Prueba3NoRecursoivo ...
***
  procedure par (x: integer; var result: boolean);
  label 1;
    procedure impar (x2: integer; var result2: boolean);
    begin
      if x2 = 0 then result2 := false
      else
        begin
          x := x2-1;
          goto 1
        end
      end; ( impar )
    begin
      1: if x = 0 then result := true
      else
        begin
          impar (x-1, result);
          writeln ('Pasa por par ', x)
        end
      end; ( par )
    begin
      ***
      par (7, resultado)
      ***
    end.

```

Programa 4.3.5 Transformación del Programa 4.3.4 en un programa no recursivo ¡pero no equivalente!

La razón es que con el salto no local en el procedimiento *impar* no solo estamos evitando la secuencia de retorno de dicho procedimiento (lo cual no está mal, al sustituir a una activación final), sino también todas las sentencias que hubiera en el llamador a continuación del punto en que se activó el otro procedimiento.

En definitiva, concluimos que la optimación, de seguir la estrategia de transformación del programa fuente, no solo dependería de que la activación de un procedimiento fuera final, sino también de que a su vez la llamada al procedimiento que contiene dicha activación final estuviera en posición final. Evidentemente, la complejidad de análisis requerida en el transformador para procesar tales situaciones es enorme, haciendo prácticamente imposible tal tarea. Por contra, en el modelo dinámico optimado que se propone como alternativa al problema, la activación final de *par* en el procedimiento *impar* sigue siendo optimable, a pesar de que no se consiga el comportamiento en espacio constante. Y, aunque ya hemos dicho que habremos de examinar en

toda su extensión el impacto del optimador dentro del compilador, es de esperar que la complejidad necesaria para detectar las activaciones optimables sea menor, al depender exclusivamente de la información del bloque que se está compilando, y no de otros bloques que pueden situarse en puntos lejanos al actual.

La discusión anterior, añadida a los problemas encontrados acerca de la inicialización de los datos locales y la actualización de los parámetros por referencia, procedimentales y funcionales, no hace más que afianzar nuestra postura acerca de la imposibilidad de aplicar una técnica transformacional para conseguir la optimación que puede alcanzarse mediante una modificación del modelo dinámico asociado al lenguaje.

Para finalizar, como en el capítulo precedente, hemos de comparar los resultados obtenidos mediante nuestra estrategia de optimación con los que se aplican en SCHEME en general, y con la estrategia de Hanson [1990], recogida en el apartado 2.4.2, en particular. Como concluimos en la sección 3.3, la Restricción 4.2.3 y la Restricción 4.2.4 surgen por las propias características de los elementos a que hacen referencia, e incluso se puede decir que la segunda existe igualmente en el lenguaje SCHEME, si lo que desea es conseguir un comportamiento iterativo puro. Lo que sí es un impedimento directamente atribuible a nuestra solución es, evidentemente, la otra restricción para la optimación de las llamadas finales no recursivas: la Restricción 4.2.2, que impone la necesidad de que no sea activado un procedimiento local al que invoca para poder aplicar el mecanismo alternativo de llamada.

La solución de Hanson, y cualquier implementación de SCHEME en general, no adolecen de este problema. Recordemos que la estrategia de Hanson consiste, en principio, en permitir un crecimiento de la pila de activaciones ante las llamadas sujetas a la Restricción 4.2.2. Concretamente, Hanson, por las mismas razones que nosotros, tiene que acumular en la pila un registro de ambiente correspondiente a la llamada: el ambiente del procedimiento llamador debe ser accesible, por lo que no puede desaparecer. Nosotros no podemos distinguir entre registros de ambiente y registros de control, lo que nos obliga a acumular un registro de activación completo. Lo que sí puede hacer Hanson es desapilar un número variable de registros de ambiente en otras situaciones, más exactamente en la situación "contraria", es decir, cuando la profundidad de anidamiento del procedimiento llamado es menor que la del llamador. Con ello,

se consigue en términos absolutos que no exista crecimiento de la memoria necesaria, tal como se mostró en el Programa 2.4.2 y en la Figura 2.4.4.

Por lo tanto, vemos que es la posibilidad de desapilar registros de ambiente lo que proporciona generalidad a la estrategia de Hanson. Sin embargo, ya mencionamos al hacer el estudio de la misma en el capítulo 2 que esta es también una de sus principales desventajas, pues la operación puede llegar a ser compleja: en el caso más general, la determinación del número de registros de ambiente a desapilar solo puede hacerse dinámicamente. Hanson solo puede llegar a procesos eficientes en algunos casos, y ello tras realizar análisis estáticos mucho más costosos que los que precisa nuestra solución.

Las implementaciones más tradicionales de SCHEME no basadas en la vinculación de datos en la pila, tal como se recogieron en el apartado 2.4.1, tampoco adolecen del problema, como se mostró en la Figura 2.4.2, también correspondiente al Programa 2.4.2. La separación entre la información de ambiente y la información de control, esta vez por la vinculación de los ambientes en el montículo, es la que posibilita, como para Hanson, que esto sea así.

En definitiva, concluimos que existe una clase de activaciones en PASCAL que no resulta optimada siguiendo nuestra estrategia, y que no presenta mayor dificultad en el lenguaje SCHEME, sea cual sea la implementación elegida. Se afianza entonces nuestra conclusión previa de que el conjunto de procesos optimables en PASCAL sería un subconjunto de los procesos que en SCHEME resultan en un comportamiento iterativo puro. Sin embargo, nuestra solución puede ser de inmediata aplicación en las implementaciones existentes del lenguaje PASCAL, dadas las características ya estandarizadas de las mismas desde hace años, y lo que es más importante, mantienen (si no mejoran, lo que será objeto de estudio en el capítulo 6) la eficiencia tradicionalmente asociada a las mismas. La utilización de la estrategia de Hanson no solo supondría un alejamiento de la implementación clásica de PASCAL poco atractivo para los diseñadores de compiladores ya existentes, sino que introduciría elementos de ineficiencia que podrían llevar a desestimar la conveniencia de considerarlo como una optimación efectiva. Es evidente que habrá de estudiarse con detenimiento el impacto de la Restricción 4.2.2 en la consecución de un comportamiento iterativo para procesos recursivos finales, sobre todo aquellos en que existe recursividad indirecta. Ello será en el capítulo 7.

5. OPTIMACIÓN DE ACTIVACIONES FINALES DE FUNCIONES

5.1 INTRODUCCIÓN

Este capítulo se dedica al estudio de la posible optimación de activaciones finales de los subprogramas conocidos en PASCAL como *funciones*, tanto en su modalidad recursiva como en la no recursiva. La idea básica es aplicar la misma estrategia optimadora que se ha desarrollado con los procedimientos. Sin embargo, la misma naturaleza conceptual de las funciones, mas la presencia a nuestro entender de una serie de "imprecisiones" en la concreción del concepto dentro de la definición del lenguaje PASCAL, va a complicar inicialmente el estudio del problema.

En esta sección introductoria se presenta la discusión mencionada, el modelo dinámico estándar de las activaciones de funciones y un primer apunte de las posibles optimaciones, para pasar en las dos secciones siguientes a estudiar estas en profundidad. Concretamente, se seguirá la misma organización que en el estudio de los procedimientos: primero se estudiará el caso particular de las activaciones recursivas, y después el caso más general de activaciones no recursivas.

En la siguiente sección se compararán los resultados con los que se dan en el lenguaje SCHEME. Se finaliza el capítulo con el estudio de otra clase de llamadas a las que podrá practicarse la optimación que, si bien no tan interesante como las de procedimientos o funciones, si permitirán observar hasta que punto es posible extender nuestras técnicas.

5.1.1 El concepto de función en PASCAL

En principio, una *función* en PASCAL es lo mismo que un procedimiento, excepto por el hecho de que la activación de una función produce un valor (de tipo simple o apuntador), que puede ser subsecuentemente utilizado dentro de una expresión. De este modo, una vez dada la sintaxis asociada a los procedimientos en sus diferentes aspectos (declaración y activación), las consecuencias puramente sintácticas de esta caracterización de las funciones son mínimas, pues las diferencias se reducen a:

- la inclusión de un nombre de tipo en la declaración, y
- la inclusión de una sentencia especial, por lo demás idéntica en sintaxis a la sentencia de asignación, para la asignación del valor a devolver por una función,

mientras que los otros aspectos significativos no varían: las funciones pueden tener las mismas clases de parámetros que los procedimientos, definen bloques como ellos, y se activan del mismo modo. Idénticamente, las consecuencias dinámicas son menores, si cabe, pues simplemente se refieren al manejo del valor de retorno, lo cual será tema principal de esta discusión en breves instantes.

Sin embargo, existe en PASCAL una filosofía de uso de las funciones que marca una profunda diferencia con los procedimientos, y que contrasta con la caracterización formal de las mismas. Según la propia definición del lenguaje:

«Assignments to non-local variables and other so-called side effects should be avoided within function declarations» [Jensen y Wirth 1991: 147]

En cierto modo, se entiende que las funciones deben usarse para realizar cálculo numérico "puro" y operaciones no numéricas de naturaleza similar. Esto es lógico si las activaciones de funciones van a formar parte de los procesos asociados a las expresiones, en que ha de mantenerse la "transparencia referencial" [Quine 1968: 155-156; MacLennan 1990: 11-12]. De ello se deduce que la definición del lenguaje debería haber hecho una de dos cosas:

- o bien distinguir sintácticamente de una manera más drástica las entidades "procedimiento" y "función", siguiendo las ideas apuntadas, y prohibiendo por ejemplo los parámetros variables para las funciones (véase en este sentido la interesante discusión de Ichbiah *et al.* [1991:

166-167] al respecto del lenguaje ADA, que solo admite parámetros por valor para las funciones),

- o bien no establecer diferencia sintáctica alguna, es decir, conceptualizar la idea de "subprograma" en una sola entidad (capaz de devolver un valor en todos los casos, aunque no sea preciso, tal como hace por ejemplo el lenguaje C [Kernighan y Ritchie 1988: 30]), poniendo entonces el énfasis en la metodología de programación para realizar la distinción entre las dos clases de procesos existentes.

De todos modos, para el constructor de un compilador, la discusión anterior puede parecer baldía, pues no puede preocuparse de las recomendaciones de estilo de programación, sino de las reglas sintácticas exactas del lenguaje, y hemos visto que estas no establecen mayor diferencia entre procedimientos y funciones. Sin embargo, consideramos lo anterior un primer aviso de las cuestiones problemáticas que van a surgir en lo que respecta a la entidad que diferencia a procedimientos y funciones en PASCAL, es decir, el "valor de retorno". De hecho, el propio concepto genérico de "función" como subprograma que produce un valor existe en otros lenguajes, evidentemente, pero los detalles exactos acerca de su manejo pueden variar notablemente respecto a los que se definen en PASCAL, por lo que es fundamental la consideración de dichos detalles (concretamente, SCHEME aplica una filosofía distinta a este respecto, que facilita la optimización de la recursividad final, como veremos más adelante).

Según la definición del lenguaje, la activación de un bloque de la clase función contiene:

"a result that is undefined when the algorithm [specified by the statement part of the block] commences" [Jensen y Wirth 1991: 180]

entendiendo la palabra "indefinido" de la misma manera que en relación con las variables (cf. apartado 3.2.3). Este "resultado" o valor de retorno debe haber sido definido cuando la activación de la función finalice:

"Upon completion of the [function] activation's algorithm, [...] it is an error if the result is undefined" [Jensen y Wirth 1991: 167]

La manera de dar definición al valor de retorno es, como hemos dicho, usar la sentencia de asignación con el identificador de la función en la parte izquierda. Se dice explícitamente:

«The block of a function declaration must contain at least one assignment to the function identifier» [Jensen y Wirth 1991: 183]

Por supuesto, esta condición es necesaria para garantizar que el valor de retorno de la función está definido al término de esta, pero no es suficiente, como se puede mostrar con funciones casi triviales. No hay manera de determinar estáticamente que una función devolverá un valor definido en todos los casos, máxime cuando la sentencia de asignación puede incluso no estar presente en la parte de sentencias de la propia función, sino aparecer en un bloque anidado en ella (ya que se dice en la norma anteriormente citada que es el *bloque* de la función el que debe contener la asignación al valor de retorno, no la parte de sentencias). Por lo tanto, debe realizarse esta labor dinámicamente.

Pensamos que existe aún mayor confusión en lo relativo a esta sentencia de asignación especial para el valor de retorno. Concretamente, se explica que:

«Within the function declaration there must be an executed assignment [...] to the function identifier to "return n" [*sic*] the result of the function» [Jensen y Wirth 1991: 123]

Esta frase parece significar que la sentencia por la que se da un valor de retorno a una función tiene también el efecto de terminar su activación. Ello querría decir que cualquier sentencia que textualmente estuviera a continuación de esa sentencia de asignación no sería efectivamente ejecutada, pero no hemos podido encontrar ninguna mención de este hecho a lo largo de la definición estándar del lenguaje PASCAL. Bajo otra perspectiva, parece que se está suponiendo que lo lógico es que la sentencia de asignación al identificador de la función aparezca exactamente al final de la parte de sentencias de la función, y de hecho casi todos los ejemplos de funciones del texto de referencia del lenguaje [Jensen y Wirth 1991: 123, 184] aplican esta idea. Esto está de acuerdo con la filosofía de uso de las funciones en la programación en PASCAL (pues en el momento en que se dispone del valor a devolver no se necesitan más operaciones si las funciones están libres de efectos laterales), pero evidentemente no tiene obligatoriamente que ser así. Explícitamente, además, se dice:

«The assignment statement serves to access the variable or function-activation result and to replace its current value by the value obtained by evaluating the expression» [Jensen y Wirth 1991: 170]

y no se menciona que esta sentencia de asignación provoque la terminación de la activación de la función (mientras que sí se indica el modo en que una activación puede terminar debido a otra causa que no sea la finalización de su parte de sentencias, concretamente debido a un salto no local [Jensen y Wirth 1991: 179]).

En definitiva, a pesar de la imprecisión inicial, entendemos que existen dos procesos a llevar a cabo durante la activación de una función: asignación de un valor a devolver, y terminación de la activación; y entendemos que estos procesos son independientes, uno llevado a efecto por la sentencia de asignación sobre el identificador de la función, y el otro por la propia terminación de la parte de sentencias; finalmente, la independencia de estos procesos implica la libre colocación de la sentencia de asignación al valor de retorno, incluso seguida de otras sentencias que puedan llegar a producir efectos laterales. Y si esto es así, se comprende más aún la imposibilidad de una verificación estática completa de que el valor de retorno de una función está definido cuando finaliza cualquier activación de la misma.

La comprobación dinámica del valor de retorno podría llevarse a cabo, en principio, de dos modos alternativos. Por un lado, podría hacerse inmediatamente después de terminar la activación de la función, y sería parte de la secuencia de retorno que ejecutaría el subprograma activador de la función; esta táctica presenta el inconveniente de producir un trozo de código cada vez que se activa la función, lo cual puede darse en numerosas ocasiones. Por otro lado, realmente no hay ninguna dificultad en que sea la propia función la que compruebe su valor de retorno: cuando termina su parte de sentencias, se genera el código para devolver el control, y antes puede perfectamente llevar a cabo la comprobación. Está claro que esta alternativa es preferible a la anterior, pues el código se genera una sola vez para una función dada. En definitiva, consideraremos que la comprobación del valor de retorno forma parte de la secuencia de retorno ejecutada por la función activada.

Finalizamos esta discusión mencionando que, lo mismo que ocurre con las normas relativas a la definición de las variables (*cf.* apartado 3.2.3), diversas implementaciones (por ejemplo, Borland [1992a] o Digital Equipment Corporation [1987]) se desvían de la definición estándar del lenguaje PASCAL, no señalando un error de ejecución si el valor de retorno de una función no está definido.

5.1.2 Modelo dinámico estándar de las funciones en PASCAL

Habiendo establecido las ideas anteriores, pasamos a describir la implementación habitual de las activaciones de funciones en PASCAL. Una activación de función provoca la acumulación en la pila de control de un registro de activación con una estructura prácticamente idéntica a la de los registros de activación correspondientes a los procedimientos (cf. Figura 2.5.1). De hecho, la única diferencia es que existe un espacio adicional para el valor de retorno.

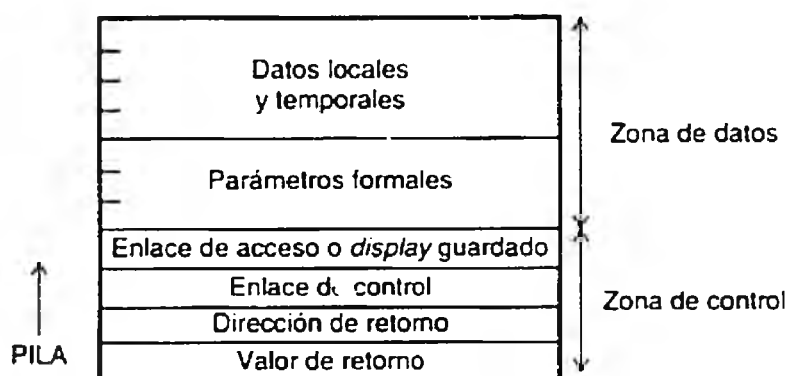


Figura 5.1.1 Estructura del registro de activación de una función en PASCAL

Incluso, puede que el registro de activación de un procedimiento se forme exactamente como el de una función, incluyendo el espacio para el valor de retorno (aunque no se use); de este modo, los accesos a los diferentes elementos del registro de activación se hacen independientes de la clase de subprograma, enfatizando más si cabe lo mínimo de la diferencia dinámica existente entre procedimientos y funciones, tal como se ha discutido al principio de la sección. Nosotros, sin embargo, mantendremos la distinción entre una y otra clase de registro de activación.

Los registros globales *BasePila* y *TopePila* siguen marcando la posición del registro de activación de la cumbre de la pila (como en la Figura 2.5.2); entonces, si corresponde a una función, un desplazamiento relativo a partir de *BasePila* permite el acceso a la localidad del valor de retorno de esa función, o también de cualquier otra que esté activada y sea visible, a través del seguimiento de la cadena estática.

El espacio para el valor de retorno resulta necesario en PASCAL, justamente a causa de lo establecido en el apartado precedente: la independencia entre el proceso de asignación del valor de retorno y el proceso de terminación de la activación. A este respecto, hay que decir que esta concepción de "función" en PASCAL se desvía de lo que ocurre en casi todos los demás lenguajes, como C, MODULA, ADA (en que los dos procesos reseñados son indivisibles, y se dan a través de la sentencia `return`), y particularmente, SCHEME, por lo que se trata de un nuevo aspecto en que la aplicación de la implementación de este último lenguaje a PASCAL no es trivial. Aún más, la filosofía de esos otros lenguajes al respecto se puede ver como un caso particular de la que posee PASCAL, por lo que este estudio adquiere un interés adicional.

El espacio para el valor de retorno tiene un tamaño fijo para todas las funciones, pues el tipo de retorno sólo puede ser ordinal, real o apuntador [Jensen y Wirth 1991: 122]; entonces, se hace que el tamaño del espacio para el valor de retorno sea el máximo de los tamaños de los tipos mencionados¹. La situación del valor de retorno en el registro de activación también es significativa: normalmente, como primera tarea de la secuencia de retorno de una activación de función, el subprograma que invocó a la función deberá tomar el valor de retorno para realizar con él las operaciones que se precisen (generalmente, se almacenará temporalmente en el propio registro de activación del subprograma llamador); entonces, teniéndolo inmediatamente encima de su registro de activación, aquel puede localizar ese valor de retorno sin ninguna dificultad [Aho, Sethi y Ullman 1990: 421].

Existe la alternativa de asignar un registro para el valor de retorno [op. cit.: 411], pero ello puede no ser viable en PASCAL, debido a la posibilidad de que un número indeterminado de funciones estén simultáneamente activadas, salvo que esa idea se entienda como en TURBO PASCAL [Borland 1992a: 255-256]: existe un espacio local para el valor de retorno en el registro de activación de la función, y solamente en el momento del retorno se copia sobre un registro predeterminado (siempre el mismo), de donde se supone que lo tomará el

¹ A pesar de que la restricción en el tipo de retorno de las funciones es coherente con el uso pretendido de las mismas, literalmente «for the use in the evaluation of an expression» [Jensen y Wirth 1991: 122], este es a nuestro parecer el típico caso en que no se sabe muy bien si la norma del lenguaje lleva a una determinada implementación o es más bien al contrario, que la restricción en la implementación fuerza una restricción en el lenguaje.

llamador. De todos modos, esta alternativa no afecta en nada a nuestro estudio, de momento.

La activación de una función se realiza a través de la secuencia de llamada que aparece a continuación.

- L1: el llamador evalúa los parámetros actuales, y coloca los valores obtenidos por encima de su registro de activación (esto es, la dirección *TopePila* más el tamaño del nuevo registro de activación);
- L2: el llamador calcula el enlace de control (simplemente es *BasePila*) y lo coloca por encima de su registro de activación (de nuevo, usando *TopePila*);
- L3: el llamador calcula el enlace de acceso (que es *BasePila* o se obtiene siguiendo la cadena de enlaces de acceso hacia la base de la pila) o toma el registro de *display* correspondiente a la profundidad de anidamiento del subprograma receptor, dejando en cualquier caso el valor por encima de su registro de activación, mediante *TopePila*;
- L4: el llamador coloca la dirección de retorno (dos o tres instrucciones más que la actual, según la siguiente operación) por encima de su registro de activación, también usando *TopePila*;
- L5: el llamador actualiza *BasePila* a su nuevo valor, que es *TopePila* más el tamaño de la zona de control; si se usa el *display*, además se hace que el registro de *display* correspondiente a la profundidad de anidamiento del subprograma receptor tenga el valor dado a *BasePila*;
- L6: el llamador salta (jump) a la primera instrucción del código del receptor;
- L7: el receptor actualiza *TopePila* a su nuevo valor, que es *BasePila* más el tamaño de la zona de datos;
- L8: el receptor asigna valores iniciales a sus datos locales y al valor de retorno.

Algoritmo 5.1.1 Secuencia de llamada estandar para activaciones de funciones en PASCAL

De nuevo, la similitud con los procedimientos hace que esta secuencia de llamada solo se diferencie de la secuencia de llamada estándar de procedimientos (Algoritmo 2.5.1) en que la función que recibe la llamada debe dar un valor indefinido al valor de retorno durante el paso L8; el resto de operaciones permanece sin modificación. Con "el receptor" nos referiremos aquí, para simplificar denominaciones, a la función que es invocada, y "el llamador" denota el subprograma (indistintamente procedimiento o función) que realiza la invocación.

En cuanto a la secuencia de retorno, la discusión previa ha mostrado que la función receptora ha de verificar que el valor de retorno ha sido definido, lo que se hará como primer paso de la secuencia de retorno; por lo demás, no hay tampoco mayor diferencia con los procedimientos (Algoritmo 2.5.2).

- R1: el receptor verifica que el valor de retorno esté definido, señalando un error en caso contrario;
- R2: el receptor, si es el caso, restablece la situación del *display* previa a la llamada, almacenando en el registro de *display* correspondiente a su nivel de anidamiento el valor guardado en su registro de activación; si se utiliza el enlace de acceso, no debe hacerse ninguna operación;
- R3: el receptor recupera el valor anterior de *TopePila*, que es *BasePila* menos el tamaño de la zona de control;
- R4: el receptor recupera el valor anterior de *BasePila*, que es el enlace de control de su registro de activación; para ello, usa el valor de *TopePila*;
- R5: el receptor salta (*jump*) a la instrucción de retorno del llamador, que se obtiene también a partir del registro *TopePila*;
- R6: el llamador toma el valor de retorno que está inmediatamente encima de su registro de activación.

Algoritmo 5.1.2 Secuencia de retorno estándar para activaciones de funciones en PASCAL

Incluimos como paso final adicional el hecho de que el llamador no puede retrasar la captura del valor de retorno, para evitar su pérdida; generalmente, esto obligará a su almacenamiento temporal, como se ha dicho antes.

5.1.3 Posibles optimaciones

Definidas las secuencias de llamada y retorno del modelo estándar de activación de funciones, pasamos a estudiar la optimación de las mismas si se trata de activaciones finales. A primera vista, parecería que la operación R6 del Algoritmo 5.1.2 de retorno invalidaría la posibilidad de aplicar una optimación similar a la conseguida para los procedimientos. En efecto, una de las características que permitía la optimación era el hecho de que el procedimiento llamador no precisaba realizar ninguna operación tras el retorno del procedimiento receptor. Sin embargo, ahora el subprograma llamador debe tomar el valor de retorno de la función receptora de la llamada, y parece que esta operación no puede eliminarse, con lo que no podría darse el proceso optimado. Sin embargo, veremos que esta impresión es errónea.

Comenzamos estudiando un ejemplo (Programa 5.1.1) en el que aparecen una activación de función recursiva final y una activación de función no recursiva final, basado en un programa para verificar si un número es palíndromo tomado de Jensen y Wirth [1991: 244].

Maticemos primero, a la vista de este programa, lo que vamos a entender por activación final de una función. En principio, debe tratarse de una llamada a función que aparezca en una sentencia en posición final, donde este término lo entendemos de la misma manera que en el apartado 3.2.1. Ahora bien, existe una dificultad añadida, y es que una llamada a función es una entidad de "parte derecha", es decir, formará parte de una expresión; esto significa que el valor de retorno se utilizará para realizar alguna otra operación, con lo que en general se perderá el sentido de activación final. Así por ejemplo, si *x* fuera una variable, y se tuviera la sentencia:

```
x := CheckSymmetry(1, Size)
```

aunque estuviera en posición final, la activación de la función no podría considerarse como activación final, pues debe realizarse la asignación a la variable *x* después de retornar la función. Esto elimina de una posible optimación todas las activaciones de funciones desde procedimientos, ya que siempre estarán asociadas a una sentencia en que han de darse otras operaciones después del retorno de la función (bien una asignación, bien una llamada a otro subprograma).

```

program IsItAPalindrome --
--
  function Palindrome --
  const
    Places = 5;
  type
    NPlaces = 1..Places;
    SingleDigit = 0..9;
    DigitVec = array [NPlaces] of SingleDigit;
  var
    Digits: DigitVec;
    Size: NPlaces;

    function CheckSimmetry (Left, Right: NPlaces): Boolean;
    begin
      if Left >= Right then CheckSimmetry := true
      else
        if Digits[Left] = Digits[Right] then
          CheckSimmetry := CheckSimmetry(Left+1, Right-1)
                               { activación recursiva final }
        else CheckSimmetry := false
      end { CheckSimmetry };

  begin
    Palindrome := CheckSimmetry(1, Size)
                               { activación no recursiva final }
  end { Palindrome };
--
begin
--
end.

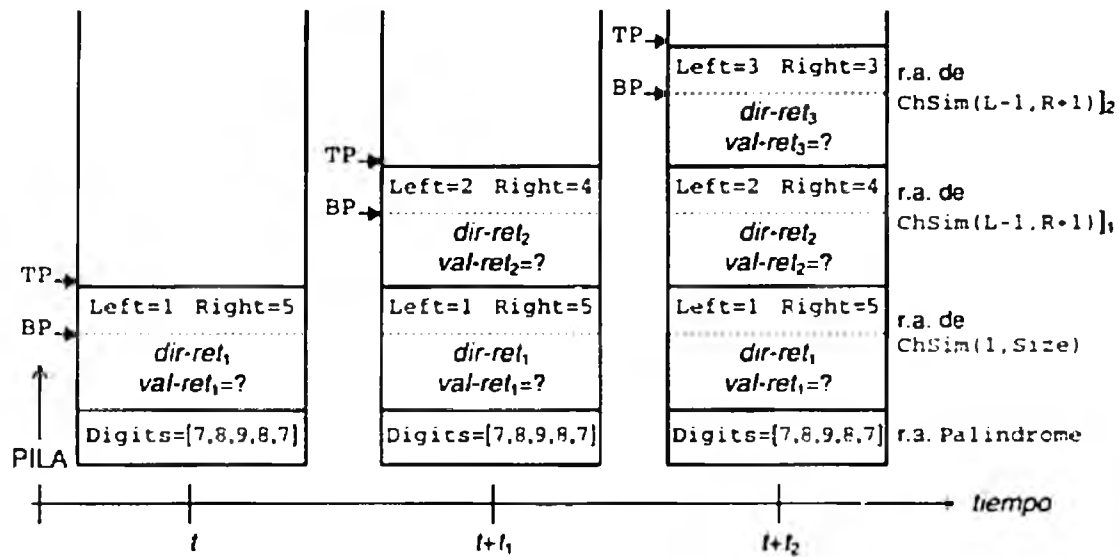
```

Programa 5.1.1 Un programa con una función recursiva final y una función no recursiva final en PASCAL (tomado de Jensen y Wirth [1991: 244])

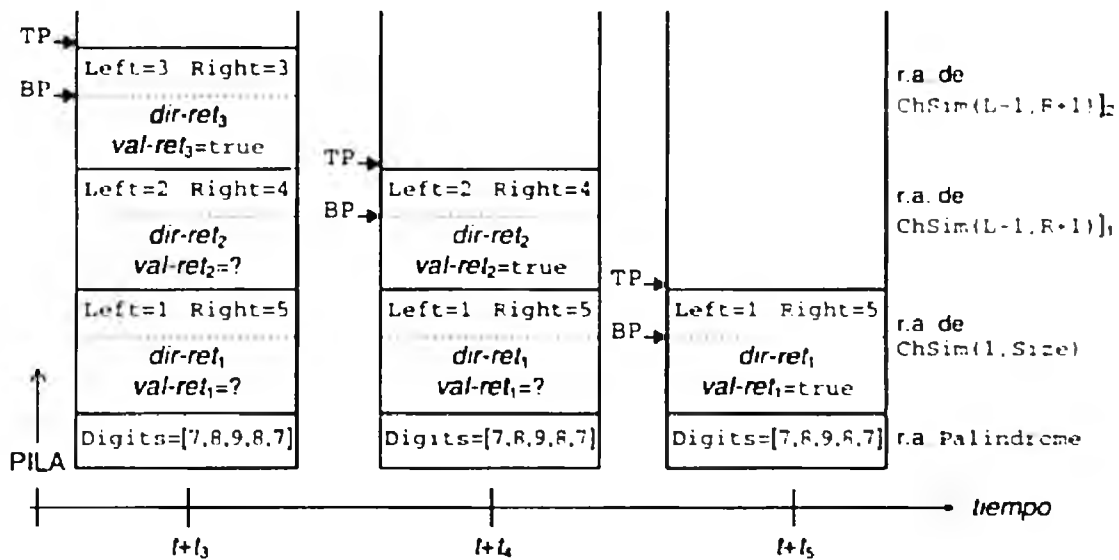
La misma idea parece poder aplicarse a las funciones, y realmente ocurre que la aparición de una activación de función en una sentencia de asignación o formando parte de un parámetro actual no es optimable, por las mismas razones, aunque la sentencia esté en posición final. Sin embargo, la sentencia especial por la que se asigna el valor de retorno de una función también ha de considerarse, y va a ser la que posibilite la optimación. Estudiemos el comportamiento del Programa 5.1.1 ante una activación de la función recursiva *CheckSimmetry*, siguiendo el modelo dinámico estándar. Suponemos que *Digits* es el vector {7, 8, 9, 8, 7}, que *Size* es 5, y que la llamada es:

CheckSimmetry(1, *Size*)

La siguiente figura muestra la situación de la pila de control a lo largo de las diferentes activaciones.



(a)



(b)

Figura 5.1.2 Instantáneas sucesivas de la pila de control durante la ejecución del Programa 5.1.1 (a) en las llamadas, y (b) en los retornos

Puede verse que todas las operaciones de retorno posteriores a la primera son, en cierto modo, redundantes. En efecto, al ser la activación recursiva:

```
CheckSymmetry := CheckSymmetry(Left+1, Right-1)
```

una vez que se ha llegado al caso no recursivo, cada activación recursiva previa solo ha de tomar el valor de retorno de la activación siguiente y colocarlo como su propio valor de retorno, y entonces retornar a su vez. No se hace uso en ningún momento de ninguno de los datos contenidos en el registro de activación, ni se aplica ninguna operación adicional al valor de retorno de la activación siguiente: simplemente se copia a otra localidad. Está claro que la activación del caso no recursivo puede producir el valor de retorno definitivo de toda la secuencia de llamadas recursivas, por lo que los registros de activación de las otras llamadas no son necesarios.

En definitiva, como en el caso de los procedimientos, podemos dar un programa no recursivo alternativo para mostrar la optimización.

```

program IsItAPalindromeNoRec ...
  function Palindrome ...
    function CheckSymmetry (Left, Right: NPlaces): Boolean;
    label 1;
    begin
      1: if Left >= Right then CheckSymmetry := true
        else
          if Digits[Left] = Digits[Right] then
            begin
              Left := Left+1;
              Right := Right-1;    { actualización de parámetros }
              goto 1              { salto al comienzo }
            end
          else CheckSymmetry := false
        end { CheckSymmetry };
    begin
      Palindrome := CheckSymmetry(1, Size)
    end { Palindrome };
  begin
  end.

```

Programa 5.1.2 Función no recursiva equivalente a la del Programa 5.1.1

La llamada recursiva se ha sustituido por una actualización de los parámetros y un salto al comienzo del cuerpo de la función. De esta manera, el crecimiento neto de la pila de control es nulo, como se aprecia en la Figura 5.1.3.

El modelo dinámico optimado que se pretende obtener debe comportarse exactamente como se refleja en la figura. Ello puede conseguirse si ante una activación como la señalada:

CheckSymmetry := CheckSymmetry(Left+1, Right-1)

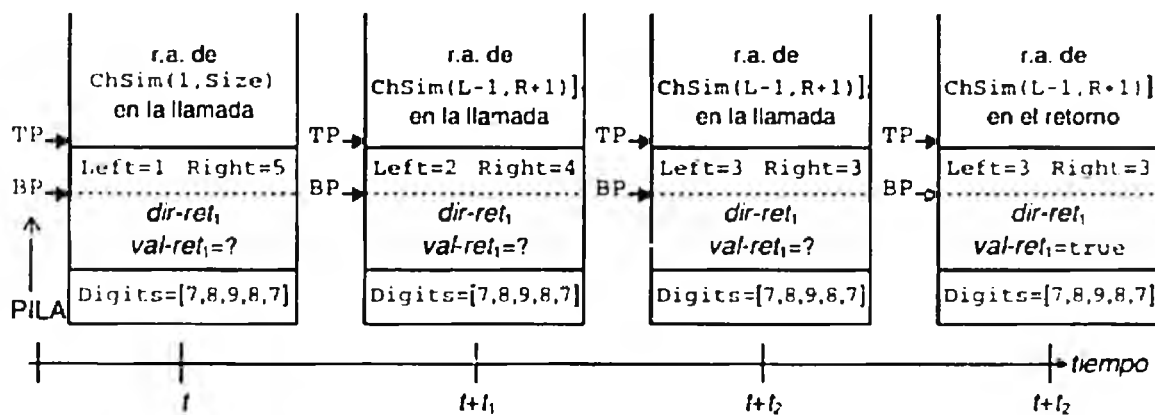


Figura 5.1.3 Instantáneas sucesivas de la pila de control durante la ejecución del Programa 5.1.2

en lugar de acumular un nuevo registro de activación en la pila, se mantiene el actual, actualizando únicamente los parámetros formales y saltando a la primera instrucción del código de la función. Es decir, siguiendo prácticamente la misma estrategia que en la optimación de las activaciones recursivas finales de procedimientos parece conseguirse una optimación de similares características para las que ya desde ahora definiremos como *activaciones finales recursivas de funciones*, que son las sentencias de la forma:

$$\langle \text{función} \rangle := \langle \text{función} \rangle (\langle \text{parámetros} \rangle)$$

situadas en posición final en la parte de sentencias del propio bloque $\langle \text{función} \rangle$.

Es importante resaltar que el requisito establecido en el apartado 5.1.1, que indicaba la necesidad de que una función verifique dinámicamente la definición de su valor de retorno antes de devolver el control, queda perfectamente satisfecho en el modelo optimado que resulta en el comportamiento de la Figura 5.1.3. Es más: el estudio de este aspecto nos hará comprender una bondad añadida de la optimación. En efecto, en la cadena de activaciones recursivas de la función *CheckSymmetry* del Programa 5.1.1, siguiendo el modelo dinámico estándar, una vez alcanzado el caso base de la recursividad, cada activación ha de verificar la definición del valor de retorno antes de devolver el control a la activación previa.

En el modelo optimado que se está proponiendo, realmente solo habrá una secuencia de retorno, que será ejecutada por la última activación de la cadena de llamadas recursivas. Es decir, no solamente se están evitando las operaciones habituales de la secuencia de retorno (tal como se efectuaban en los

procedimientos), sino que se está suprimiendo incluso la verificación del valor de retorno, lo que permite ya afirmar que la optimación será más intensa que la correspondiente a los procedimientos.

Por supuesto, la optimación no puede darse si esta sentencia especial de asignación recursiva del valor de retorno contiene en la parte derecha una expresión que conste de algo más que la activación de la función, como en el caso paradigmático de la función para el cálculo del factorial:

$$\text{fact} := \text{fact}(i-1) * i$$

La obligación de aplicar una operación sobre el valor de retorno impide considerar esta clase de activaciones como finales.

La semejanza encontrada en la optimación de las activaciones finales recursivas de funciones con las homónimas de procedimientos nos permite conjeturar que existirán activaciones finales no recursivas de funciones que se podrán optimar del mismo modo. Las consideraciones del estudio anterior nos llevan a entender una *activación final no recursiva de función* como la sentencia de la forma:

$$\langle \text{función1} \rangle := \langle \text{función2} \rangle (\langle \text{parámetros} \rangle)$$

situada en posición final en la parte de sentencias del bloque de $\langle \text{función1} \rangle$. Si se da que $\langle \text{función2} \rangle$ no precisa acceder a ninguno de los datos de la activación de $\langle \text{función1} \rangle$ (lo que se cumple si $\langle \text{función2} \rangle$ no es local en $\langle \text{función1} \rangle$), entonces no es preciso mantener el registro de activación de $\langle \text{función1} \rangle$, en las mismas condiciones que con los procedimientos, con lo que se logra un crecimiento nulo de la pila de control y una mejora temporal de la ejecución. De esta forma, también los procesos de recursividad mutua de funciones y otros similares pueden resultar optimados.

Queda solamente por establecer si existen otras clases de activaciones en posición final que pudieran ser optimadas. En el caso de procedimientos, la aparición de una activación de función en una sentencia en posición final siempre ha de estar asociada al menos a una asignación (ya sea a una variable o a un nombre de función) o dentro de un parámetro actual de activación de procedimiento; por lo tanto, amén de otras cuestiones, nunca podría darse una optimación, por la necesidad de realizar otras operaciones después del retorno de la función.

En el caso de funciones, la aparición de una activación de función en una sentencia en posición final solo permite la optimación en los casos que se han analizado antes, y que se estudiarán con profundidad en las próximas secciones. Pero además, debemos considerar la posibilidad de una activación de procedimiento como sentencia en posición final en el cuerpo de una función.

```

program Prueba ...
  ...
  procedure p ...
  ...
  function f ...
  ...
  begin
    ...
    p
  end ( f );
  ...
begin
  ...
end.

```

Programa 5.1.3 Función con una activación no recursiva final de procedimiento

Este caso tiene, a primera vista, características similares a los estudiados, pero la necesidad de verificar el valor de retorno de la función después de terminar la activación final del procedimiento parece impedir la optimación. En efecto, esta consistiría en evitar la secuencia de retorno de la función, pero esto no es posible, pues durante esta secuencia de retorno ha de comprobarse que el valor de retorno de la función ha sido definido, operación que no tiene equivalente en la secuencia de retorno del procedimiento activado en posición final. La propia diferencia entre las secuencias de retorno de funciones y procedimientos es la que impide la optimación.

Obviamente, esto nos hace pensar que si una implementación no observa la norma del lenguaje relativa a la comprobación del valor de retorno, entonces sí podría efectuarse, en principio, una optimación de esta clase de activaciones, debido a que en tal caso las secuencias de retorno de funciones y procedimientos son iguales. Veremos, sin embargo, que aún podría haber una salida a esta cuestión, sin necesidad de incumplir ninguna norma del lenguaje.

A pesar de ello, aún queda otro problema más grave: se espera que la activación de la función deje un valor de retorno en la cumbre de la pila. Ahora bien, eso significa que la optimación ha de preservar el valor de retorno de la activación de función terminada prematuramente durante la activación del

procedimiento al que aquella llama. Y la optimación se consigue en esta clase de activaciones no recursivas sustituyendo el registro de activación del llamador (la función) por el registro de activación del receptor (el procedimiento). Si, como estamos suponiendo nosotros, no existe en el registro de activación de los procedimientos un espacio para el valor de retorno, el valor de retorno de la función va a perderse irremediablemente. ¿Sería posible preservar de alguna manera ese valor de retorno? Como vemos que la solución no parece simple, dejaremos la profundización en este problema para la última sección del capítulo.

5.2 MODELO DINÁMICO OPTIMADO: ACTIVACIONES FINALES RECURSIVAS DE FUNCIONES

Las ideas básicas apuntadas en la sección anterior nos llevan a conjeturar que exactamente las mismas acciones que se llevan a cabo para la optimación de las activaciones finales de procedimientos pueden servir sin ninguna modificación para obtener la optimación correspondiente de las activaciones finales de funciones. El objetivo de esta sección y la siguiente es mostrar la corrección de tal postura, que de confirmarse, supondría, amén de lo interesante de ampliar la optimación a las funciones, la existencia de un único y simple mecanismo para proceder a la optimación de los subprogramas en PASCAL, independientemente de la clase que sean, lo cual, por supuesto, es una cualidad muy a tener en cuenta a la hora de simplificar el compilador optimador.

La estructura de esta sección, en la que se estudiará el caso de las activaciones recursivas, es similar a las secciones correspondientes de capítulos anteriores: primero se definirán lo que son las activaciones finales de funciones; después, se presentará la secuencia de llamada que realiza la optimación y se establecerán los demás requisitos que han de darse para poder llevar a cabo la optimación.

5.2.1 Definición de activación final de función en PASCAL

La activación de una función es, en el lenguaje PASCAL, una clase de expresión; como tal, por tanto, nunca puede encontrarse en posición final. Obsérvese que hablaremos simplemente de "activación de función" en los mismos términos que Jensen y Wirth [1991: 179], es decir, como lo también denominado "llamada a función". Esto no debe confundirse con lo que denominaremos "activación final de función", que no es simplemente una activación de función en posición final, porque eso, como decimos, nunca es posible.

En efecto, una activación final de función será, para nosotros, un caso particular de la sentencia especial por la que se establece un valor a retornar por una función. La sintaxis de esta sentencia, que llamaremos *sentencia de asignación de retorno de función*, es:

$$\langle \text{unlab-stment} \rangle \rightarrow \text{id} := \langle \text{expr} \rangle$$

(la sintaxis completa para la parte de sentencias de un bloque, de donde se ha tomado la anterior regla de producción, aparece en el apartado 3.2.1). Obviamente, esta sentencia solo puede aparecer dentro del bloque de una función cuyo nombre sea *id*; ahora bien, no tiene que hacerlo obligatoriamente en la propia parte de sentencias de la función, sino que puede estar en la parte de sentencias de algún bloque anidado en ella. Requeriremos justamente que esto no sea así, es decir, la primera condición necesaria para poder afirmar que una sentencia de asignación de retorno es una activación final de función será que tal sentencia aparezca estrictamente en la parte de sentencias de la función que se nombra en la parte izquierda de la sentencia, y que lo haga en posición final, obviamente, tal como se definió el concepto de posición final para una sentencia cualquiera en PASCAL (Restricción 3.2.1).

Por otro lado, la segunda condición necesaria consiste en que la expresión $\langle \text{expr} \rangle$ de la parte derecha de la sentencia de asignación de retorno debe ser exactamente una activación de función. Si la activación es directamente recursiva, es decir, si la función llamada es la propia función a la que se asignará el valor de retorno, diremos que la sentencia al completo es una *activación final recursiva de función*. Si la función que es llamada en la sentencia

de asignación de retorno es otra función cualquiera distinta de la nombrada en la parte izquierda, entonces se trata de una *activación final no recursiva de función*.

Resumimos esta definición para el caso de las activaciones finales recursivas, dejando para la próxima sección el caso de las activaciones no recursivas:

Restricción 5.2.1. Si la sentencia de asignación de retorno de una función no está en posición final en la parte de sentencias del bloque de la propia función, o si no consta en su parte derecha exclusivamente de la activación de la función, entonces la optimación no puede llevarse a cabo.

Pasamos entonces a estudiar la secuencia de llamada optimada y el resto de requisitos de la optimación.

5.2.2 Optimación de la secuencia de llamada

La impresión recogida del estudio introductorio de esta clase de activaciones nos hizo conjeturar que exactamente la misma secuencia de llamada optimada para las activaciones finales recursivas de procedimientos (Algoritmo 3.2.1) podría servir para la misma clase de activaciones de funciones. El establecimiento con todo rigor de este hecho es el objetivo de este apartado.

Esta situación se debe a la escasa diferencia "real" entre el concepto de procedimiento y función en PASCAL, tal como se discutió en la introducción del presente capítulo. En lo que respecta a la estructura del registro de activación, la única distinción radica en la existencia de un espacio adicional para el valor de retorno (y ya hemos dicho que probablemente muchos compiladores ni siquiera hagan tal diferencia). Sin embargo, la incorporación de este espacio no influye para nada en el estudio de los demás elementos del registro de activación, pues estos ocupan exactamente las mismas posiciones en el registro de activación del llamador y en el registro de activación del receptor. Los registros *BasePila* y *TopePila*, la dirección de retorno, el enlace de control y el enlace de acceso han de mantenerse con los valores con que se activó por primera vez la función recursiva final, debido a exactamente los mismos argumentos recogidos en el apartado 3.2.2. Por lo tanto, las diversas condiciones para las activaciones finales recursivas de procedimientos obtenidas en aquel lugar siguen siendo

válidas ahora, y ello se consigue simplemente evitando ejecutar las operaciones de la secuencia de llamada estándar de funciones que calculan dichos componentes, resultando una optimación de idénticas características.

La discusión sobre el punto del código del subprograma al que debe establecerse el salto y que dio lugar a la Condición 3.2.4 sigue sirviendo para las funciones, pero ahora precisamos una matización. Recordemos que la parte de la secuencia de llamada ejecutada por la función receptora incluye la actualización de *TopePila*, y la inicialización de las variables y el valor de retorno (que es la única diferencia respecto al caso de los procedimientos). Igual que para estos era importante asegurar que se producía efectivamente la asignación de un valor indefinido a los datos locales (lo cual sigue siendo necesario para las funciones), ahora también es preciso que ocurra lo mismo con el valor de retorno. En efecto, puede ser que una activación de función asigne su valor de retorno en un momento determinado, y aun así continúe su código, dándose una activación final recursiva; la nueva activación debe comenzar con su propio valor de retorno indefinido, y al ser una activación final recursiva, el valor de la activación llamadora se va a perder realmente al retornar la receptora. Por lo tanto, en el modelo dinámico optimado es importante que se asegure que el paso L8 de la secuencia de llamada estándar es ejecutado. Ello queda reflejado sin ningún problema en la versión correspondiente a la Condición 3.2.4 para las funciones, al indicar que el salto puede obviar las instrucciones de la secuencia de llamada que actualizan *TopePila*, pero no las que dan valores indefinidos a los datos locales y el valor de retorno.

En cuanto a los parámetros formales, vamos a encontrar también los mismos resultados que en el caso de los procedimientos, tanto respecto a las restricciones (es decir, imposibilidad de aplicar la optimación si un parámetro actual por referencia es un dato local o parámetro formal por valor local, o si un parámetro actual procedimental o funcional es un procedimiento o función local) como en lo relativo a la optimación del proceso de actualización de los parámetros. Realmente, todas las discusiones de los apartados correspondientes de la sección 3.2 pueden aplicarse sin ninguna salvedad a funciones en lugar de procedimientos, por lo que concluimos afirmando nuestra impresión de que exactamente la misma secuencia de llamada optimada que resolvía las activaciones finales recursivas de procedimientos sirve para la misma clase de activaciones de funciones:

- LO1: el llamador evalúa los parámetros actuales, y coloca los valores obtenidos en el lugar correspondiente a sus propios parámetros formales;
- LO2: el llamador salta (jump) a la instrucción del código de la función que sigue a la actualización del valor de `TopePila`;
- LO3: el receptor (realmente, la misma función) asigna valores iniciales a sus datos locales y su valor de retorno.

Algoritmo 5.2.1 *Secuencia de llamada optimada para activaciones finales recursivas de funciones*

Por supuesto, sigue existiendo la pequeña diferencia en el paso LO3 sobre la asignación de un valor indefinido al valor de retorno, pero esa es la parte de la secuencia de llamada ejecutada por el receptor, que no ha de modificarse, como se sabe.

En definitiva, la parte de secuencia de llamada optimada que debe ejecutar la función llamadora (donde realmente se aplica la optimación) es exactamente la misma que la correspondiente a la misma clase de activaciones de procedimientos. Esto supone, como se ha dicho, una virtud añadida del optimador, pues con un único esquema de generación de código se podrán resolver dos aspectos en principio distintos del lenguaje fuente.

5.3 **MODELO DINÁMICO OPTIMADO: ACTIVACIONES FINALES NO RECURSIVAS DE FUNCIONES**

Lo que se ha concluido en la sección anterior nos hace confiar más fuertemente en que las denominadas activaciones finales no recursivas de funciones se resuelvan del mismo modo que las activaciones homónimas de procedimientos (*cf.* sección 4.2, y en particular, el Algoritmo 4.2.1). Sin embargo, aparece un aspecto novedoso que habremos de tener en cuenta, y que de hecho, podríamos decir que se nos "escapó" en cierto modo en el estudio de las activaciones recursivas en la sección anterior, aunque no invalidará para nada lo allí obtenido.

Supongamos que recogemos, para empezar, la definición de activación final no recursiva de función en la siguiente restricción, a partir del estudio realizado en el apartado 5.2.1:

Restricción 5.3.1. Si la sentencia de asignación de retorno de una función no está en posición final en la parte de sentencias del bloque de la propia función, o si no consta en su parte derecha exclusivamente de la activación de otra función, entonces la optimación no puede llevarse a cabo.

Desde luego, la siguiente cuestión a abordar ha de ser la que se refiere a la profundidad de anidamiento de la función llamada. El mismo argumento que dio lugar a la Restricción 4.2.2 sirve ahora para decir que la función activada no debe ser una función definida localmente en la que realiza la llamada (aunque sí puede ser un parámetro formal funcional de ella): solo en ese caso existiría la posibilidad de un acceso a la información del registro de activación de la función llamadora, que por tanto no podría desaparecer de la pila. Por tanto, existirá una versión de la restricción citada para las funciones:

Restricción 5.3.2. Si una activación final no recursiva de función es de una función definida en la parte de declaración de procedimientos y funciones de la función llamadora, entonces la optimación no puede llevarse a cabo.

Ahora bien, hemos de recordar que la condición de posición final no solo se ha de cumplir sintácticamente, sino también dinámicamente, es decir, la optimación es posible cuando, durante la ejecución, la función llamadora no hace más que copiar el valor producido por la función receptora en su propio valor de retorno. Con las activaciones de procedimientos, esto se cumplía trivialmente, pero con las funciones resulta que no siempre que se encuentra en un programa la estructura definida en la Restricción 5.3.1 se da exclusivamente el proceso mencionado. Ello es debido a que en la sentencia de asignación de retorno de función se exige, como en la asignación normal, que:

«The value of the expression must be assignment-compatible [...] with the type of the variable or function identifier» [Jensen y Wirth 1991: 170]

Un tipo *t* es compatible en asignación con otro tipo *t'* si básicamente son el mismo tipo, pero también se admite, entre otras cosas, que *t'* sea el tipo predefinido *Real* y *t* el tipo predefinido *Integer* [Jensen y Wirth 1991: 160], es decir, puede asignarse un dato entero a una variable real. Esto tiene una

consecuencia importante, como es que el proceso habitual de ejecución de la asignación resulta modificado: ha de realizarse una conversión de tipo antes de llevar el valor a la localidad de la variable o del valor de retorno de la función.

Consideremos el siguiente programa, que contiene una pretendida activación final no recursiva.

```

program Prueba2 ...
  function f (...): Integer;
  function g (...): Real;
  begin
    g := f (...) { supuesta activación final no recursiva }
  end ( g );
begin
end.

```

Programa 5.3.1 Un programa con una supuesta activación final no recursiva de función

Sintácticamente, parece que la única operación a realizar en la sentencia de asignación de retorno en posición final para la función *g* es la colocación del valor devuelto por la función *f* en la localidad del valor de retorno de *g*.

Sin embargo, se trata del fenómeno conocido como *coerción*, es decir, existe una conversión de tipo implícita adicional, como ya hemos dicho. Debido a que la representación máquina de los valores de tipo real suele ser muy diferente de la representación de los valores de tipo entero, y no solo por la longitud asociada habitualmente a cada clase de valor, es preciso que exista una instrucción en el código objeto que realice explícitamente la conversión del valor devuelto por *f*, antes de ser colocado en las posiciones correspondientes al valor de retorno de *g*.

La optimación no es entonces posible: esta consistiría en dar como resultado de la activación de *g* el resultado de la activación de *f*, y ello estaría omitiendo la transformación del valor entero a la representación máquina de los valores de tipo real, con lo que el programa optimado no sería equivalente. Por lo tanto, hemos de estudiar en profundidad las operaciones implícitas que pueden aparecer al procesar una sentencia de asignación de retorno de función.

Empezamos con la definición exacta de la compatibilidad de tipos en el lenguaje PASCAL:

«Two types are said to be *compatible* if any of the following four conditions is true.

- (a) They are the same type.
- (b) One is subrange of the other, or both are subrange of the same host type.
- (c) [...].
- (d) [...].

»A type is called *assignable* if it is neither a file type nor a structured type with a component type that is not assignable.

»A value possessing type T2 is called *assignment-compatible* with a type T1 if any of the following four conditions is true.

- (a) T1 and T2 are the same assignable type.
- (b) T1 is Real and T2 is Integer.
- (c) T1 and T2 are compatible ordinal types [...], and the value is a member of the set of values determined by T1.
- (d) [...].

»Wherever assignment-compatibility is required, and T1 and T2 are either compatible ordinal types or [...], it is an error if the value is not a member of the set of values determined by T1» [Jensen y Wirth 1991: 160-161]

El texto original no incluido no es relevante para el caso de la sentencia de asignación de retorno de función, pues se refiere a los tipos *set* y *string*. En resumidas cuentas, lo anterior viene a decir que hay exactamente seis posibles casos en lo que a compatibilidad en asignación se refiere. Si escribimos el tipo de las entidades de las partes izquierda y derecha de la sentencia de asignación en lugar de las propias entidades, y tenemos en cuenta que *tipo* es cualquier tipo que pueda ser resultado de una función pero que no sea subrango (por tanto, ordinal no subrango, apuntador o el tipo predefinido *Real*), que *host* es un tipo que admite formación de un tipo subrango (por lo tanto, enumerado o los tipos predefinidos *Integer*, *Char* o *Boolean*), y que *rg* y *rg'* son ambos tipos subrango de *host*, los seis casos posibles son:

- | | |
|----------------------------------|--|
| 1. <i>tipo</i> := <i>tipo</i> | [condición (a) de la compatibilidad en asignación] |
| 2. <i>Real</i> := <i>Integer</i> | [condición (b) de la compatibilidad en asignación] |
| 3. <i>rg</i> := <i>rg</i> | [condición (c) de la compatibilidad en asignación] |
| 4. <i>rg</i> := <i>host</i> | [<i>idem</i>] |
| 5. <i>host</i> := <i>rg</i> | [<i>idem</i>] |
| 6. <i>rg</i> := <i>rg'</i> | [<i>idem</i>] |

No hemos de considerar el caso 3 como incluido dentro del caso 1 por una razón que será aparente en unos instantes (y que también nos lleva a incluir la asignación entre *Real* y un subrango de *Integer* dentro del caso 2). Estudiaremos cada caso para saber si existen procesos implícitos asociados.

Desde luego, el caso 1, en que el tipo de las funciones llamadora y receptora es exactamente el mismo, no presenta ningún problema, así que la optimización puede llevarse a cabo sin dificultad, ya que es realmente la suposición hecha hasta el momento. De la misma forma, ya sabemos que el caso 2 imposibilita la optimización, por la necesidad de practicar una coerción después del retorno de la función llamada. Los otros casos son un poco más complejos, y para ellos hay que tener presente el último párrafo de la cita anterior, que indica que deben existir comprobaciones para asegurar que el valor que se asigna a una variable o función cuyo tipo es subrango está efectivamente incluido entre los valores posibles de ese tipo.

Como se sabe, estas comprobaciones pueden ser estáticas en algunos casos, pero también pueden darse situaciones (las más de ellas, por otro lado) en que la comprobación haya de ser dinámica. Tomemos el caso 4: es preciso asegurar que el valor de tipo *host* de la parte derecha de la asignación está comprendido entre los valores posibles del tipo subrango *rg* de la parte izquierda, debiendo señalar un error en caso contrario. Cuando se trata de una sentencia de asignación de retorno de función, y el valor de la parte derecha es el resultado de la activación de otra función, es claro que la comprobación ha de ser dinámica, por lo que tras el retorno de dicha función ha de llevarse a efecto tal comprobación *antes* de llevar el resultado a la localidad del valor de retorno de la función llamadora. En definitiva, la optimización no es posible de ninguna manera, pues omitiría esta verificación.

El caso contrario, el quinto, no ofrece ningún problema para la optimización, pues no ha de realizarse ninguna comprobación. El número 6 es similar al 4, e imposibilita la optimización, salvo que el tipo *rg'* sea en la práctica un subrango del tipo *rg*, es decir, que el conjunto de valores determinado por el tipo *rg'* esté incluido en el conjunto de valores determinado por el tipo *rg*. Y nos queda el caso 3, que en principio admite la optimización de la misma forma que el caso 1. Sin embargo, otro aspecto del lenguaje PASCAL va a complicar aún más este asunto.

A la hora de establecer el tipo de las expresiones, han de observarse las normas relativas a los diferentes operadores, pero los tipos de los operandos se obtienen primeramente a partir de su constitución como factores. A este respecto, la definición del lenguaje dice:

«The type of a factor is derived from the type of its constituent (e.g., variable or function). If the constituent's type is a subrange, then the type of the factor is the host type of the subrange; [...] otherwise, the type of the factor is the same as the type of the constituent» [Jensen y Wirth 1991: 166]

Esto significa que los casos 3, 5 y 6 anteriores no serían realmente posibles, pues nunca una expresión puede tener un tipo subrango, con lo cual los casos 3 y 6 se encontrarían englobados en el caso 4, donde la optimación no es posible, y el caso 5 sería un caso particular del caso 1, manteniendo la posibilidad de optimación, tal como antes se ha concluido.

A pesar de ello, resulta una lástima que una activación final de función como la que contiene el siguiente programa no resulte optimada.

```

program Prueba3 ...
...
type rg: 1 .. 10;
...
var x, y: rg;
...
    function f (_): rg;
    ...
    function g (_): rg;
    begin
        ...
        g := f (_) { casi una activación final no recursiva }
    end { g };
...
begin
    ...
    x := y;      { asignación con verificación redundante }
    ...
end.

```

Programa 5.3.2 Un programa con una activación de función que casi es final no recursiva

Obsérvese que el compilador, siguiendo la norma de ampliar el tipo subrango de un factor a su tipo *host*, generará entonces una comprobación dinámica para asegurar que el valor devuelto por la función *f* está comprendido entre los valores posibles del tipo de la función *g*. Claramente, esta verificación es redundante, al ser el tipo de ambas funciones el mismo, y es posible modificar el compilador sin mucha dificultad para que omita tales verificaciones, y no

solo para las activaciones de funciones, sino en general para cualquier proceso en que la compatibilidad en asignación sea requerida (asignaciones y parámetros actuales por valor), como la sentencia marcada en el programa anterior. Por lo tanto, es posible también que el optimador aplique una técnica similar para detectar que activaciones finales no recursivas de función como la del ejemplo puedan considerarse como optimables.

Si esto es así, podemos considerar como definitivamente válidas para practicar la optimación las situaciones 1, 3 y 5, y como no válidos los casos 2 y 4. El caso 6 puede resultar optimado si el conjunto de valores del tipo de la función llamada está contenido en el conjunto de valores del tipo de la función llamadora. La detección de este hecho es algo más compleja que la del caso 3, pero posible sin duda. Con ello, tenemos un requisito adicional para la optimación de activaciones finales no recursivas de funciones:

Restricción 5.3.3. Si T1 y T2 son los tipos de las funciones llamadora y receptora, respectivamente, en una activación final no recursiva de función, entonces la optimación solo puede llevarse a cabo si alguna de las siguientes dos condiciones es cierta:

- (a) T1 y T2 son el mismo tipo.
- (b) T2 es un subrango de T1.
- (c) T1 y T2 son tipos subrango compatibles y puede determinarse estáticamente que el conjunto de valores determinado por T2 está incluido en el conjunto de valores determinado por T1.

La condición (a) incluye los casos 1 y 3 del estudio anterior, la condición (b) el caso 5, mientras que las situaciones válidas del caso 6 quedan reflejadas en la condición (c). Del mismo modo, las situaciones 2 y 4 quedan rechazadas como optimables definitivamente.

Ahora podemos entender también porque al principio dijimos que se nos había "escapado" algo del estudio de las activaciones finales recursivas de funciones. Si tenemos una función recursiva final cuyo tipo sea subrango, ocurrirá exactamente lo mismo que se ha obtenido hace un momento: si el compilador no es cuidadoso, puede marcar como no optimable la sentencia de asignación de retorno recursiva en posición final, pues aparecerá en la parte derecha de la asignación la necesidad de una comprobación dinámica debida a la promoción del tipo subrango a su tipo *host*. Por lo tanto, el optimador ha de estar igualmente atento a tales situaciones, pues esa comprobación es

redundante, y la posición final, dinámicamente hablando, sigue garantizada. Por lo demás, todo el estudio de la sección 5.2 sigue siendo completamente válido, y más en concreto la Restricción 5.2.1 como definición de activación final recursiva de función.

Existe, a pesar de todo, otra perspectiva adicional para enfocar este asunto, que viene provocada por el mismo concepto de tipo asociado a las funciones. Este tipo es el del valor de retorno, y habitualmente, diferentes tipos están vinculados a diferentes representaciones máquina de los valores correspondientes. La característica del modelo dinámico estándar asociado a las funciones, tal como se estableció en el apartado 5.1.2, de dar un tamaño fijo independiente del tipo para el espacio reservado al valor de retorno de las funciones en el registro de activación se vuelve de primordial importancia para aplicar la optimación. Supongamos el caso de una activación final no recursiva sujeta a la situación 5 de las definidas antes, en que la función llamada tiene un tipo subrango del tipo de la función llamadora. Es muy probable que, según sean el tipo *host* y el tipo subrango, el compilador realice una optimación de tamaño en los datos del tipo subrango: si se trata de un subrango del tipo predefinido *Integer*, expresado por $1 \dots 10$, basta con un *byte* para almacenar los valores correspondientes (incluso cuatro *bits* son suficientes). Si el modelo dinámico estándar del lenguaje, en lo referente a las funciones, reservara un espacio exacto para el valor de retorno, entonces los mismos problemas encontrados entre el tipo real y el tipo entero surgirían en este caso, pues tras la finalización de la activación de las dos funciones involucradas en el proceso, el subprograma que lo inició esperaría encontrar un valor de retorno de tipo *host* en la pila, mientras que realmente hay un valor de un tipo subrango, de menor tamaño, con lo cual se produciría un error al tomar el valor de retorno. Por lo tanto, enunciemos una nueva restricción para la optimación:

Restricción 5.3.4. Los espacios destinados a los valores de retorno de las funciones llamadora y receptora deben tener exactamente el mismo tamaño, y las representaciones máquina de los datos correspondientes a los tipos de retorno de ambas funciones deben ser idénticas.

Con ella expresamos que no solo el tamaño, sino el mismo modo en que se representan los datos correspondientes a los tipos subrango han de ser exactamente iguales a los que se utilizan para los tipos *host* a que corresponden.

Obsérvese que esta restricción no es necesaria para el caso (a) de la Restricción 5.3.3, que será ciertamente el más habitual.

La técnica de mantener el valor de retorno de una función en un registro, en lugar de la pila de control, también ofrecería dificultades similares, aunque es probable que en ese caso no se realizara la optimación de tamaño sobre los subrangos, por razones obvias, con lo que la Restricción 5.3.4 se cumpliría sin problemas. En cualquiera de las dos alternativas, no supone mucha dificultad, pensamos, modificar ligeramente un compilador dado para que esa restricción pueda aplicarse en toda su extensión: simplemente se trata de no realizar la posible optimación de tamaño de los tipos subrango en lo relativo al valor de retorno de una función.

Una vez establecidas las restricciones básicas de la optimación de activaciones finales no recursivas de funciones, se puede comprobar que el resto de aspectos siguen sin ninguna variación los resultados obtenidos para las activaciones homónimas de procedimientos en el capítulo 4. Recordemos que el valor de `BasePila` no ha de modificarse (algo que ahora depende directamente de que el espacio para el valor de retorno no varíe entre la función llamadora y la receptora, lo que viene asegurado por la Restricción 5.3.4), que el valor de `TopePila` tampoco ha de modificarse si la actualización es realizada por la función llamada a partir de `BasePila`, que la dirección de retorno y el enlace de control han de mantenerse invariantes, y que el enlace de acceso ha de calcularse como normalmente. También se obtienen las mismas restricciones que impiden la optimación si un objeto local es parámetro actual por referencia, procedimental o funcional. Finalmente, si la función activada es un parámetro formal funcional de la función llamadora, entonces es preciso preservar su valor durante el cálculo de los parámetros actuales y su asignación a las posiciones definitivas.

Todas las consideraciones que se realizaron en el capítulo 4 pueden aplicarse sin problemas a las activaciones de funciones para concluir que lo anterior determina el modelo dinámico optimado para esta clase de activaciones. Por lo tanto, la secuencia de llamada optimada (Algoritmo 4.2.1) sirve de la misma manera. Solo hay que recalcar que ello es posible gracias a la Restricción 5.3.4, lo mismo que para el registro `BasePila`: los elementos de la zona de control del registro de activación de la función receptora ocuparán exactamente la misma posición que los elementos correspondientes de la

función que llama si el valor de retorno (que se encuentra por debajo de ellos) ocupa exactamente el mismo tamaño, y por eso la secuencia de llamada optimada no precisa ninguna instrucción si estos elementos se mantienen invariantes a lo largo de la llamada.

5.4 CONSIDERACIONES ADICIONALES

A modo de resumen, podemos decir que los resultados de la optimación de activaciones finales de funciones en PASCAL son básicamente los mismos que se obtuvieron para los procedimientos en los capítulos 3 y 4, salvo el problema que se ha recogido en la Restricción 5.3.3 y la Restricción 5.3.4. Estas restricciones surgen como efecto directo del sistema de tipos definido en el lenguaje PASCAL, tal como se ha comprobado. Por lo tanto, podemos concluir que existen aspectos del lenguaje en principio no directamente relacionados con el problema de la recursividad final que resultan determinantes para la posibilidad de llevar a cabo su optimación. En cierto modo, estamos confirmando nuestra hipótesis inicial de que la aplicación de las técnicas conocidas en un lenguaje a otro lenguaje distinto (pero no tan distinto) no es una labor tan trivial como a veces se concluye en la literatura.

Ahora nos interesa, tal como hemos hecho en los capítulos anteriores, poner en comparación los resultados obtenidos con los que se conocen para el lenguaje SCHEME, particularmente respecto a la estrategia de Hanson [1990], referente ya constante de nuestro trabajo. En el caso de las funciones, esto será todavía más clarificador, pues nos permitirá ver que otras semánticas asociadas al concepto de "función" permiten la optimación, como es el caso de SCHEME.

En el lenguaje SCHEME el concepto de subprograma siempre está asociado al retorno de un valor. Quizá esto no esté dicho de un modo correcto, pues lo que realmente ocurre es que SCHEME, como LISP, trabaja en el dominio de las *expresiones* (entidades que se asocian principalmente a la producción de un valor), y ocurre que la activación de un subprograma (*aplicación*, en la terminología funcional) es una clase particular de expresión, por lo que, como cualquier otra, produce un valor. El concepto de "procedimiento", tal como se entiende en PASCAL, es en LISP el caso particular de una función que devuelve

un "valor nulo". Por consiguiente, la implementación de esta filosofía va a distanciarse notablemente de la que existe en PASCAL (a pesar de que, como dijimos, trata sobre los mismos conceptos).

Recordemos del apartado 5.1.1 que en PASCAL se diferenciaba claramente entre el proceso que asignaba un valor de retorno para la activación de una función y el proceso que producía el retorno de la función como subprograma. La principal consecuencia de la semántica de SCHEME a este respecto es que no se va a dar la distinción entre ambos procesos: la operación que determina la terminación de una función como subprograma es exactamente la misma que produce el valor de retorno, o tal como se dice más habitualmente, el resultado que produce una función es el resultado de la última operación que se realiza dentro de la misma. Esta estrategia, por otro lado, es similar en filosofía a la que existe en otros lenguajes, como C o MODULA, al utilizar una sentencia especial (*return*) para ello, como se comentó en el apartado 5.1.2. Por consiguiente, en todos esos lenguajes no existe necesidad de reservar un espacio de memoria durante la activación de cada función para almacenar el valor de retorno, sino que basta con adoptar un lugar global en el que colocarlo justo antes del retorno, independientemente de que existan muchas funciones activadas en un momento determinado. Realmente, ese lugar global puede ser el que utilice cualquier expresión para colocar su resultado. Por ello, la información dinámica necesaria en SCHEME para la activación de una función (o sea, el equivalente al registro de activación en PASCAL de la Figura 5.1.1) se parece más a lo que en PASCAL se necesita para los procedimientos (Figura 2.5.1), en el sentido de que no es precisa la existencia de espacio explícito para el valor de retorno.

En este esquema, el tratamiento de la recursividad final es sencillo: cuando una función retorna, no solamente esta produciendo el valor de retorno correspondiente a su propia activación, sino el de toda la secuencia de posibles llamadas en posición final que la precedieron. La secuencia de retorno que ejecute esa función (si es el caso) solo ha de actuar como habitualmente (en consonancia con nuestras propias Condición 3.2.1 y Condición 4.2.1 acerca de no modificar las operaciones ejecutadas por los receptores de las llamadas finales): el valor de retorno ya habrá quedado en el lugar global acordado (como efecto de la evaluación de la última expresión de su código) y solo habrá de recuperarse la dirección de retorno y saltar a ella. Por lo tanto, el

comportamiento conjunto en lo relativo a la recursividad final es similar al de PASCAL, pero es más sencillo comprender que es correcto.

Ahora debemos investigar lo que ocurre con la Restricción 5.3.3 y la Restricción 5.3.4, que no tienen parangón en SCHEME. En realidad, igual que afirmamos que tales restricciones surgen directamente como consecuencia del sistema de tipos de PASCAL, podemos decir que su no existencia en SCHEME se debe a la misma razón, lo que afianza nuestra conclusión. Muchas veces se dice [Sethi 1992: 286] que LISP, y SCHEME como descendiente, carecen de tipos (en inglés, *untyped languages*). Pensamos que esto no es del todo cierto. Un lenguaje como PASCAL tiene un *sistema de tipos primordialmente estático*, esto es, existe una gran cantidad de verificaciones estáticas (en tiempo de compilación) de una serie de normas que rigen las relaciones entre unos tipos y otros; y decimos "primordialmente" porque algunas de esas normas solo pueden verificarse dinámicamente. En LISP y SCHEME los valores tienen un "tipo", eso es indudable, y existen normas que gobiernan las relaciones entre esos valores y las operaciones que pueden aplicarse a ellos, así que no puede decirse que estos lenguajes "carezcan de tipos". Lo que ocurre es que la verificación de tales normas se lleva a cabo por entero durante la ejecución del programa: tienen un *sistema de tipos dinámico*, por consiguiente.

Una de las características de un sistema de tipos dinámico es que la verificación de las diferentes normas es realizada implícitamente por las operaciones que se aplican a los valores. Así, la aplicación del operador de suma de cantidades numéricas a un dato de tipo lista produce la violación de una de las normas del sistema de tipos, que se señala con el correspondiente error de ejecución. Sin embargo, no se asocia el concepto de tipo a los nombres del programa (algo requerido obviamente por un sistema de tipos primordialmente estático, de donde surge el concepto de *declaración*), y ello es particularmente cierto para los nombres de función, con lo que no existe la idea de "función que produce un valor de un cierto tipo", sino solamente de "función que produce un valor". Se supone que el programador es responsable de que una función produzca valores lógicamente correctos en términos de la utilización posterior de esos valores. Por otro lado, sí es posible afirmar que, en general, existen menos normas en el sistema de tipos de LISP o SCHEME que en un lenguaje como PASCAL: el sistema de tipos de PASCAL es *fuerte*, y el de LISP, no siendo tampoco *débil*, sí es "algo menos" fuerte que el del primer lenguaje.

La recursividad final se obtiene muy sencillamente si esto es así, por el mero hecho de que una función nunca se preocupa de lo que produce: eso queda para quien recoja ese valor. Por ello, ninguna función tiene necesidad en SCHEME de verificar norma alguna relativa al sistema de tipos en la producción de su valor de retorno, algo que sí ocurre en PASCAL, resultando por consiguiente la Restricción 5.3.3. Por supuesto, esto tiene un inconveniente: responsabilizar al programador de ciertos aspectos relativos a la corrección del programa. No es este el momento de discutir la conveniencia de esta filosofía, o de abogar por un sistema de tipos estático y *fuerte* en un lenguaje de programación, pero lo cierto es que lenguajes funcionales modernos (ML [Milner, Tofte y Harper 1990; Clack, Myers y Poon 1993], HASKELL [Hudak y Fasel 1992; Hudak, Peyton Jones y Wadler 1992]) parecen adoptar esta misma idea, acompañada, eso sí, de *polimorfismo* y un sistema de *inferencia de tipos* que resuelven los problemas más comúnmente criticados de esta filosofía.

Por ello, se concluye que existirán restricciones para la optimación de la recursividad final de funciones, similares a las encontradas en PASCAL, en todos los lenguajes que dispongan de un sistema de tipos fuerte, siempre que la sintaxis para la determinación del valor de retorno de una función lleve implícita la verificación dinámica de alguna norma del sistema de tipos. Debe recalarse que en PASCAL los casos en que la Restricción 5.3.3 prohíbe la optimación son relativamente pocos (surgen por la relación entre enteros y reales y por las normas acerca de los tipos subrango), pero más que nada porque está muy restringido el tipo del valor de retorno que puede producir una función. En otros lenguajes, en que los valores de retorno de las funciones puedan ser de un conjunto mucho más amplio de tipos (si no todos), es de esperar que también existan más normas para la relación entre ellos en el sistema de tipos, con lo que el equivalente de la Restricción 5.3.3 contendrá más situaciones que prohíben la optimación de las activaciones finales.

La discusión anterior admite otro punto de vista para comprender el alcance de las restricciones mencionadas. En cierto modo, es posible entender las normas que forman un sistema de tipos como operaciones que implícitamente serán ejecutadas sin que el programador tenga que añadirlas explícitamente en el programa. Por lo tanto, que SCHEME sea "menos fuerte" en su sistema de tipos que PASCAL significa que para conseguir la misma seguridad al programar cierto algoritmo, es preciso escribir explícitamente en SCHEME

cierto código que en PASCAL se generará implícitamente como verificación dinámica. Entonces, los casos que en PASCAL no pueden ser optimados por estar sujetos a la Restricción 5.3.3 podrían obligar, de ser trasladados a SCHEME, a la aparición explícita de un código que desharía la posición final. En definitiva, es posible pensar que esta restricción no tendrá más impacto en la programación en PASCAL que el que ya existe en SCHEME, donde, aunque no existe tal requisito, sería necesario adoptar soluciones no recursivas finales para cierta clase de problemas, si se quieren mantener las mismas cotas de seguridad en los programas.

5.5 MODELO DINÁMICO OPTIMADO: ACTIVACIONES FINALES MIXTAS DE PROCEDIMIENTOS

La introducción del capítulo (apartado 5.1.3) mostró que sería interesante estudiar la posibilidad de la optimación de una última clase de activaciones finales: las activaciones de un procedimiento desde una función, que denominaremos *activaciones finales mixtas*. El objetivo, como todas las optimaciones que se han estudiado hasta el momento, sería la sustitución del registro de activación de la función que está en la cumbre de la pila por el registro de activación del procedimiento llamado en posición final. Existen varios problemas para que esto pueda darse, que analizaremos en profundidad en esta sección. Primero, definiremos exactamente las condiciones sintácticas de la clase de activaciones que se van a considerar. Después, mostraremos las condiciones necesarias para que la optimación pueda llevarse a cabo.

5.5.1 Definición de activación final mixta en PASCAL

La definición de *activación final mixta* necesita realmente poco más de lo que se ha dicho ya: ha de ser una sentencia de activación de procedimiento situada en posición final en la parte de sentencias del bloque de una función; la condición de posición final sigue siendo la misma que se recogió en la

Restricción 3.2.1. Por lo tanto, enunciaremos la siguiente primera condición sintáctica para esta nueva clase de optimaciones:

Restricción 5.5.1. Si la sentencia de activación de un procedimiento no está en posición final en la parte de sentencias del bloque de una función, entonces la optimación no puede llevarse a cabo.

Desde luego, ha de verificarse también que el procedimiento llamado no sea local en el bloque de la función, por las mismas razones que dieron lugar a la Restricción 4.2.2 y la Restricción 5.3.2:

Restricción 5.5.2. Si una activación final mixta es de un procedimiento definido en la parte de declaración de procedimientos y funciones de la función llamadora, entonces la optimación no puede llevarse a cabo.

Por lo tanto, pasamos a estudiar los otros aspectos de la optimación.

5.5.2 Optimación de la secuencia de llamada

Existen dos problemas principales a la hora de permitir la optimación de una activación final mixta, los dos asociados al hecho de que una función es un subprograma que produce un valor de retorno, mientras que un procedimiento no lo produce. Primero, al evitar la secuencia de retorno de la función, ha de asegurarse que ninguna operación de las que allí se realizan se omite incorrectamente. Segundo, después de la activación del procedimiento, el estado de la máquina debe ser el mismo que si hubiera finalizado la activación de la función, es decir, debe existir un valor de retorno encima del registro de activación del subprograma que activó el proceso. Estos dos problemas son un poco más complejos que los que hemos encontrado hasta ahora, ciertamente, pero hemos de intentar solucionarlos.

El primer problema es claro: durante la secuencia de retorno de la función se verifica que su valor de retorno está definido, señalándose un error de ejecución en caso contrario. Puesto que esta secuencia de retorno no existirá durante el proceso optimado, ha de realizarse esta comprobación en algún otro momento. Desde luego, no puede ser durante el proceso de retorno del procedimiento, pues sigue siendo válida una condición básica de todas las optimaciones, que repetimos para mantener la completitud:

Condición 5.5.1. En el modelo dinámico optimado, las operaciones de las secuencias de llamada y de retorno que son realizadas por el receptor de la llamada han de mantenerse sin variación respecto al modelo dinámico estándar.

El único punto que nos queda es la propia secuencia de llamada del procedimiento, justo el proceso que queremos optimar, tal como ya sucedió en algún caso anteriormente (actualización del *display* en las activaciones finales no recursivas de procedimientos, cf. apartado 4.2.2). Por lo tanto, implícitamente estamos admitiendo la hipótesis de que el procedimiento llamado no puede modificar el valor de retorno de la función, es decir, que este realmente ya está definido *antes* de la activación del procedimiento. Y esto es cierto, pero siempre ocurre en esta vida, solo hasta cierto punto.

En efecto, pudiera ser que el código del procedimiento contuviera una sentencia de asignación de retorno para la función llamadora. En ese caso, desde luego, no sería admisible la optimación, lo mismo que no lo sería si se hiciera alguna referencia a un objeto local de la función. Pero eso justamente está prohibido por la Restricción 5.5.2, así que no hay que tener ninguna duda a este respecto. Uno estaría tentado de concluir que, efectivamente, basta con incluir la verificación del valor de retorno de la función llamadora *antes* de activar el procedimiento de la llamada final mixta para solucionar el problema.

Consideremos, sin embargo, el Programa 5.5.1, en que un parámetro actual por valor del procedimiento es una llamada a una función local de la función llamadora. Efectivamente, observamos que el valor de retorno de la función llamadora f puede no estar definido *antes* de la activación del procedimiento en posición final, mientras que sí lo está tras ella, y eso sin que el procedimiento tenga acceso a ese valor de retorno, al no ser local en f . Y ello debido al efecto lateral producido durante la evaluación de los parámetros actuales de la llamada al procedimiento. Por lo tanto, concluimos que la verificación del valor de retorno de la función ha de realizarse exactamente después de haber calculado en su totalidad los parámetros actuales de la llamada al procedimiento. Una vez alcanzado ese punto, sí podemos asegurar, mediante la Restricción 5.5.2, que el procedimiento no puede modificar de ningún modo el valor de retorno de la función que lo activó, y que por ello su verificación puede avanzarse sin problemas.

```

program Prueba4 ...
  procedure p (...);
  function f ...
    function g ...
    begin
      f := 7;
    end ( g );
  begin
    { el valor de retorno de f aun puede estar no definido }
    p ( g(...), ...) { activación final mixta }
  end ( f );
begin
end.

```

Programa 5.5.1 Un programa con una activación final mixta en PASCAL.

Tenemos, en definitiva, solución a la primera dificultad planteada para esta clase de activaciones:

Condición 5.5.2. La verificación del valor de retorno de la función que contiene la activación final mixta ha de realizarse estrictamente después de la evaluación de todos los parámetros actuales de llamada.

y podemos ver la estructura de la primera parte de la secuencia de llamada optimada:

- LO1: la función llamadora evalúa los parámetros actuales, y coloca los valores obtenidos en el lugar correspondiente de su propio registro de activación, mediante desplazamientos respecto a BasePila;
- LO2: la función llamadora verifica su valor de retorno, señalando un error si no está definido;
- LO3: ...

Veamos ahora el otro asunto.

Después de la activación final del procedimiento, el subprograma que inició el proceso espera encontrar un valor de retorno de la función a la que llamó, situado exactamente encima de su registro de activación (es decir, accesible directamente por el registro TopePila). Tal como se ha definido el modelo dinámico estándar del lenguaje, los registros de activación de los procedimientos no poseen espacio propio para un valor de retorno. Por ello, el

problema no consiste solamente en la mera pérdida del valor de retorno de la función, sino en la diferencia estructural de los registros de activación de funciones y procedimientos, que hará que elementos correspondientes se sitúen en posiciones relativas diferentes.

Obviamente, si se hubiera elegido uniformizar la estructura de los registros de activación de todos los subprogramas y mantener un espacio para el valor de retorno en los registros de activación de los procedimientos, nuestras dificultades estarían resueltas de un plumazo: simplemente, el valor de retorno de la función ocupará el espacio (normalmente inútil) del valor de retorno del registro de activación del procedimiento, con lo que cuando este finalice, seguirá ofreciéndose el valor de retorno correcto al subprograma llamador del proceso; además, los elementos de la zona de control (algunos de los cuales no habrá que modificar, como nos imaginamos) seguirían situados en las mismas posiciones, con lo que al fin se obtendría una optimación tan sencilla y elegante como las anteriormente estudiadas. Ciertamente, como se dijo en la introducción del capítulo, dadas las mínimas diferencias sintácticas y dinámicas entre funciones y procedimientos en PASCAL, es probable que casi todos los compiladores del lenguaje elijan esta estructura, pues el compilador resulta incluso más sencillo, y la presencia de espacio inútil en la pila de control durante la ejecución no es un precio nada alto: si la pila se va consumir en algún momento, será debido a problemas relacionados con una profundidad excesiva de la recursividad, no con la presencia de unos pocos *bytes* adicionales por cada registro de activación.

Sin embargo, nuestro interés en este trabajo no es solo mostrar que la optimación puede llevarse a cabo con un esquema particular de organización del programa objeto, sino saber si puede aplicarse con pocas dificultades a modelos dinámicos alternativos para el lenguaje PASCAL. Por esa razón decidimos diferenciar la estructura de los registros de activación de los subprogramas, con lo que podremos observar, como en este caso, el impacto exacto de las elecciones que realiza un diseñador de un compilador para cubrir las necesidades del lenguaje.

Supongamos entonces que simplemente se mantiene el espacio para el valor de retorno de la función llamadora durante la activación del procedimiento, de manera que el nuevo registro de activación ocupe el espacio por encima de ese valor de retorno, o más exactamente, la zona de control del

nuevo registro de activación se coloca exactamente en las mismas posiciones que la zona de control del registro de activación de la función llamadora, y la zona de datos comienza también a partir del mismo punto. Con ello, los parámetros actuales de la llamada han de colocarse en la misma zona de parámetros formales de la función (aunque pueden ser de diferente tamaño, como sabemos): salvo el paso LO2 anterior, el resto de secuencia de llamada consiste exactamente en las mismas operaciones que se realizan en la secuencia de llamada optimada que se definió para las activaciones finales no recursivas de procedimientos. Como la actualización de los diferentes elementos depende del registro *BasePila*, y este señala el punto de separación entre la zona de control y la zona de datos del registro de activación, las operaciones que se realicen no dependen para nada de la existencia o no existencia de un espacio adicional por debajo del registro de activación. Evidentemente, si *BasePila* señalase estrictamente al comienzo del registro de activación, no sería posible hacerlo de modo tan sencillo, pero en principio bastaría con aumentar el valor de ese registro para que apunte justo por encima del valor de retorno de la función llamadora, y entonces aplicar las operaciones de la misma manera.

Obsérvese que entonces es fácil cumplir las condiciones de la optimación, que evidentemente van a ser las mismas que para las activaciones finales no recursivas de procedimientos: *BasePila*, la dirección de retorno y el enlace de control no han de modificarse, y solo ha de calcularse el enlace de acceso nuevo. También es igual de fácil ver que el procedimiento no va a poder modificar nunca ese valor de retorno, pues un procedimiento no tiene acceso a tal localidad desde *BasePila*, y el subprograma llamador no puede entender tales posiciones como propias, pues realmente se encuentran *fuera* de su registro de activación. Por lo tanto, la optimación parece igual de posible que en los otros casos, y de la misma manera sencilla y uniforme.

Sin embargo, aún queda un problema que va a resultar bastante grave. La optimación sustituye una secuencia de retorno del procedimiento llamado y otra secuencia de retorno de la función llamadora por la simple secuencia de retorno del procedimiento. La necesidad de verificar la definición del valor de retorno de la función durante secuencia eliminada nos obligó a realizar tal operación durante la secuencia de llamada optimada. Ahora, la conjunción de este hecho con la diferencia estructural entre los registros de activación de funciones y procedimientos nos causará un pequeño trastorno. En efecto, al

ejecutar el retorno del procedimiento, se actualiza el valor de *TopePila*, y este registro queda apuntando al que era supuestamente el final del registro de activación del subprograma iniciador del proceso. En el modelo dinámico estándar, en que los registros de activación de procedimientos y funciones son diferentes, eso se realiza restando al valor de *BasePila* el tamaño de la zona de control del registro de activación. Pero como quien está ejecutando la secuencia de retorno es el procedimiento, la actualización deja *TopePila* apuntando *encima* de las posiciones reservadas al valor de retorno de la función que llamó al procedimiento: el valor de retorno sigue estando ahí, con el valor exacto que le asignó la función, pero *debajo* de la posición señalada por *TopePila*, no por encima, como supone cualquier subprograma que active una función tras el retorno de esta.

Como las secuencias de retorno de los subprogramas no pueden modificarse en el modelo dinámico alternativo, tal como establece la Condición 5.5.1, ocurre que no puede darse una acomodación del registro *TopePila* para que señale correctamente al valor de retorno de la función tras la activación final mixta, así que finalmente hemos de concluir que no es posible llevar a cabo la optimación de esta clase de activaciones bajo el modelo dinámico estándar elegido para el lenguaje PASCAL. Sin embargo, es interesante haber hecho todo estudio, porque nos va a permitir comprobar, tal como decíamos antes, el impacto de las decisiones que el diseñador del compilador toma acerca de la estructura del programa objeto. En este caso, concretamente, son la utilización de un registro, como es *TopePila*, para marcar la cumbre de los registros de activación, y la diferencia estructural, por pequeña que pareciera, entre los registros de activación de procedimientos y funciones.

En efecto, vamos a ver que diferentes soluciones al problema encontrado pasan por una modificación del modelo dinámico estándar elegido para el lenguaje que incide en uno de los dos aspectos señalados. Obviamente, la primera alternativa sería elegir una misma estructura para los registros de activación de procedimientos y funciones, de acuerdo a lo ya comentado en el apartado 5.1.2. Según ello, el registro de activación de un procedimiento tendría también un espacio para el valor de retorno, del mismo tamaño que el correspondiente a las funciones. Este espacio nunca sería utilizado realmente por un procedimiento, así que la permanencia en ese lugar de un valor correspondiente a la función llamadora en una activación final mixta no sería

ningún problema. Obsérvese que el coste temporal de este modelo alternativo es exactamente idéntico al original, y únicamente se paga un precio en espacio de memoria que a veces se reserva inútilmente, pero es un precio ciertamente ínfimo, máxime cuando incluso el propio compilador puede verse notablemente beneficiado, por la uniformidad que se consigue. Por otro lado, es evidente que las optimaciones obtenidas hasta el momento no se ven alteradas en lo más mínimo, así que estudiaremos finalmente la optimación de las activaciones finales mixtas siguiendo esta alternativa, pero antes echemos un vistazo a otras.

Otra posibilidad realmente sencilla consistiría en que una función, al retornar, dejara el registro *TopePila* apuntando *encima* de su valor de retorno, no directamente a él. El subprograma llamador de la función debería tomar el valor de retorno, llevarlo a la localidad adecuada de su propio registro de activación, y después decrementar *TopePila* en el tamaño reservado para el valor de retorno. Esto no es nada complejo, a pesar de que tiene asociado un pequeño coste dinámico, al incorporar una instrucción adicional en el código que no existe en el modelo estándar considerado hasta ahora. La optimación de las activaciones finales mixtas sería entonces posible tal como se ha estudiado a lo largo del apartado, y se puede ver fácilmente que las optimaciones de las otras clases de activaciones finales mantienen su validez de forma plena, así que podría ser también una alternativa a considerar.

La tercera y última alternativa incide de lleno en el propio uso del registro *TopePila*. Realmente, si no existen datos de longitud variable no es necesario el uso de tal apuntador [Aho, Sethi y Ullman 1990: 416-422], tal como se indicó en el apartado 2.5.1, pudiéndose llevar a cabo todas las tareas sin mayor dificultad, a expensas, eso sí, de una pequeña complejidad adicional en el compilador. Entonces, cada subprograma sería capaz de establecer la posición del final de su registro de activación, con lo que desaparece sin más el problema planteado, y la optimación sería posible. Sin embargo, el lenguaje PASCAL admite en el denominado «nivel 1» los *arrays* de longitud variable, los llamados «*conformant-arrays*» [Jensen y Wirth 1991: 200]. Por ello, la solución mencionada puede ser solo posible en una implementación del lenguaje PASCAL que sea de «nivel 0», es decir, que no hace uso de esa clase de parámetros. Nuestra implementación es de nivel 0, pero no hay problema, en principio, en aplicar la optimación de las activaciones finales a una implementación de

nivel 1, por lo que la solución de eliminar el registro `TopePila` puede no ser admisible.

En definitiva, concluimos que la simple elección de una estructura uniforme para los registros de activación de todos los subprogramas, concretamente añadiendo para los procedimientos un espacio del mismo tamaño que el valor de retorno de las funciones, y que normalmente no se usará, posibilita la optimación de las que se han llamado activaciones finales mixtas, de un procedimiento desde una función en posición final. Suponiendo entonces este modelo dinámico alternativo, se podrían enunciar el resto de condiciones de la optimación, que no van a ser otras que las obtenidas para las activaciones finales no recursivas de procedimientos y funciones, por lo que no las repetiremos.

Como resultado de estas condiciones y de la uniformización de los registros de activación de procedimientos y funciones, la secuencia de llamada optimada (Algoritmo 5.5.1 a continuación) es muy simple, también básicamente similar a la correspondiente a activaciones finales no recursivas de procedimientos, pero incluyendo la verificación adicional del valor de retorno, como ya concluimos antes. De nuevo, es claro que se obtendrá una mejora temporal del proceso respecto al comportamiento normal, no tan drástica como en optimaciones anteriores, pero aún visible, que ha de añadirse a la propia mejora espacial que resulta de la no acumulación de un nuevo registro de activación en la pila de control.

- LO1: si el procedimiento receptor es un parámetro formal procedimental de la función que efectúa la llamada, esta coloca en lugar seguro el enlace de acceso y la dirección de entrada asociados a dicho parámetro formal procedimental; después, evalúa los parámetros actuales, y coloca los valores obtenidos en el lugar correspondiente de su propio registro de activación, mediante desplazamientos respecto a *BasePila*;
- LO2: la función llamadora verifica su valor de retorno, señalando un error si no está definido;
- LO3: la función llamadora calcula el nuevo enlace de acceso, si se usa la cadena estática, y lo coloca en el lugar correspondiente de su propio registro de activación; si se usa el *display*, restablece el registro de *display* que le corresponde, según se haría normalmente en su propia secuencia de retorno, coloca en el lugar correspondiente de su propio registro de activación el registro de *display* a guardar para la profundidad de anidamiento del receptor, y actualiza dicho registro;
- LO4: la función llamadora salta (jump) a la primera instrucción del código del procedimiento receptor;
- LO5: el procedimiento receptor actualiza *TopePila* a su nuevo valor, que es *BasePila* más el tamaño de la zona de datos;
- LO6: el procedimiento receptor asigna valores iniciales a sus datos locales.

Algoritmo 5.5.1 Secuencia de llamada optimada para activaciones finales mixtas

6. EVALUACIÓN DE LA OPTIMACIÓN

6.1 INTRODUCCIÓN

Este capítulo presenta el estudio del impacto estático y dinámico de la optimación. Esto es, se estudiarán en profundidad, por un lado, las mejoras espaciales y temporales que incorporan los programas generados por un compilador que realice la optimación, y por otro lado, los requisitos que ha de cumplir ese mismo compilador para detectar las activaciones de procedimientos y funciones que puedan resultar optimadas y generar el código adecuado. Este estudio se realiza en las dos secciones a continuación, y aunque trata de ser lo suficientemente general, ha de basarse lógicamente en determinados momentos en elecciones de diseño particulares, que permitan la evaluación cuantitativa exacta de los aspectos considerados. Como quiera que otros compiladores pueden desviarse de las soluciones elegidas, una tercera sección muestra un pequeño estudio de las modificaciones que serían necesarias para llevar a cabo la optimación ante implementaciones del lenguaje diferentes de la considerada. Finalmente, se presentan las características principales del compilador que se ha construido para acompañar de forma práctica el trabajo teórico desarrollado.

6.2 COMPORTAMIENTO DINÁMICO OPTIMADO

Esta sección tiene por objeto determinar definitivamente, desde el punto de vista dinámico, las mejoras introducidas en los programas optimados, principalmente en comparación con el comportamiento estándar. En la introducción del capítulo 3 ya se avanzaron las ideas básicas, cualitativamente

hablando, sobre todo en lo referente a las activaciones recursivas: un comportamiento en espacio constante, por un lado, y una mejora temporal significativa (aunque dependiente de la profundidad de la recursividad). En lo siguiente se estudiarán estos aspectos a la luz de las secuencias de llamada optimadas definitivas, y se fijarán además cuantitativamente las mejoras.

Desde el punto de vista espacial, realmente no hay mucho más que decir. Una secuencia de k activaciones finales recursivas consume exactamente k registros de activación en el modelo dinámico estándar, mientras que solo existe un registro de activación en el modelo dinámico optimado, si todas las activaciones pueden resultar optimadas. Obsérvese que este número k puede ser tan grande como se quiera, incluso potencialmente infinito; de esta manera, es posible crear procesos repetitivos ilimitados con ayuda de la recursividad final, sin necesidad de recurrir al esquema iterativo clásico:

```
while true do ...
```

con lo que puede enfatizarse la relación entre iteratividad y recursividad, algo que impide el modelo estándar, por el mero límite de la memoria.

En cuanto a las activaciones finales no recursivas, la mejora espacial dependerá de la clase de proceso en que aparezcan. Desde luego, los más interesantes son los que corresponden a procesos recursivos expresados mediante subprogramas con recursividad indirecta, como en el Programa 4.3.1; en esos casos, sigue siendo válido el resultado anterior: todo el proceso se ejecuta en espacio constante, en un único registro de activación; quizá el tamaño del registro de activación puede tener ligeras modificaciones a lo largo del proceso, si los subprogramas involucrados tienen diferente tamaño de datos locales, temporales o parámetros formales, pero la base del registro de activación es constantemente la misma.

Por supuesto, esta característica espacial del comportamiento dinámico optimado es, a pesar de su sencillez, donde radica su interés principal, por las drásticas diferencias que pueden darse respecto al comportamiento dinámico estándar, y por las alternativas que ofrece para la metodología de programación en PASCAL, que estudiaremos más adelante. De hecho, recordemos que es en esta característica espacial donde se coloca el énfasis en la definición de los lenguajes que son propiamente recursivos finales:

«Implementations of Scheme are required to be properly tail-recursive. This allows the execution of an iterative computation in constant space, even if the iterative computation is described by a syntactically recursive procedure» [Clinger y Rees 1991: 3]

Además, está claro que esta mejora es independiente del modelo dinámico optimado que se elija como alternativa al estándar, con lo que podría incluirse en la definición del lenguaje PASCAL de la misma forma que en el lenguaje SCHEME, con la salvedad, eso sí, de las restricciones encontradas que impiden sintácticamente la optimación.

Sin embargo, como ya hemos avanzado, existe un interés adicional del comportamiento dinámico optimado, que radica en una posible mejora temporal en comparación con el comportamiento estándar, lo cual pasamos a estudiar a continuación. Esta mejora temporal, al contrario que la anterior, depende directamente del modelo dinámico estándar elegido para el lenguaje PASCAL, con lo que podría darse el caso de que una determinada organización del programa objeto condujera a una mejora poco apreciable, aunque también ocurre que una elección adecuada lleva a un comportamiento considerablemente más rápido. Estudiaremos cuantitativamente la mejora temporal que se obtiene con el modelo dinámico elegido para el trabajo en el apartado 2.5.1, que va a resultar posiblemente la más drástica.

En primer lugar, definimos la máquina objeto para la cual se genera el código. Hemos supuesto, siguiendo las ideas de Aho, Sethi y Ullman [1990: 535-538] y otras referencias, un computador CISC genérico, direccionable por *bytes*, con palabra de dos *bytes*, y un cierto número de registros de cuatro *bytes*, aunque esta elección de tamaño es más que nada una aproximación al tipo de máquina más habitual en los últimos años, no siendo especialmente necesaria para nuestro propósito. Nos interesan principalmente las operaciones de movimiento de datos, expresadas en el lenguaje objeto adoptado por:

$$\langle destino \rangle := \langle fuente \rangle$$

En general, debería indicarse el tamaño del dato a mover, pero para nuestro problema basta con suponer que se trata de direcciones de memoria (cuatro *bytes*) en todos los casos. Del mismo modo, consideraremos aritmética de direcciones:

$$\langle destino \rangle := \langle fuente1 \rangle \pm \langle fuente2 \rangle$$

Finalmente, expresamos la instrucción de salto incondicional por:

IR-A (*dirección*)

Los operandos de estas instrucciones se pueden elegir entre un dato literal, que se escribe precedido de #, incluso si representan direcciones absolutas de memoria, o un direccionamiento por registro. Se permite, para facilitar la generación de código, el uso de etiquetas para señalar direcciones, que se especifican con la pseudoinstrucción:

etiqueta (*nombre-de-etiqueta*)

de forma que un operando que no esté en posición de lado izquierdo puede ser una etiqueta.

Los direccionamientos admisibles recogen el repertorio necesario para la generación de código, presente en todas las máquinas actuales:

- *inmediato*, representado simplemente por el nombre del registro: el dato es el contenido del registro;
- *directo*, representado por [*(registro)*]: el dato está en la posición de memoria *contenido((registro))*;
- *directo indexado*, representado por [*(registro)±#(despl)*], donde *#(despl)* es una dirección que especifica un desplazamiento relativo: el dato está en la posición de memoria *contenido((registro))±#(despl)*;
- *indirecto*, representado por [[*(registro)*]]: el dato está en la posición de memoria *contenido(contenido((registro)))*;
- *indirecto indexado*, representado por [[*(registro)±#(despl)*]]: el dato está en la posición de memoria *contenido(contenido((registro))±#(despl))*;

Suponemos que, dada la organización del programa objeto para el lenguaje PASCAL, se han reservado tres registros de direcciones para el manejo de la pila de control, cuyo nombre se particulariza de la siguiente manera:

- *Datos* señala el comienzo de la pila, y por lo tanto, sirve para el acceso directo a los datos globales⁴;
- *BasePila* señala en cada momento el comienzo del registro de activación de la cumbre de la pila; recordemos que se eligió (Figura 2.5.2) que

⁴ No es estrictamente necesario inutilizar un registro a lo largo de todo el programa para este propósito, pero el acceso a los datos resulta notablemente mejorado, a la luz de ciertos estudios [Fischer y LeBlanc 1988: 307].

BasePila apuntara por encima de la zona de control del registro de activación actual, aunque también podría elegirse que apuntara exactamente a su comienzo;

- TopePila señala en cada momento la primera posición libre de la pila, justo por encima del registro de activación actual; recordamos (cf. apartado 2.5.1) que el uso de este registro es obligatorio si existen datos de longitud variable en el lenguaje, pero en cualquier caso puede simplificar la tarea.

Si se usa la estrategia de la cadena estática para el acceso a los datos no locales, que es lo adecuado en PASCAL, supondremos que se utiliza un registro de direcciones adicional, denominado específicamente Enlace, para el cálculo del enlace de acceso en cada activación, registro que por lo demás, podría ser usado para otros cálculos en cualquier otro momento, al contrario que Datos, BasePila y TopePila.

También es posible el uso del *display* para conseguir el mismo objetivo. Sobre la colocación de la matriz de *display* existen varias alternativas [Aho, Sethi y Ullman 1990: 436]: mantenerla en registros (si hay un número suficiente) o situarla en memoria y acceder de modo indexado a través de un único registro apuntador al comienzo de la matriz. Contrariamente a lo que mantienen los anteriores autores [*ibid.*], es difícil que un compilador de PASCAL, habitualmente de un solo paso, pueda conocer con antelación a la generación de código el tamaño máximo de la matriz de *display*, así que colocar sus elementos en registros puede no ser factible, amén del reducido número de estos, y a pesar de que el compilador pueda imponer una limitación en la profundidad máxima de anidamiento, como afirman Fischer y LeBlanc [1988: 294]. Al colocar la matriz de *display* en la zona global de memoria, señalada con un registro, puede darse una limitación mucho más relajada y difícil de ser alcanzada, por ejemplo de 50, sin más coste adicional que unas pocas direcciones de memoria. A pesar de todo, y para asegurar que la mejora de los programas optimados se establece respecto a los programas que mejor se comportan en el modelo estándar, mantendremos la matriz en registros, que denominaremos Display1, Display2, etc.

La evaluación cuantitativa de los programas se basa en los costes asociados a las instrucciones, tanto espaciales como temporales. En principio, no hemos considerado importante, y por eso no lo hemos incluido entre los aspectos a

estudiar de la optimación, el tamaño del programa objeto generado, por ser una cuestión que cada vez suscita menos problemas dado el gran avance producido en el *hardware* en los últimos años. En cuanto a los costes temporales de las instrucciones, pueden surgir varias dificultades para obtener un resultado exacto. En primer lugar, paradójica pero sensatamente, Aho, Sethi y Ullman [1990: 537] proponen aproximar el coste temporal de las instrucciones por su coste espacial, debido a que el tiempo para traer la instrucción de la memoria puede exceder el tiempo necesario para su ejecución; de este modo, es lógico pensar que instrucciones más largas (que necesiten palabras adicionales para los operandos) requieran más tiempo. Este será nuestro primer criterio de medida.

En segundo lugar, podrían elegirse diversas máquinas concretas para establecer tiempos exactos; al ser un estudio comparativo entre programas similares, no se perdería generalidad. Sin embargo, la evaluación exacta puede ser difícil debido a varios factores. Así, para el MC68020 [Motorola 1985: 9/1-9/3], la existencia de *caché*, la desalineación de instrucciones y el solapamiento de la ejecución de instrucciones puede variar significativamente el tiempo empleado para la ejecución de una serie de ellas. Concretamente, Motorola [*op. cit.*: 9/10] explícitamente advierte que las tablas de coste temporal de las instrucciones no pueden predecir exactamente el comportamiento de un determinado programa; sin embargo, ofreciendo los valores del "mejor caso" (en que los factores anteriormente mencionados son todos favorables) y del "peor caso" (en que son desfavorables) pueden establecerse las cotas inferior y superior del número de ciclos de reloj que invertirá el procesador en ejecutar una secuencia de instrucciones.

La evaluación de las instrucciones del lenguaje máquina MC68020 que serían generadas por el compilador si esa fuera la máquina objeto será nuestro segundo criterio de medida, incluyendo como subcriterios la consideración de los valores de "mejor caso" y "peor caso". Hemos de decir que la amplia información ofrecida para el MC68020 en cuanto a la evaluación temporal de las instrucciones contrasta con la proporcionada por otros fabricantes, por ejemplo, en el caso del 80386 [Intel 1989] (concretamente, solo se indican los tiempos de ejecución de las instrucciones, sin informar de los tiempos de carga ni de cálculo de direcciones efectivas), debido a lo cual no hemos considerado posible utilizar esta otra máquina. De todos modos, las buenas características propias del MC68020 como procesador moderno y fácilmente adaptable a numerosas

situaciones, unidas a su amplia difusión en muy diversos entornos, desde el ordenador personal hasta las estaciones de trabajo, hacen que consideremos como suficiente su elección para la evaluación comparativa del código.

Entonces, según el primer criterio de Aho, Sethi y Ullman, el coste de una instrucción es uno más el coste asociado a los direccionamientos de los operandos; este coste asociado es uno para los direccionamientos indexados, pues el desplazamiento ocupa una palabra adicional. En segundo lugar, expresaremos el código elegido en el lenguaje máquina del MC68020, y usaremos las tablas de cálculo temporal, en sus dos variantes de "mejor caso" y "peor caso" [Motorola 1985: 9/11-9/28], para establecer el coste de cada programa. La evaluación de un repertorio particular de instrucciones según ambos criterios se muestra en la Tabla 6.2.1. En ella, r y r' representan nombres de registros, $\#dir$ es una dirección inmediata (doble palabra), y $\#d$ y $\#d'$ son desplazamientos relativos inmediatos (se supone que ocupan una palabra, porque las versiones con doble palabra son significativamente más costosas).

Instrucción	Coste criterio 1	Instrucción MC68020	Coste criterio 2	
			(a) mejor caso	(b) peor caso
$r := r'$	1	MOVEA r', r	0	3
$r := r' \pm \#d'$	3	MOVEA r', r	0	3
		ADDQ $r, \#d'$	0	3
$[r \pm \#d] := r'$	2	MOVE $r', (\#d, r)$	3	7
$[r \pm \#d] := [r' \pm \#d']$	3	MOVE $(\#d', r'), (\#d, r)$	6	13
$r := [r' \pm \#d']$	2	MOVE $(\#d', r'), r$	3	9
$[r \pm \#d] := \#dir$	3	MOVE $\#dir, (\#d, r)$	6	14
IR-A $\#dir$	2	JMP $\#dir$	1	9
IR-A $[r \pm \#d]$	2	JMP $(\#d, r)$	2	11

Tabla 6.2.1

Costes de un repertorio de instrucciones del lenguaje máquina genérico, según dos criterios: (1) el coste definido según Aho, Sethi y Ullman [1990: 536-537], (2) el coste exacto de las instrucciones equivalentes del MC68020 [Motorola 1985: 9/11-9/28] bajo suposición de (a) "mejor caso", y (b) "peor caso"

Bajo estos supuestos, comenzamos el estudio temporal comparativo de los programas optimados respecto a los no optimados.

6.2.1 Activaciones finales recursivas de procedimientos

Estudiamos primero la optimización de las activaciones recursivas finales de un procedimiento, tal como se estudió a lo largo del capítulo 3. Para empezar, mostramos el código generado típicamente para un procedimiento según el modelo dinámico estándar, siguiendo las secuencias de llamada y retorno que aparecen en el Algoritmo 2.5.1 y el Algoritmo 2.5.2, respectivamente.

```

; punto de entrada al procedimiento
etiqueta (punto-entrada-proc)
; parte de la secuencia de llamada como receptor
; L7: actualización de TopePila
(1) TopePila := BasePila + #(tamaño-zona-datos-proc)
; L8: inicialización de datos locales
etiqueta (punto-entrada-alternativo-proc)
...
; fin de la secuencia de llamada como receptor
; comienzo del código del procedimiento
...
; secuencia de llamada correspondiente a una activación recursiva final
; L1: evaluación de los parámetros actuales
...
; L2: enlace de control
(2) [TopePila + #(CNTL-LINK-PROC)] := BasePila
; L3: enlace de acceso
(3) [TopePila + #(ACCS-LINK-PROC)] := [BasePila + #(ACCS-LINK-LOC)]
; L4: dirección de retorno
(4) [TopePila + #(DIR-RET-PROC)] := (etiqueta-retorno)
; L5: actualiza BasePila
(5) BasePila := TopePila + #(tamaño-zona-control-proc)
; L6: salto al receptor, en este caso, el mismo
(6) IR-A (punto-entrada-proc)
; fin de la secuencia de llamada como llamador
; comienza la secuencia de retorno como llamador, que no incluye nada
etiqueta (etiqueta-retorno)
; continuación del código
...
; la parte final del código del procedimiento es la parte de la secuencia
; de retorno como receptor
; R2: recupera TopePila
(7) TopePila := BasePila - #(tamaño-zona-control-proc)
; R3: recupera BasePila con enlace de control
(8) BasePila := [TopePila + #(CNTL-LINK-PROC)]
; R4: salto a la dirección de retorno
(9) IR-A [TopePila + #(DIR-RET-PROC)]
; fin del código del procedimiento

```

Programa 6.2.1 Esquema del código objeto generado para un procedimiento directamente recursivo según el modelo dinámico estándar, usando la cadena estática

Por contra, si se produce la optimación, siguiendo la secuencia de llamada optimada (Algoritmo 3.2.1), el código del procedimiento quedaría como en el siguiente programa; se han mantenido los números de las instrucciones según aparecen en el Programa 6.2.1, pero se ha eliminado, para simplificar, la secuencia de retorno, que por lo demás, es idéntica a la del primer programa, debido a la Condición 3.2.1.

```

                                ; punto de entrada al procedimiento
etiqueta (punto-entrada-proc)
                                ; parte de la secuencia de llamada como receptor
                                ; L7: actualización de TopePila
(1) TopePila := BasePila + #(<tamaño-zona-datos-proc>)
                                ; L8: inicialización de datos locales
etiqueta (punto-entrada-alternativo-proc)
...
                                ; fin de la secuencia de llamada como receptor
                                ; comienzo del código del procedimiento
...
                                ; secuencia de llamada para una activación recursiva final optimada
                                ; LO1: evaluación de los parámetros actuales
...
                                ; LO2: salto al receptor, en este caso, el mismo, pero sin paso L7
(6) IR-A (punto-entrada-alternativo-proc)
                                ; fin de la secuencia de llamada optimada como llamador
                                ; comienza la secuencia de retorno como llamador, que no incluye nada
etiqueta (etiqueta-retorno)
                                ; continuación del código
...
                                ; la parte final del código del procedimiento es la parte de la secuencia
                                ; de retorno como receptor
...
                                ; fin del código del procedimiento

```

Programa 6.2.2 Esquema del código objeto generado para un procedimiento directamente recursivo según el modelo dinámico optimado, usando la cadena estática

Se ha supuesto que se utiliza la cadena estática para el acceso a los datos no locales. Todos los datos literales son conocidos para el compilador, y se nombran para facilitar la comprensión de los mismos; así, #(<ACCS-LINK-LOC>) es el desplazamiento necesario para situar el enlace de acceso de un registro de activación a partir del registro BasePila, mientras que #(<ACCS-LINK-PROC>) es el desplazamiento necesario para situar el enlace de acceso de un registro de activación a partir del valor de TopePila para el registro de activación que está por debajo. Lo mismo puede decirse para el enlace de control y la dirección de retorno.

Para el cálculo del enlace de acceso en el Programa 6.2.1 se ha supuesto que el compilador optimiza, ya en el modelo dinámico estándar, el seguimiento de la cadena estática: recordemos (cf. apartado 4.2.2) que si la distancia léxica de la llamada a un procedimiento es la unidad, o lo que es lo mismo, si el procedimiento llamador tiene la misma profundidad de anidamiento que el procedimiento receptor (como es el caso en una activación recursiva como la considerada), entonces el enlace de acceso del receptor es el mismo que el enlace de acceso del llamador, por lo que basta con tomarlo del registro de activación de este último; de ahí la instrucción (3) del Programa 6.2.1.

Con ayuda de la Tabla 6.2.1, puede calcularse el coste de cada instrucción, y con ello los costes de las llamadas completas en el modelo estándar y en el modelo optimado, que denominaremos $c[ce]pr-llam_{est}$ y $c[ce]pr-llam_{opt}$, respectivamente. La secuencia de llamada estándar está formada, olvidándonos de momento de la evaluación de los parámetros actuales y su asignación a los formales, por las instrucciones (2) a (6) seguidas de la (1) del Programa 6.2.1; en la secuencia de llamada optimada solo se ejecuta la instrucción (6) del Programa 6.2.2 (recordemos que se salta al punto de entrada alternativo del procedimiento, pues tampoco es necesaria la actualización de *TopoPila*). Los costes en cada modelo dinámico se recogen en las dos siguientes tablas.

Secuencia de llamada estándar	Coste según criterio 1	Coste según criterio 2a	Coste según criterio 2b
(2)	2	3	7
(3)	3	6	13
(4)	3	6	14
(5)	3	0	6
(6)	2	1	9
(1)	3	0	6
$c[ce]pr-llam_{est}$	16	16	55

Tabla 6.2.2

Coste de una secuencia de llamada recursiva final de procedimiento (usando la cadena estática) con el modelo dinámico estándar (los números de instrucciones se refieren al Programa 6.2.1)

Secuencia de llamada optimada	Coste según criterio 1	Coste según criterio 2a	Coste según criterio 2b
(6)	2	1	9
$c[ce]pr-llam_{opt}$	2	1	9

Tabla 6.2.3 Coste de una secuencia de llamada recursiva final de procedimiento (usando la cadena estática) con el modelo dinámico optimado (los números de instrucciones se refieren al Programa 6.2.2)

Podríamos definir entonces la razón de optimación en una llamada, que denominaríamos $r-opt[ce]pr_{llam}$, como el cociente entre los costes de las secuencias de llamada en cada modelo dinámico:

$$r-opt[ce]pr_{llam} = \frac{c[ce]pr-llam_{est}}{c[ce]pr-llam_{opt}}$$

De ser así, la razón de optimación resultaría en unos valores de 8 según el criterio aproximado de costes de Aho, Sethi y Ullman [1990], y entre 16 (!) y un poco más de 6 con el criterio, en principio más exacto, de suponer código del MC68020.

	Coste según criterio 1	Coste según criterio 2a	Coste según criterio 2b
$c[ce]pr-llam_{est}$	16	16	55
$c[ce]pr-llam_{opt}$	2	1	9
$r-opt[ce]pr_{llam}$	8	16	$6\frac{1}{2}$

Tabla 6.2.4 Razón de optimación (usando la cadena estática) para una llamada final recursiva de procedimiento

Sin embargo, pensamos que no debemos establecer la comparación solamente en este proceso particular de llamada; en efecto, sabemos que la optimación evita, por cada activación, la realización de la secuencia de retorno correspondiente al llamador, así que ese aspecto ha de tenerse en cuenta. Llamamos $c[ce]p-ret$ al coste de la secuencia de retorno (que sirve, si se usa la cadena estática, para cualquier procedimiento independientemente del modelo dinámico que se considere, por la Condición 3.2.1); este proceso está formado por las instrucciones (7) a (9) del Programa 6.2.1. Su coste se recoge en la Tabla 6.2.5.

Secuencia de retorno	Coste según criterio 1	Coste según criterio 2a	Coste según criterio 2b
(7)	3	0	6
(8)	2	3	9
(9)	2	2	11
$c[ce]p-ret$	7	5	26

Tabla 6.2.5 Coste de una secuencia de retorno de procedimiento (usando la cadena estática) en los dos modelos dinámicos (los números de instrucciones se refieren al Programa 6.2.1)

En tal caso, la razón de optimación de un proceso de activación recursiva final completo está dada por:

$$r-opt[ce]pr_{llam-ret} = \frac{c[ce]pr-llam_{est} + c[ce]p-ret}{c[ce]pr-llam_{opt}} \quad [6.2.1]$$

pues no se realiza la secuencia de retorno correspondiente al llamador. Por lo tanto, podemos plantear las siguientes cifras según los diferentes criterios de coste elegidos.

	Coste según criterio 1	Coste según criterio 2a	Coste según criterio 2b
$c[ce]pr-llam_{est}$	16	16	55
$c[ce]p-ret$	7	5	26
$c[ce]pr-llam_{opt}$	2	1	9
$r-opt[ce]pr_{llam-ret}$	11'5	21	9

Tabla 6.2.6 Razón de optimación (usando la cadena estática) para un proceso completo de activación recursiva final de un procedimiento

Las cifras son ciertamente sorprendentes. No vamos a insistir, desde luego, en supeditar nuestras posturas en el valor de 21 que se obtiene con el criterio 2a, a pesar de que pueda ser perfectamente plausible. Pensamos que una medida más aplicable a otro tipo de situaciones puede ser la de los otros dos criterios, que se muestran similares. Aun así, creemos no pecar de vanidad al decir que una razón de 10 es una optimación realmente importante.

Debe recalcar que las cifras anteriores se pueden aplicar tal como aparecen a cualquier secuencia de activaciones recursivas finales consecutivas. En efecto, ante una secuencia de profundidad k de llamadas finales del mismo procedimiento, en que se considera que solo la primera es especial (la que comienza el proceso, realizada por otro subprograma), existen $k-1$ activaciones recursivas finales. En el modelo dinámico estándar, existen $k-1$ llamadas y $k-1$ retornos, mientras que en el modelo optimado, solo existen $k-1$ llamadas optimadas, y el único retorno se puede considerar como el que corresponde a la primera activación, no recursiva, por lo que la razón de optimación del proceso completo sigue siendo la obtenida.

Para hacerse una idea del alcance de estos resultados, si se considera un procedimiento recursivo final que se activa 10 veces, entonces los procesos de llamada y retorno ocupan 810 ciclos de reloj en el modelo estándar, mientras que se ejecutan en 90 ciclos en el modelo dinámico (habiendo tomado el criterio 2b para esta comparación). Y evidentemente, cuanto mayor sea la profundidad de la recursividad, más drásticamente se mejorará el comportamiento del programa optimado en términos absolutos.

El uso del *display* para el acceso a los nombres no locales va a conducir a resultados similares. El código generado con el modelo dinámico estándar para una activación de procedimiento será el que aparece en el Programa 6.2.3 a continuación, en donde *Display<profundidad>* representa el registro de *display* de la profundidad de anidamiento que corresponde al procedimiento. Realmente, este esquema se va a aplicar a todas las activaciones de procedimientos en el modelo estándar, si se usa el *display*, ya que no existe mejora en el caso particular en que los procedimientos involucrados tienen el mismo anidamiento, como ocurría con el enlace de acceso.

La optimación de la llamada radica en este caso, además de en evitar las operaciones ya conocidas, en evitar también la actualización del *display*, por lo que el código del procedimiento se transforma en el Programa 6.2.4 que se recoge después. En él no se incluye, para simplificar como en el caso precedente, la secuencia de retorno, que será exactamente la misma que aparece al final del Programa 6.2.3.

```

; punto de entrada al procedimiento
etiqueta (punto-entrada-proc)
; parte de la secuencia de llamada como receptor
; L7: actualización de TopePila
(1) TopePila := BasePila + #(tamaño-zona-datos-proc)
; L8: inicialización de datos locales
etiqueta (punto-entrada-alternativo-proc)
...
; fin de la secuencia de llamada como receptor
; comienzo del código del procedimiento
...
; secuencia de llamada correspondiente a una activación recursiva final
; L1: evaluación de los parámetros actuales
...
; L2: enlace de control
(2) [TopePila + #(CNTL-LINK-PROC)] := BasePila
; L3: display guardado
(3) [TopePila + #(DISPLAY-PROC)] := Display<profundidad>
; L4: dirección de retorno
(4) [TopePila + #(DIR-RET-PROC)] := (etiqueta-retorno)
; L5(a): actualiza BasePila
(5) BasePila := TopePila + #(tamaño-zona-control-proc)
; L5(b): actualiza display
(6) Display<profundidad> := BasePila
; L6: salto al receptor, en este caso, el mismo
(7) IR-A (punto-entrada-proc)
; fin de la secuencia de llamada como llamador
; comienza la secuencia de retorno como llamador, que no incluye nada
etiqueta (etiqueta-retorno)
; continuación del código
...
; la parte final del código del procedimiento es la parte de la secuencia
; de retorno como receptor
; R1: recupera display
(8) Display<profundidad> := [TopePila + #(DISPLAY-PROC)]
; R2: recupera TopePila
(9) TopePila := BasePila - #(tamaño-zona-control-proc)
; R3: recupera BasePila con enlace de control
(10) BasePila := [TopePila + #(CNTL-LINK-PROC)]
; R4: salto a la dirección de retorno
(11) IR-A [TopePila + #(DIR-RET-PROC)]
; fin del código del procedimiento

```

Programa 6.2.3 Esquema del código objeto generado para un procedimiento directamente recursivo según el modelo dinámico estándar, usando el *display*

```

; punto de entrada al procedimiento
etiqueta (punto-entrada-proc)
; parte de la secuencia de llamada como receptor
; L7: actualización de TopePila
(1) TopePila := BasePila + #(< tamaño-zona-datos-proc >)
; L8: inicialización de datos locales
etiqueta (punto-entrada-alternativo-proc)
...
; fin de la secuencia de llamada como receptor
; comienzo del código del procedimiento

...
; secuencia de llamada para una activación recursiva final optimada
; LO1: evaluación de los parámetros actuales

...
; LO2: salto al receptor, en este caso, el mismo
(7) IR-A (punto-entrada-alternativo-proc)
; fin de la secuencia de llamada como llamador
; comienza la secuencia de retorno como llamador, que no incluye nada
etiqueta (etiqueta-retorno)
; continuación del código

...
; la parte final del código del procedimiento es la parte de la secuencia
; de retorno como receptor

...
; fin del código del procedimiento

```

Programa 6.2.4 Esquema del código objeto generado para un procedimiento directamente recursivo según el modelo dinámico optimado, usando el *display*

La misma clase de cálculos que se han hecho para el caso de la cadena estática permite construir la Tabla 6.2.7; el coste de la secuencia de llamada estándar con el *display* es el mismo independientemente de si la activación es recursiva o no, como hemos dicho.

	Coste según criterio 1	Coste según criterio 2a	Coste según criterio 2b
$c[d]p\text{-}llam_{est}$	16	13	52
$c[d]pr\text{-}llam_{opt}$	2	1	9
$r\text{-}opt[d]pr_{llam}$	8	13	57

Tabla 6.2.7 Razón de optimación (usando el *display*) para una llamada final recursiva de procedimiento

Con el *display*, la secuencia de llamada es un poco menos costosa (aunque con el criterio 1, más aproximado, se obtiene el mismo coste, y en el criterio 2 existe muy poca diferencia); por ello, la razón de optimación es un poco más baja. Sin embargo, el uso del *display* penaliza las secuencias de retorno:

Secuencia de retorno	Coste según criterio 1	Coste según criterio 2a	Coste según criterio 2b
(8)	2	3	9
(9)	3	0	6
(10)	2	3	9
(11)	2	2	11
$c[d]p-ret$	9	8	35

Tabla 6.2.8 Coste de una secuencia de retorno de procedimiento (usando el *display*) en los dos modelos dinámicos (los números de instrucciones se relieren al Programa 6.2.3)

con lo cual se puede obtener la razón de optimación para una activación completa (aplicando la ecuación [6.2.1] antes conseguida):

	Coste según criterio 1	Coste según criterio 2a	Coste según criterio 2b
$c[d]p-llam_{est}$	16	13	52
$c[d]p-ret$	9	8	35
$c[d]pr-llam_{opt}$	2	1	9
$r-opt[d]pr_{llam-ret}$	12'5	21	9'6

Tabla 6.2.9 Razón de optimación (usando el *display*) para un proceso completo de activación recursiva final de procedimiento

Salvo con el criterio 2a, la razón de optimación se ha incrementado en una unidad, algo lógico al evitarse justamente las secuencias de retorno, que son ahora más costosas. Incluso, si la matriz de *display* tuviera que colocarse en memoria global, estas cifras se verían notablemente incrementadas:

	Coste según criterio 1	Coste según criterio 2a	Coste según criterio 2b
$c[dm]p-llam_{est}$	18	19	62
$c[dm]p-ret$	10	11	37
$c[dm]pr-llam_{opt}$	2	1	9
$r-opt[dm]pr_{llam}$	9	19	6'8
$r-opt[dm]pr_{llam-ret}$	14	30	11

Tabla 6.2.10 Razón de optimación (con el *display* en memoria) en un proceso de activación recursiva final de procedimiento

Estamos tentados de decir que la razón de optimación conseguida no precisa casi ningún comentario más. Sin embargo, no es así, y es necesario justo uno muy importante. Todo lo anterior (tanto en el caso de la cadena estática como en este último del *display*) representa la relación de mejora en el tiempo que consumen las operaciones propias de la secuencia de llamada y retorno de activaciones finales recursivas de procedimientos. Por supuesto, este resultado no puede extenderse, y ello debe recalcar y recordarse también para todos los demás casos que evaluemos, a todo el proceso de activación (una o más veces) del procedimiento. Concretamente, si el cuerpo del procedimiento es relativamente simple, es muy probable que la mejora de los procesos de llamada y retorno provoque una mejora visible del tiempo de ejecución de todo el programa; sin embargo, si el procedimiento es complejo, es también posible que esta mejora no se haga patente o represente un porcentaje ínfimo del tiempo de ejecución total de cada activación.

En este sentido, hemos omitido deliberadamente dos aspectos fundamentales de la llamada, como son la evaluación de los parámetros actuales y su asignación a los parámetros formales, por un lado, y la inicialización de los datos locales, por otro. Lo mismo que con el código, la complejidad de estos procesos será determinante del efecto total de la mejora de las secuencias de llamada. Para ser exactos, la razón de optimación real, observable a partir de la ejecución del programa, debería calcularse por la ecuación:

$$r\text{-opt}[\bullet]pr_{\text{real}} = \frac{c[\bullet]pr\text{-}par_{\text{est}} + c[\bullet]pr\text{-}llam_{\text{est}} + c[\bullet]p\text{-}ini + c[\bullet]p\text{-}cod + c[\bullet]p\text{-}ret}{c[\bullet]pr\text{-}par_{\text{opt}} + c[\bullet]pr\text{-}llam_{\text{opt}} + c[\bullet]p\text{-}ini + c[\bullet]p\text{-}cod}$$

donde $c[\bullet]pr\text{-}par$ es el coste de la evaluación de los parámetros actuales y su asignación a los parámetros formales, $c[\bullet]p\text{-}ini$ es el coste de la inicialización de los datos locales, y $c[\bullet]p\text{-}cod$ es el coste del propio código del procedimiento. Desde luego, insistimos en que cuanto más grande sea la suma de estos costes, debido a cualquiera de los factores por separado o a una conjunción de ellos, menor será el efecto de mejora introducido por la optimación de las secuencias de llamada y retorno. Por lo tanto, es a la luz de estas matizaciones que hay que entender las cifras obtenidas a lo largo del apartado, y las que se obtendrán en lo que sigue.

A pesar de ello, hay que resaltar que también pueden manifestarse comportamientos diferentes en la evaluación de los parámetros actuales, para

mejor y para peor, y por ello se han consignado dos costes diferentes en la ecuación anterior ($c[\bullet]pr-par_{est}$ y $c[\bullet]pr-par_{opt}$). Efectivamente, recordemos la Condición 3.2.10, la Condición 3.2.11, la Condición 3.2.12 y las discusiones que condujeron a ellas: no es necesaria la actualización de parámetros en determinadas condiciones bajo el modelo dinámico optimado; por lo tanto, para ciertos procedimientos (aquellos a los que pueda aplicarse esta optimación) es posible que el proceso de llamada se vea mejorado no solo por las operaciones consideradas a lo largo del apartado, sino también debido al hecho de que se evitan algunos procesos de actualización.

Es cierto que estos son en cualquier caso muy sencillos, pues se refieren a un parámetro actual que consta de un solo nombre, pero es indudable que su desaparición mejora el comportamiento temporal. Por ejemplo, si el parámetro es por referencia, realmente no hay mucha mejora, pues en el modelo estándar la actualización se resuelve con la copia de una dirección, que es el objeto típicamente manejado eficientemente por los procesadores. Si el parámetro es por valor, y su tipo no es estructurado, ocurre algo parecido, pero si es estructurado (*array* o *record*, generalmente) la cosa puede cambiar significativamente, más cuanto mayor sea el tamaño del objeto. Con ello, las razones de optimación obtenidas podrían verse incrementadas para los procedimientos que fueran susceptibles de esta otra optimación.

Por el contrario, también es plausible pensar que, a causa de la Condición 3.2.9, la actualización de parámetros resulte más compleja en el modelo optimado que en el estándar, dependiendo de la estrategia elegida por el compilador para ese proceso (debe recapitularse aquí toda la discusión del apartado 3.2.4). En efecto, en el modelo estándar podría seguirse la estrategia de evaluar los parámetros actuales directamente sobre las localidades definitivas, ahorrándose el uso de posiciones temporales. En el modelo optimado, ello no es posible, debido a la condición mencionada: al ser las localidades definitivas de los parámetros formales las mismas que las de los actuales, ha de asegurarse que no se escribe un parámetro formal antes de ser usado en otro parámetro actual; ello obligaría a usar localidades temporales, con lo que el proceso de actualización sería más lento que en el modelo estándar. Y esa mayor lentitud podría contrarrestar, o incluso sobrepasar ampliamente, la mejora producida en el resto de operaciones de la llamada y/o el retorno.

En cuanto al proceso de inicialización de los datos locales, está claro que consumirá el mismo tiempo en cualquiera de los modelos dinámicos. Recordemos que, más que una inicialización, se trata de asignar un valor indefinido a esos datos locales, para cumplir las normas del lenguaje PASCAL acerca del acceso al valor de un dato no definido previamente. Este asunto tiene cierta complejidad intrínseca, y no lo trataremos hasta más adelante, en que será necesario tenerlo en cuenta en relación con las funciones, concretamente con su valor de retorno.

En definitiva, las razones de optimación obtenidas deben entenderse *exclusivamente* como referidas a las operaciones de las secuencias de llamada y retorno que afectan a los elementos de la zona de control del registro de activación (enlace de control, enlace de acceso o *display*, dirección de retorno) y al salto propiamente dicho. La incorporación de la actualización de parámetros puede mejorar esas cifras pero también empeorarlas significativamente. Y en general, el resto de elementos a considerar (los datos locales y el propio código del subprograma) impiden una evaluación exacta del comportamiento del programa completo, por lo que las razones de optimación anteriores *nunca* pueden extrapolarse hasta tal punto, aunque sí considerarse como indicativas de una mejora, más o menos pequeña, de ese comportamiento.

Finalmente, hay otra perspectiva importante bajo la que analizar la mejora que introduce la optimación, y es la comparación con un programa iterativo (semánticamente) equivalente. *A priori*, puede entenderse que un proceso iterativo realiza exactamente las mismas operaciones que se realizan en una secuencia de activaciones recursivas finales optimadas: cada paso del proceso se resuelve con una actualización de las variables involucradas (lo que es equivalente a la actualización de los parámetros en la llamada final optimada) y un salto al comienzo del bucle (similar al salto al comienzo del procedimiento recursivo final). Por otra parte, la eliminación de todas las secuencias de retorno en el proceso recursivo final optimado se corresponde con la terminación de un proceso iterativo una vez que se alcanza la condición de salida. Esto quiere decir que el comportamiento de las activaciones recursivas finales optimadas está muy cerca del comportamiento típico de los programas iterativos, aun admitiendo que algunos detalles (que dependerán de cada caso concreto) puedan hacerlas algo más lentas, pero muy ligeramente. Por lo tanto, al menos la recursividad final directa se ha convertido en una

solución tan eficiente como la iteratividad para la expresión de los programas. En el siguiente apartado veremos que la recursividad final indirecta también puede tener un comportamiento eficiente.

6.2.2 Activaciones finales no recursivas de procedimientos

Estudiamos a continuación el alcance temporal de la optimación de las activaciones finales no recursivas de procedimientos. Consideramos primero que se usa la cadena estática. Recordemos también antes de nada la hipótesis planteada al final del apartado 4.2.2, en el sentido de que parecía haber diferencias interesantes a tener en cuenta en el caso particular de activación no recursiva de un procedimiento con la misma profundidad de anidamiento que el que lo invoca. Allí denominamos a esta clase de llamadas *activaciones casi recursivas*, que se guiaban, en el modelo dinámico optimado, por la secuencia optimada de llamada general del Algoritmo 4.2.1, pero en la cual el paso LO2 no era preciso. Para confirmar o refutar la hipótesis, distinguiremos a partir de ahora este caso particular, dejando la denominación de *activación no recursiva* para el caso general de activación que no es recursiva y que se realiza de un procedimiento de menor profundidad de anidamiento que el que invoca.

Las denominaciones elegidas parecen, *a priori*, aplicables solo al modelo dinámico optimado. Sin embargo, también pueden establecerse estas diferencias en el modelo dinámico estándar, por exactamente las mismas razones. El cálculo del enlace de acceso, que depende de la distancia léxica entre la llamada (n_{act}) y el lugar de definición del procedimiento activado (n_{def}), es lo que establece la distinción. Recordemos que el caso $n_{def}=n_{act}$ es el que corresponde a la activación de un procedimiento local, cuya optimación está prohibida por la Restricción 4.2.2, así que no lo consideraremos. En el caso en que $n_{act}-n_{def}$ es 1 (es decir, cuando llamador y receptor tienen la misma profundidad de anidamiento), el enlace de acceso es exactamente el mismo, lo cual se resuelve en la secuencia de llamada estándar tomando directamente el enlace de acceso del llamador (lo cual es válido trivialmente para las activaciones recursivas, como ya quedó reflejado en el Programa 6.2.1), y en la secuencia de llamada optimada evitando la operación. Finalmente, si $n_{act}-n_{def}$ es más que 1, ha de utilizarse un registro temporalmente (el que denominamos

Enlace en la introducción de esta sección) para el seguimiento de la cadena estática, independientemente del modelo dinámico. Consideraremos los dos casos por separado, pero ha de quedar con esto clara nuestra intención de realizar una comparación "limpia" entre los dos modelos dinámicos alternativos: elegimos las mejores opciones en el modelo dinámico optimado, pero también en el modelo dinámico estándar.

En el caso de las llamadas casi recursivas, el código que se genera bajo el modelo dinámico estándar es en todo similar al Programa 6.2.1, por lo que hemos discutido. Entonces, el coste de la llamada, que denominaríamos $c[ce]pr-llam_{est}$ por "casi recursiva", viene dado exactamente por $c[ce]pr-llam_{est}$. Ahora bien, la optimación de la llamada consiste en este caso (Algoritmo 4.2.1) en evitar las operaciones relativas al enlace de control, enlace de acceso, dirección de retorno y actualización de BasePila; sin embargo, sí es necesaria la actualización de TopePila, por lo que sabemos que el salto ha de hacerse al punto de entrada habitual del procedimiento, lo mismo que en el modelo estándar (cf. Condición 4.2.4). El código generado para la optimación se representa en el siguiente programa.

```

; punto de entrada al procedimiento receptor
etiqueta (punto-entrada-proc-receptor)
; parte de la secuencia de llamada como receptor
; L7: actualización de TopePila
(1) TopePila := BasePila + #(<tamaño-zona-datos-proc>)
; L8: inicialización de datos locales
...
; fin de la secuencia de llamada como receptor
; comienzo del código del procedimiento receptor
...
; fin del código del procedimiento receptor
...
; punto de entrada al procedimiento llamador
...
; secuencia de llamada correspondiente a una activación final casi recursiva
; LO1: evaluación de los parámetros actuales
...
; LO3: salto al receptor
(6) IR-A (punto-entrada-proc-receptor)
; fin de la secuencia de llamada como llamador
; comienza la secuencia de retorno como llamador, que no incluye nada
etiqueta (etiqueta-retorno)
; continuación del código
...
; fin del código del procedimiento llamador

```

Programa 6.2.5 Esquema del código objeto generado para una activación final casi recursiva según el modelo dinámico optimado, y usando la cadena estática

El coste de la llamada optimada, que denominamos $c[ce]pcr-llam_{opt}$, es un poco mayor que en las activaciones recursivas, debido a la instrucción (1) que ahora ha de ejecutarse, por lo que la mejora temporal será un poco menor.

Secuencia de llamada optimada	Coste según criterio 1	Coste según criterio 2a	Coste según criterio 2b
(6)	2	1	9
(1)	3	0	6
$c[ce]pcr-llam_{opt}$	5	1	15

Tabla 6.2.11 Coste de una secuencia de llamada final casi recursiva optimada, usando la cadena estática (los números de instrucciones se refieren al Programa 6.2.5)

La razón de optimación en este caso sigue dada por una ecuación similar a la [6.2.1], pues cada llamada optimada supone también la desaparición de una secuencia de retorno:

$$r-opt[ce]pcr_{llam-ret} = \frac{c[ce]pcr-llam_{est} + c[ce]p-ret}{c[ce]pcr-llam_{opt}} \quad [6.2.2]$$

Los valores pueden reflejarse inmediatamente en la siguiente tabla.

	Coste según criterio 1	Coste según criterio 2a	Coste según criterio 2b
$c[ce]pcr-llam_{est}$	16	16	55
$c[ce]p-ret$	7	5	26
$c[ce]pcr-llam_{opt}$	5	1	15
$r-opt[ce]pcr_{llam-ret}$	4'6	21	5'4

Tabla 6.2.12 Razón de optimación (usando la cadena estática) para una activación final casi recursiva

Aun siendo estas cifras la mitad aproximadamente de las que se obtenían para las activaciones finales recursivas (Tabla 6.2.6), pensamos que no deben considerarse despreciables de ninguna manera.

Consideremos ahora el caso de las activaciones denominadas "no recursivas", en que el procedimiento receptor tiene menor profundidad de anidamiento que el llamador. Como se ha dicho, es preciso calcular el enlace de acceso siguiendo la cadena estática tantas veces como distancia léxica existe

entre la llamada y la definición del procedimiento a activar, es decir, el valor $n_{act}-n_{def}$, y ello tanto en el modelo estándar como en el optimado. Mostramos el programa generado por el compilador sin optimación para observar el modo en que se calcula el enlace de acceso.

```

; punto de entrada al procedimiento receptor
etiqueta (punto-entrada-proc-receptor)
; parte de la secuencia de llamada como receptor
; L7: actualización de TopePila
(1) TopePila := BasePila + #(tamaño-zona-datos-proc)
; L8: inicialización de datos locales
...
; fin de la secuencia de llamada como receptor
; comienzo del código del procedimiento receptor
...
; fin del código del procedimiento receptor
...
; punto de entrada al procedimiento llamador
...
; secuencia de llamada correspondiente a una activación final no recursiva
; L1: evaluación de los parámetros actuales
...
; L2: enlace de control
(2) [TopePila + #(CNTL-LINK-PROC)] := BasePila
; L3: enlace de acceso
(3a) Enlace := [BasePila + #(ACCS-LINK-LOC)]
Enlace := [Enlace + #(ACCS-LINK-LOC)] }
(3b) ...                               } (nact-ndef-2 veces)
Enlace := [Enlace + #(ACCS-LINK-LOC)] }
(3c) [TopePila + #(ACCS-LINK-PROC)] := [Enlace + #(ACCS-LINK-LOC)]
; L4: dirección de retorno
(4) [TopePila + #(DIR-RET-PROC)] := (etiqueta-retorno)
; L5: actualiza BasePila
(5) BasePila := TopePila + #(tamaño-zona-control-proc)
; L6: salto al receptor
(6) IR-A (punto-entrada-proc-receptor)
; fin de la secuencia de llamada como llamador
; comienza la secuencia de retorno como llamador, que no incluye nada
etiqueta (etiqueta-retorno)
; continuación del código
...
; fin del código del procedimiento llamador

```

Programa 6.2.6 Esquema del código objeto generado para una activación final no recursiva, según el modelo dinámico estándar, y usando la cadena estática

La instrucción (3a) recorre una vez la cadena estática, la instrucción (3c) otra, y por ello la instrucción (3b) ha de realizarse $n_{act}-n_{def}-2$ veces. La secuencia de llamada optimada no puede ahora evitar estas instrucciones (ha de aplicarse el Algoritmo 4.2.1 en toda su extensión), pero sí las demás, como en el caso anterior.

```

; punto de entrada al procedimiento receptor
etiqueta (punto-entrada-proc-receptor)
; parte de la secuencia de llamada como receptor
; L7: actualización de TopePila
(1) TopePila := BasePila + #(<tamaño-zona-datos-proc>)
; L8: inicialización de datos locales

...

; fin de la secuencia de llamada como receptor
; comienzo del código del procedimiento receptor

...

; fin del código del procedimiento receptor

; punto de entrada al procedimiento llamador

...

; secuencia de llamada correspondiente a una activación final no recursiva
; LO1: evaluación de los parámetros actuales

...

; LO2: enlace de acceso
(3a) Enlace := [BasePila + #(<ACCS-LINK-LOC>)]
Enlace := [Enlace + #(<ACCS-LINK-LOC>)] }
(3b) ... } (nact-ndef-2 veces)
Enlace := [Enlace + #(<ACCS-LINK-LOC>)] }
(3c) [BasePila + #(<ACCS-LINK-LOC>)] := [Enlace + #(<ACCS-LINK-LOC>)]
; LO3: salto al receptor
(b) IR-A (punto-entrada-proc-receptor)
; fin de la secuencia de llamada como llamador
; comienza la secuencia de retorno como llamador, que no incluye nada
etiqueta (etiqueta-retorno)
; continuación del código

...

; fin del código del procedimiento llamador

```

Programa 6.2.7 Esquema del código objeto generado para una activación final no recursiva, según el modelo dinámico optimado, y usando la cadena estática

Claramente, los costes de las secuencias de llamada estándar y optimada, que llamamos $c[\text{cel}]\text{pnr-llam}_{\text{est}}$ y $c[\text{cel}]\text{pnr-llam}_{\text{est}}$ (por "no recursiva") respectivamente, dependerán de la distancia léxica entre los procedimientos involucrados; denominamos d , para simplificar, al valor $n_{\text{act}}-n_{\text{def}}$, con lo que los costes se pueden calcular como en la Tabla 6.2.13 y la Tabla 6.2.14. Entonces, la razón de optimación en estas circunstancias, que viene dada como siempre por:

$$r\text{-opt}[\text{cel}]\text{pnr}_{\text{llam-ret}} = \frac{c[\text{cel}]\text{pnr-llam}_{\text{est}} + c[\text{cel}]\text{p-ret}}{c[\text{cel}]\text{pnr-llam}_{\text{opt}}} \quad [6.2.3]$$

va a bajar significativamente, debido al gran coste de las operaciones del cálculo del enlace de acceso, que ahora no pueden evitarse en el programa optimado; además, está claro que esa razón será tanto menor cuanto mayor sea la diferencia de anidamiento entre los procedimientos involucrados (Tabla 6.2.15).

Secuencia de llamada estándar	Coste según criterio 1	Coste según criterio 2a	Coste según criterio 2b
(2)	2	3	7
(3a)	2	3	9
(3b)	$2 \times (d-2)$	$3 \times (d-2)$	$9 \times (d-2)$
(3c)	3	6	13
(4)	3	6	14
(5)	3	0	6
(6)	2	1	9
(1)	3	0	6
$c[ce]pnr-llam_{est}(d)$	$14 + 2 \times d$	$13 + 3 \times d$	$46 + 9 \times d$

Tabla 6.2.13 Coste de una secuencia de llamada final no recursiva usando la cadena estática, según el modelo dinámico estándar (los números de instrucciones se refieren al Programa 6.2.6)

Secuencia de llamada optimada	Coste según criterio 1	Coste según criterio 2a	Coste según criterio 2b
(3a)	2	3	9
(3b)	$2 \times (d-2)$	$3 \times (d-2)$	$9 \times (d-2)$
(3c)	3	6	13
(6)	2	1	9
(1)	3	0	6
$c[ce]pnr-llam_{opt}(d)$	$6 + 2 \times d$	$4 + 3 \times d$	$19 + 9 \times d$

Tabla 6.2.14 Coste de una secuencia de llamada final no recursiva usando la cadena estática, según el modelo dinámico optimado (los números de instrucciones se refieren al Programa 6.2.7)

	Coste según criterio 1	Coste según criterio 2a	Coste según criterio 2b
$c[ce]pnr-llam_{est}(d)$	$14 + 2 \times d$	$13 + 3 \times d$	$46 + 9 \times d$
$c[ce]p-ret$	7	5	26
$c[ce]pnr-llam_{opt}(d)$	$6 + 2 \times d$	$4 + 3 \times d$	$19 + 9 \times d$
$r-opt[ce]pnr_{llam-ret}(d)$	$\frac{21 + 2 \times d}{6 + 2 \times d}$	$\frac{18 + 3 \times d}{4 + 3 \times d}$	$\frac{72 + 9 \times d}{19 + 9 \times d}$

Tabla 6.2.15 Razón de optimación (usando la cadena estática) para una activación final no recursiva

Como es difícil, *a priori*, encontrar programas en que haya una profundidad máxima de anidamiento de 5 o 6, damos la razón de optimación para los primeros valores posibles de d .

$r\text{-opt}[ce]pnr_{\text{llam-ret}}(d)$	Coste según criterio 1	Coste según criterio 2a	Coste según criterio 2b
$d=2$	25/10=2'50	24/10=2'4	90/37=2'43
$d=3$	27/12=2'25	27/13=2'07	99/46=2'15
$d=4$	29/14=2'07	30/16=1'87	108/55=1'96
$d=5$	31/16=1'93	33/19=1'73	117/64=1'82

Tabla 6.2.16 Primeros valores de la razón de optimación (usando la cadena estática) para una activación final no recursiva de procedimiento según la diferencia de anidamiento

Observamos que la *ratio* de optimación se mantiene alrededor de 2 en cualquiera de los criterios, para procedimientos de profundidad cercana al llamador. Desde luego, estas cifras parecen una nimiedad comparadas con las obtenidas hasta ahora, pero debemos recordar que una razón de 2 supone que la ejecución del proceso consume la *mitad* de tiempo en el programa optimado que en el programa estándar, lo cual de por sí es una optimación considerable. Es sorprendente que el criterio 2 produzca los mismos resultados tanto para el "mejor caso" como para el "peor caso", pero debemos recordar que se trata de una distinción que sirve para aproximarse al comportamiento temporal de los programas para el MC68020, y que solo una medición directa podría dar el valor exacto, dada la cantidad de factores involucrados en tal tarea. Lo importante es que se mantiene la constante, observada a lo largo de tablas anteriores, de una similitud entre el criterio 1 y el criterio 2b, lo que viene a reforzar la validez de los resultados.

El uso del *display* supone una cierta dificultad en las activaciones no directamente recursivas que quieren optimarse. Primero, el código generado en el modelo estándar para el proceso es realmente el mismo que el presentado en el Programa 6.2.3, ya que con el *display* no hay en ese modelo una optimación de las activaciones casi recursivas (ni en las recursivas, que a este respecto son un casi particular), como ya dijimos. Por otro lado, de cara a la optimación recordemos (Condición 4.2.8) que, básicamente es necesario, durante la secuencia de llamada optimada, restablecer el registro de *display* del llamador (la

operación que se realizaría en el retorno de este, que desaparece en el modelo optimado). Sin embargo, si el anidamiento del receptor es el mismo que el del llamador (activación casi recursiva), no hay que hacer realmente nada, simplemente basta con mantener el *display* guardado del llamador en el registro de activación del receptor. Este último caso conduce al siguiente programa.

```

; punto de entrada al procedimiento receptor
etiqueta (punto-entrada-proc-receptor)
; parte de la secuencia de llamada como receptor
; L7: actualización de TopePila
(1) TopePila := BasePila + #(< tamaño-zona-datos-proc >)
; L8: inicialización de datos locales
...
; fin del código del procedimiento receptor
; punto de entrada al procedimiento llamador
; secuencia de llamada correspondiente a una activación final no recursiva
; LO1: evaluación de los parámetros actuales
...
; LO3: salto al receptor
(7) IR-A (punto-entrada-proc-receptor)
; fin de la secuencia de llamada como llamador
; comienza la secuencia de retorno como llamador, que no incluye nada
etiqueta (etiqueta-retorno)
; continuación del código
...
; fin del código del procedimiento llamador

```

Programa 6.2.8 Esquema del código objeto generado para una activación final casi recursiva, según el modelo dinámico optimado, y usando el *display*

Los costes de la secuencia de llamada y retorno estándar con el *display* son los ya calculados en el apartado anterior (Tabla 6.2.7 y Tabla 6.2.8), y el de la secuencia de llamada optimada es el mismo que cuando se usa la cadena estática (Tabla 6.2.11). Por lo tanto, aplicando la ecuación [6.2.2], obtenemos:

	Coste según criterio 1	Coste según criterio 2a	Coste según criterio 2b
$c[d]p\text{-}llam_{ost}$	16	13	52
$c[d]p\text{-}ret$	9	8	35
$c[d]pcr\text{-}llam_{opt}$	5	1	15
$r\text{-}opt[d]pcr_{llam-ret}$	5	21	5'8

Tabla 6.2.17 Razón de optimación (usando el *display*) para una activación final casi recursiva

Estos valores son casi idénticos a los correspondientes a la cadena estática (Tabla 6.2.12), porque el mayor coste de la secuencia de retorno se compensa con el menor coste de la secuencia de llamada, tal como ocurría en el caso de las activaciones recursivas (Tabla 6.2.6 y Tabla 6.2.9).

Pasemos al caso en que el procedimiento llamador tiene una profundidad de anidamiento estrictamente inferior a la del receptor (activación no recursiva). Además de que ha de procederse a actualizar el *display* en la llamada tal como se hace en el modelo estándar, ha de incorporarse la recuperación del *display* para el anidamiento del llamador que se haría en la secuencia de retorno de este. El esquema del código aparece a continuación.

```

                                ; punto de entrada al procedimiento receptor
etiqueta (punto-entrada-proc-receptor)
                                ; parte de la secuencia de llamada como receptor
                                ; L7: actualización de TopePila
(1) TopePila := BasePila + #(<tamaño-zona-datos-proc>)
                                ; L8: inicialización de datos locales
...
                                ; fin del código del procedimiento receptor
...
                                ; punto de entrada al procedimiento llamador
...
                                ; secuencia de llamada correspondiente a una activación final no recursiva
                                ; LO1: evaluación de los parámetros actuales
...
                                ; LO2(a). recupera display
(2a) Display(profundidad-llamador) := [BasePila + #(<DISPLAY-LOC>)]
                                ; LO2(b): display guardado
(2b) [BasePila + #(<DISPLAY-LOC>)] := Display(profundidad-receptor)
                                ; LO2(c): actualiza display
(6) Display(profundidad-receptor) := BasePila
                                ; LO3: salto al receptor
(7) IR-A (punto-entrada-proc-receptor)
                                ; fin de la secuencia de llamada como llamador
                                ; comienza la secuencia de retorno como llamador, que no incluye nada
etiqueta (etiqueta-retorno)
                                ; continuación del código
...
                                ; fin del código del procedimiento llamador

```

Programa 6.2.9 Esquema del código objeto generado para una activación final no recursiva, según el modelo dinámico optimado, y usando el *display*

Está claro que no importa en esta estrategia (comparando la cadena estática) la distancia léxica entre los procedimientos involucrados, lo que se aduce habitualmente como una de las virtudes del uso del *display*. El coste de esta secuencia de llamada se sumaría en la siguiente tabla.

Secuencia de llamada optimada	Coste según criterio 1	Coste según criterio 2a	Coste según criterio 2b
(2a)	2	3	9
(2b)	2	3	7
(6)	1	0	3
(7)	2	1	9
(1)	3	0	6
$c[d]pnr-llam_{opt}$	10	7	34

Tabla 6.2.18 Coste de una secuencia de llamada final no recursiva usando el *display*, según el modelo dinámico optimado (los números de instrucciones se refieren al Programa 6.2.9)

Por lo tanto, obtenemos la *ratio* de optimación que aparece a continuación.

	Coste según criterio 1	Coste según criterio 2a	Coste según criterio 2b
$c[d]p-llam_{est}$	16	13	52
$c[d]p-ret$	9	8	35
$c[d]pnr-llam_{opt}$	10	7	34
$r-opt[d]pnr_{llam-ret}$	2'5	3	2'55

Tabla 6.2.19 Razón de optimación (usando el *display*) para una activación final no recursiva

Vemos que se obtienen valores un poco superiores al mejor caso usando la cadena estática (Tabla 6.2.16), independientemente de la distancia léxica entre los procedimientos, como ya se ha apuntado, pero que son cualitativamente similares. Tal como allí dijimos, no debe despreciarse el haber obtenido una *ratio* de optimación de 2'5, a pesar de la diferencia que haya en la comparación con las activaciones finales recursivas.

El resultado más claro de este estudio (amén de las cifras obtenidas) es la confirmación de la hipótesis planteada sobre la conveniencia de la distinción entre las dos grandes clases de activaciones finales no directamente recursivas: las que involucran procedimientos con el mismo anidamiento (casi recursivas) y las que se aplican a un procedimiento de anidamiento menor que el llamador (no recursivas). Ello se puede ver si representamos gráficamente el efecto absoluto de la optimación, eso sí, siempre teniendo presente que nos referimos

exclusivamente a los procesos de llamada y retorno (sin incluir la evaluación de los parámetros actuales ni la inicialización de los datos locales). Para ello, elegimos el criterio de coste 2b, que es el término medio entre todos los criterios de coste, y dibujamos el coste absoluto de las diferentes secuencias de llamada y retorno.

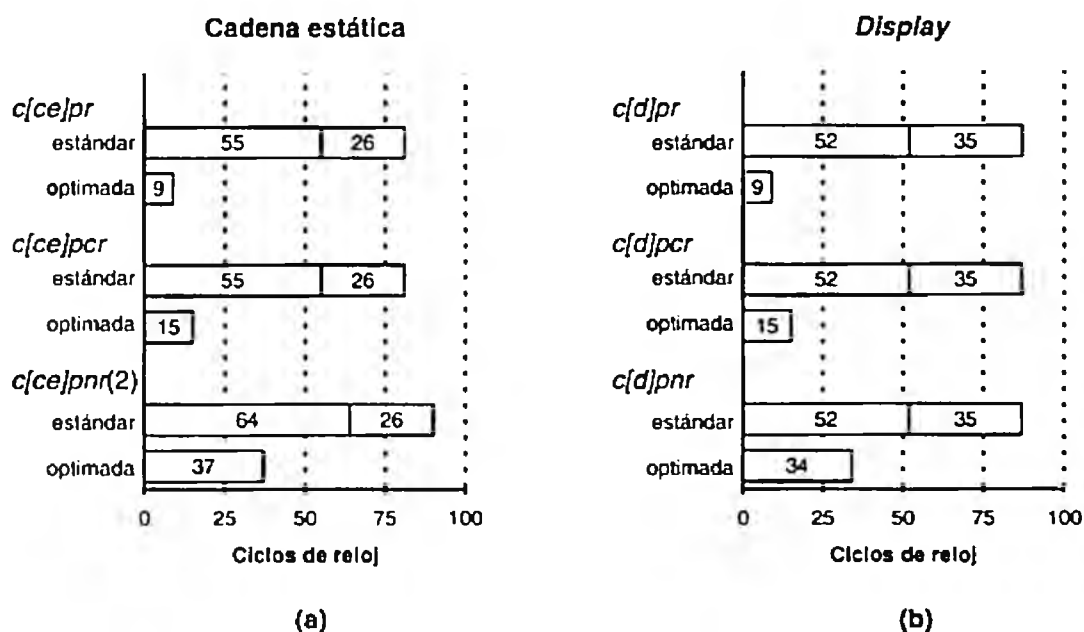


Figura 6.2.1 Comparación entre los costes (usando el criterio 2b) de las distintas clases de activaciones finales de procedimientos, (a) usando la cadena estática, y (b) usando el *display*

En los casos etiquetados con "estándar", aparece en la barra primero el coste de la secuencia de llamada y después el coste de la secuencia de retorno. Con ello, podemos hacernos una idea más clara del alcance de la optimización. Pero todavía mejor, podemos dibujar los valores de las razones de optimación. Antes, resumimos en la Tabla 6.2.20 los resultados alcanzados hasta el momento, para poder visualizarlos en conjunto. Con ella, se obtienen las gráficas comparativas de la Figura 6.2.2.

Podemos decir que el alcance de la optimación en las llamadas casi recursivas está a medio camino entre la optimación máxima conseguida para las activaciones finales recursivas y la optimación significativamente menor para las activaciones no recursivas en que el procedimiento llamado está léxicamente lejos del que lo invoca.

	Coste según criterio 1	Coste según criterio 2a	Coste según criterio 2b
$r-opt[ce]pr_{llam-ret}$	11'5	21	9
$r-opt[ce]pcr_{llam-ret}$	4'6	21	5'4
$r-opt[ce]pnr_{llam-ret}(2)$	2'5	2'4	2'43
$r-opt[d]pr_{llam-ret}$	12'5	21	9'6
$r-opt[d]pcr_{llam-ret}$	5	21	5'8
$r-opt[d]pnr_{llam-ret}$	2'5	3	2'55

Tabla 6.2.20 Resumen de las razones de optimación para las distintas clases de activaciones finales de procedimientos

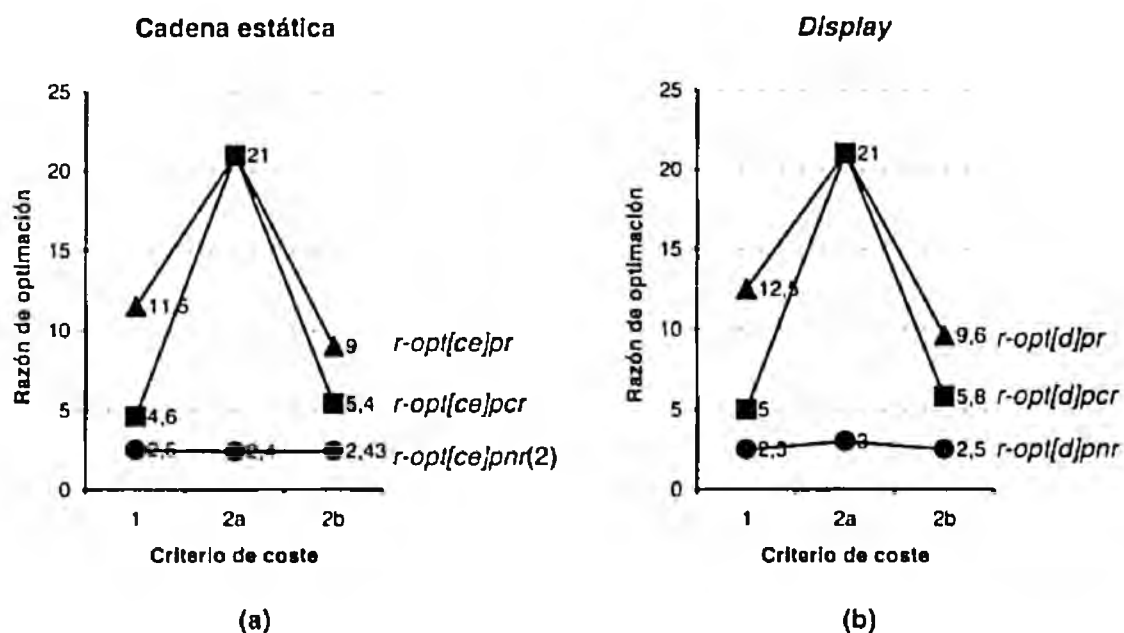


Figura 6.2.2 Comparación entre las razones de optimación de las distintas clases de activaciones finales de procedimientos, (a) usando la cadena estática, y (b) usando el display

Este hecho es muy importante, pues la otra clase de procesos recursivos típicos, además de la recursividad directa, es la recursividad mutua, o en general, la recursividad indirecta de cualquier grado. Si esta recursividad indirecta consigue expresarse mediante procedimientos del mismo nivel de anidamiento (usando la directiva `forward` del lenguaje PASCAL), el proceso recursivo completo, además de ejecutarse en espacio constante, será

significativamente más rápido (siempre considerando exclusivamente los procesos de llamada y retorno).

A este respecto, ha de recordarse la discusión de la sección 4.3, y concretamente el Programa 4.3.1. Tal programa es, como allí se decía, el ejemplo paradigmático de recursividad no directa que puede resultar altamente optimada, al estar involucrados procedimientos del mismo anidamiento. Otro ejemplo lo podemos encontrar justamente en relación con los compiladores, en la escritura de un *parser* descendente recursivo, en que cada variable de la gramática independiente del contexto se asocia a un procedimiento que activa otros procedimientos correspondientes a las variables que aparecen en las partes derechas de sus reglas [Wirth 1971b; Aho, Sethi y Ullman 1990: 44-48; Fischer y LeBlanc 1988: 36-40]. Si los procedimientos correspondientes a todas las variables se definen con el mismo nivel de anidamiento, todas las activaciones finales no recursivas resultarán optimadas de un modo significativo. En el capítulo 7 estudiaremos más profundamente estos ejemplos y el impacto en general de la optimación en la metodología de programación en el lenguaje PASCAL.

Aún no podemos terminar, sin embargo, el estudio de las activaciones finales no recursivas de procedimientos sin considerar dos cuestiones adicionales importantes. La primera, como en las activaciones recursivas, tiene que ver con el problema del cálculo de los parámetros actuales y su asignación a los formales, lo que quedó reflejado en la Condición 4.2.9 de la optimación de esta clase de activaciones. De nuevo, la estrategia que se use para realizar este proceso determinará la mejora o empeoramiento temporal del proceso completo de llamada. Desde luego, el caso mejor es ahora un comportamiento idéntico al estándar, pues no existe la optimación del cálculo de ciertos parámetros que podía darse en determinadas condiciones en las activaciones finales recursivas; el caso peor es que se produzca un notable empeoramiento si resultase necesario utilizar localidades temporales que no se precisarían en el modelo estándar.

Hay que considerar, como parte del asunto, el hecho de que las zonas de parámetros formales del llamador y del receptor pueden tener diferente tamaño; concretamente, sería un pequeño problema que los parámetros del receptor ocupasen más tamaño que los del llamador. Si se utilizasen localidades temporales en el registro de activación del llamador para acumular los valores

de los parámetros actuales, habría que asegurar que tales localidades no se solapan con la que será zona de parámetros formales del receptor; sin embargo, el compilador puede muy sencillamente hacer que esa zona de posiciones temporales empiece más allá del punto en que acabará la zona de formales del receptor, por lo que esto realmente no es problema. En cualquier caso, repetimos que solo conociendo la estrategia exacta del proceso de cálculo de los parámetros actuales y su asignación a los formales podría determinarse si va a ocurrir un empeoramiento del proceso en los programas optimados.

Por otro lado, ha de tenerse en cuenta un caso particular, poco frecuente, pero desde luego necesario para mantener completas nuestras posturas. Se trata de la posibilidad de que el procedimiento activado en posición final sea un parámetro formal procedimental del propio llamador, lo que dio lugar a la Condición 4.2.10 de la optimación. Mostramos en primer lugar el código generado en el modelo dinámico estándar para la activación general de un parámetro formal procedimental, recordando que la implementación habitual de esta clase de parámetros consiste en que el parámetro formal procedimental contenga el enlace de acceso calculado para el procedimiento "real" al que se asoció por primera vez, y el punto de entrada de dicho procedimiento. Entonces, el enlace de acceso para el nuevo registro de activación será el que contiene el parámetro formal, y el salto se hará también al punto de entrada guardado.

Esto aparece en el Programa 6.2.10. Los valores *<despl-ca-proc-formal>* y *<despl-pe-proc-formal>* representan los desplazamientos respecto de *BasePila* en que se encuentran los dos elementos, enlace de acceso y punto de entrada, contenidos en el parámetro formal procedimental, y que son conocidos para el compilador. Obsérvese que la solución elegida consiste en usar un registro temporalmente (*Enlace*) para señalar al registro de activación donde se encuentra el parámetro formal; en el caso de que este sea local, la instrucción (3b) no se ejecuta ninguna vez, pero sí es necesario seguir utilizando el registro *Enlace*, a pesar de que tiene el mismo valor que *BasePila*, porque este se modifica en la instrucción (5), y no podría usarse para el salto de la instrucción (6). Si el parámetro formal procedimental no es local, se sigue la cadena estática como habitualmente para otros objetos no locales.

en el que se sitúa el parámetro formal procedimental; entonces, no sería precisa ninguna operación para preservar dicho parámetro. Por supuesto, esta situación puede ser esporádica, y no soluciona nuestro problema. Entonces, la estrategia para preservar el parámetro formal puede ser de dos clases: o bien se copia el parámetro formal procedimental en registros (generalmente dos, puesto que los registros habitualmente son del tamaño de las direcciones, y el parámetro formal procedimental contiene dos direcciones), o bien se copia en la zona temporal del registro de activación del llamador. En el siguiente programa generado en el modelo dinámico optimado escogemos la segunda posibilidad, que será más costosa, al implicar operaciones sobre memoria.

```

; punto de entrada al procedimiento receptor
etiqueta (punto-entrada-proc-receptor)
; parte de la secuencia de llamada como receptor
; L7: actualización de TopePila
(1) TopePila := BasePila + #(<tamaño-zona-datos-proc>)
; L8: inicialización de datos locales
...
; fin del código del procedimiento receptor
...
; punto de entrada al procedimiento llamador
...
; secuencia de llamada para activación de parámetro formal procedimental
; LO1(a): copia segura del parámetro formal procedimental
(1a) [BasePila + #(<despl-temp1>)] := [BasePila + #(<despl-ca-proc-formal>)]
(1b) [BasePila + #(<despl-temp2>)] := [BasePila + #(<despl-pe-proc-formal>)]
; LO1(b): evaluación de los parámetros actuales
...
; LO2: enlace de acceso
(3c) [BasePila + #(<CNTRL-LINK-PROC>)] := [BasePila + #(<despl-temp1>)]
; LO3: salto al receptor
(6) IR-A [BasePila + #(<despl-temp2>)]
; fin de la secuencia de llamada como llamador
; comienza la secuencia de retorno como llamador, que no incluye nada
etiqueta (etiqueta-retorno)
; continuación del código
...
; fin del código del procedimiento llamador

```

Programa 6.2.11 Esquema del código objeto generado para una activación final de un parámetro formal procedimental local, según el modelo dinámico optimado, y con la cadena estática

Los valores *<despl-temp1>* y *<despl-temp2>* son los desplazamientos relativos donde se situarán temporalmente los elementos del parámetro formal. Las dos instrucciones (1a) y (1b) podrían hacerse en una sola si existiese la instrucción para mover un número cualquiera de *bytes*; sin embargo, algunos procesadores (como el MC68020) no cuentan con tal posibilidad.

```

; punto de entrada al procedimiento receptor
etiqueta (punto-entrada-proc-receptor)
; parte de la secuencia de llamada como receptor
; L7: actualización de TopePila
(1) TopePila := BasePila + #(<tamaño-zona-datos-proc>)
; L8: inicialización de datos locales

; fin del código del procedimiento receptor

; punto de entrada al procedimiento llamador

; secuencia de llamada para activación de parámetro formal procedimental
; L1: evaluación de los parámetros actuales

; L2: enlace de control
(2) [TopePila + #(<CNTL-LINK-PROC>)] := BasePila
; L3: enlace de acceso
(3a) Enlace := BasePila
Enlace := [Enlace + #(<ACCS-LINK-LOC>)] }
(3b) ... } (Nact-1) veces
Enlace := [Enlace + #(<ACCS-LINK-LOC>)] }
(3c) [TopePila + #(<CNTL-LINK-PROC>)] := [Enlace + #(<despl-ca-proc-formal>)]
; L4: dirección de retorno
(4) [TopePila + #(<DIR-RET-PROC>)] := (<etiqueta-retorno>)
; L5: actualiza BasePila
(5) BasePila := TopePila + #(<tamaño-zona-control-proc>)
; L6: salto al receptor
(6) IR-A [Enlace + #(<despl-pe-proc-formal>)]
; fin de la secuencia de llamada como llamador
; comienza la secuencia de retorno como llamador, que no incluye nada
etiqueta (<etiqueta-retorno>)
; continuación del código

; fin del código del procedimiento llamador

```

Programa 6.2.10 Esquema del código objeto generado para una activación final de un parámetro formal procedimental, según el modelo dinámico estándar, y usando la cadena estática

La discusión del apartado 4.2.3 que condujo a la Condición 4.2.10 y al Algoritmo 4.2.2 debe recapitularse en estos momentos para el modelo dinámico optimado: básicamente, se concluía que si el parámetro formal procedimental era local, resultaba preciso preservar dicho parámetro formal hasta realizar la actualización del enlace de acceso y el salto al procedimiento receptor. Este requisito puede resolverse de diferentes maneras, pero influyen mucho más que en otras ocasiones los detalles concretos de implementación que se hayan elegido. En el modelo dinámico que venimos usando, podría haber varias situaciones posibles, que pasamos a analizar.

Primeramente, podría darse el caso de que el tamaño de los parámetros formales del procedimiento receptor fuera más pequeño que el desplazamiento

en el que se sitúa el parámetro formal procedimental; entonces, no sería precisa ninguna operación para preservar dicho parámetro. Por supuesto, esta situación puede ser esporádica, y no soluciona nuestro problema. Entonces, la estrategia para preservar el parámetro formal puede ser de dos clases: o bien se copia el parámetro formal procedimental en registros (generalmente dos, puesto que los registros habitualmente son del tamaño de las direcciones, y el parámetro formal procedimental contiene dos direcciones), o bien se copia en la zona temporal del registro de activación del llamador. En el siguiente programa generado en el modelo dinámico optimado escogemos la segunda posibilidad, que será más costosa, al implicar operaciones sobre memoria.

```

                                ; punto de entrada al procedimiento receptor
etiqueta (punto-entrada-proc-receptor)
                                ; parte de la secuencia de llamada como receptor
                                ; L7: actualización de TopePila
(1) TopePila := BasePila + #(<tamaño-zona-datos-proc>)
                                ; L8: inicialización de datos locales

                                ; fin del código del procedimiento receptor

                                ; punto de entrada al procedimiento llamador

                                ; secuencia de llamada para activación de parámetro formal procedimental
                                ; LO1(a): copia segura del parámetro formal procedimental
(1a) [BasePila + #(<despl-temp1>)] := [BasePila + #(<despl-ca-proc-formal>)]
(1b) [BasePila + #(<despl-temp2>)] := [BasePila + #(<despl-pe-proc-formal>)]
                                ; LO1(b): evaluación de los parámetros actuales

                                ; LO2: enlace de acceso
(3c) [BasePila + #(<CNTRL-LINK-PROC>)] := [BasePila + #(<despl-temp1>)]
                                ; LO3: salto al receptor
(6) IR-A [BasePila + #(<despl-temp2>)]
                                ; fin de la secuencia de llamada como llamador
                                ; comienza la secuencia de retorno como llamador, que no incluye nada
etiqueta (etiqueta-retorno)
                                ; continuación del código

                                ; fin del código del procedimiento llamador

```

Programa 6.2.11 Esquema del código objeto generado para una activación final de un parámetro formal procedimental local, según el modelo dinámico optimado, y con la cadena estática

Los valores *<despl-temp1>* y *<despl-temp2>* son los desplazamientos relativos donde se situarán temporalmente los elementos del parámetro formal. Las dos instrucciones (1a) y (1b) podrían hacerse en una sola si existiese la instrucción para mover un número cualquiera de *bytes*; sin embargo, algunos procesadores (como el MC68020) no cuentan con tal posibilidad.

El coste de la secuencia de llamada estándar, dependiendo de la distancia léxica d entre la llamada y la definición del parámetro formal ($d=n_{act}-n_{def}$), se calcula en la siguiente tabla.

Secuencia de llamada estándar	Coste según criterio 1	Coste según criterio 2a	Coste según criterio 2b
(2)	2	3	7
(3a)	1	0	3
(3b)	$2 \times d$	$3 \times d$	$9 \times d$
(3c)	3	6	13
(4)	3	6	14
(5)	3	0	6
(6)	2	2	11
(1)	3	0	6
$c[ce]pf-llam_{est}(d)$	$17 + 2 \times d$	$17 + 3 \times d$	$60 + 9 \times d$

Tabla 6.2.21 Coste de una secuencia de llamada final (usando la cadena estática) de un parámetro formal procedimental, según el modelo dinámico estándar (los números de instrucciones se refieren al Programa 6.2.10)

El coste de la secuencia de llamada optimada para el caso en que el parámetro formal procedimental sea local al llamador se obtiene a continuación.

Secuencia de llamada optimada	Coste según criterio 1	Coste según criterio 2a	Coste según criterio 2b
(1a)	3	6	13
(1b)	3	6	13
(3c)	3	6	13
(6)	2	2	11
(1)	3	0	6
$c[ce]pfl-llam_{opt}$	14	20	56

Tabla 6.2.22 Coste de una secuencia de llamada final (usando la cadena estática) de un parámetro formal procedimental local, según el modelo dinámico optimado (los números de instrucciones se refieren al Programa 6.2.11)

Solamente con el criterio 2a obtenemos un empeoramiento del proceso, debido al alto coste de las operaciones de movimiento de datos entre memoria, que

equilibra el coste de las instrucciones eliminadas; sin embargo, teniendo en cuenta que se elimina la secuencia de retorno del llamador, aún resulta una pequeña mejora del proceso completo, tal como se refleja en la tabla siguiente.

	Coste según criterio 1	Coste según criterio 2a	Coste según criterio 2b
$c[ce]pf\text{-}llam_{est}(0)$	17	17	60
$c[ce]p\text{-}ret$	7	5	26
$c[ce]pfi\text{-}llam_{opt}$	14	20	56
$r\text{-}opt[ce]pfi\text{-}llam\text{-}ret$	1'71	1'1	1'53

Tabla 6.2.23 Razón de optimación (usando la cadena estática) para una activación final no recursiva de un parámetro formal procedimental del llamador

También podemos analizar, para completar, las activaciones de un parámetro formal procedimental no local al llamador. En el modelo estándar, el Programa 6.2.10 sigue expresando el código en esta situación, ahora con $d > 0$; en el modelo optimado, ya no es preciso salvaguardar el parámetro formal procedimental, pues este no se encuentra en el registro de activación del llamador, que va a desaparecer. El código generado para la llamada optimada es como en el Programa 6.2.10, pero sin las instrucciones (2), (5) y (6), por lo que no lo repetimos. Por lo tanto, podemos calcular el coste de la secuencia de llamada optimada en este caso, y la razón de optimación correspondiente, que aparecen en las tablas que siguen.

Secuencia de llamada optimada	Coste según criterio 1	Coste según criterio 2a	Coste según criterio 2b
(3a)	1	0	3
(3b)	$2 \times d$	$3 \times d$	$9 \times d$
(3c)	3	6	13
(6)	2	2	11
(1)	3	0	6
$c[ce]pfi\text{-}llam_{opt}(d)$	$9 + 2 \times d$	$8 + 3 \times d$	$33 + 9 \times d$

Tabla 6.2.24 Coste de una secuencia de llamada final (usando la cadena estática) de un parámetro formal procedimental no local, según el modelo dinámico optimado (los números de instrucciones se refieren al Programa 6.2.12)

	Coste según criterio 1	Coste según criterio 2a	Coste según criterio 2b
$c[ce]pf\text{-}llam_{est}(d)$	$17 + 2 \times d$	$17 + 3 \times d$	$60 + 9 \times d$
$c[ce]p\text{-}ret$	7	5	26
$c[ce]pfn\text{-}llam_{opt}(d)$	$9 + 2 \times d$	$8 + 3 \times d$	$33 + 9 \times d$
$r\text{-}opt[ce]pfn_{llam\text{-}ret}(d)$	$\frac{24 + 2 \times d}{9 + 2 \times d}$	$\frac{22 + 3 \times d}{8 + 3 \times d}$	$\frac{86 + 9 \times d}{33 + 9 \times d}$

Tabla 6.2.25 Razón de optimación (usando la cadena estática) para una activación final de un parámetro formal procedimental no local

Tal como se hizo en otro caso anteriormente, mostramos la razón de optimación para los primeros valores posibles de d .

$r\text{-}opt[ce]pfn_{llam\text{-}ret}(d)$	Coste según criterio 1	Coste según criterio 2a	Coste según criterio 2b
$d=1$	$26/11=2'36$	$25/11=2'27$	$95/42=2'26$
$d=2$	$28/13=2'15$	$28/14=2$	$104/51=2'03$
$d=3$	$30/15=2$	$31/17=1'82$	$113/60=1'88$
$d=4$	$32/17=1'88$	$34/20=1'7$	$122/69=1'76$

Tabla 6.2.26 Primeros valores de la razón de optimación (usando la cadena estática) para una activación final de un parámetro formal procedimental no local, según la distancia léxica con el llamador

Las cifras obtenidas son prácticamente iguales que las correspondientes a la activación final de un procedimiento de distinto anidamiento que el llamador (cf. Tabla 6.2.16, teniendo en cuenta que $d=2$ en esa tabla es lo mismo que $d=1$ en la Tabla 6.2.26, y así sucesivamente).

No es sorprendente que estos casos resulten un poco mejor optimados que cuando el parámetro formal procedimental es local en el llamador (cf. Tabla 6.2.23), ya que la obligación de preservar el valor del parámetro formal es un proceso costoso, si se realiza sobre memoria temporal del registro de activación del llamador, como hemos visto. Desde luego, podría ser que se usaran registros para salvaguardar el parámetro formal, con lo que los costes de la secuencia de llamada optimada disminuirían, y la optimación sería mejor. Sin embargo, tomando el caso del MC68020, resulta que no es posible realizar un

salto a la dirección contenida directamente en un registro, así que en todo caso podría llevarse a registro el enlace de acceso guardado para el parámetro formal, mientras que la dirección de retorno debería copiarse en memoria, como en el Programa 6.2.11. La mejora de esta variación no es demasiado significativa (de 1'71/1'1/1'53 de la Tabla 6.2.23 a 2/1'57/1'87, según los diferentes criterios de coste), y aún se mantienen por debajo de lo obtenido cuando el parámetro formal procedimental no es local.

Concluimos, por tanto, que la mejora introducida en la activación de parámetros formales procedimentales es pequeña, incluso no puede casi hablarse de mejora si el parámetro formal es local en el llamador. De todas las maneras, la optimación sigue siendo interesante aunque solo sea por la contrapartida de mejora espacial que se consigue, lo mismo que para las activaciones finales no recursivas de procedimientos. Además, estos casos son, por un lado, menos frecuentes, y por otro, menos significativos en lo que a procesos recursivos, genéricamente hablando, se refiere.

6.2.3 Activaciones finales recursivas y no recursivas de funciones

Los resultados obtenidos para los procedimientos podrían, en principio, aplicarse de igual modo a las funciones, y de hecho, así lo dijimos al comienzo del apartado 6.2.1. Sin embargo, es preciso realizar una consideración adicional para completar el estudio. Básicamente, el aspecto a analizar es la influencia de la optimación en las operaciones que se refieren al valor de retorno de la función, que son las únicas que establecen la diferencia entre procedimientos y funciones en PASCAL, como se concluyó en el capítulo 5. Tal como allí se obtuvo, las partes de las secuencias de llamada optimadas de procedimientos ejecutadas por los que invocan son las que deben ejecutar las funciones en las activaciones finales. Por ello, la primera conclusión parecería ser que las razones de optimación van a ser las mismas, y efectivamente es así, pero solo si se considera el propio proceso de llamada.

Hemos de recordar que la optimación, primero, elimina una secuencia de retorno por cada activación optimada, y segundo, que se aplica a una clase concreta de sentencias en que intervienen funciones, esto es, no solo a la propia activación de una función. En lo que respecta al primer asunto, hay una

operación que se realiza durante la secuencia de retorno de una función que no tiene parangón en los procedimientos: es la verificación de que el valor de retorno ha sido definido. Por lo tanto, la secuencia de retorno de funciones será algo más costosa que la de procedimientos, y viniendo la razón de optimación dada por ecuaciones similares a la ecuación [6.2.1], la ecuación [6.2.2] y la ecuación [6.2.3]:

$$r_{\text{-opt}} \cdot f_{\text{llam-ret}} = \frac{c \cdot f_{\text{-llam-est}} + c \cdot f_{\text{-ret}}}{c \cdot f_{\text{-llam-opt}}} \quad [6.2.4]$$

está claro que se obtendrían cifras aún más altas que las anteriores. Para ser rigurosos, también habremos de considerar como parte de la secuencia de llamada la operación que asigna un valor indefinido al valor de retorno de la función entrante, tanto en el modelo estándar como en el modelo optimado, con lo que la secuencia de llamada optimada no va a ser tan sencilla como en el caso de los procedimientos, y las cifras pueden, al final, empeorar.

En cualquier caso, existe cierta dificultad a la hora de cuantificar estas operaciones, debido a que el concepto de "valor definido" ha sido uno de los más difíciles de implementar del lenguaje PASCAL desde su misma concepción. Welsh, Sneeringer y Hoare [1977] ya avisan de este problema:

«Uninitialized variables are also difficult to detect, and all hardware detection mechanisms known to us are quite expensive. Possible solutions are to require that every variable be initialized when it is declared or that every variable be assigned in such a way that the compiler can easily verify that there are no references to uninitialized variables. The latter might work very well in a language without jumps, and deserves further investigation»

Pensamos, sin embargo, que no es posible determinar estáticamente en todos los casos que un cierto dato está definido, y desde luego sería extremadamente complejo aún en los casos posibles. Por ello, la detección habría de ser dinámica, y ello supone un gran coste temporal.

Quizá, con los procesadores modernos, que disponen generalmente de la posibilidad de manejar campos de *bits*, podría darse una solución relativamente económica: se habilita un campo de *bits* de longitud suficiente para que cada variable (y cada componente de una variable de tipo estructurado, recursivamente) quede representada por exactamente uno de los *bits*. Este campo se colocaría justo al final de la zona de datos locales del registro de activación de cada subprograma, porque su tamaño será conocido exactamente

cuando se hayan procesado todas las declaraciones del bloque. El compilador puede, sin demasiada dificultad, igual que asigna un desplazamiento a cada variable, asignar el *bit*, o conjunto de *bits*, que le corresponden. Cuando se accede al valor de lado derecho de una variable, es posible realizar la comprobación del *bit* de definición que tiene asignado, mediante una sencilla instrucción; cuando se asigna una variable, ese *bit* puede colocarse a su valor preciso. En el MC68020, la comprobación podría realizarse con las instrucciones:

```
BTST #(<despl-bit>), [BasePila + #(<despl-campo-bits>)]
BEQ <error>
```

y la asignación con:

```
BSET #(<despl-bit>), [BasePila + #(<despl-campo-bits>)]
```

Desde luego, el compilador podría realizar el análisis estático comentado antes para evitar generar cualquiera de las dos operaciones cuando sea posible conocer su resultado, de modo que el empeoramiento de la ejecución fuera mínimo. Los costes de las operaciones de verificación del centinela para cualquier variable, incluido el correspondiente al valor de retorno de una función, son los siguientes:

Instrucción	Coste criterio 1	Coste criterio 2	
		(a) mejor caso	(b) peor caso
BTST #(<d-bit>), [BasePila + #(<d-c-bits>)]	3	7	12
BEQ <error>	2	3	9
c-verif-def	5	10	21

Tabla 6.2.27 Costes de las instrucciones de manejo de *bits* para la verificación del centinela de definición de un dato, según los criterios habituales

Ahora bien, ¿cuándo se establece como indefinido el centinela del valor de retorno? Sabemos que es, como para todos los datos locales, al comienzo del código de la función, concretamente se trata de la última operación de la secuencia de llamada, tanto en el modelo estándar como en el optimado (*vide* Algoritmo 5.1.1 y Algoritmo 5.2.1, respectivamente). Por lo tanto, con la sencilla instrucción de llevar ceros (o unos) a un *byte* o conjunto de *bytes*, que en el MC68020 es:

CLR [BasePila + #(<despl-campo-bits>)]

se puede resolver el problema de la asignación inicial de los centinelas. Pensamos que esta solución es mucho menos costosa que cualquier otra basada en asignar un valor especial a cada dato, que exigiría un gran coste en el proceso de inicialización (aunque el coste de los procesos de verificación y asignación fuera el mismo). Concretamente, su coste viene dado por la siguiente tabla.

Instrucción	Coste criterio 1	Coste criterio 2	
		(a) mejor caso	(b) peor caso
CLR [BasePila + #(<d-c-bits>)]	2	5	9
c-asign-def	2	5	9

Tabla 6.2.28 Costes del proceso de asignación de los centinelas de definición de los datos locales y el valor de retorno, según los criterios habituales

Desde luego, verificar y asignar el valor de cada uno van a ser operaciones costosas, como vemos en la Tabla 6.2.27, por lo que muchas implementaciones del lenguaje PASCAL omiten esta especificación.

En lo que concierne a nuestro problema, está claro que las instrucciones de verificación del valor de retorno (BTST, BEQ) han de añadirse a la secuencia de retorno. Por ello, el coste de la secuencia de retorno para las funciones será:

$$c[\bullet]f-ret = c[\bullet]p-ret + c-verif-def \quad [6.2.5]$$

que se muestra en la tabla siguiente para el caso de la cadena estática.

	Coste según criterio 1	Coste según criterio 2a	Coste según criterio 2b
c[ce]p-ret	7	5	26
c-verif-def	5	10	21
c[ce]f-ret	12	15	47

Tabla 6.2.29 Coste de una secuencia de retorno de función (usando la cadena estática) en los dos modelos dinámicos

También vamos a incluir en la secuencia de llamada, tanto estándar como optimada, la operación de inicialización de los datos locales y el valor de

retorno. Esto puede parecer un tanto incoherente con posturas que hemos mantenido previamente: para los procedimientos, no incluimos este proceso entre los analizados para obtener los costes de las secuencias de llamada y las razones de optimización consiguientes. Sin embargo, para los procedimientos este proceso puede no darse, si no existen datos locales, mientras que siempre aparece en las funciones, al menos para el valor de retorno. De esa manera, pensamos, quedará reflejada la obligatoriedad de esta operación para esta clase de subprogramas. Según ello:

$$c[\bullet]f\bullet\text{-llam.} = c[\bullet]p\bullet\text{-llam.} + c\text{-asign-def} \quad [6.2.6]$$

tanto para los dos modelos dinámicos como para las diferentes clases de activaciones. Así, podríamos calcular con ayuda de la ecuación [6.2.4], por ejemplo para las activaciones finales recursivas, la siguiente razón de optimización.

	Coste según criterio 1	Coste según criterio 2a	Coste según criterio 2b
$c[ce]fr\text{-llam}_{est}$	18	21	64
$c[ce]f\text{-ret}$	12	15	47
$c[ce]fr\text{-llam}_{opt}$	4	6	18
$r\text{-opt}[ce]fr_{llam-ret}$	7.5	6	$6.1\hat{6}$

Tabla 6.2.30 Razón de optimización tentativa (usando la cadena estática) para un proceso completo de activación recursiva final de una función

Claramente vemos que el coste de la verificación del valor de retorno es alto: la razón de optimización se reduce bastante respecto a la obtenida para las activaciones finales recursivas de procedimientos.

Sin embargo, antes de continuar con las otras clases de activaciones, hemos de tener en cuenta otro asunto. Recordemos (apartado 5.2.1) que se denomina activación final de función a una sentencia de asignación del valor de retorno de una función con una expresión formada exclusivamente por la llamada a otra (o la misma) función:

$$\langle \text{función} \rangle := \langle \text{función} \rangle (\langle \text{parámetros} \rangle)$$

Esto significa que la optimización no solo evita la secuencia de retorno de la función llamada, sino también, en este caso, las operaciones que realizaría la

función llamadora para capturar el valor de retorno de la función llamada y copiarlo en el espacio destinado a su propio valor de retorno. Llamaremos $c[ce/d]f\text{-copia}$ al coste de esta operación, válido tanto para la cadena estática como para el *display*, y que es independiente de la clase de activación; realmente, se podría incluir en la secuencia de retorno, pero así enfatizamos la diferencia con los procedimientos. Supondremos que el proceso se realiza de la forma más simple posible, aunque puede ocurrir que implementaciones alternativas consuman más tiempo. Concretamente, puede hacerse en una única operación, pues la función llamadora accede al valor de retorno de la función receptora directamente a través del registro *TopePila*, y la localidad de destino se obtiene con un desplazamiento a partir de *BasePila*:

(1) $[BasePila + \# \langle displ\text{-}valor\text{-}retorno \rangle] := [TopePila]$

La única salvedad es que el número de *bytes* a copiar depende del tipo de retorno de la función, y evidentemente, eso podría influir en el coste de la operación. No ocurre así con el MC68020, siempre que el dato ocupe como mucho cuatro *bytes*; esto suele servir para casi todos los posibles valores de retorno, salvo para el tipo real, que suele implementarse en ocho *bytes*; en ese caso, se precisarían dos operaciones de copia:

(2a) $[BasePila + \# \langle displ\text{-}v\text{-}ret1 \rangle] := [TopePila]$

(2b) $[BasePila + \# \langle displ\text{-}v\text{-}ret2 \rangle] := [TopePila + \# \langle displ\text{-}v\text{-}ret2 \rangle]$

Por ello, establecemos dos posibles valores para el coste de esta operación: el primero se denominará $c[ce/d]f\text{-copia}_c$, por "copia corta", y el segundo $c[ce/d]f\text{-copia}_l$, por "copia larga". Estos valores se obtienen en la tabla siguiente.

Secuencia de copia del valor de retorno	Coste según criterio 1	Coste según criterio 2a	Coste según criterio 2b
(1)	2	6	11
$c[ce/d]f\text{-copia}_c$	2	6	11
(2a)	2	6	11
(2b)	3	6	13
$c[ce/d]f\text{-copia}_l$	5	12	24

Tabla 6.2.31 Coste de la operación de copia del valor de retorno en una activación final de función, según el tamaño del valor de retorno

Entonces, la ecuación [6.2.4] que expresaba la razón de optimación en un proceso de activación final de función debe modificarse para tener en cuenta esta operación:

$$r\text{-opt}[\bullet]f_{\text{llam-ret}} = \frac{c[\bullet]f_{\text{est}} - \text{llam}_{\text{est}} + c[\bullet]f_{\text{-ret}} + c[ce/d]f\text{-copia.}}{c[\bullet]f_{\text{est}} - \text{llam}_{\text{opt}}} \quad [6.2.7]$$

colocando en el espacio reservado a $c[ce/d]f\text{-copia.}$ el valor correcto dependiendo del tamaño del valor de retorno de la función receptora.

Con todo ello, se pueden obtener ya los valores exactos de la optimación de las activaciones de funciones, teniendo en cuenta las ecuaciones [6.2.5], [6.2.6] y [6.2.7], y los valores obtenidos para los procedimientos. Los resultados se obtienen en las siguientes tablas, en donde se consignan solo los valores para el caso sencillo de copia del valor de retorno (es decir, se ha usado $c[ce/d]f\text{-copia.}$); los otros valores son en general medio punto más altos, pero por lo demás no muy diferentes.

	Coste según criterio 1	Coste según criterio 2a	Coste según criterio 2b
$c[ce]f_{\text{r-llam}_{\text{est}}}$	18	21	64
$c[ce]f_{\text{cr-llam}_{\text{est}}}$	18	21	64
$c[ce]f_{\text{nr-llam}_{\text{est}}}(d)$	$16 + 2 \times d$	$18 + 3 \times d$	$55 + 9 \times d$
$c[ce]f_{\text{r-llam}_{\text{opt}}}$	4	6	18
$c[ce]f_{\text{cr-llam}_{\text{opt}}}$	7	6	24
$c[ce]f_{\text{nr-llam}_{\text{opt}}}(d)$	$8 + 2 \times d$	$9 + 3 \times d$	$28 + 9 \times d$

Tabla 6.2.32 Costes de las distintas clases de secuencias de llamada para activaciones finales de funciones, usando la cadena estática

	Coste según criterio 1	Coste según criterio 2a	Coste según criterio 2b
$r\text{-opt}[ce]f_{\text{r-llam-ret}}$	8	7	$6,7$
$r\text{-opt}[ce]f_{\text{cr-llam-ret}}$	$4,57$	7	$5,08$
$r\text{-opt}[ce]f_{\text{nr-llam-ret}}(d)$	$\frac{30 + 2 \times d}{8 + 2 \times d}$	$\frac{39 + 3 \times d}{9 + 3 \times d}$	$\frac{113 + 9 \times d}{28 + 9 \times d}$

Tabla 6.2.33 Razones de optimación (usando la cadena estática) para las distintas clases de activaciones finales de una función

$r\text{-opt}[ce]/nr_{llam-ret}(d)$	Coste según criterio 1	Coste según criterio 2a	Coste según criterio 2b
$d=2$	$34/12=2'8\hat{3}$	$45/15=3$	$131/46=2'84$
$d=3$	$36/14=2'57$	$48/18=2'6$	$140/55=2'54$
$d=4$	$38/16=2'37$	$51/21=2'42$	$149/64=2'32$
$d=5$	$40/18=2'2$	$54/24=2'25$	$158/73=2'16$

Tabla 6.2.34 Primeros valores de la razón de optimación (usando la cadena estática) para una activación final no recursiva de función según la diferencia de anidamiento

Los valores para el *display* no son tampoco muy diferentes, y se pueden calcular de la misma manera. Con los obtenidos para la cadena estática, podemos representar gráficamente, en términos absolutos y relativos, el efecto de la optimación en las funciones, como se hizo con los procedimientos.

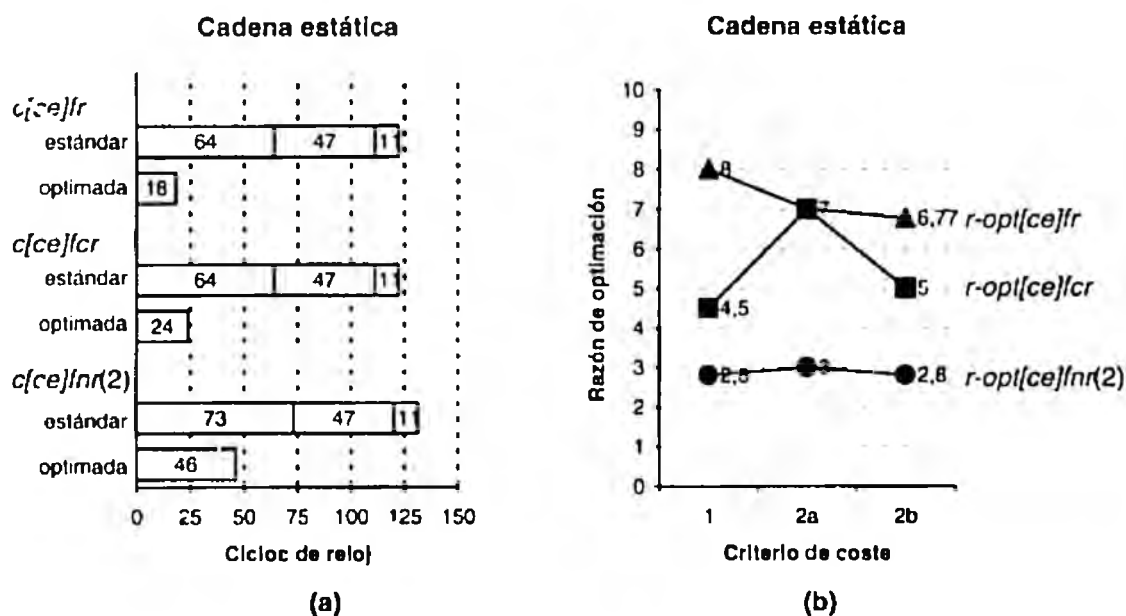


Figura 6.2.3 Comparación (a) entre los costes (usando el criterio 2b), y (b) entre las razones de optimación de las distintas clases de activaciones finales de funciones, usando la cadena estática

Siguen observándose, como para las activaciones de procedimientos, tres patrones bien diferenciados de optimación de activaciones finales de funciones: las mejores son las recursivas, a medio camino quedan las que estamos llamando "casi recursivas", y las peor paradas, lógicamente, son las no recursivas que involucran funciones de distinto anidamiento. Sin embargo,

existe menor separación entre ellas, por lo que, para completar este análisis, conviene comparar los resultados obtenidos para las dos clases de subprogramas, lo que se recoge en la Figura 6.2.4 y la Figura 6.2.5.

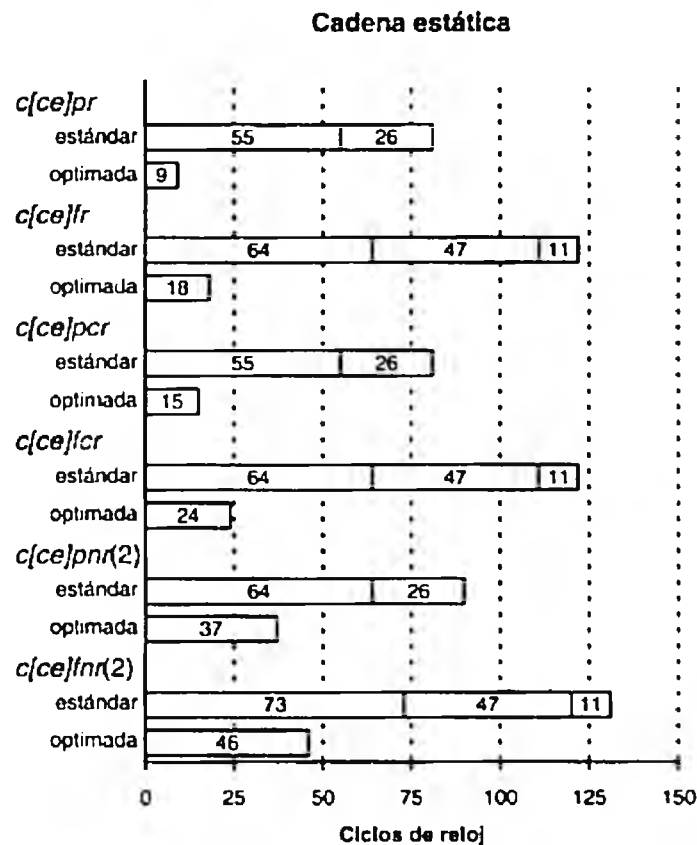


Figura 6.2.4 Comparación entre los costes (usando el criterio 2b) de las distintas clases de activaciones finales de procedimientos y funciones, usando la cadena estática

Se pueden extraer diversas conclusiones. Primero, en cuanto a las activaciones recursivas, claramente la optimación es peor para las funciones que para los procedimientos, aunque el criterio de coste en que más diferencia existe, el criterio 2a, puede no ser el más indicado para establecer la comparación, por los valores extremos que produce en este caso. En términos absolutos vemos que la llamada recursiva optimada de funciones cuesta el doble que la de procedimientos; ello es debido, lógicamente, a que esta última solo incluye el salto al comienzo del procedimiento, mientras que para las funciones debe realizarse obligatoriamente la colocación del centinela de definición del valor de retorno, operación igual de costosa que el propio salto.

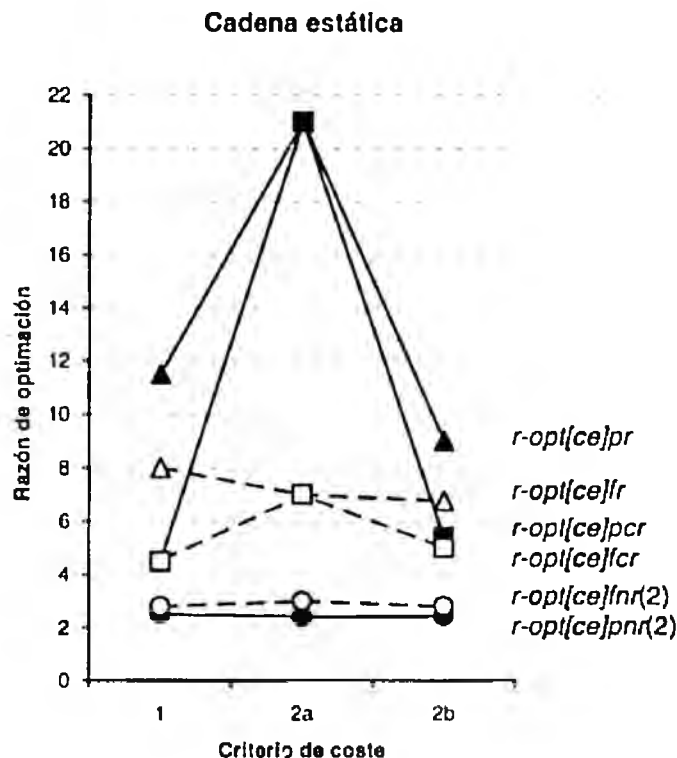


Figura 6.2.5 Comparación entre las razones de optimación de las distintas clases de activaciones finales de procedimientos y funciones, usando la cadena estática

Si en los procedimientos se considerase la operación de inicialización de los centinelas de definición de los datos locales (en caso de existir), esa diferencia se eliminaría totalmente, e incluso la razón de optimación para los procedimientos disminuiría, haciéndose menor que la de las funciones. Así y todo, consideramos una razón de optimación de 6 (con el criterio 2b) como un resultado extremadamente interesante. Por otro lado, también hay que tener en cuenta que el número de ciclos de reloj que se evitan en la llamada optimada de procedimientos es de 72, mientras que se eliminan 104 en cada llamada recursiva optimada de funciones, aunque luego el porcentaje relativo de mejora sea menor en el último caso.

¿Quiere decir esto que es mejor utilizar procedimientos que funciones, al menos desde el punto de vista temporal? Pensamos que no se puede tomar tal conclusión tan a la ligera. Desde luego, vemos en la Figura 6.2.4 que los costes de las llamadas a funciones son algo más costosas que las de procedimientos, en cualquier modelo que se considere. Pero recordemos que estas cifras no tienen

en cuenta la presencia de parámetros formales, ni de datos locales, y en cierto modo, el valor de retorno de una función puede considerarse como un dato local. Es decir, la comparación habría que establecerla en todo caso entre funciones y procedimientos destinados a los mismos cálculos. Concretamente, el concepto de función como subprograma con valor de retorno puede simularse con un procedimiento añadiendo un parámetro formal por referencia que equivale al valor de retorno. Es interesante discutir esta variación, porque nos permitirá completar ciertos aspectos sobre la diferencia entre procedimientos y funciones que analizamos en el apartado 5.1.1, pero lo dejaremos para el próximo capítulo, en que evaluaremos el impacto de la optimación en la metodología de programación habitual en el lenguaje PASCAL.

En cuanto a las activaciones finales casi recursivas, que involucran funciones del mismo anidamiento, obtenemos una razón de optimación casi tan buena como para los procedimientos. Y esta mejora se hace ya evidente al considerar las activaciones finales no recursivas entre funciones de distinto anidamiento: los resultados son un poco mejores que con la otra clase de subprogramas, y nunca bajan de 2. Esta tendencia tiene una razón clara: cuando existen pocas operaciones en la secuencia de llamada optimada, añadir la instrucción para colocar el centinela de definición del valor de retorno tiene un gran efecto, pero a medida que la secuencia de llamada optimada precisa más operaciones, esa influencia se va atenuando, y empieza a prevalecer en el grado de mejora el alto coste de la secuencia de retorno de función que se elimina con la optimación.

6.2.4 Comentarios finales

Terminamos la sección con una serie de consideraciones que recapitulan los resultados conseguidos. El primer punto en el que hay que insistir es que el interés primordial de la optimación de las activaciones finales de procedimientos y funciones en PASCAL reside en el comportamiento en espacio constante, es decir, las activaciones de esta clase (en las restricciones conocidas) pueden procesarse de manera que no se consuma espacio en la pila de control durante la ejecución. Ello permite, como se sabe, utilizar la recursividad sin necesidad de recurrir a las construcciones iterativas, y sin preocuparse por una

posible terminación anormal del programa durante la ejecución debida a un desbordamiento de la pila.

Por otro lado, la optimación consigue un mejor comportamiento temporal de las activaciones finales, al menos en lo referente a los procesos de manejo de los elementos del registro de activación que no sean parámetros o datos. Esto se debe a la eliminación de ciertas operaciones de la secuencia de llamada, y a que se evita una secuencia de retorno por cada activación optimada; en el caso de las funciones, se elimina además la operación de copia del valor de retorno. Las activaciones recursivas son las que conducen a la mejor *ratio* de optimación, pues la secuencia de llamada optimada consta prácticamente solo del salto al comienzo del código del subprograma; por ello, su comportamiento es similar a lo que se entiende por "comportamiento iterativo" en todos los aspectos. Las activaciones no recursivas de un subprograma con el mismo anidamiento que el llamador también resultan en una alta optimación temporal, y el peor comportamiento lo tienen las activaciones no recursivas de subprogramas de distinto anidamiento que el llamador, pero aun la *ratio* de optimación está en torno a 2.

El efecto de esta mejora temporal, sin embargo, puede no ser observable dependiendo de diversos factores: número de parámetros del subprograma, mecanismo de evaluación de los parámetros actuales y asignación a los formales, datos locales, mecanismo de vigilancia de la definición de los mismos, y complejidad del propio código. Por ello, insistimos de nuevo en que las cifras conseguidas nunca pueden aplicarse sin más a una activación o secuencia de activaciones completas de un procedimiento o función, sino exclusivamente a los procesos de llamada. En todos los factores anteriores es determinante la estrategia de diseño elegida en el compilador, por lo que solo un estudio de cada compilador concreto puede indicar si se produce mejora en el comportamiento temporal.

En cualquier caso, también es preciso matizar que consideramos esta mejora en el tiempo de ejecución, a pesar del exhaustivo análisis que se le ha dedicado, como secundaria en el interés de la optimación. Es decir, aún en el caso de que el proceso de llamada optimada tuviera un comportamiento peor que en el modelo dinámico habitual para el lenguaje, seguiríamos considerando la optimación interesante por el mero hecho del crecimiento nulo de la pila de control que se consigue. Y desde luego, puede ser que pequeñas variaciones del

modelo dinámico de partida, como veremos dentro de poco, impidan que la optimación produzca la mejora temporal observada, pero sin duda podrá conseguirse el comportamiento en espacio constante, por lo que la optimación propuesta podrá aplicarse a una amplia variedad de implementaciones alternativas, y seguirán siendo válidas las consecuencias metodológicas de este hecho.

6.3 REQUISITOS ESTÁTICOS DE LA OPTIMACIÓN

La sección que ahora iniciamos se dedica a establecer los requisitos del compilador, en sus diferentes fases, para llevar a cabo la optimación. Como se ha avanzado en capítulos anteriores, existen dos aspectos primordiales: por un lado, la detección de las activaciones que pueden resultar optimadas, y por otro, la generación de código necesaria. Iniciaremos el análisis presentando la estructura habitual de un compilador típico de PASCAL; después estudiaremos las variaciones necesarias para la optimación, que indudablemente van a existir, y evaluaremos el alcance de las mismas.

6.3.1 Un compilador clásico de PASCAL

El aspecto más notable del lenguaje PASCAL en lo referente a su procesamiento quizá sea la uniformidad de las diferentes implementaciones, es decir, hay una estructura más o menos constante en los compiladores, sobre todo en comparación con otros lenguajes. Welsh, Sneeringer y Hoare [1977] ya indican que uno de los objetivos de diseño del lenguaje, según el propio Wirth, era "to keep the compiler relatively compact and efficient" [Jensen y Wirth 1975: 8]. Welsh, Sneeringer y Hoare concluyen entonces que este objetivo favorecía, a pesar de no indicarse explícitamente en el "Report", la organización del compilador en un paso, lo que fue seguido casi al pie de la letra por los diferentes diseñadores de compiladores. Ello conduce, según estos autores muestran, a una serie de pequeños problemas, que se han corregido totalmente en la definición normalizada del lenguaje [Jensen y Wirth 1991], justamente a

nuestro entender, para seguir favoreciendo la estructura de un paso del compilador.

Una mirada a los primeros compiladores de PASCAL permite obtener otro punto de acuerdo prácticamente unánime: la técnica descendente (con un símbolo de anticipación, es decir, LL(1)) para el análisis sintáctico, y por tanto, como guía del proceso de compilación. El propio primer compilador de Wirth [1971a] y la segunda versión del mismo [Ammann 1981: 65], que incorporaba los cambios de la revisión del lenguaje, utilizaban procedimientos (posiblemente) recursivos asociados a cada uno de los no terminales de la gramática independiente del contexto que representa la sintaxis del lenguaje. Estos dos compiladores son fundamentales para entender que se haya producido tal acuerdo en la estructura de los compiladores de PASCAL a lo largo de la historia, no solo por favorecerlo la propia definición del lenguaje, sino por otra razón: son uno de los primeros ejemplos, después de LISP [Allen 1978: 307-309], de definición "metacircular", y de uso de la técnica conocida como *bootstrapping*.

Por ello, al estar los compiladores originales para el CDC 6000 escritos en el propio lenguaje PASCAL, subsecuentes versiones para otras máquinas se construyeron utilizando los primeros, y heredando de manera prácticamente total su estructura. Así, Welsh [1981: 171-173] explica que los compiladores para el ICL 1900 construidos en la Queen's University de Belfast se realizaron, el primero (1971) mediante *bootstrapping* con el compilador de Wirth, y el segundo (1974-76) siguiendo la técnica de construcción propuesta por Ammann, es decir, justamente los dos compiladores "originales" del ETH de Zurich para el CDC 6000. Del mismo modo, Rees y colaboradores [1981: 261] indican, al explicar otro compilador de PASCAL, que ya era típico para esta tarea usar *bootstrapping* a partir del compilador existente más cercano.

Creemos que los compiladores comerciales del lenguaje no se alejan casi nada de estas ideas; la razón es que la organización comentada es la que conduce en efecto a una mayor eficiencia del compilador, un requisito obvio desde el punto de vista comercial; la calidad del código generado, que es otro de los requisitos, evidentemente, puede incrementarse con diversas técnicas de optimización conocidas, que pueden aplicarse generalmente (salvo algún caso particular) sin problemas en dicha estructura. Quizá el único punto de diferencia sería el lenguaje de escritura del compilador: es posible que no se

utilice la definición metacircular ni el *bootstrapping*, sobre todo, si se desarrolla para una única máquina, y si se quiere sacar el máximo partido de las características de la máquina que ejecuta el compilador.

Nuestra hipótesis básica acerca de la estructura clásica de un compilador de PASCAL radica, por lo tanto, en la organización en un paso con un analizador sintáctico descendente. Dado esto, realmente no existen muchas alternativas. Probablemente, y más si el lenguaje de escritura no es el propio PASCAL, ya no se usen procedimientos recursivos para implementar el analizador sintáctico, sino que se utiliza el esquema predictivo no recursivo [Aho, Sethi y Ullman 1990: 190-191], basado en una pila y guiado por la tabla de análisis sintáctico, que se construye automáticamente a partir de la gramática, lo que reduce los recursos necesarios por el compilador, como se sabe.

Donde sí puede haber diferencias es en la metodología de implementación del procesamiento semántico. Llamamos así a las tareas del compilador que se realizan después del análisis sintáctico; engloba, por supuesto, el análisis semántico, y también una parte más o menos extensa de la generación de código, dependiendo esto de la estructura elegida. Considerando que el compilador es de un paso, el procesamiento semántico podría incluir la generación de una representación intermedia, la cual sería dada a una fase de generación de código objeto posterior; también podría evitarse esta representación intermedia y realizar directamente la generación de código.

Como es ampliamente conocido, existen dos filosofías principales para la especificación del procesamiento semántico: una es la *gramática con atributos*, que es la técnica más formal, pero por ello muy compleja de implementar; otra es una técnica más *ad hoc*, conocida algunas veces como *esquema de traducción*, que es teóricamente una concreción de la clase de gramáticas con atributos por la izquierda bajo un analizador descendente, y que consiste básicamente en insertar las reglas semánticas en lugares adecuados de las partes derechas de las reglas de producción de la gramática independiente del contexto, de manera que las reglas semánticas son ejecutadas cuando se accede a ellas durante el análisis sintáctico. Estas reglas semánticas son las que realizan el procesamiento semántico. Supondremos entonces que el compilador típico de PASCAL está basado en un esquema de traducción, ya sea producto de una formalización de la semántica a través de una gramática con atributos por la izquierda, o en una

serie de acciones que se han insertado en las reglas de la gramática que refleja la sintaxis sin una formalización previa.

El procesamiento semántico, en tal caso, tiene una característica muy típica en PASCAL, al estar perfectamente separadas las secciones que definen las características de los nombres (la parte de declaraciones de cada bloque) y las secciones que expresan los cálculos a efectuar (la parte de sentencias). Quiere esto decir que el trabajo necesario durante el procesamiento de las declaraciones se limita prácticamente a rellenar la tabla de símbolos con la información pertinente, mientras que el esfuerzo de análisis semántico y posible generación de código se efectúa exclusivamente en la parte de sentencias. Aunque el procesamiento de las declaraciones puede necesitar estructuras de datos particulares, no es realmente determinante para nuestro objetivo. Nos interesa el modo en que se tratan las partes de sentencias de los bloques, pues ahí es donde ha de detectarse y procesarse la optimización que planteamos.

A este respecto, podría en principio pensarse en un procesamiento totalmente "local", en el sentido de que solo se necesitaría mantener en memoria en cada momento una parte pequeña de información sobre el programa fuente; desde el punto de vista abstracto, eso quiere decir que solo se precisa construir explícitamente porciones muy limitadas del árbol sintáctico: por ejemplo, toda una expresión o partes concretas de algunas sentencias.

El enfoque del extremo contrario consistiría en mantener en memoria todo el árbol correspondiente a la parte de sentencias de un bloque o subprograma; este árbol se construiría a la vez que se realiza el análisis semántico, por lo que ya quedaría "decorado" con la información adicional precisa, y una vez terminado se recorre para realizar la generación de código. Desde luego, esta alternativa exige muchos más recursos de memoria para el compilador, y en principio, resultaría desechada frente a la primera, que solo precisa la acumulación de información de forma muy limitada, en una estructura de datos, que típicamente es una pila, conocida como *pila semántica*, y que no ha de confundirse con la pila del analizador sintáctico.

En cuanto a la generación de código, ya sea en forma intermedia o código objeto, el único aspecto a reseñar es muy conocido: la generación de los saltos hacia adelante, si se utiliza algún código intermedio sin etiquetas simbólicas para las direcciones de los saltos, o es directamente código objeto.

Estos saltos son necesarios en el código de las expresiones booleanas, y de las sentencias de control del flujo, incluyendo la propia sentencia de salto incondicional. Entonces, es preciso utilizar la técnica conocida como *relleno de retroceso*, o *backpatching* [Aho, Sethi y Ullman 1990: 515-520]. Así, Ammann [1981: 76] ya comenta que era necesario mantener en memoria el código de cada subprograma hasta terminar la compilación del cuerpo del mismo: entonces, se dejan sin rellenar las posiciones que se refieren a direcciones aún no conocidas, y se mantiene una lista de los huecos; cuando se resuelve la dirección, se rellena el hueco, y se elimina de la lista. Al llegar al final del subprograma, se escribe su código de una vez en el dispositivo de salida. Esto supone, según el propio Ammann [*op. cit.*: 81], unos requisitos de memoria relativamente altos (también hay que considerar que estamos hablando del año 1972). Por supuesto, esto no es necesario si se genera un programa en ensamblador, pero entonces el compilador no es realmente de un paso, al ser precisa la fase de ensamblado posterior. Aunque muchos compiladores pueden ofrecer una salida en ensamblador como opción, lo habitual es que se genere código objeto relocizable, lo que facilita la compilación separada (no existente en la definición estándar del lenguaje PASCAL pero presente en casi todas las implementaciones actuales de una forma u otra), por lo que el requisito de mantener el código de cada subprograma en memoria es prácticamente de obligado cumplimiento.

En definitiva, la hipótesis completa acerca de un compilador clásico de PASCAL es la estructura en un paso, guiado por un analizador descendente que lanza acciones de procesamiento semántico que actúan localmente sobre la estructura del programa fuente (esto es, sin construir explícitamente más que pequeñas porciones del árbol sintáctico) y que generan código que ha de mantenerse en memoria para cada subprograma hasta su terminación. Esta estructura se puede representar gráficamente como en la Figura 6.3.1.

En ella, se engloban en cajas elípticas las principales partes del programa que realiza la compilación (esto es, analizador léxico, analizador sintáctico, y acciones de procesamiento semántico), y en cajas rectangulares las estructuras de datos necesarias para el mismo, tal como se ha explicado anteriormente (o bien se trata de estructuras ampliamente conocidas y utilizadas, como el autómata finito determinista para el análisis léxico o la gramática independiente del contexto para el análisis sintáctico).

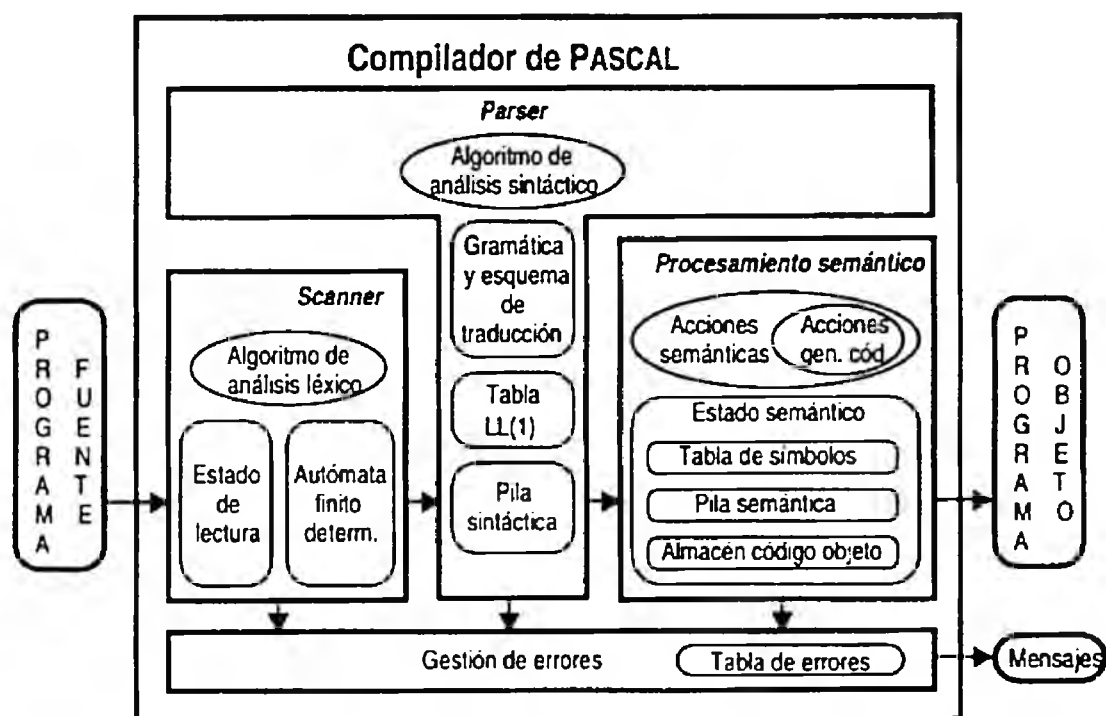


Figura 6.3.1 Estructura típica de un compilador de PASCAL

6.3.2 Detección de las activaciones optimables

El estudio de los capítulos anteriores permitió reflejar en una serie de restricciones las condiciones necesarias y suficientes, desde el punto de vista sintáctico, para que una activación de procedimiento o función fuera optimable. En este apartado debemos explicar la manera en que el compilador detecta tales condiciones y procede a marcar la activación como optimable. Para ello, clasificaremos primero las condiciones:

- La Restricción 3.2.1, la Restricción 4.2.1, la Restricción 5.2.1, y la Restricción 5.3.1 expresan la característica de finales que han de tener tales activaciones (con la condición adicional de aparecer en una sentencia de asignación del valor de retorno para las funciones); las denominaremos *restricciones de posición final*.
- La Restricción 4.2.2 y la Restricción 5.3.2 expresan el requisito adicional de las activaciones no recursivas de no ser de un subprograma local en el llamador; nos referiremos a ellas como *restricciones de anidamiento*.

- Por su parte, la Restricción 3.2.2, la Restricción 3.2.3, la Restricción 4.2.3, la Restricción 4.2.4 y las correspondientes para las funciones recogen los requisitos acerca de los parámetros actuales por referencia (que no han de ser datos locales o parámetros formales por valor del llamador), procedimentales y funcionales (que no han de ser subprogramas locales del llamador); se llamarán *restricciones de parámetros*.
- Finalmente, la Restricción 5.3.3 y la Restricción 5.3.4 señalan la relación que ha de haber entre los tipos de los valores de retorno de las funciones involucradas en una activación final no recursiva; se denominarán *restricciones del valor de retorno*.

Estudiaremos cada uno de estos requisitos paso por paso, pero vamos a dejar el primero (la restricción de posición final) para el final (como su nombre indica).

En principio, verificar las tres últimas clases de restricciones solo exige mantener en memoria como mucho el árbol de la sentencia correspondiente, e incluso podría no ser necesario mantenerlo al completo. Tomemos por ejemplo la sentencia de activación de un procedimiento:

$$\langle \text{proc-id} \rangle \ (\ \langle \text{par}_1 \rangle, \ \langle \text{par}_2 \rangle, \dots \langle \text{par}_n \rangle \)$$

y olvidemos de momento la necesidad de verificar primero la restricción de posición final. La aparición del identificador del procedimiento permite ya decidir si se trata de una activación recursiva o no, con una muy sencilla comprobación de la tabla de símbolos; al mismo tiempo prácticamente, puede ya establecerse la restricción de anidamiento, y en el caso de ser una activación no recursiva, si se trata de un procedimiento del mismo anidamiento o no. Por lo tanto, antes de llegar los parámetros ya se sabría el tipo de procesamiento que debería darse a los mismos.

La manera de hacer esta tarea justamente es un tanto peliaguda, pues es quizá donde más diferencias puede haber entre distintas implementaciones, y pueden tener mucha influencia cuestiones relativas a la máquina objeto, como ya hemos avanzado en varias ocasiones. En principio, la verificación de las restricciones de parámetros no son nada complicadas, pues solo hay que comprobar, si el parámetro formal es variable, procedimental o funcional, que el objeto que se proporciona como parámetro actual no sea uno de los no admitidos, y para ello basta una mirada a la tabla de símbolos.

En cuanto a la propia generación de código, un compilador estándar podría simplemente hacerla según van apareciendo los parámetros, con lo que no se precisaría mantener la lista de árboles de estos en memoria; tampoco lo necesitaría en principio el compilador optimador, pues solo ha de observar la Condición 3.2.9 y la Condición 4.2.9, que indica que no ha escribirse un parámetro formal antes de haberlo usado como parte de un parámetro actual, lo cual afecta al código que se generará. Finalmente, la optimación consistente en no actualizar un parámetro actual que solo conste del identificador del parámetro formal a que corresponde es bastante sencilla de detectar, a pesar de requerir algo de esfuerzo adicional en el compilador respecto al normal.

Con las activaciones finales de funciones, expresadas sintácticamente por el esquema:

$$\langle \text{func-id}_1 \rangle := \langle \text{func-id}_2 \rangle (\langle \text{par}_1 \rangle, \langle \text{par}_2 \rangle, \dots \langle \text{par}_n \rangle)$$

se tiene la dificultad adicional de la Restricción 5.2.1 y la Restricción 5.3.1, que obliga a comprobar que el lado derecho de la sentencia de asignación consta exclusivamente de la activación de una función. En cierto modo, el caso de los procedimientos es más sencillo porque la sintaxis para las llamadas optimadas es la misma que para las no optimadas. Ahora, el esquema de las activaciones finales de funciones es solo un caso particular de la sintaxis para la sentencia de asignación:

$$\langle \text{id} \rangle := \langle \text{expresión} \rangle$$

pues no solo se trata de verificar que el identificador del lado izquierdo es el de una función (lo cual ha de hacerlo el compilador estándar y no supone ningún problema), sino que ha de comprobarse que la expresión del lado derecho consta solamente de una llamada a una función. Supongamos dos ejemplos concretos:

$$\begin{aligned} f &:= g (\langle \text{par}_1 \rangle, \langle \text{par}_2 \rangle, \dots \langle \text{par}_n \rangle) \\ f &:= g (\langle \text{par}_1 \rangle, \langle \text{par}_2 \rangle, \dots \langle \text{par}_n \rangle) * k \end{aligned}$$

Durante el procesamiento de la activación de g , es imposible distinguir si ella por sí sola formará la expresión completa o actúa como operando de otra expresión más compleja. Por lo tanto, es preciso obtener el árbol completo de la expresión del lado derecho de la sentencia de asignación del valor de retorno para verificar que consta solamente de una activación de función, antes de poder generar código, ya sea una llamada optimable o no.

Si se consideran además las restricciones del valor de retorno, este hecho es aún más evidente, pues la estructura de las expresiones en PASCAL hace que las comprobaciones (y en su caso, generación de código) relativas a la compatibilidad de tipos en la asignación se realice *después* de procesar la expresión. Por lo tanto, en la medida que en el compilador optimador ese conocimiento debe ser previo a la generación de código, no puede discernirse durante el procesamiento de la expresión si ha de actuarse de un modo u otro. Antes de pensar en las consecuencias de este resultado, debemos analizar la manera en que se resuelven las restricciones de posición final.

La restricción de posición final se ofreció en forma completa en la Restricción 3.2.1. Recordemos que:

“Una sentencia está en posición final si es la última que aparece en la sentencia compuesta del bloque de un subprograma, o si, recursivamente, es la sentencia de una sentencia con etiqueta en posición final, o es una rama de una sentencia condicional en posición final, o es la sentencia de una sentencia *with* en posición final, o es la última sentencia de una sentencia compuesta en posición final”

Tomemos la parte de la gramática independiente del contexto del lenguaje PASCAL que corresponde a un bloque; esta gramática es un poco distinta de la ofrecida en el apartado 3.2.1, porque está ya preparada para hacer un análisis sintáctico descendente. A este respecto, es necesario indicar que se han elegido, como habitualmente se hace, unas reglas para las sentencias condicionales que producen ambigüedad [Díaz Labrador 1994: 188-190]; los conflictos que surgen en la tabla de análisis sintáctico descendente se resuelven del modo conocido [Aho, Sethi y Ullman 1990: 196-197], por lo que aunque la gramática no es estrictamente LL(1), es válida para la realización de un análisis sintáctico correcto.

$$\langle \text{block} \rangle \rightarrow \langle \text{label-decl-part} \rangle \langle \text{const-decl-part} \rangle \langle \text{type-decl-part} \rangle \\ \langle \text{var-decl-part} \rangle \langle \text{proc-func-decl-part} \rangle \langle \text{comp-stment} \rangle$$

$$\langle \text{comp-stment} \rangle \rightarrow \text{begin } \langle \text{stment-list} \rangle \text{ end}$$

$$\langle \text{stment-list} \rangle \rightarrow \langle \text{stment} \rangle \langle \text{resto-stment-list} \rangle$$

$$\langle \text{resto-stment-list} \rangle \rightarrow ; \langle \text{stment-list} \rangle$$

$$| \lambda$$

$$\langle \text{stment} \rangle \rightarrow \langle \text{unlab-stment} \rangle$$

$$| \text{label} : \langle \text{unlab-stment} \rangle$$

$$\langle \text{unlab-stment} \rangle \rightarrow \langle \text{comp-stment} \rangle$$

$$| \text{id } \langle \text{resto-id-stment} \rangle$$

$$| \text{goto label}$$

$$| \text{if } \langle \text{expr} \rangle \text{ then } \langle \text{stment} \rangle \langle \text{resto-if-stment} \rangle$$

$$| \text{case } \langle \text{expr} \rangle \text{ of } \langle \text{case-list} \rangle \text{ end}$$

$$| \text{while } \langle \text{expr} \rangle \text{ do } \langle \text{stment} \rangle$$

$$| \text{repeat } \langle \text{stment-list} \rangle \text{ until } \langle \text{expr} \rangle$$

$$| \text{for id} := \langle \text{expr} \rangle \text{ for-dir } \langle \text{expr} \rangle \text{ do } \langle \text{stment} \rangle$$

$$| \text{with } \langle \text{record-var-list} \rangle \text{ do } \langle \text{stment} \rangle$$

$$| \lambda$$

$$\langle \text{resto-id-stment} \rangle \rightarrow := \langle \text{expr} \rangle$$

$$| \langle \text{var-ref} \rangle := \langle \text{expr} \rangle$$

$$| \lambda \quad \quad \quad (\dagger)$$

$$| (\langle \text{param-list} \rangle) \quad \quad \quad (\dagger\dagger)$$

$$\langle \text{resto-if-stment} \rangle \rightarrow \lambda$$

$$| \text{else } \langle \text{stment} \rangle$$

$$\langle \text{case-list} \rangle \rightarrow \langle \text{case-label-list} \rangle : \langle \text{stment} \rangle \langle \text{resto-case-list} \rangle$$

$$\langle \text{resto-case-list} \rangle \rightarrow ; \langle \text{case-list} \rangle$$

$$| \lambda$$

Obsérvese que la regla marcada con $\dagger$ es exactamente la activación de un procedimiento sin parámetros, y la regla con $\dagger\dagger$ es la activación de un procedimiento con parámetros. Por lo tanto, debería ser como mucho en los puntos señalados a continuación con $\bullet$:

$$\langle \text{resto-id-stment} \rangle \rightarrow \bullet \lambda$$

$$| \bullet (\langle \text{param-list} \rangle)$$

donde debería ser ya conocido si la activación es una activación final o no, pues la generación de código para la evaluación de los parámetros actuales depende

de ello. Obviamente, ese conocimiento no es posible en el caso de la segunda sentencia, y realmente tampoco en la primera, como veremos. Ello obliga a que la decisión sobre la posible característica de final de la llamada deba posponerse al menos hasta la finalización de la misma:

$$\langle \text{resto-id-stment} \rangle \rightarrow \lambda \bullet \\ | (\langle \text{param-list} \rangle) \bullet$$

Sin embargo, esto ya nos obliga a mantener el árbol correspondiente a la llamada, con los parámetros actuales incluidos, sin haber podido generar ni un solo trozo de código. Lo peor es que la cosa no acaba ahí.

Supongamos que la sentencia de activación de un procedimiento es una de las ramas de una sentencia *if-then-else*. Si fuera la rama de la parte *else*, aún podría pensarse en retrasar la generación de código hasta asegurarse de algún modo que esa sentencia condicional está en posición final. Pero el esfuerzo sería inútil, porque si la llamada aparece en la rama de la parte *then*, no hay modo de saber hasta quizás mucho después si se trata de una sentencia en posición final o no. Evidentemente, no es posible retrasar la generación de código de la parte *then* más que hasta el comienzo de la parte *else*, y en ese punto aún no se puede conocer absolutamente nada. Un argumento similar sirve para la sentencia *case* y para la sentencia *with*, que son los otros puntos en que pueden darse activaciones finales.

Como consecuencia de ello, obtenemos uno de los resultados más negativos de la optimación que estamos estudiando:

Condición 6.3.1. El compilador optimador precisa, para detectar las sentencias en posición final, mantener durante la compilación el árbol decorado completo de la parte de sentencias de cada subprograma y retrasar la generación de código hasta el término de dicha parte de sentencias.

Este es, en efecto, un requisito fuerte, dada la estructura clásica de un compilador de PASCAL, que presupone la necesidad de almacenar como mucho nada más que trozos locales del árbol decorado. Desde luego, las consecuencias de este resultado deben ser analizadas cuidadosamente, lo cual se hará en un próximo apartado de esta misma sección. Antes, concluiremos explicando la manera de detectar las activaciones optimables una vez que se tiene el árbol decorado de la parte de sentencias de un bloque.

Por supuesto, el árbol se construye paralelamente al análisis semántico, y cuando se finalice se habrán detectado todos los posibles errores del programa fuente; en el árbol queda reflejada por tanto toda la información semántica necesaria para la generación de código. Entonces, el análisis de las activaciones optimables se realiza recorriendo dicho árbol, sustituyendo los nodos que correspondan a esa clase de activaciones por los nodos particulares adecuados. Una vez hecho eso, el árbol se recorre de nuevo para generar código, tras lo cual se puede liberar la memoria dinámica reservada para el árbol.

Para empezar, también es conveniente almacenar en el árbol la mayor cantidad de información que ayude a la detección de las restricciones de las llamadas optimables. Por ejemplo, empezando por los procedimientos, el nodo correspondiente a una activación, en general, se puede representar por:

(PROC-CALL *<proc-id>* *<par₁>* *<par₂>* ... *<par_n>*)

En general, denotaremos los árboles mediante listas, como en LISP y SCHEME, en que el primer elemento de la lista representa la clase del nodo, y los demás elementos son los hijos que cuelgan o dependen del primero; estos a su vez pueden ser otras listas, es decir, otros árboles. Sin embargo, puesto que para la optimación existen cinco clases de procedimientos, conviene distinguir ya desde la misma construcción del árbol la clase de que se trata:

- si es una activación directamente recursiva, el nodo se denominará REC-PROC-CALL;
- si es la activación de un procedimiento del mismo anidamiento que el llamador, el nodo se denominará CREC-PROC-CALL (por "casi recursiva");
- si es la activación de un procedimiento de menor anidamiento que el llamador, el nodo se denominará NREC-PROC-CALL;
- si es la activación de un parámetro formal procedimental (ya sea del llamador o de otro procedimiento de jerarquía superior), el nodo se denominará FORM-PROC-CALL;
- finalmente, si es la activación de un procedimiento local del llamador, el nodo se denominará LOCL-PROC-CALL;

La necesidad de estos cuatro nodos es clara a partir del estudio que se realizó: el último corresponde al caso en que no es posible la optimación, y los tres primeros son los que producen comportamientos significativamente

distintos, según se ha encontrado en la sección previa. La detección de la clase de nodo es muy sencilla, pues una operación de consulta de la tabla de símbolos similar a la necesaria para establecer que el identificador es de un procedimiento permite conocer si ese identificador cae en uno u otro caso. La distinción, por otro lado, es interesante aunque no se lleve a cabo la optimación, pues ayuda a la generación de código en lo relativo al cálculo del enlace de acceso.

Para el caso de las funciones, la misma idea anterior sirve para los nodos que reflejan las llamadas propiamente dichas: se tendrían REC-FUNC-CALL, CREC-FUNC-CALL, NREC-FUNC-CALL, FORM-FUNC-CALL y LOCL-FUNC-CALL. Sin embargo, recordemos que la optimación se realiza sobre las sentencias de asignación de valor de retorno, cuya sintaxis es:

$$\langle unlabeled-statement \rangle \rightarrow id := \langle expr \rangle$$

con lo que el nodo del árbol será:

$$(\text{FUNC-SET } \langle func-id \rangle \langle expr \rangle)$$

Sin embargo, esta sentencia de asignación, para conducir a la optimación, ha de aparecer estrictamente en la parte de sentencias del bloque de la propia función, y no en la parte de sentencias de algún bloque local, por lo que conviene distinguir los dos casos:

$$\begin{aligned} & (\text{LOCAL-FUNC-SET } \langle func-id \rangle \langle expr \rangle) \\ & (\text{NOLOCAL-FUNC-SET } \langle func-id \rangle \langle expr \rangle) \end{aligned}$$

lo cual también es en general de ayuda aunque no se realice la optimación, pues cada caso conduce a diferente código (en el segundo ha de seguirse la cadena estática, no así en el primero, en que se genera una asignación local).

A continuación (Programa 6.3.1) mostramos ya la acción semántica encargada de recorrer el árbol decorado de la parte de sentencias de un procedimiento, y detectar las activaciones que resultan optimables. Debemos indicar que está escrita en el lenguaje SCHEME (esa es una de las razones para representar los árboles mediante listas), que es el lenguaje en que se ha implementado el prototipo de compilador optimador (*vide* sección 6.5 para los detalles), y que se han simplificado algunas cuestiones para facilitar la apreciación de la estructura principal.

```

; contain-tail-proc-calls? devuelve la existencia de optimaciones;
; como efecto lateral, modifica los nodos correspondientes

(define contain-tail-proc-calls?
  (lambda (proc-id tree ...)
    ; contain-loop? es un proceso recursivo de recorrido de tree

    (let (contain-loop? ((tree tree))
          (let ((stment-code (first tree)))
            (case stment-code

; casos base:
;   * llamada recursiva en p.f.: se comprueban parámetros, se optiman
;   si es el caso, y se modifica el nodo
;   * resto de llamadas (salvo local) en p.f.: se comprueban parámetros
;   y se cambia el nodo (no cabe la optimación de parámetros)

              ((<REC-PROC-CALL> <CREC-PROC-CALL>
                <NREC-PROC-CALL> <FORM-PROC-CALL>)
               (let ((actual-list (cddr tree))
                     (formal-list ...))
                 (cond
                  ((valid-params? actual-list formal-list)
                   (if (eq? stment-code '<REC-PROC-CALL>)
                       (optimize-params actual-list formal-list)
                       (set-car! tree
                                (case stment-code
                                  (<REC-PROC-CALL> '<TAIL-REC-PF-CALL>)
                                  (<CREC-PROC-CALL> '<TAIL-CREC-PF-CALL>)
                                  (<NREC-PROC-CALL> '<TAIL-NREC-PF-CALL>)
                                  (<FORM-PROC-CALL> '<TAIL-FORM-PF-CALL>))))
                   #t)
                  (else #f))))))

; casos recursivos:
;   * si una sent. con etiq. está en p.f., la sent. sin etiq. también
;   * si una if-then está en p.f., la rama then también
;   * si una with está en p.f., la rama then también

              ((<IF-THEN> <LABEL> <WITH>) (contain-loop? (third tree)))

;   * si una if-else está en p.f., la rama then y la rama else también
;   * si una case está en p.f., cada una de sus ramas también

              ((<IF-ELSE> <CASE>)
               (there-exists?
                (map contain-loop? (cddr tree)) (lambda (x) x)))

;   * si una sent. compuesta está en p.f., la última también

              ((<SEQ>) (contain-loop? (car (last-pair tree)))))

; otros casos: o bien no conduce a p.f. (e.g. while) o no es <PROC-CALL>
; (e.g. asignación) o es <LOCAL-PROC-CALL>

              (else #f))))))

; fin contain-tail-proc-calls?

```

Programa 6.3.1 Esquema de la acción semántica de recorrido del árbol decorado de la parte de sentencias de un procedimiento que verifica las restricciones de posición final de las activaciones de procedimientos

Siguiendo el enunciado de la Restricción 3.2.1 es sencillo entender la estructura de esta acción semántica. Recursivamente, se examina la sentencia de una sentencia con etiqueta, o la última sentencia de una sentencia compuesta, o las ramas de una sentencia condicional, o la sentencia de una sentencia with. Cuando se llega al nodo que representa una activación de un procedimiento (siempre que no sea local en el procedimiento llamador), se realizan las comprobaciones de parámetros, mediante la función `valid-params?`, y en caso afirmativo, se modifica el nodo correspondiente a la activación; en el caso de ser una activación recursiva optimable, se realiza además la optimación de los parámetros expresada por la Condición 3.2.10, la Condición 3.2.11 y la Condición 3.2.12, mediante la función `optimize-params`. Las dos operaciones son muy fáciles de implementar: simplemente se trata de consultar la tabla de símbolos, y no es necesario ofrecer siquiera el esquema de las mismas; con ello, consideraremos el análisis de las restricciones de parámetros un problema resuelto.

La verificación de las restricciones de posición final para las funciones requiere un poco más de trabajo, a pesar de compartir la estructura básica de la acción semántica del Programa 6.3.1, debido a que ahora nos enfrentamos a una sentencia, la de asignación del valor de retorno, de estructura más compleja. El esquema de la acción semántica aparece a continuación, en el Programa 6.3.2. Se trata de buscar en este caso el nodo `LOCAL-FUNC-SET` (mientras que el nodo `NOLOCAL-FUNC-SET` se rechaza implícitamente); si se encuentra, para que resulte una activación de función optimable, la expresión ha de contener únicamente una llamada a una función no local.

Es necesario comentar que no se encuentra en la función del Programa 6.3.2 ninguna referencia a las restricciones del valor de retorno, concretamente a la Restricción 5.3.3. Para comprenderlo, debemos recapitular la discusión de la sección 5.3 que condujo a las restricciones mencionadas, acerca del tratamiento de las expresiones con tipos subrango en PASCAL. Concretamente, supongamos que f es una función cuyo tipo es subrango:

$$\text{function } f \text{ (} \langle par_1 \rangle; \dots \langle par_n \rangle \text{) : rg}$$

Si se tiene una llamada a esta función, el nodo de la clase expresión que se construye será:

$$(\text{FUNC-CALL } rg \ f \ \langle par_1 \rangle \dots \langle par_n \rangle)$$

```

; contain-tail-func-calls? devuelve la existencia de optimaciones;
; como efecto lateral, modifica los nodos correspondientes

(define contain-tail-func-calls?
  (lambda (tree ...)
    ; contain-loop? es un proceso recursivo de recorrido de tree

    (let contain-loop? ((tree tree))
      (let ((stment-code (first tree)))
        (case stment-code

          ; caso base: asignación local del valor de retorno

          (<LOCAL-FUNC-SET>
            (let* ((expr (third tree))
                  (expr-code (first expr)))
              (case expr-code

                ; si es exclusivamente una llamada a función (salvo local)

                ((<REC-FUNC-CALL> <CREC-FUNC-CALL>
                  <NREC-FUNC-CALL> <FORM-FUNC-CALL>)
                  (let ((actual-list (cdddr tree))
                        (formal-list ...))
                    (cond
                     ((valid-params? actual-list formal-list)
                      (if (eq? expr-code <REC-FUNC-CALL>)
                          (optimize-params actual-list formal-list)
                          (set-car! tree
                            (case expr-code
                              (<REC-FUNC-CALL> <TAIL-REC-PF-CALL>)
                              (<CASIREC-FUNC-CALL> <TAIL-CREC-PF-CALL>)
                              (<NOREC-FUNC-CALL> <TAIL-NREC-PF-CALL>)
                              (<FORMAL-FUNC-CALL> <TAIL-FORM-PF-CALL>)))
                        (set-cdr! tree (cdddr expr))
                        #t)
                     (else #f))))))

          ; si la expresión es otra cosa

          (else #f))))))

; casos recursivos: similares a los de contain-tail-proc-calls?

; otros casos: o bien no conduce a p.f. (e.g. while) o no es <FUNC-SET>
; (e.g. asignación) o es <NOLOCAL-FUNC-SET>

(else #f))))))

; fin contain-tail-func-calls?

```

Programa 6.3.2 Esquema de la acción semántica de recorrido del árbol decorado de la parte de sentencias de una función que verifica las restricciones de posición final de las sentencias de asignación del valor de retorno de funciones

donde el segundo hijo está ocupado en general por el tipo de la expresión. Sin embargo, ya dijimos que las normas del lenguaje obligan a promocionar el tipo

subrango de un factor a su tipo *host* correspondiente [Jensen y Wirth 1991: 166]. Por ello, ante una sentencia de asignación del valor de retorno como:

$$f := f (\langle par_1 \rangle, \dots \langle par_n \rangle)$$

el árbol que se construiría siguiendo al pie de la letra las indicaciones del lenguaje será:

```
( LOCAL-FUNC-SET
  f
  ( RANGE-CHECK
    rg
    ( HOST-PROM
      host
      ( FUNC-CALL rg f (par1) ... (parn) ) ) ) )
```

Entonces, se perdería la posibilidad de considerar esta sentencia como optimable, pues la expresión no consta exclusivamente de una llamada a función.

En general, el nodo HOST-PROM no producirá ningún código particular, pero sí lo hará el nodo RANGE-CHECK, concretamente una verificación dinámica de que el resultado de la expresión se mantiene entre los valores admitidos para el tipo *rg*. Claramente, esta verificación es redundante en este caso, porque realmente el tipo de los dos lados de la asignación es el mismo, lo mismo que lo sería en general en una sentencia de asignación cualquiera:

$$a := b$$

si las variables *a* y *b* son del tipo *rg*. Obsérvese que, si no se pone cuidado en esta cuestión, un compilador de PASCAL podría evitar el nodo HOST-PROM para ahorrar espacio (o podría no guardar la información necesaria si no construye explícitamente el árbol):

```
( LOCAL-FUNC-SET
  f
  ( RANGE-CHECK
    rg
    ( FUNC-CALL host f (par1) ... (parn) ) ) )
```

pero entonces es obligatorio generar el código de verificación de los límites del rango.

Sin embargo, en un compilador cuidadoso, al procesar la sentencia de asignación en lo relativo a la compatibilidad de tipos, puede aplicarse la regla evidente de sustituir:

```
( RANGE-CHECK
  rg
  ( HOST-PROM
    host
    ( expr-code rg ... ) ) )
```

por:

```
( expr-code rg ... )
```

evitando así una operación innecesaria (y relativamente costosa) de comprobación dinámica. De este modo, el árbol que realmente se construye para la sentencia de asignación del valor de retorno considerada es:

```
( LOCAL-FUNC-SET
  f
  ( FUNC-CALL rg f <par1> ... <parn> ) ) )
```

con lo que la sentencia es perfectamente optimable (al menos en lo que a los requisitos considerados se refiere). Por lo tanto, la verificación de las restricciones del valor de retorno quedan integradas en la comprobación general de la compatibilidad de tipos ante la asignación, no precisando ninguna otro análisis particular.

En principio, no hay más que comentar a este respecto. Realmente, el módulo de detección de la optimación resultó muy sencillo de implementar, en comparación con los demás módulos del procesamiento semántico, presentes en cualquier compilador del lenguaje. A pesar de utilizar la recursividad, el coste espacial y temporal que añade al compilador no es demasiado grande, pues la verificación se dirige directamente a las sentencias en posición final, que claramente no pueden ser muchas en cualquier subprograma. Por supuesto, el coste añadido respecto a un compilador sin esta clase de optimación viene recogido en la Condición 6.3.1: la necesidad de almacenar el árbol completo correspondiente a la parte de sentencias de un bloque es un requisito muy fuerte, cuyas consecuencias deberemos analizar.

En principio, es difícil calibrar en qué medida van a incrementarse las necesidades de recursos del compilador para mantener el árbol decorado de cada subprograma en memoria. Sin embargo, a pesar de que el compilador ha de estar preparado para todas las posibles situaciones, lo cierto es que el propio estilo de programación en PASCAL (y en otros lenguajes) recomienda que los subprogramas no sean de gran tamaño. Por ello, es de esperar que en la mayor parte de los casos, los recursos necesarios no sean excesivos.

También es preciso tener en cuenta que los requisitos acerca de la economía en espacio consumido por un compilador se están relajando fuertemente desde hace ya varios años: es lógico que tales requisitos fueran fundamentales cuando los computadores disponían de 256 Kb de memoria, pero en la actualidad, en que cualquier ordenador personal dispone de 8 Mb de memoria primaria, ese criterio ha pasado a muy segundo plano, a nuestro entender. Por la misma razón, se incorporan cada vez más optimaciones, lo cual tiene evidentemente un precio en recursos de compilación, pero con ello se consigue generar código de mayor calidad. En cierto modo, por lo tanto, aceptando que la presencia del árbol abstracto de cada subprograma en memoria supone innegablemente un consumo de recursos en tiempo de compilación, no pensamos que sea un aspecto que impida, hoy en día, la aplicación de la optimación que estamos considerando en este trabajo.

De todas formas, este requisito todavía podría relajarse en cierto modo. Recordemos del apartado 6.3.1 que habitualmente es necesario mantener también todo el código generado para un subprograma en memoria, para resolver las referencias hacia adelante mediante el relleno de retroceso. Esto en principio podría ser negativo, puesto que los recursos necesarios se incrementan por adición a los anteriores. Sin embargo, puede extraerse un resultado muy positivo si se intenta hacer una mezcla de ambas cosas. En efecto, en numerosas ocasiones, es seguro que no va a producirse una posible optimación; por ejemplo, cualquier sentencia repetitiva deshace la característica de posición final. Entonces, cuando se ha construido el árbol que corresponde a una tal sentencia, es posible lanzar las acciones de generación de código, y guardar el trozo de código generado en lugar del árbol abstracto, cuya memoria puede liberarse.

Es posible encontrar una gran gama de situaciones en que esto es posible, de manera que queden en forma de árbol abstracto prácticamente solo las

estructuras que intervienen en la detección de la optimación, es decir, sentencias de activación de procedimientos y de asignación de valor de retorno de funciones, sentencias condicionales y la última sentencia de las sentencias compuestas. Así, el consumo de recursos de memoria del compilador no sería más que un poco mayor del habitual, con lo que el impacto de la Condición 6.3.1 resultaría mínimo.

6.4 INFLUENCIA DE LA ELECCIÓN DEL MODELO DINÁMICO DEL LENGUAJE EN LA OPTIMACIÓN

Esta sección pretende mostrar la facilidad o dificultad con que la optimación de las activaciones finales de procedimientos y funciones puede aplicarse en modelos dinámicos alternativos al elegido en el apartado 2.5.1. De esta manera, podremos saber la dependencia de los resultados obtenidos de elecciones particulares que ha de realizar el diseñador de un compilador o de un lenguaje de programación. Esta cuestión es importante, a nuestro entender, por lo siguiente. Podría aducirse que haber encontrado una organización del programa objeto que conduce al comportamiento en espacio constante para las activaciones finales, y con una posible mejora temporal del proceso de activación sería suficiente para garantizar la posibilidad de la optimación. Aunque esto es cierto en parte, también hay que tener presente que la elección de esta organización depende de varios factores, desde las características de la máquina objeto, pasando por los criterios habituales de generación de código sencillo y rápido, hasta la propia facilidad con que el compilador realiza dicha organización.

Por lo tanto, la postura más correcta en este asunto sería considerar las necesidades de la optimación como un posible criterio más a tener en cuenta a la hora de decidir una organización del programa objeto, el cual debería valorarse conjuntamente con los demás criterios para llegar a una decisión adecuada. Desde luego, podría darse el caso de que la inclusión de la optimación forzará un modelo dinámico muy desfavorable en comparación con otras alternativas, lo que llevaría a un impedimento *de facto* de la optimación.

Nuestro objetivo en la presente sección es hacer una introducción a este problema. Para ello, examinaremos diferentes variaciones más o menos drásticas del modelo dinámico del lenguaje, y estudiaremos la posibilidad de la aplicación de la optimación en las mismas. Un desarrollo más exhaustivo exigiría, por ejemplo, la consideración concreta de las máquinas objeto para las que se construyen compiladores de PASCAL, amén de otras cuestiones particulares, que pensamos que al fin y al cabo, han de ser valoradas por los constructores de los compiladores. Los resultados que obtengamos, por ello, se tomarán como indicativos de la mayor parte de situaciones, aunque nunca como absolutos o definitivamente asentados fuera de toda duda.

6.4.1 Variaciones simples

Las primeras variaciones que consideraremos son muy simples. Por ejemplo, la manera en que se indica el comienzo de cada registro de activación, a través del registro que llamábamos *BasePila*, puede hacerse señalando estrictamente al comienzo, y no al punto de separación entre la zona de control y la zona de datos, tal como se recoge en la figura siguiente (cf. Figura 2.5.1).

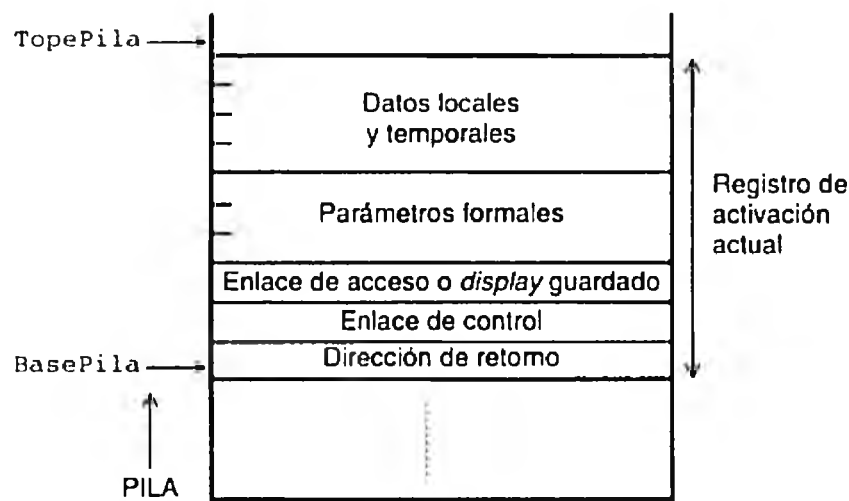


Figura 6.4.1 Variación del modelo dinámico en que se señala estrictamente el comienzo de cada registro de activación

Esta elección puede estar motivada porque las operaciones de actualización de *BasePila* y *TopePila* en las secuencias de llamada y retorno resultan un

poco más sencillas. Concretamente, el paso L5 del Algoritmo 2.5.1 de llamada estándar con la cadena estática (instrucción 5 del Programa 6.2.1) pasa de ser:

$$\text{BasePila} := \text{TopePila} + \#(\text{tamaño-zona-control-proc})$$

a ser simplemente:

$$\text{BasePila} := \text{TopePila}$$

y el paso R2 del Algoritmo 2.5.2 de retorno (instrucción 7 del mismo programa) pasa de ser:

$$\text{TopePila} := \text{BasePila} - \#(\text{tamaño-zona-control-proc})$$

a ser simplemente:

$$\text{TopePila} := \text{BasePila}$$

Con ello, evidentemente, las secuencias de llamada y de retorno resultan un poco menos "caras" en el modelo estándar, lo que siempre es uno de los criterios más importantes a tener en cuenta.

Nuestra elección, sin embargo, vino motivada por la misma razón que aducen [Aho, Sethi y Ullman 1990: 420], aunque usan una organización un poco distinta: si BasePila se hace apuntar al punto intermedio entre la zona de control y de datos del registro de activación, se pueden tener diferentes organizaciones de la zona de control, y aun así los elementos de esa zona de control se van a encontrar en los mismos desplazamientos relativos a partir de BasePila. Por ejemplo, convinimos (apartado 5.1.2) en que el registro de activación de las funciones ha de tener un espacio para el valor de retorno; si ese espacio no se reserva en los registros de activación de los procedimientos, y BasePila apunta al final de la zona de control, los desplazamientos para encontrar el enlace de control, el enlace de acceso y la dirección de retorno son los mismos independientemente de que se esté ejecutando un procedimiento o una función; si BasePila apunta estrictamente al comienzo del registro de activación, por contra, deberían darse diferentes desplazamientos relativos en cada caso para encontrar el mismo elemento.

En cualquier caso, la optimación no resulta afectada en ningún modo por esta variación: el registro BasePila no ha de modificarse en la secuencia de llamada optimada, sea esta de la clase que sea, y el resto de condiciones se mantienen idénticas; además, justamente por ello, la instrucción L5 de la secuencia de llamada estándar es la que resulta eliminada en las activaciones

optimadas, por lo que las distintas secuencias de llamada optimadas son exactamente las mismas que las ofrecidas en capítulos anteriores.

Otra variación realmente simple tiene que ver también con esa instrucción L5 de actualización de `BasePila` en la secuencia de llamada estándar (Algoritmo 2.5.1 y Algoritmo 5.1.1). Podría ser que la realizase el receptor, en lugar del llamador. Esto es interesante porque debe tratarse de que el llamador realice el menor número de operaciones posibles, aunque este es un criterio meramente espacial (de tamaño del código objeto), y no tiene ninguna influencia en el comportamiento temporal del programa. A primera vista, esta variación parece tener un efecto negativo en la optimación, a causa de la Condición 3.2.1 y similares: las operaciones que ejecuta el receptor (en la llamada o el retorno) no pueden modificarse, y justamente la primera de ellas (la operación L5) es una de las que debe eliminarse. Sin embargo, este problema ya lo tuvimos con la actualización de `TopePila` en las activaciones recursivas, y se resolvió (*vide* Condición 3.2.4) simplemente efectuando el salto, no al punto de entrada normal del subprograma, sino a la instrucción L8 directamente. Ahora podemos hacer lo mismo:

- en las activaciones recursivas optimadas, el salto se efectuaría a la instrucción L8, como siempre, con lo que `BasePila` no se modificaría;
- en las activaciones no recursivas optimadas, el salto se efectuaría a la instrucción L7, antes punto de entrada del subprograma, con lo que `BasePila` no se modificaría, pero sí `TopePila`.

En cualquiera de los casos, la optimación puede aplicarse, y además con las mismas mejoras temporales obtenidas en la sección 6.2. La situación peor se daría si el compilador no pudiera generar los saltos a esos puntos de entrada alternativos. Entonces, no quedaría más remedio que preparar los valores de `BasePila` y `TopePila` para que la ejecución de la parte de secuencia de llamada realizada por el receptor siguiera cumpliendo la Condición 3.2.2 y similares, acerca de la invariancia de `BasePila`. Pero eso se conseguiría simplemente con la instrucción:

`TopePila := BasePila - #(tamaño-zona-control-proc)`

previa al salto. El comportamiento en espacio constante de las activaciones optimadas seguiría garantizado, y la única diferencia estribaría en que la mejora temporal no sería tan fuerte como la obtenida con el modelo elegido.

6.4.2 Parámetros antes de la zona de control

Estudiamos ahora un modelo dinámico que supone, al contrario que los anteriores, una variación bastante drástica respecto al usado a lo largo del trabajo. Es el utilizado por Aho, Sethi y Ullman [1990: 410-411, 420-422] (también se puede encontrar en Pittman y Peters [1992: 176-182]), cuya principal característica distintiva es la colocación de los parámetros formales en la base del registro de activación, antes de la zona de control.

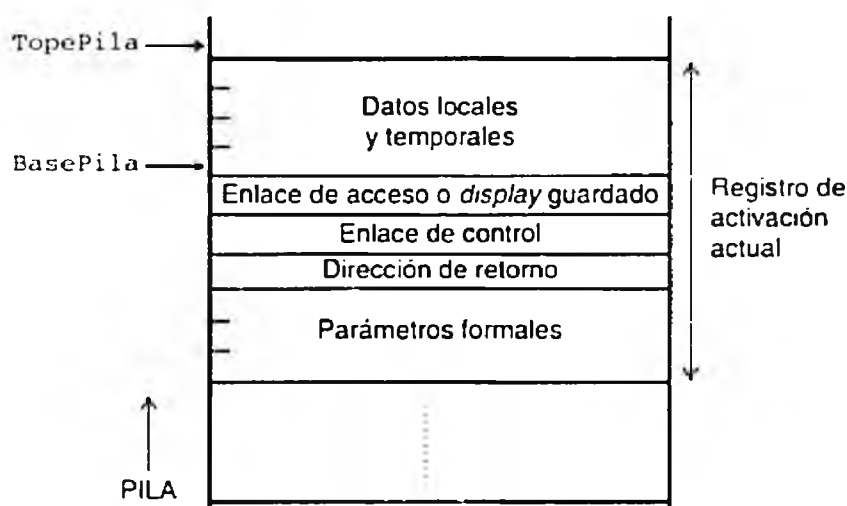


Figura 6.4.2 Estructura del registro de activación en que la zona de parámetros se sitúa antes de la zona de control (reelaborado a partir de Aho, Sethi y Ullman [1990: 411, figura 7.8])

En principio, la razón principal de esta organización parece ser la de permitir procedimientos con un número variable de parámetros. Así, los autores mencionados [Aho, Sethi y Ullman 1990: 421] señalan el ejemplo de la función `printf` del lenguaje C. Ocurre que en PASCAL no es posible definir subprogramas con un número variable de parámetros, y los procedimientos predefinidos que tienen esa característica (por ejemplo, `Read` y `Write`) no se van a tratar siguiendo la estrategia habitual de llamada a procedimientos, por lo que en principio no sería necesaria una estructura como la de la Figura 6.4.2, aunque puede haber alguna otra razón de bastante importancia, como veremos más adelante. De hecho, muchas de las demás estructuras alternativas que se han encontrado, ya sean orientadas específicamente al lenguaje PASCAL.

([Ammann 1981: 75; Rees *et al.* 1981: 272; Wirth 1981: 204-205; Rees y Robson 1988: 111]) o a otros lenguajes ([Fischer y LeBlanc 1988: 291-292; Hanson 1990]) coinciden en colocar la zona de control en la base del registro de activación, tal como nosotros hemos elegido (Figura 2.5.1). A pesar de ser, por lo tanto, una organización poco habitual para PASCAL, estudiaremos el efecto de la misma en la optimación.

Empezamos considerando las activaciones finales recursivas, ya sean de procedimiento o de función. La única dificultad que se encuentra en este caso es la misma que realmente aparecía en nuestro modelo dinámico, a saber, la actualización de los parámetros formales. En efecto, los elementos de la zona de control no se modifican, por lo que eliminando las operaciones relativas a ellos de la secuencia de llamada se cumplen las condiciones requeridas. En cuanto a la actualización de los parámetros, la Condición 3.2.9 obliga a la misma clase de cuidados que se conocen, para que un parámetro formal no sea actualizado antes de ser utilizado en otro parámetro actual. Por lo tanto, las mismas estrategias que resuelven el problema en la organización elegida sirven para este caso. En definitiva, no se encuentran diferencias para las activaciones recursivas, y las mejoras temporales de las llamadas optimadas serán exactamente las mismas que se obtuvieron en la sección 6.2.

Donde sí surgen problemas que pueden ser graves es en las activaciones no recursivas. Al reutilizar el registro de activación del llamador para el receptor, puede ocurrir que ciertos elementos no se encuentren en las mismas posiciones, por lo que se precisa un movimiento de datos, a pesar de que su valor no se modifique. Concretamente, supongamos que el procedimiento receptor tiene un mayor tamaño de la zona de parámetros que el llamador. Los registros de activación que deberían estar en la pila de control en el modelo dinámico optimado durante la activación del procedimiento llamador y la activación del procedimiento receptor, junto con sus zonas de parámetros, control y datos, quedan reflejadas en la Figura 6.4.3. Antes de nada, obsérvese que el registro *BasePila* cambia de valor, contradiciendo la Condición 4.2.2, que ya no es válida: en su lugar, debe producirse una actualización de dicho registro. Sin embargo, debemos considerar si es posible realizar una variación como la primera mostrada en el apartado anterior: que *BasePila* apunte al comienzo del registro de activación.

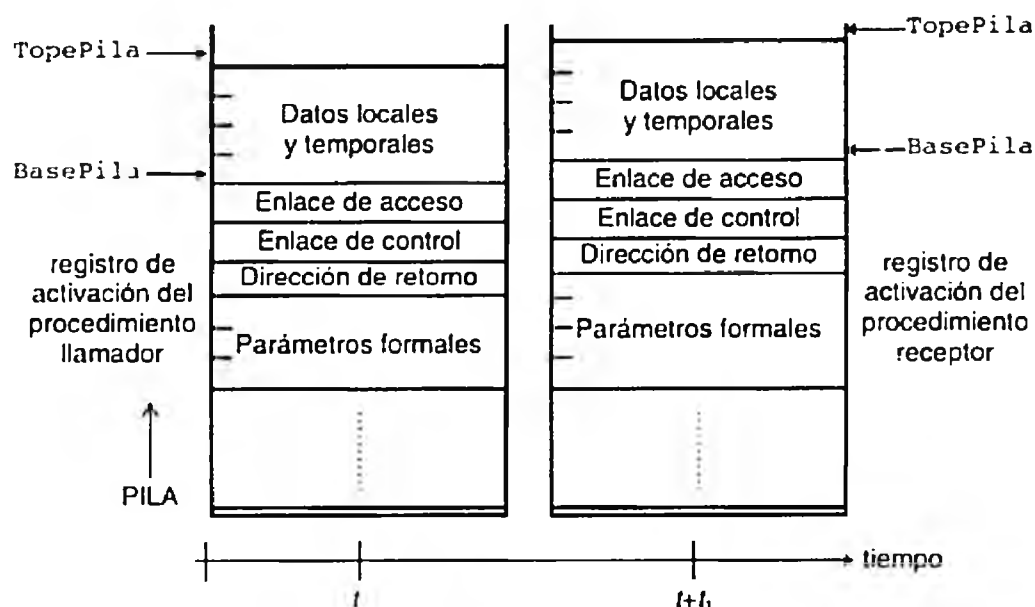


Figura 6.4.3 Instantáneas de la situación requerida en la pila de control en el modelo dinámico optimado durante una activación final no recursiva de procedimiento

Realmente, esto no sería posible si efectivamente el lenguaje admitiese subprogramas con un número variable de parámetros; como esto no ocurre en PASCAL, podría variarse ligeramente la organización de Aho, Sethi y Ullman para hacer que BasePila señale el comienzo del registro de activación, y con ello, la Condición 4.2.2 seguiría siendo válida, y este problema no existiría. Sin considerar tal variación, aún es posible realizar la actualización, pero a costa de más instrucciones.

Antes de verlo, echemos una mirada al otro problema: los elementos de la zona de control del registro de activación del llamador no cambian de valor, en consonancia con la Condición 4.2.5, la Condición 4.2.6 y la Condición 4.2.7 (salvo el enlace de acceso, si se trata de una llamada a un procedimiento de distinto anidamiento), pero cambian de lugar, y lo que es peor, los parámetros formales del receptor pueden ocupar parte de las posiciones reservadas a esos valores en el registro de activación previo. Este hecho prácticamente exige salvaguardar los valores de la zona de control del registro de activación del llamador en localidades seguras antes de proceder a la evaluación de los parámetros actuales, para ser recuperados después de ello a sus localidades definitivas, tras lo cual es cuando puede procederse a la actualización de

BasePila. Más en concreto, podemos dar el siguiente esquema para la secuencia de llamada optimada:

- LO1: el llamador evalúa los parámetros actuales, colocando los valores obtenidos en localidades seguras; estas localidades seguras pueden estar en la zona de datos temporales de su registro de activación, con la condición adicional de que su desplazamiento respecto de la base del registro de activación sea mayor que el tamaño de los parámetros y la zona de control del receptor;
- LO2: el llamador coloca también sus propios valores del enlace de control y la dirección de retorno en localidades seguras; si se usa el enlace de acceso, calcula el nuevo valor (si el receptor es de menor anidamiento) o usa su propio enlace de acceso, colocando el valor que sea en otra localidad segura;
- LO3: el llamador disminuye TopePila hasta señalar a la base de su propio registro de activación; utilizando ese valor, el llamador coloca los valores de los parámetros actuales, enlace de control, enlace de acceso y dirección de retorno guardados antes en las localidades seguras (para ello, aún utiliza el valor de BasePila para acceder a los localidades seguras);
- LO4: el llamador actualiza BasePila como el valor de TopePila más el tamaño de los parámetros actuales y de la zona de control;
- LO5: el llamador salta (jump) a la primera instrucción del código del receptor;
- LO6: el receptor actualiza TopePila como el valor de BasePila más el tamaño de los datos locales;
- LO7: el receptor asigna valores iniciales a sus datos locales.

Algoritmo 6.4.1 Secuencia de llamada optimada para activaciones finales no recursivas cuando los parámetros formales se encuentran en la base del registro de activación

Desde luego, ha sido posible llegar a una secuencia de llamada optimada que cumpla los requerimientos de la optimación, pero el adjetivo "optimada" habremos de entenderlo en este caso en su aspecto espacial, porque el comportamiento temporal resultará indudablemente muy degradado respecto a

la secuencia de llamada estándar. Las operaciones de movimiento de los elementos de la zona de control ya son de por sí bastante costosas, lo que unido a la necesidad de actualizar el registro BasePila hace que esta solución no sea tan atractiva como en la organización elegida a lo largo del trabajo.

Sin embargo, no todo son malas noticias. Hay un caso particular importante, como es el caso en que los parámetros del llamador ocupan lo mismo que los del receptor, algo que el compilador puede detectar muy fácilmente. En esa situación, la Condición 4.2.2 sigue cumpliéndose, y lo que es más importante, no se precisa salvaguardar la información de la zona de control. Con ello, la secuencia de llamada optimada sería prácticamente la misma que en el modelo dinámico original: solo sería necesaria la actualización de parámetros, con el cuidado habitual, el cálculo del enlace de acceso (si es preciso) y el salto al receptor. La mejora temporal en este caso particular vendría dada exactamente por los mismos valores que en la sección 6.2 para este tipo de activaciones finales.

Este caso particular, lejos de representar una situación esporádica, engloba a nuestro entender un gran número de situaciones interesantes de cara a la optimación. En efecto, cuando los procesos recursivos se expresan mediante procedimientos o funciones mutuamente recursivos, o con cualquier grado de recursividad indirecta, es lógico pensar que los diferentes subprogramas tengan exactamente los mismos parámetros. Entonces, además de conseguir el comportamiento en espacio constante, objetivo final de la optimación, también resulta una mejora temporal interesante, teniendo en cuenta que es de la misma forma muy probable que los subprogramas de definan en el mismo nivel de anidamiento, con lo que la razón de optimación temporal viene dada por las cifras conseguidas para esta clase de activaciones. Por lo tanto, la otra clase de procesos recursivos más interesantes de cara a la optimación, además de los expresados mediante recursividad directa, quedan tan bien tratados como en el modelo dinámico original, a pesar de que otras activaciones no recursivas resulten en un comportamiento temporal pobre.

Otro caso particular es aquel en que los parámetros del llamador son de mayor tamaño que los del receptor. Aquí tampoco se cumple la Condición 4.2.2, pero como es seguro que la actualización de los parámetros no afectará a la zona de control, no es necesario el paso LO2 del Algoritmo 6.4.1 que salvaguarda estos valores; sin embargo, si es necesaria su traslación a las nuevas localidades

(operación LO3) y la actualización de BasePila (paso LO4) por lo que el comportamiento temporal seguirá siendo probablemente peor que en el modelo dinámico sin optimización.

En definitiva, concluimos que la estructuración del registro de activación consistente en colocar los parámetros formales en la base del registro de activación, antes de la zona de control, no impide en ningún modo la optimización, aunque la empeora desde el punto de vista temporal en aquellos casos en que no se cumpla la Condición 4.2.2, no por el propio hecho de la necesidad de actualizar el registro BasePila, sino por la obligación de salvaguardar y/o trasladar la información de la zona de control.

6.4.3 Uso de instrucciones especializadas

La última clase de variaciones del modelo dinámico responde a la duda que surge sobre la posibilidad de la optimización cuando se utilizan en las secuencias de llamada y retorno instrucciones específicas de la máquina objeto para el manejo de la pila de control, como las propias instrucciones que activan y retornan de una subrutina (de máquina objeto, para diferenciarlas de los subprogramas del lenguaje de alto nivel) o las que apilan y desapilan objetos de la pila de la máquina. Es de suponer que el uso de estas instrucciones específicas conduce a una mejor calidad de la ejecución, por lo que se preferirían antes que las instrucciones de manejo explícito de la pila de control que aparecen en el modelo dinámico utilizado a los largo del trabajo para el lenguaje PASCAL. Sin embargo, en este asunto hay una serie de cuestiones interrelacionadas que hacen bastante difícil su análisis.

En primer lugar, es habitual que los procesadores actuales dispongan de instrucciones de manejo de una estructura de datos en forma de pila, a través de un registro específico que señala la cima de la pila, que llamaremos genéricamente SP; esas instrucciones son básicamente las de acumulación (push) y desapilado (pop) de operandos. Adicionalmente, se proporcionan instrucciones de manejo de subrutinas, precisamente mediante la zona de la memoria destinada a la pila, que es referida por el registro SP; así, una instrucción call consiste típicamente en acumular en la pila la dirección de retorno y saltar a otra dirección del código, mientras que una instrucción ret

desapila una dirección y salta a la misma. El problema que surge es saber hasta qué punto esta pila interna del microprocesador, que denominaremos la *pila máquina*, puede servir para alojar la *pila de control* necesaria en un programa objeto de cualquier lenguaje de programación que admita subprogramas y activaciones en forma de árbol de los mismos. Ningún texto de compiladores por nosotros conocido muestra de modo explícito cuál puede ser esta relación, y evidentemente pueden encontrarse compiladores que tomen una u otra solución.

En este problema intervienen diferentes factores, desde la propia máquina, pasando por el sistema operativo subyacente, hasta las necesidades del propio lenguaje, que pueden entremezclarse entre sí. Por ejemplo, la pila máquina puede ser también utilizada para el procesamiento de interrupciones o excepciones, que pueden producirse en cualquier momento, y que esperan encontrar la pila de una determinada manera a través del registro SP. El sistema operativo, por su parte, asigna la memoria a los programas y puede asociar la pila máquina con la pila de control, pero también puede no hacerlo, por ejemplo, en un sistema multitarea o multiusuario. Finalmente, el procesador fija el crecimiento y decrecimiento de la pila máquina de una determinada forma respecto a la memoria, y ese sentido puede ser o no ser el adecuado para las necesidades de la pila de control en un determinado lenguaje de programación. Estos, y quizás otros factores que olvidamos, determinarán la posibilidad de usar la pila máquina como pila de control del programa objeto.

Supongamos que en un determinado sistema, los programas objeto utilizan la pila máquina como pila de control, es decir, el registro SP es el que se utiliza para lo que se ha denominado en capítulos anteriores *TopePila*. Tomemos entonces el caso de las interrupciones. Generalmente, los procesos que se ejecutan en el manejo de las interrupciones utilizan la pila máquina (por lo menos para acumular la dirección a la cual retornar), por lo que, si el programa objeto utiliza esa pila máquina como pila de control, es obligado que en todo momento el registro SP señale al menos más allá de cualquier información útil que el programa objeto haya colocado en la pila de control, y que sea necesaria para continuar la ejecución tras el retorno desde la interrupción. En ese caso, el modelo dinámico elegido en los capítulos anteriores no serviría desde el momento que, durante el proceso de llamada a un subprograma, se acumula información en la pila de control antes de la

actualización del registro *TopoPila*: esa información podría perderse si antes de esa actualización ocurre una interrupción.

Diferentes procesadores adoptan diferentes posturas a este respecto. Por ejemplo, en el MC68020 pueden diferenciarse claramente la pila máquina de usuario (*user stack*) de la pila máquina de servicio para interrupciones. Concretamente, el MC68020 [Motorola 1985: 6/1-6/8] define dos estados de procesamiento, de diferente privilegio, y lo que es más importante, que pueden tener diferentes estructuras de datos para el procesamiento. Los programas se ejecutan normalmente en el *estado de usuario*, que es el de más baja prioridad; en él, se tiene un registro para señalar la cima de la pila, llamado USP. Cuando ocurre una excepción, y solo de esa manera, el procesador cambia al *estado de supervisor*, en el que pueden ejecutarse algunas instrucciones adicionales; más importante, el estado de supervisor tiene su propio registro de pila, diferente de USP, que se llama SSP. Por lo tanto, es posible asignar diferentes partes de la memoria a la pila de usuario y a la pila de supervisor, cosa que debería hacer, por ejemplo, el sistema operativo, y entonces:

«During the processing of traps and interrupts, both the program counter and the status register are saved on the supervisor stack [...]. Thus, the execution of supervisor state code is not dependent on the behavior of user code or condition of the user stack, and user programs may use the user stack pointer independent of supervisor stack requirements» [Motorola 1985: 2/21]

Obvia decir que esta organización de la máquina posibilita mucho más la asignación de la pila de control a la pila máquina, permitiendo incluso las licencias acerca del manejo de la pila de control implícitas en el modelo dinámico elegido.

La implementación del modelo dinámico utilizado hasta ahora en una máquina que carezca de estas características, como el 80386 [Intel 1989], queriendo mantener la pila de control en la pila máquina, a pesar de ser posible, requiere una serie de modificaciones para asegurar el hecho de que SP (*TopoPila*) nunca señale por debajo de información colocada en la pila en los procesos de llamada y retorno de subprogramas. Una solución sencilla consistiría en comenzar la secuencia de llamada estándar utilizando un registro temporalmente para señalar el punto al que apuntará *BasePila* en la activación entrante, y entonces incrementar *TopoPila* para reservar espacio para la zona de control y los parámetros; la evaluación de los parámetros

actuales y la asignación de los elementos de control pueden hacerse con desplazamientos respecto al registro temporal, y el resto de la secuencia de llamada es prácticamente similar a lo conocido, incluyendo la colocación definitiva de `TopePila` más allá de la zona de datos locales. En la secuencia de retorno, se puede decrementar `TopePila` solo hasta donde están el enlace de control y la dirección de retorno, y por ello, es preciso que el llamador establezca el valor correcto de `TopePila` después del retorno mismo, algo que no era necesario en la implementación de este proceso que se ha estudiado hasta ahora.

El coste de esta variación puede ser alto debido a que se utiliza un registro más, lo que puede ser problemático (recordemos que se precisan dos registros para el registro de activación, otro para los datos globales y posiblemente otro para el cálculo del enlace de acceso), y a que se añaden instrucciones adicionales en los procesos de llamada y retorno. Sin embargo, en principio con la optimación no hay problema, e incluso se obtendrían mayores razones de optimación que las de la sección 6.2, ya que justo las nuevas instrucciones son algunas de las que pueden eliminarse.

Ahora examinamos las consecuencias de utilizar instrucciones específicas de manejo de pila, llamadas genéricamente `push` y `pop`, para la implementación de los procesos de llamada y retorno, manteniendo el requisito de que `TopePila` ha de señalar en todo momento la cumbre de la pila máquina. Obsérvese que el uso de tales instrucciones conlleva una modificación implícita de este registro. Desde luego, esto prácticamente obliga a una organización muy concreta del registro de activación, pues ello vendrá determinado por el orden de apilamiento de los elementos del mismo. Por ejemplo, si mantenemos nuestra elección de que la evaluación de los parámetros actuales y su asignación a los formales se haga como primera operación de la secuencia de llamada, la zona de parámetros en el registro de activación debe situarse *antes* de la zona de control, cuyos elementos se gestionan temporalmente después en la llamada. Así, esta es otra razón para elegir una organización del registro de activación como la de la Figura 6.4.2; allí decíamos que en principio esa organización era necesaria cuando existía un número variable de parámetros, algo que no ocurre en PASCAL, pero vemos que el estudio mereció la pena, pues se aplica a esta otra variación, que puede ser relativamente común.

Si el uso de `push` y `pop` obliga a colocar la zona de parámetros debajo de la zona de control en el registro de activación, las conclusiones respecto a la optimación son básicamente las mismas que en el apartado anterior, con la salvedad de que ahora no es posible utilizar la instrucción `push` en la secuencia de llamada optimada, pues no se trata de acumular nueva información, sino reutilizar y actualizar la existente. Para verlo con mayor claridad, mostraremos el código generado para la llamada y retorno de un procedimiento, siguiendo el modelo estándar y suponiendo una organización del registro de activación parecida a la de la Figura 6.4.2.

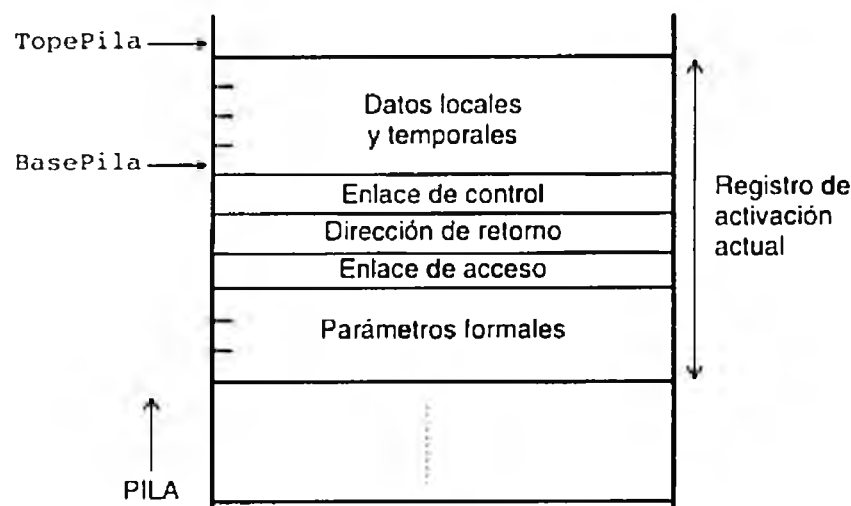


Figura 6.4.4 Estructura del registro de activación en que la zona de parámetros se sitúa antes de la zona de control (variación de la Figura 6.4.2)

Realmente, solo se ha variado la posición de los elementos de la zona de control; ello facilitará las secuencias de llamada y retorno. Veámoslo en el Programa 6.4.1. Obsérvese que la colocación del enlace de control como último elemento de la zona de control del registro de activación permite realizar una operación `pop` para actualizar el registro `BasePila` en la secuencia de retorno. Incluso, si se puede usar la instrucción `pop` para cargar el contador de programa (registro que denominamos `Instrucción`), la operación de salto al procedimiento llamador (paso R3) podría ser simplemente:

POP Instrucción


```

; punto de entrada al procedimiento receptor
etiqueta (punto-entrada-proc-receptor)
; parte de la secuencia de llamada como receptor
; L7: actualización de TopePila
TopePila := BasePila + #(tamaño-zona-datos-proc)
; L8: inicialización de datos locales

; fin de la secuencia de llamada como receptor
; comienzo del código del procedimiento receptor

; la parte final del código del procedimiento es la parte de la secuencia
; de retorno como receptor
; R1: pierde zona datos locales
TopePila := BasePila
; R2: recupera BasePila con enlace de control
POP BasePila
; R3: salto a la dirección de retorno
IR-A [TopePila]
; fin del código del procedimiento receptor

; punto de entrada al procedimiento llamador

; secuencia de llamada correspondiente a una activación final no recursiva
; L1: evaluación de los parámetros actuales
; se usa PUSH para cada parámetro

PUSH (parámetro-actual)

; L2: enlace de acceso
Enlace := [BasePila + #(ACCS-LINK-LOC)]
Enlace := [Enlace + #(ACCS-LINK-LOC)] } (n_act - n_dct - 2 veces)
Enlace := [Enlace + #(ACCS-LINK-LOC)] }
PUSH [Enlace + #(ACCS-LINK-LOC)]
; L3: dirección de retorno
PUSH (etiqueta-retorno)
; L4: enlace de control
PUSH BasePila
; L5: actualiza BasePila
BasePila := TopePila
; L6: salto al receptor
IR-A (punto-entrada-proc-receptor)
; fin de la secuencia de llamada como llamador
; comienza la secuencia de retorno como llamador
etiqueta (etiqueta-retorno)
; R4: reposición de TopePila
TopePila := TopePila - #(tamaño-parámetros+enlace-acceso+dirección-retorno-proc)
; continuación del código

; fin del código del procedimiento llamador

```

Programa 6.4.1 Esquema del código objeto generado para las secuencias de llamada y retorno usando instrucciones especializadas de manejo de la pila máquina, según el modelo dinámico estándar, usando la cadena estática, y con la estructura del registro de activación de la Figura 6.4.4

Es interesante realizar un cálculo de costes de los procesos de llamada y retorno, similar a los de la sección 6.2, con esta variación del modelo dinámico. Se encontraría que la secuencia de llamada estándar es un poco menos costosa (pero solo en una relación de 1'27 con el criterio de coste 2b), mientras que la secuencia de retorno cuesta exactamente lo mismo. Por lo tanto, la mejora introducida por el uso de las instrucciones *push* y *pop* no es demasiado significativa.

En cuanto a la optimización, solo hay que insistir en que ahora no es posible utilizar esas instrucciones para actualizar el registro de activación, pero en el caso de las activaciones finales recursivas, realmente no hay que hacer nada más que actualizar los parámetros y saltar al comienzo del procedimiento, por lo que la razón de optimización en ese caso se mantiene en valores similares a los obtenidos. Donde sí aparecerán problemas, ya lo sabemos, es en las activaciones finales no recursivas en que los procedimientos involucrados tengan diferente tamaño de la zona de parámetros en el registro de activación, debido a la necesidad de salvaguardar la información de control. Las consecuencias son las mismas a que se llegó al final del apartado previo.

El último paso que daremos, dentro del uso de la pila máquina para implementar la pila de control del programa objeto, será la consideración de las instrucciones habituales del microprocesador para el manejo de subrutinas. Típicamente, como hemos dicho al comienzo del apartado, existe una instrucción *call* que acumula en la pila máquina la dirección de retorno, y salta a la instrucción que se le proporciona como operando, y existe una instrucción *ret* que desapila una dirección y salta a la misma. Estas instrucciones son muy adecuadas para realizar los saltos en la llamada y retorno de los procedimientos y las funciones. La variación del Programa 6.4.1 que utiliza estas instrucciones se muestra a continuación en el Programa 6.4.2.

Las diferencias más notables con el programa anterior radican en que es preciso colocar algunas operaciones de la secuencia de llamada estándar en el receptor, concretamente la acumulación del enlace de control y la actualización del registro *BasePila*. Ello es debido a que la instrucción *call* acumula la dirección de retorno, y es conveniente que el enlace de control esté por encima en el registro de activación (justamente por lo que debe hacerse en la secuencia de retorno), así que ese paso debe darlo el receptor. Ello no es problema, más al

contrario, se recomienda que sea el receptor el que haga el mayor número de operaciones posible.

```

; punto de entrada al procedimiento receptor
etiqueta (punto-entrada-proc-receptor)
; parte de la secuencia de llamada como receptor
; L4: enlace de control
PUSH BasePila
; L5: actualiza BasePila
BasePila := TopePila
; L7: actualiza TopePila
TopePila := BasePila + #(tamaño-zona-datos-proc)
; L8: inicialización de datos locales

; fin de la secuencia de llamada como receptor
; comienzo del código del procedimiento receptor

; la parte final del código del procedimiento es la parte de la secuencia
; de retorno como receptor
; R1: pierde zona datos locales
TopePila := BasePila
; R2: recupera BasePila con enlace de control
POP BasePila
; R3: salto a la dirección de retorno
RET
; fin del código del procedimiento receptor

; punto de entrada al procedimiento llamador

; secuencia de llamada correspondiente a una activación final no recursiva
; L1: evaluación de los parámetros actuales
; se usa PUSH para cada parámetro

PUSH (parámetro-actual)

; L2: enlace de acceso
Enlace := [BasePila + #(ACCS-LINK-LOC)]
Enlace := [Enlace + #(ACCS-LINK-LOC)] } (nact-nder-2 veces)
Enlace := [Enlace + #(ACCS-LINK-LOC)]
PUSH [Enlace + #(ACCS-LINK-LOC)]
; L3: salto al receptor
CALL (punto-entrada-proc-receptor)
; fin de la secuencia de llamada como llamador
; comienza la secuencia de retorno como llamador
etiqueta (etiqueta-retorno)
; R4: reposición de TopePila
TopePila := TopePila - #(tamaño-parámetros+enlace-acceso-proc)
; continuación del código

; fin del código del procedimiento llamador

```

Programa 6.4.2 Esquema del código objeto generado para las secuencias de llamada y retorno usando instrucciones especializadas de manejo de la pila máquina y de subrutinas, según el modelo dinámico estándar, usando la cadena estática, y con la estructura del registro de activación de la Figura 6.4.4

Por otro lado, la optimación no resulta afectada en esencia por esta modificación, a pesar de la Condición 3.2.1 y similares; se aplica exactamente el mismo argumento que en el apartado 6.4.1, donde precisamente se consideró la variación de que fuera el receptor el que actualiza el registro BasePila.

En cuanto a la mejora temporal, hay que empezar diciendo que el uso de estas instrucciones sí supone un ahorro interesante en la secuencia de llamada estándar respecto a la versión original que hemos manejado a lo largo del trabajo; concretamente, utilizando las instrucciones JSR y RTS del MC68020, la razón es de 1'57 con el criterio 2b, aunque no tanto (1'23) con el criterio 2a. La secuencia de retorno, por contra, no se ve modificada (salvo el criterio 2a).

	Coste según criterio 1	Coste según criterio 2a	Coste según criterio 2b
<i>c[ce]pr-llam_{est}</i>	8 (16)	13 (16)	35 (55)
<i>c[ce]p-ret</i>	5 (7)	12 (5)	26 (26)

Tabla 6.4.1

Coste de la secuencia de llamada recursiva estándar de procedimiento (usando la cadena estática) cuando se utilizan instrucciones de manejo de la pila máquina y de subrutinas (entre paréntesis aparecen los valores originales, cuando no se usan esas instrucciones, de la Tabla 6.2.2 y la Tabla 6.2.5)

Por lo tanto, la razón de optimación de las activaciones finales recursivas se verá decrementada, pero aún supone una mejora sustancial del proceso: se obtiene un valor cercano a 7 con el criterio 2b, cuando en la Tabla 6.2.6 se obtuvo un valor de 9. De todas formas, es una razón de optimación indudablemente generosa, que no nos hace reconsiderar ninguno de los comentarios realizados a lo largo de las secciones previas. Y de nuevo, lo que ocurre con las activaciones finales no recursivas no es más grave que lo que ya hemos comentado en ocasiones anteriores.

Terminamos este apartado mencionando que parece ser que la estructura del registro de activación de la Figura 6.4.4, y la implementación de las secuencias de llamada y retorno de subprogramas recogida en el Programa 6.4.2, son las que genera, con escasas diferencias, el compilador TURBO PASCAL [Borland 1992a] para el microprocesador Intel 8086 en el sistema operativo MS-DOS. Y decimos "parece ser" porque la obtención de esta conclusión ha sido posible gracias principalmente a métodos conocidos como "ingeniería

inversa": aunque la guía del lenguaje especifica la acumulación de parámetros en la pila de control y el uso de la instrucción `call` para la llamada [ibíd.: 253], y las partes de las secuencias de llamada y retorno de los procedimientos y funciones como receptores [ibíd.: 261-262], no se muestra en ningún momento la estructura del registro de activación, ni se indica explícitamente si se usa la cadena estática para el acceso a los datos no locales.

En efecto, hemos podido concluir que se utiliza este método, pero una explicación referente a la necesidad de acumular un cierto enlace (que no es denominado en ningún momento "enlace de acceso") para los llamados «procedimientos y funciones anidados» [ibíd.: 257], que no son otros que procedimientos y funciones no globales, es claramente errónea, al pretender que simplemente se acumula el registro `BasePila`, y que siguiéndolo cierto número de veces se accede a los datos no locales. En descargo de ello, hemos comprobado que el código generado efectúa el cálculo del enlace de acceso del modo correcto. Pero en general, es una muestra de la dificultad para encontrar información detallada de las características internas de los compiladores comerciales.

6.4.4 Paso de parámetros

El problema del paso de los parámetros actuales del subprograma llamador al subprograma receptor se ha convertido, ya desde el mismo inicio del estudio, en uno de los aspectos más cruciales y determinantes de la posibilidad de la optimación de las activaciones finales, posiblemente el más importante después de las propias restricciones sintácticas que se han encontrado para el lenguaje PASCAL. En este último apartado de la sección trataremos de ofrecer un estudio más profundo del problema.

Recapitulando, obtuvimos que el principio de reutilización de parte del registro de activación del llamador en una activación final conducía en primer lugar a la Condición 3.2.9 y la Condición 4.2.9 (y similares para las funciones): la obligación de que no se modifique (o se pierda), en el modelo dinámico optimado, un parámetro formal del llamador antes de haber sido utilizado en cualquier parámetro actual con los que se activa al receptor. Este es un

resultado similar al que parece haber obtenido Hanson [1990] en el trabajo analizado en el apartado 2.4.2, pero lo resuelve con la siguiente escueta frase:

«Compute the access link and bindings of the new environment record, saving them somewhere other than the stack» [Hanson 1990: 110]

Puede entenderse la razón de que esto no sea un problema en SCHEME: este lenguaje hace un uso más intensivo de la memoria dinámica, en el montículo, y no es problema utilizar algunas localidades temporalmente para los valores de los parámetros actuales. Esto no ralentiza la ejecución, *a priori*, porque tal memoria no se libera explícitamente, sino que esto lo hará el recolector de basura.

Sin embargo, en PASCAL esta estrategia no es posible, porque las localidades temporales usadas en el montículo deberían liberarse una vez llevado su contenido al nuevo registro de activación, lo que es un proceso más costoso, por ejemplo, que hacer lo mismo sobre la propia pila, habiendo determinado durante la compilación el espacio necesario e incluyéndolo en el registro de activación del subprograma que invoca. Una vez más, se afianza nuestra idea de que la aplicación de las ideas de SCHEME a PASCAL, en lo que respecta a la recursividad final, no es trivial, como parecería a primera vista.

Podemos suponer, antes de estudiar estrategias concretas de paso de parámetros en diferentes compiladores que no aplican la optimación, que el criterio común de este proceso trata cada parámetro actual de forma independiente: se toma cada parámetro en un determinado orden (que no está fijado por el lenguaje, por otro lado), y se realiza su evaluación y su asignación a la localidad del parámetro formal correspondiente antes de pasar al siguiente parámetro en el orden elegido. Esta estrategia es la que necesita claramente menos recursos tanto espaciales como temporales.

Entonces, el comportamiento optimado que resulta de la Condición 3.2.9 y la Condición 4.2.9 se aleja de esta estrategia, pues la forma más sencilla de cumplir esos requisitos precisa en la práctica la evaluación completa de todos los parámetros actuales antes de ser asignados los valores a las localidades definitivas; según ello, aparecen dos necesidades no presentes en el comportamiento estándar: espacialmente, localidades volátiles en donde recoger los valores actuales durante la evaluación, y temporalmente, procesos de copia de las localidades temporales a las definitivas. El impacto de estas

necesidades depende obviamente de sus atributos cuantitativos, lo que al fin y al cabo depende de cada subprograma concreto, tal como ya se ha comentado en otras ocasiones: un subprograma con pocos y sencillos parámetros resultará en un buen comportamiento, mientras que si tiene muchos o son complejos el resultado puede ser altamente insatisfactorio.

En principio, las necesidades de espacio temporal adicional no suponen ningún problema, pues este espacio tiene un tiempo de vida extremadamente restringido: si el compilador genera un manejo eficiente de las localidades temporales, no manteniéndolas más que lo estrictamente necesario, no deben observarse comportamientos diferentes durante la optimación. El único problema que podría existir sería un posible desbordamiento de la pila de control, pero ello querría decir que ya el programa original se movía en los límites del espacio dedicado a la pila, por lo que en sí ya se trataba de un programa "problemático". Además, el espacio que se ahorra al reutilizar el registro de activación del llamador en las activaciones optimadas probablemente compense, con mucho, la necesidad del espacio temporal en la gran mayoría de las situaciones, por lo que en general, es casi seguro que los programas optimados consumirán mucho menos espacio de pila de control que los no optimados, sin necesidad de recurrir al caso límite de procesos recursivos de gran profundidad que solo podrían llevarse a cabo si se ha realizado la optimación.

El comportamiento temporal, por otro lado, es el más afectado, o el que más impedimentos puede poner a la optimación. Recordemos, primero, que la mejora en el consumo de memoria es el objetivo primero de la optimación, y que el consumo temporal es un segundo criterio en el que pueden influir diversos factores. Es decir, puede que un constructor de un compilador no esté dispuesto a soportar absolutamente ningún empeoramiento temporal de los programas ejecutables, mientras que otro constructor esté dispuesto a sacrificar una pequeña parte de sus resultados para, al fin y al cabo, poder ejecutar más programas. En esta decisión pueden influir las propias características del compilador original, los entornos en que se utiliza, y otros factores de difícil análisis.

El análisis de la sección 6.2 nos permitió concluir que puede existir una mejora apreciable de los procesos de activación en lo referente al manejo de la información de control. Después de examinar diferentes alternativas en los

apartados anteriores, podemos garantizar que esa mejora será significativa en las activaciones finales recursivas en casi cualquier modelo dinámico que se elija, mientras que pueden existir variaciones importantes en las activaciones finales no recursivas dependiendo de la organización del registro de activación y de la secuencia de llamada elegida. Por lo tanto, está claro que existirá todo un rango de comportamientos temporales de los programas optimados respecto a los no optimados, aún con la necesidad de utilizar temporales para la evaluación de los parámetros: desde programas en que se observará una mejora significativa del tiempo de ejecución, pasando por programas en que la mejora de los procesos propios de la llamada se compensará con la copia de los temporales en las localidades definitivas, hasta programas que resultan significativamente empeorados.

Las causas del posible empeoramiento son dos: primero, un gran número de parámetros formales en el subprograma receptor, y segundo, un gran tamaño en conjunto de dichos parámetros. Ante la conjunción de las dos causas, la necesidad de realizar copias seguras de los parámetros actuales se convierte en una tarea costosa temporalmente, en una cantidad proporcional a la suma de los tamaños máquina de los parámetros. Por lo tanto, como ya hemos avanzado al final del apartado 6.2.1, las situaciones más graves se producen en relación con parámetros por valor, sobre todo de tipo estructurado, es decir, *arrays*, *records* o conjuntos. Desde luego, este es un contratiempo (nunca mejor dicho), pero puede que aún resulte interesante la optimación para el programador por el comportamiento espacial que se consigue. Hay que tener en cuenta que en tales situaciones el programa no optimado ya puede adolecer de por sí de un comportamiento temporal bastante pobre, porque se sigue precisando un proceso de copia de objetos de gran tamaño; o sea, que si las especificaciones de un programa exigen una buena calidad temporal, es muy probable que los primeros esfuerzos se hayan dedicado a evitar justamente el factor que provoca los peores comportamientos en la optimación.

Recordemos que es posible aplicar, solo en el caso de las activaciones recursivas, una optimación adicional cuando el parámetro actual consta únicamente del propio parámetro formal correspondiente, según viene dado por la Condición 3.2.10, la Condición 3.2.11 y la Condición 3.2.12; de nuevo, por lo tanto, las activaciones finales recursivas son las que mejor paradas resultan en la optimación, en comparación con las activaciones no recursivas.

En lo que resta de este apartado, examinaremos algunas alternativas del proceso de paso de parámetros, pues hasta ahora hemos considerado que ese proceso se llevaba a cabo al completo antes de establecer la información de control. Concretamente, Ammann, explicando la implementación del segundo compilador de PASCAL para el CDC 6000, señala lo siguiente:

«The parameter descriptors of the first four parameters are always passed in X registers (and not in memory)» [Ammann 1981: 78]

El subprograma receptor copia, como primera actividad, los «parameter descriptors» en las localidades definitivas de su registro de activación [Ammann 1977], aunque no se explica el funcionamiento si el parámetro actual no puede ser almacenado en un registro. En cualquier caso, TURBO PASCAL [Borland 1992a: 253-255] realiza algo similar, pues aunque todos los parámetros actuales se acumulan en la pila en el momento de la llamada, para los parámetros por valor de tipo estructurado de tamaño superior a 4 *bytes* no se copia el objeto completo, sino un apuntador al mismo, de modo que el subprograma receptor ha de realizar la copia correspondiente. Esto son buenas noticias pero también malas.

En efecto, con el criterio de Ammann, suponiendo que hay como mucho cuatro parámetros y ninguno es estructurado si es por valor, el paso de parámetros en el modelo dinámico optimado sería idéntico al que se realiza normalmente, con lo cual no habría absolutamente ninguna penalización temporal por este proceso durante la optimación. Esta técnica puede ser relativamente habitual, y si no la hemos mencionado antes ha sido para mantener un criterio de «peor caso» a la hora de evaluar el coste del proceso de paso de parámetros. Además, los supuestos anteriores pueden ser los que engloben a la mayor parte de los subprogramas en PASCAL, por lo que la optimación producirá una indudable mejora del comportamiento temporal en cualquier clase de activaciones finales con tales características, y más si se utiliza la estructura del registro de activación en que la zona de control se coloca en la base, que es la utilizada por Ammann. Por supuesto, quedaría por saber lo que ocurre con los parámetros por valor de tipo estructurado, lo que estudiaremos bajo la otra alternativa.

La variante mencionada presente en TURBO PASCAL sí puede representar un impedimento absoluto de la optimación. Claramente, si el objeto de tipo estructurado cuya copia se difiere es local en el subprograma llamador,

entonces sería imposible aplicar la optimación, y ello a causa de que el subprograma receptor intentaría, como parte final de su secuencia de llamada, realizar la copia de un objeto que ya no existe como tal. Como no es posible modificar la parte de la secuencia de llamada ejecutada por el receptor (Condición 3.2.1 y similares), este problema no puede evitarse. En tal caso, debería añadirse una nueva restricción a la optimación, que diría que si un dato de tipo estructurado local en el llamador aparece como parámetro actual por valor en la llamada, entonces no es posible realizar la optimación.

Sin embargo, pensamos que solo hay una razón para elegir una estrategia de paso de parámetros como la que estamos tratando, y es el menor tamaño del código generado para un programa. En efecto, el proceso de copia de un objeto estructurado es relativamente complejo (aunque muchos procesadores, entre ellos MC68020 e Intel 80386, cuentan con instrucciones repetitivas que simplifican notablemente el proceso); si el subprograma que tiene como parámetro formal ese objeto de tipo estructurado es llamado en varias ocasiones, y si se realizase el proceso de copia en cada subprograma llamador, el código generado sería más grande que si se genera una sola vez en el receptor. Por contra, el programa será incluso un poco más lento, pues la copia del apuntador por parte del llamador no existiría si este directamente efectuase la copia del objeto completo.

En resumidas cuentas, diferir la copia de algunos parámetros actuales por valor hasta que el control es tomado por el receptor es una estrategia discutible, más cuando pensamos que si la llamada resulta optimada, se van a eliminar a su vez algunas instrucciones, que pueden compensar las que se añaden si es el subprograma llamador el que realiza la copia de los parámetros actuales en la pila. No es nada complejo, creemos, modificar el compilador para que se produzca una evaluación completa de todos los parámetros actuales durante la llamada, tal como se ha supuesto a lo largo de todo este trabajo, con lo que el problema planteado desaparecería. Pero, en general, podemos concluir que diferir hasta el receptor algunos procesos de copia de ciertos parámetros por valor puede impedir la optimación, si los parámetros actuales son objetos locales en el llamador.

6.4.5 Consideraciones adicionales

La elección de los diferentes aspectos que venimos englobando en lo que denominamos modelo dinámico del lenguaje puede determinar la facilidad con que la optimación puede aplicarse, como hemos visto. Una organización del registro de activación en que la zona de parámetros formales se sitúa en la base, antes de la zona de control, no impide de ninguna manera la optimación, aunque sí la dificulta, sobre todo en comparación con la alternativa que coloca la zona de control al comienzo del registro de activación. Estas dificultades aparecen solamente en relación con las activaciones finales no recursivas, y pueden conducir a un comportamiento temporal no tan mejorado como el que posibilita la organización que estudiamos originalmente. Del mismo modo, la utilización de instrucciones específicas de la máquina objeto para la implementación de las operaciones de llamada y retorno pueden precisar una organización del registro de activación como la señalada, pero no aparece ninguna otra dificultad adicional. Finalmente, la estrategia de paso de parámetros debe evitar, y es el único aspecto en que se produciría un impedimento de la optimación en algunas ocasiones, procesos "mixtos" en que parte de trabajo sea realizado por el receptor como final de su secuencia de llamada a partir de datos que estén en el registro de activación del llamador; sin embargo, la estrategia en que los parámetros actuales se colocan en registros de la máquina, a pesar de exigir la copia por parte del receptor, resulta ser de gran interés, pues conduce posiblemente al mejor comportamiento relativo de la optimación.

Nuestra postura, conocidos estos resultados, es que la optimación de las activaciones finales de procedimientos y funciones en PASCAL debería pasar al menos a formar parte de los criterios que un diseñador de un compilador para el lenguaje ha de observar. Por supuesto, criterios habituales como la generación de código compacto y de calidad, dinámicamente hablando, han de ser los preponderantes, igual que la sencillez del propio compilador. Sin embargo, el estudio que hemos llevado a cabo a lo largo de esta sección demuestra que una elección cuidadosa de los aspectos involucrados en el problema puede conducir a una optimación de las activaciones finales que puede mejorar notablemente el comportamiento espacial y temporal de muchos programas, sin penalizar más que en un pequeño factor (si es que existe tal

factor) el comportamiento temporal de las activaciones que no resultan optimadas. En todo caso, sería conveniente ofrecer esta optimación como una opción del compilador, como se hace con la inmensa mayoría de optimaciones que se practican, de manera que sea el programador y usuario del compilador el que pueda elegir su aplicación.

En este sentido, existe por lo menos un ámbito en el que la aplicación de la optimación debería ser prácticamente obligatoria, a nuestro entender, y es en un compilador orientado a la enseñanza de la programación. Recogiendo los argumentos de la introducción de esta tesis, resulta altamente interesante ofrecer a los estudiantes una herramienta que puede sacar partido de la recursividad (o de ciertas clases de recursividad, para ser más exactos), sin requerirles la confección de procesos iterativos equivalentes, a menudo más complejos y difíciles de aprehender en los primeros estadios del aprendizaje.

6.5 EL COMPILADOR OPTIMADOR PASCAL-JK

Para finalizar el capítulo se mostrarán las ideas más relevantes sobre el compilador de un subconjunto de PASCAL que se ha construido para (*soivitur ambulando*) acompañar de forma práctica la demostración de la posibilidad de optimación de las activaciones finales de subprogramas en el lenguaje. Primero, hay que mencionar que no se trata de un compilador competitivo (no solo por procesar un subconjunto del lenguaje), sino que es un prototipo en el que se han evitado una serie de cuestiones que no atañían directamente al problema, lo que ha permitido, por un lado, simplificar la construcción, y por otro lado, tener una visión más abstracta de los problemas realmente importantes. Es evidente que hemos de explicar en primer lugar las razones de haber obrado así.

En principio, parecería más lógico disponer de un compilador que practicara la optimación y que pudiera compararse cuantitativamente con otros compiladores no optimadores del lenguaje (de hecho, esta misma cuestión ya podría haberse planteado cuando realizamos la evaluación temporal de los programas optimados). Sin embargo, esta comparación, por muy buenos resultados que produjera, no sería a nuestro entender demasiado correcta. Es decir, puestos a realizar una evaluación temporal comparativa entre los

programas optimados y los programas no optimados, se podría llegar a que el programa X optimado en nuestro compilador A se ejecuta en, digamos, 15 segundos, y el mismo programa X obtenido sin optimación con otro compilador B se ejecuta en 30 segundos, vamos a decir; y entonces, se podría afirmar que el compilador A es mucho mejor que el compilador B porque practica la optimación (lo más seguro, por otro lado, es que se obtuvieran cifras contrarias, debido a la gran complejidad y eficiencia que introducen los compiladores comerciales hoy en día). Pensamos que esto no es correcto, porque en la ejecución de un programa producido por un compilador están influyendo un gran repertorio de aspectos que pueden llegar a enmascarar totalmente los resultados al comparar dos compiladores diferentes: la máquina y el sistema operativo subyacente (se podría hacer que estos factores no influyeran tomándolos iguales en los dos compiladores A y B, pero ya estaríamos excluyendo otras máquinas y otros sistemas operativos de los resultados), la organización general del programa objeto, el modelo dinámico elegido (en todos sus factores, como estructura del registro de activación, secuencias de llamada y retorno, mecanismo de paso de parámetros), el manejo de los registros y de las localidades temporales necesarias, *etc.* Tal como hemos visto en la sección 6.4, estos factores pueden variar en gran medida de un compilador a otro, y su elección puede estar motivada por argumentos relativamente complejos y variados. Y ello, al fin y al cabo, haría que las cifras poco tuvieran que ver con la aplicación de la optimación.

Si esto es así, la comparación debería plantearse en un mismo compilador, con idénticos programas fuente, y entre los programas objeto optimados y los no optimados. Por las mismas razones que antes, extrapolar esos resultados a otros compiladores no sería nada correcto: aspectos del compilador que pueden producir una significativa mejora al practicar la optimación pueden no ser válidos en otro compilador (tal como insistimos en diversos apartados de la sección mencionada, al respecto de la estructura del registro de activación, del uso de instrucciones específicas y del mecanismo de paso de parámetros), y convertir la mejora en nada, incluso.

Hay todavía otra perspectiva para esta cuestión, y tiene que ver con los programas que se elijan para la evaluación. Desde luego, es posible encontrar programas en que la optimación producirá un comportamiento dinámico realmente bueno, mientras que en otros quizás no se aprecie diferencia, e

incluso algunos pueden verse empeorados en ciertas condiciones. Ya identificamos con anterioridad los factores que determinan esas diferencias (número de parámetros y modo de los mismos, complejidad del propio subprograma, *etc.*), y eso es lo que realmente importa, a nuestro entender. Por supuesto, casi ninguno de los *benchmarks* comúnmente usados para la medida del rendimiento de los sistemas informáticos es aplicable al problema (Jain [1991: 45-59] presenta una sinopsis de los más populares; también Gabriel [1985] expone una evaluación de sistemas LISP, y contiene además una serie de interesantes discusiones sobre el problema de *benchmarking*). En todo caso lo podría ser el test de Wichmann [1977], que utiliza la función de Ackermann para medir la eficiencia del mecanismo de llamada a subprogramas en lenguajes de la familia de ALGOL. Sin embargo, está claro que no se pueden sacar conclusiones sobre nuestro problema exclusivamente a partir de un único programa recursivo. En definitiva, la preparación de una serie de programas de prueba no haría más que informarnos de lo que ya conocemos, y no se podrían obtener ninguna clase de generalizaciones sobre el comportamiento de *cualquier* proceso recursivo.

Por todas estas razones, se consideró adecuado desde un primer momento simplificar la construcción del compilador, fijando la atención en los detalles realmente significativos, y evitando todos los factores no directamente involucrados en el problema. Nuestra postura es que el compilador construido apoya la factibilidad de introducir las optimaciones de las activaciones finales, siempre con el objetivo de facilitar la ejecución de procesos recursivos. Hemos encontrado que cierta elección de los aspectos internos de la compilación (la que se aplica en el compilador que vamos a exponer) conduce a un comportamiento temporal mejorado para, posiblemente, un amplio rango de programas recursivos finales. Ahora bien, también hemos dedicado un gran esfuerzo a examinar distintas alternativas de implementación, que pueden encontrarse en diferentes compiladores, y hemos encontrado que casi ninguna de ellas (salvo el problema de la transferencia "mixta" de los parámetros por valor) impide la optimación, aunque las mejoras introducidas pueden no ser tan generosas. Pensamos que con ello, este trabajo puede ser de gran ayuda para los diseñadores de otros compiladores ya comercializados, pero han de ser ellos quienes al fin y al cabo evalúen definitivamente la posibilidad de la implantación de esta optimación, pues solo el conocimiento completo y preciso

de todos los detalles y necesidades internas del compilador y del sistema en que se soporta permitirán tomar esa decisión.

Esta simplificación, por otro lado, tampoco debía ser excesiva, pues existe otra perspectiva bajo la cual el compilador puede tener un claro interés: desde el punto de vista docente. Por un lado, como apoyo de los cursos introductorios a la programación basados en el lenguaje PASCAL (y también de materias como programación metódica), y por otro lado, como apoyo a la propia materia de compiladores. En ese sentido, el compilador construido podría ser la semilla de proyectos encaminados a disponer de tales herramientas. Este segundo criterio de extensibilidad nos llevó entonces a excluir solamente aquellos aspectos que no obligasen a un fuerte replanteamiento posterior de la estructura del compilador, de manera que la adición de los mismos al compilador se pueda realizar con el mínimo esfuerzo, y sin necesidad de una modificación radical de alguno de sus componentes.

Teniendo en cuenta estos supuestos, pasamos a describir las características del compilador, que llamaremos *compilador PASCAL-JK*. Se trata de un compilador que sigue las directrices habituales de los compiladores del lenguaje (tal como se presentaron en el apartado 6.3.1): se organiza en un paso a partir de un analizador sintáctico descendente no recursivo, que controla la rutina de análisis léxico, por un lado, y lanza acciones semánticas que realizan el análisis semántico y la generación de código. Desde luego, la Figura 6.3.1 describe exactamente la estructura básica del compilador. El código que se genera es un código virtual, a la vez generalización y simplificación de los posibles códigos máquina más habituales, similar al que utilizan Aho, Sethi y Ullman [1990: 535-538] y otros autores. Este código se interpreta después de haber sido generado, con lo que se puede observar el comportamiento de los programas. El compilador dispone de la optimación como opción, lo que también permite comparar los programas optimados y los no optimados.

El lenguaje fuente es un subconjunto de PASCAL. Evidentemente, elegir un subconjunto simplifica la tarea de construcción del compilador, pero debe hacerse de modo que no se omitan los factores realmente significativos. Por esa razón, no consideramos conveniente utilizar el subconjunto más conocido, PASCAL-S [Wirth 1981: 199-259], pues no incorpora algunas de las construcciones necesarias en el problema. Como es obvio, para ello el subconjunto a elegir no debería preocuparse demasiado por las estructuras de

datos presentes, sino más bien por todos los aspectos relacionados con el control, en todas sus implicaciones; ese fue el criterio utilizado para la exclusión de los siguientes aspectos (la definición del lenguaje utilizada ha sido la correspondiente a la revisión del estándar ISO 7185-1983 tal como queda reflejada en la versión de Mickel y Miner de 1991 del texto original de Jensen y Wirth [1974], y que ha sido citada en otras ocasiones como [Jensen y Wirth 1991]):

- En cuanto a los tipos de datos, se excluyen los *records*, los conjuntos (*sets*), y los apuntadores, además de todas las construcciones del lenguaje que se refieren al manejo de entidades de tales tipos (referencias de campos, constructores y operadores de conjuntos, sentencia *with*, etc.). Se mantienen todos los tipos simples (predefinidos, enumeraciones y subrangos), por la relativa facilidad de su implementación (aunque en el caso de los subrangos, hemos visto la importancia que pueden tener en el caso de las activaciones finales de funciones). En cuanto a los tipos estructurados, se eligió uno de ellos (los *arrays*) como representante del grupo, pero finalmente también se incluyeron los ficheros para contar con los mecanismos de entrada y salida, y en previsión de que fuera conveniente disponer de datos de entrada para algunos programas en memoria secundaria. El aspecto más conflictivo se planteó sobre la inclusión de los apuntadores; se decidió no hacerlo, pues en principio tales tipos de datos se refieren a la memoria dinámica, típicamente alojada en el montículo, que, en primer lugar, no plantea ninguna problemática en relación con el uso de la pila de control, y en segundo lugar, exige un esfuerzo considerable (ya en el propio procesamiento de las declaraciones, como se sabe).
- En cuanto al conjunto de funciones y procedimientos predeclarados, se excluyeron *Put*, *Get*, *New*, *Dispose*, *Pack* y *Unpack*, por no ser esenciales al problema. De la misma manera, no se consideraron los parámetros de escritura del procedimiento predeclarado *Write*, por suponer una mayor complejidad innecesaria en el análisis sintáctico.

Se mantuvieron en el subconjunto elegido, por las razones anteriormente descritas, dos aspectos significativos relacionados con el control y que habitualmente son los primeros candidatos para la exclusión cuando se trata de establecer un subconjunto del lenguaje, como ocurre con PASCAL-S: la sentencia

de salto incondicional, tanto local como no local y los parámetros procedimentales y funcionales⁵.

El código objeto tiene las características ya mencionadas al comienzo de la sección 6.2. Se supone que es una máquina basada en registros, al estilo de máquinas actuales como el MC68020 [Motorola 1985] o el i80386 [Intel 1989]. Los modos de direccionamiento ya se recogieron en la sección señalada: inmediato, directo, directo indexado, indirecto o a través de memoria e indirecto indexado. Existen instrucciones específicas para caracteres, enteros, reales, booleanos y direcciones. Las clases de instrucciones utilizadas son las siguientes (todos los operandos se pueden expresar con cualquier clase de direccionamiento):

- la instrucción de movimiento de datos, en que puede especificarse el tipo o tamaño del objeto a copiar:

$$\langle \text{destino} \rangle := \langle \text{fuente} \rangle [\langle \text{tamaño} \rangle]$$

- instrucciones aritméticas binarias, en que también puede especificarse el tipo de los objetos involucrados (op puede tomar diferentes valores según el tipo, pero básicamente se incluyen las necesarias para la aritmética de PASCAL, además de la suma y resta de direcciones):

$$\langle \text{destino} \rangle := \langle \text{fuente1} \rangle \text{ op } \langle \text{fuente2} \rangle [\langle \text{tamaño} \rangle]$$

- instrucciones aritméticas unarias, como cambio de signo, incremento y decremento, coerción de entero a real y viceversa, coerción de entero a carácter y viceversa, etc.::

$$\langle \text{destino} \rangle := \text{op } \langle \text{fuente} \rangle [\langle \text{tamaño} \rangle]$$

⁵ Es necesario comentar que, curiosamente, estos dos aspectos (ciertamente problemáticos en un compilador de PASCAL) son tratados de una forma bastante alejada de la definición estándar en algunos compiladores, sobre todo en el que sea posiblemente el más extendido: TURBO PASCAL [Borland 1992a: 91] no admite saltos no locales y de los saltos locales no se verifican las condiciones necesarias de aplicación [Jensen y Wirth 1991: 171-172], sino que se genera sin más un salto a la dirección asociada a la etiqueta; en cuanto a los parámetros procedimentales y funcionales [Borland 1992a: 47-50], el alejamiento del estándar es total, a pesar de que se ofrece una solución que, desde cierto punto de vista, se podría calificar de muy interesante; sin embargo, no se admite un subprograma no global (anidado en otro subprograma) como parámetro actual procedimental o funcional. En general, el seguimiento de ambos aspectos según la definición estándar puede empobrecer en bastante medida la compilación y/o la ejecución, pero pensamos que no es razón suficiente para cometer semejante "atentado" a la definición del lenguaje (con todas las consecuencias que ello acarrea, por ejemplo, de cara a la portabilidad).

- instrucciones de salto condicional, basadas en los operadores relacionales y lógicos presentes en PASCAL:

si *<fuente1>* op *<fuente2>* [*<tamaño>*] ir-a *<destino>*

- la instrucción de salto incondicional:

ir-a *<destino>*

- instrucciones adicionales como la de terminación del programa, la de no operación, y similares.

Para simplificar la interpretación, todos los procedimientos y funciones predeclarados de la librería de ejecución (salvo algunas funciones aritméticas) se resuelven mediante pseudoinstrucciones con los parámetros necesarios; de este modo evitamos preocuparnos de un aspecto ciertamente complejo que no tiene mayor interés de cara a la optimación.

Los saltos se realizan normalmente a puntos señalados por etiquetas, es decir, no se complica la generación de código con el problema del cálculo de direcciones, que obligaría, como ya señalamos en el apartado 6.3.1, a mantener en memoria el código de cada subprograma. Antes de la interpretación se aplica una fase equivalente al ensamblado, en que se genera un programa objeto muy similar pero sin etiquetas, que han sido sustituidas por las direcciones absolutas.

Por otro lado, tampoco se utilizan los supuestos registros disponibles en la máquina para la evaluación de expresiones, sino que se adopta una estrategia muy sencilla de manejo de localidades temporales del registro de activación de cada subprograma; en cierto modo, en este aspecto el comportamiento es similar al de una máquina de pila [Rees y Robson 1988: 108-123; Pittman y Peters 1992:159-166], pero sin variar el valor del registro *TopePila*, que se mantiene inmóvil señalando la cumbre de la pila de control durante toda la activación de cualquier subprograma. Así, también se evita uno de los aspectos más complejos de la generación de código (aunque, por supuesto, esto debería ser una de las primeras cosas a cambiar si el compilador se dedicase a alguno de los objetivos docentes mencionados antes).

La estructura del programa objeto generado es la habitual: existe una sección con el código y otra sección con los datos globales, que puede verse como el registro de activación principal, a partir del cual se colocará la pila de

control. Al no incluir los tipos apuntadores, no es necesario el montículo ni su manejo. Los demás aspectos son los ya conocidos por los capítulos anteriores: la estructura del registro de activación con la zona de control en la base, el uso de los registros *BasePila* y *TopePila* para su demarcación (junto con el registro *Datos* para señalar los datos globales), secuencias de llamada y retorno, *etc.* De hecho, gran parte de los programas objeto mostrados en la sección 6.2 son remodelaciones de los que produce el compilador. Solo diremos que se ha implementado el acceso a los datos no locales a través del establecimiento de la cadena estática para cada subprograma. La estrategia del *display*, que hemos estudiado teóricamente a lo largo de todo el trabajo, puede implementarse con igual facilidad.

En cuanto a la propia estructura del código objeto que se genera, la posibilidad de utilizar etiquetas evita cierta falta de secuencialidad que se produciría si se tuvieran que generar direcciones absolutas. Los trozos de código de cada subprograma están bien delimitados, y cada uno comienza con la parte de secuencia de llamada como receptor y termina con la parte de secuencia de retorno también como receptor. Es de resaltar que se ha encontrado el mismo problema reseñado por Rees *et al.* [1981: 270-273]: a la hora de actualizar el registro *TopePila* como penúltima operación de la secuencia de llamada, el subprograma receptor debe conocer el tamaño que ha de destinar a sus datos locales y temporales; los locales son conocidos por la declaración, pero los temporales necesarios no se conocerán hasta finalizar el procesamiento de la parte de sentencias del subprograma. Los autores mencionados colocan entonces el punto de entrada real del subprograma al final del código del mismo, con un salto al comienzo una vez que se han ejecutado las operaciones necesarias de la llamada. Nuestra solución evita esta ruptura en la secuencialidad del código a través de una etiqueta especial cuyo valor se define al finalizar el subprograma; de todas formas, el problema podría igualmente resolverse aplicando la técnica del relleno de retroceso.

El lenguaje de programación del compilador es SCHEME. Esta puede parecer una elección sorprendente, pero trataremos de argumentarla. Conocíamos en cierta medida el lenguaje LISP con anterioridad al comienzo del trabajo, y su descendiente SCHEME tuvo que ser uno de los temas a estudiar durante la primera parte del trabajo, como ya se ha señalado en el capítulo 2, debido a su característica de ser propiamente recursivo final. Hemos de

reconocer que tras una temporada de toma de contacto con el lenguaje SCHEME, nos encontrábamos muy gratamente sorprendidos por el mismo, debido a sus cualidades; como Friedman escribe en el prólogo del libro de Dybvig [1987]:

«A computer language should provide an environment in which the programmer can creatively produce good computational models or paradigms for the problems at hand. The programmer's ability to construct paradigms should not be restricted. I call the ease of constructing paradigms the language's "paradigmaticity." Languages with low paradigmicity are *boring*. Programmers should not be straitjacketed. The creativity of humans is too diverse, our computing needs too varied, and our love of aesthetics too great to be hobbled by languages with low paradigmicity» [Friedman 1987: vii (las cursivas están en el original)]

Posiblemente fue la novedad de las construcciones del lenguaje lo que nos atrajo (pues nuestro trabajo hasta entonces se había desarrollado principalmente en el ámbito de los lenguajes imperativos, sobre todo PASCAL y C), pero el hecho es que teníamos ante nosotros un mundo inexplorado por conocer.

Ello hizo que nos interesáramos, de forma personal y sin relación en principio con el trabajo que estábamos empezando, por el siguiente paso, la programación funcional, y sucedió algo prácticamente similar. Una de las razones más comúnmente aducidas para adoptar el paradigma funcional es la facilidad de desarrollo de especificaciones ejecutables y de la implementación de prototipos [MacLennan 1990: 4; Hudak 1989]. A pesar de que SCHEME no es rigurosamente un lenguaje funcional, es lo más parecido, y está mucho más extendido y desarrollado que otros lenguajes funcionales "puros". El uso de SCHEME en relación con la construcción de compiladores de lenguajes de diferentes paradigmas (no necesariamente funcionales) es reseñado por Pleban, también en un prólogo, esta vez del trabajo doctoral de Lee [1989]:

«Languages such as Scheme, ML, or even the Mathematica metalanguage are superb tools for compiler (generator) writing. Quite frankly, I find the almost superstitious misconceptions of the Pascal/C/Ada/Modula community concerning the efficiency and practicality of these "almost functional" languages rather disturbing» [Pleban 1989]

Más recientemente, Orbaek [1994] utiliza SCHEME de forma similar, en el mismo contexto de generación de compiladores.

Finalmente, hay dos aspectos adicionales que justifican todavía más la elección. Primero, LISP y SCHEME son dos lenguajes altamente cualificados para el cálculo simbólico, y ello desde su misma concepción, lo que ha posibilitado el gran desarrollo de la inteligencia artificial durante las cuatro décadas pasadas. Si hay un proceso que necesite cálculo simbólico, ese es la compilación; ocurre que las implementaciones habituales de los compiladores en lenguajes no preparados para el cálculo de esta clase realizan una codificación de los nombres, y unidades léxicas en general, del programa fuente, pero esto indudablemente oscurece la programación del compilador. La presencia de objetos de tipo símbolo en LISP y SCHEME es de gran ayuda para mantener un alto nivel de abstracción, e incide en ese criterio de posible ayuda a la docencia de la materia de compiladores.

En segundo lugar, otra de las necesidades habituales en un compilador clásico es el manejo de estructuras de datos de muy diferentes clases y contenidos: árboles, pilas, tablas de símbolos. Las listas, tipo de dato básico en LISP y SCHEME, son una de las estructuras más cómodas y simples jamás creadas para el desarrollo de toda clase de estructuras de datos, incluso bajo la filosofía de tipos abstractos, y ese fue uno de los aspectos que más nos sorprendió desde el principio. Evidentemente, el manejo implícito (totalmente escondido para el programador) de los apuntadores de las listas en LISP y SCHEME debe hacerse explícito en lenguajes que no incorporan listas entre su repertorio de tipos, razón por la cual la implementación de estructuras de datos basadas en listas es notablemente simple.

En definitiva, elegimos el lenguaje SCHEME para la implementación del compilador. Durante un primer periodo, se desarrolló con ayuda del sistema PC-SCHEME 3.0 [Texas Instruments 1987a, 1987b] para el sistema operativo MS-DOS. Más adelante, se dispuso de una herramienta mucho más completa desarrollada en el MIT, llamado MIT-SCHEME, en su versión, primero para el sistema operativo MS-DOS [Hanson 1991a, 1991b], y luego para el sistema operativo WINDOWS 3.1 de Microsoft [Adams y Hanson 1993; Hanson 1993]. Todo ello en máquinas compatibles IBM-PC, basadas en el Intel 80486 a 33 MHz. En general, la impresión obtenida acerca del uso de SCHEME para la programación del compilador ha sido muy grata, y podemos decir que no ha disminuido el entusiasmo que teníamos desde los momentos iniciales. Obviamente, el seguimiento de una metodología adecuada en la programación

ha de ser la responsable última de que esto haya sucedido así (y lo sería en cualquier otro lenguaje de programación), pero ha permanecido, sin que podamos explicarlo de una manera consciente o lógica, la impresión de que existía cierta facilidad intrínseca en el uso de lenguaje (una vez conocido en toda su extensión), o bien en los métodos de programación, que comentamos más adelante, debido probablemente al contraste con nuestra experiencia previa de programación en lenguajes como el propio PASCAL o C.

Al utilizar SCHEME como lenguaje de programación, se construyó una primera versión del prototipo en la que el lenguaje objeto era el propio SCHEME, pero donde las estructuras internas del compilador ya estaban en su forma prácticamente definitiva (salvo que no se almacenaba el árbol completo de cada subprograma, con lo que el conversor era de un paso en el sentido más estricto del concepto). Así, el código SCHEME generado era recursivo final si el programa fuente era recursivo final, con lo que se pudieron hacer las primeras observaciones del comportamiento. Hay que decir que en la construcción de esta primera versión del prototipo se encontraron una serie de cuestiones muy interesantes al respecto de las diferencias entre ambos lenguajes (PASCAL y SCHEME), pero no es este el momento de entrar en ellas. Solamente indicaremos que cuestiones similares han sido hace un tiempo abordadas en la literatura (por ejemplo, por Moynihan y Wallis [1991]). En general, fue posible reflejar todos los aspectos del lenguaje PASCAL en el código SCHEME, siendo la excepción más notable la correspondiente a los saltos incondicionales (sentencia goto) no locales. Se utilizaron los mecanismos de manejo de continuaciones del lenguaje SCHEME [Dybvig 1987: 78-83] para la implementación de los saltos locales, poniendo en práctica uno de los usos más habituales de tal mecanismo, con una solución que podemos calificar de muy elegante. Sin embargo, la implementación de los saltos no locales no fue posible, o más bien deberíamos decir que abandonamos el problema por no ser relevante en aquellos momentos.

El prototipo definitivo exigió que las acciones de procesamiento semántico almacenaran el árbol completo de cada subprograma, lo cual fue una modificación bastante simple, pero también la construcción de un generador de código completamente distinto, por razones obvias (más bien, completamente nuevo, pues en el primer prototipo las acciones de generación de código se entremezclaban con las propias acciones de procesamiento semántico). La

versión final ocupa algo más de 10.000 líneas, sin incluir los comentarios a los programas. Esto puede parecer una exageración en comparación con las aproximadamente 1.500 líneas que ocupa el compilador de PASCAL-S de Wirth [1981], escrito en el propio PASCAL. Sin embargo, se incluyen en nuestro subconjunto bastantes más elementos del lenguaje original que en PASCAL-S, además justamente elementos que son relativamente problemáticos para su procesamiento, como los ficheros, los parámetros procedimentales y funcionales y la sentencia goto. Pero sobre todo ha sido la metodología de programación utilizada la que ha provocado este tamaño en el programa, cosa que no debe considerarse negativa a nuestro entender, pues al fin y al cabo, se trata de un prototipo en el que prevalecen los criterios de abstracción y modularidad sobre la compacidad y rapidez de la compilación.

La estructura del compilador se eligió de acuerdo a dos criterios primordiales: primero, la estructura clásica de los compiladores del lenguaje PASCAL (tal como se recogió en el apartado 6.3.1 con las salvedades necesarias para poder practicar la optimización), y segundo, la posibilidad de que fuera un compilador con altas cualidades desde el punto de vista docente (en consonancia con lo comentado anteriormente). Por ello, se ha enfatizado la consecución de un alto nivel de abstracción en la estructura del programa, además de la ya proporcionada por el lenguaje SCHEME. En principio, la estructura es, como hemos dicho, la reflejada en la figura Figura 6.3.1. Sin embargo, se han creado toda una serie de subestructuras internas para reflejar el estado de la compilación en cada momento, que aparecen ya explícitamente en dicha figura; son el estado de lectura, el estado de error y el estado semántico, controlados todos ellos por el programa principal, que es el analizador sintáctico, y que podríamos decir que observa la misma estructura, incluyendo los anteriores y sus propios recursos necesarios.

En general, entendemos el estado de lectura, el estado de error y el estado semántico como tipos abstractos de datos. Es decir, la información necesaria en cada instante de la compilación se encapsula en una entidad, de manera que solamente es posible acceder a ella a través de operaciones especiales. Otras estructuras de datos en un sentido más clásico que se necesitan en el compilador, concretamente pilas y tablas de símbolos, también se gestionan de la misma forma. Sin embargo, sorprendente pero lógicamente, no resultó necesario en modo alguno el diseño de una estructura de datos para

almacenar al árbol decorado de cada subprograma: las listas hacen el trabajo por nosotros. Los cinco objetos mencionados se programaron entonces siguiendo la filosofía particular de implementación de tipos abstractos de datos conocida en la comunidad del lenguaje SCHEME como "*message-passing style*" [Abelson, Sussman y Sussman 1985: 140-141; Dybvig 1987: 209-214; Friedman, Wand y Haynes 1992: 214], consistente en representar una clase de objetos mediante una función de orden superior que produce una función que acepta mensajes sobre el objeto particular creado. La aplicación de esta filosofía a estructuras de información como pilas o tablas de símbolos es habitual, pero hacerlo a estructuras que reflejan la situación dinámica de un proceso, como el de la compilación, es todavía más interesante. Por ello, nos extenderemos un momento en presentar la idea.

Tomamos el ejemplo del estado de lectura. Como se sabe, es conveniente disponer de un subsistema específico de lectura por diferentes razones. Quizá la más importante es la necesidad de contar con caracteres de anticipación para la realización del análisis léxico. En PASCAL, la gran mayoría de lexemas se delimitan con el examen de un único carácter de anticipación, pero existe una situación concreta en que son precisos dos caracteres de anticipación: la diferenciación de una constante real y una especificación de tipo subrango como `1..10` [Fischer y LeBlanc 1988: 76-77]. Entonces, se elige un sistema de entrada en que se maneja un *buffer*, que almacena en cada momento una línea completa del programa fuente, con dos marcadores finales especiales para posibilitar en cada momento dos caracteres de anticipación. Además, el control de la lectura se establece gracias a una serie de variables, tal como queda reflejado en la siguiente figura.

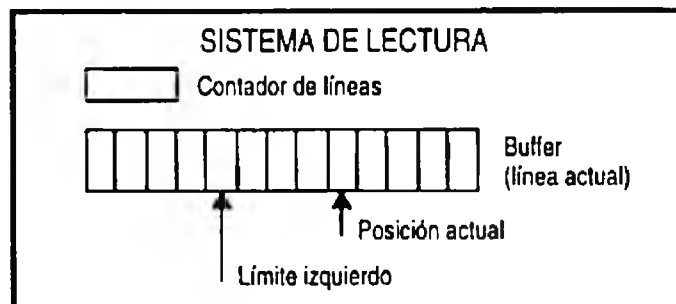


Figura 6.5.1 Estructura del subsistema de lectura del compilador PASCAL-JK

Sobre esta estructura se precisan las siguientes operaciones básicas:

- obtención del carácter de la posición actual;
- obtención del carácter de la posición siguiente a la actual;
- avance de la posición de lectura;
- colocación de la marca de comienzo de lexema;
- obtención del lexema, que es la cadena situada entre el límite izquierdo (inclusive) y la posición actual (exclusive).

Estas especificaciones permiten definir un tipo abstracto de datos, que llamaremos *read-state*, con la siguiente signature:

```

univers  read-state
import  char, string, file
type    read-state
operators
  read-state/maker:      file -> read-state
  get-char: read-state -> char
  get-next-char: read-state -> char
  get-lexema: read-state -> string
  next-pos!: read-state -> read-state
  set-first!: read-state -> read-state
equations
   $\forall r-s \in \text{read-state}, \forall f \in \text{file}:$ 
  get-char(read-state/maker(f)) = current(init(f))
  get-char(next-pos!(r-s)) = get-next-char(r-s)
  get-char(set-first!(r-s)) = get-char(r-s)
  get-next-char(r-s) = get-char(next-pos!(r-s))
  get-lexema(read-state/maker(f)) = null-string
  get-lexema(set-first!(r-s)) = null-string
  get-lexema(next-pos!(r-s)) = concat(get-lexema(r-s), get-char(r-s))
  if eof(f) then next-pos!(r-s) = r-s
end-univers

```

Figura 6.5.2 Signature del tipo abstracto de datos que refleja el estado de lectura

La implementación en SCHEME de este tipo abstracto de datos, que aparece en el Programa 6.5.1, se basa en definir una función de orden superior para el constructor *read-state/maker*. Esta función devuelve un estado de lectura, que es a su vez una función que acepta mensajes para realizar los diferentes operadores. Las estructuras de datos internas que aparecen en la Figura 6.5.1 no son, sin embargo, accesibles más que a través de los operadores, con lo que se consigue el encapsulamiento deseado.

Entonces, el programa principal del compilador será el encargado de crear (al comenzar la compilación) un estado de lectura, mediante:

```
(define rd-state (read-state/maker "source.pas"))
```

```

(define rd/state-maker
  (lambda (filename)

    ; variables internas que definen la implementación del TAD
    (let* ((source-port (open-input-file filename))
           (current-cont 1)
           (current-line ...)
           (current-pos 0)
           (current-char (string-ref current-line 0))
           (next-char (string-ref current-line 1))
           (first-pos 0))

      ; fin de las variables

      ; read-state/maker produce una función que acepta mensajes
      (lambda (op)
        (case op
          (get-char      current-char)
          (get-next-char next-char)
          (get-lexema    ...)
          (next-pos!    ...)
          (set-first!    ...)
          (else
           (error "READER-PROCESSOR: bad operation:" op)))))))

```

Programa 6.5.1 Implementación en SCHEME del subsistema de lectura como tipo abstracto de datos, basada en la signatura de la Figura 6.5.2

Este objeto es transferido como argumento, entre otras, a la función que realiza el análisis léxico, la cual puede enviar mensajes a dicho objeto para obtener la operación necesaria en cada momento:

```

(rd-state 'get-char)
(rd-state 'get-next-char)
(rd-state 'set-first!)

```

son ejemplos de obtención del primer carácter de anticipación, del segundo carácter de anticipación y de colocación del límite izquierdo para un lexema, respectivamente.

Este mecanismo de implementación de tipos abstractos de datos en SCHEME se utiliza, como hemos dicho, para las pilas (existen la pila sintáctica como parte del analizador sintáctico, y la pila semántica y una pila especial para el procesamiento de la sentencias goto, ambas como parte del estado semántico), la tabla de símbolos (que es parte también del estado semántico, y se implementa como una pila de subtablas, una por bloque léxico, tal como explican Fischer y LeBlanc [1988: 262-264]), el estado de lectura, el estado de error y el propio estado semántico.

La parte funcional del compilador queda entonces compuesta por el algoritmo de análisis sintáctico (que hace las veces de programa principal, al lanzar las operaciones que ejecutan el resto de tareas de la compilación), el algoritmo de análisis léxico, y las acciones de procesamiento semántico. Estas acciones semánticas se dividen en cuatro grandes clases, aunque funcionalmente no hay distinción entre ellas:

- las acciones de procesamiento de la parte de declaraciones de cada bloque, y que no generan código;
- las acciones de procesamiento de la parte de sentencias de cada bloque, que construyen el árbol decorado sin generar tampoco código;
- la acción de optimación, que recorre al final de cada bloque el árbol decorado construido, marcando de la manera adecuada los nodos en que puede practicarse la optimación;
- la acción de generación de código, que es lanzada al final de la parte de sentencias de cada bloque, después de la optimación, y que utiliza el árbol decorado construido y posiblemente modificado por esta última acción.

Evidentemente, hacer que la optimación sea opcional resulta muy sencillo, pues se trata de activar o desactivar un centinela que puede examinar la propia acción semántica de optimación para ejecutarse o no.

En definitiva, estamos viendo la manera en que se cumplen los diferentes requisitos estáticos de la optimación, tal como se presentaron en la sección 6.3, concretamente los reflejados en la Condición 6.3.1. La estructura básica de la acción semántica que detecta la optimación ya se presentó en el Programa 6.3.1 y el Programa 6.3.2. Quizá solo sea necesario ya mostrar la parte del generador de código que corresponde a las activaciones de subprogramas, sean optimadas o no. Empezamos con la activación normal (no optimada) de un procedimiento, siguiendo el Algoritmo 2.5.1, que se recoge en el Programa 6.5.2. Realmente, prácticamente el mismo esquema se utiliza para las activaciones de funciones, con la salvedad de que para estas se genera al final la captación del valor de retorno producido por la función llamada a una posición segura.

La activación final recursiva optimada de un procedimiento o una función (indistintamente, como se sabe) se consigue con la acción de generación de código que proviene del Algoritmo 3.2.1, y que está contenida en el Programa 6.5.3 más adelante.

```

; las sentencias de activación no final de procedimiento (no formal):
;   (<[cl]-PROC-CALL> proc-id [(mode par)]*)
; donde [cl] es LOCL, REC, CREC o NREC

(define gen/proc-call-stment
  (lambda (sem-st err-st p-call)
    (let* ((symb-tab (cod/symb sem-st))
           (tmp-mgr (tmp/maker symb-tab))
           (p-class (first p-call))
           (p-id (second p-call))
           (par-list (cddr p-call))
           (etiq (gensym))) ; etiqueta para dir. de retorno
      ; evalúa los parámetros actuales y crea los formales
      (gen/proc-func-par sem-st err-st tmp-mgr (cddr p-call)
        (symb/get-attr symb-tab p-id 'TYPE-PAR)
        PS/PROC-CNTL-SIZE)
      (tmp/delete! tmp-mgr)
      ; empieza el resto de la secuencia de llamada
      ; añade el enlace de control
      (cod/gen sem-st "(PS/COPIA (PS/DDIRX TopePila " PS/CNTL-LINK-PROC
        ") (PS/DINM BasePila) PS/DIR)" 'line)
      ; añade el enlace de acceso: la cosa depende de p-class
      (case p-class
        ; si es local, el enlace de acceso es BasePila
        (<LOCL-PROC-CALL>
         (cod/gen sem-st "(PS/COPIA (PS/DDIRX TopePila "
           PS/ACCS-LINK-PROC
           ") (PS/DINM BasePila) PS/DIR)" 'line))
        ; si es REC o CREC (distancia=1), el ea es el del llamador
        (<REC-PROC-CALL> <CREC-PROC-CALL>)
        (cod/gen sem-st "(PS/COPIA (PS/DDIRX TopePila "
          PS/ACCS-LINK-PROC
          ") (PS/DDIRX BasePila "
          PS/ACCS-LINK-LOC
          ") PS/DIR)" 'line))
        ; si es NREC (distancia>1), hay que calcularlo
        (<NREC-PROC-CALL>
         (let ((ea (gen/access-link sem-st err-st p-id)))
           (cod/gen sem-st "(PS/COPIA (PS/DDIRX TopePila "
             PS/ACCS-LINK-PROC
             ") " ea " PS/DIR)" 'line))))
      ; añade la dirección de retorno
      (cod/gen sem-st "(PS/COPIA (PS/DDIRX TopePila " PS/DIR-RET-PROC
        ") (PS/ETQ " etiq ") PS/DIR)" 'line)
      ; coloca BasePila en su nuevo valor
      (cod/gen sem-st "(PS/DIR* (PS/DINM BasePila) (PS/DINM TopePila) "
        "(PS/DIR " PS/PROC-CNTL-SIZE ") )" 'line)
      ; salto al comienzo del procedimiento llamado
      (cod/gen sem-st "(PS/IR-A "
        (list 'PS/ETQ (symb/get-attr symb-tab p-id 'ENTRY))
        ") " 'line)
      ; la dirección de retorno es la siguiente
      (cod/gen sem-st "(PS/ETIQ " etiq ") " 'line)
      ; ha terminado la parte de la secuencia de llamada que está
      ; bajo la responsabilidad del llamador; comienza ahora la
      ; parte de la secuencia de retorno que es del llamador;
      ; para un procedimiento, el llamador no ha de hacer nada más,
      ; así que se borran los temporales y yastá
      (tmp/delete! tmp-mgr)))

```

Programa 6.5.2 Implementación en SCHEME de la generación de código para una sentencia de activación de procedimiento, sin optimación

```

; la sentencia de activación recursiva final de procedimiento o función:
; (<TAIL-REC-PF-CALL> pf-id [(mode par)]*)

(define gen/tail-rec-pf-call
  (lambda (sem-st err-st pf-call)
    (let* ((symb-tab (cod/symb sem-st))
           (tmp-mgr (tmp/maker symb-tab))
           (pf-id (second pf-call))
           (par-list=(caddr pf-call))
           ; evalúa los parámetros actuales bajo las restricciones conocidas
           (gen/tail-rec-pf-par sem-st err-st tmp-mgr (caddr pf-call))
           (symb/get-attr symb-tab pf-id 'TYPE-PAR))
      ; salto al punto de entrada alternativo del propio proc/func
      (let ((entry (symb/get-attr symb-tab pf-id 'ENTRY)))
        (cod/gen sem-st "(PS/IR-A (PS/ETQ " (symbol-append entry '-real)
          ")))" 'line))
      ; ya está: superfácil y superóptimo
      (tmp/delete! tmp-mgr))))

```

Programa 6.5.3 Implementación en SCHEME de la generación de código para una sentencia de activación final recursiva optimada de procedimiento o función

Es evidente la facilidad de esta acción de generación de código en comparación con la normal. El punto de entrada alternativo al procedimiento o función, para evitar la actualización innecesaria del registro *TopePila*, se supone que es generado en la cabecera del código del propio subprograma. Evidente, la rutina para la evaluación de parámetros actuales y asignación a los formales es diferente de la que se utiliza en la llamada estándar (no optimada).

Para las activaciones finales no recursivas, se engloban en una sola acción de generación de código los dos casos posibles (sin contar con la activación de un procedimiento formal, por lo que se sigue el Algoritmo 4.2.1), pero se distinguen dentro de ella, concretamente a la hora de establecer el enlace de acceso (tarea que concluimos no era necesaria en las activaciones llamadas "casi recursivas"). La acción se presenta en el Programa 6.5.4.

En las acciones anteriores no se contemplan las activaciones (optimadas o no) de parámetros formales procedimentales o funcionales, pero no son muy diferentes de lo mostrado: únicamente varía la forma en que se obtiene el enlace de acceso y la dirección de salto. De todos modos, el apéndice A recoge el listado completo del módulo de generación de código del compilador PASCAL-JK, donde pueden consultarse, entre otras cosas, estas rutinas. También se incluye en dicho apéndice el módulo de optimación, para comprobar los detalles acerca de la misma.

```

; la sentencia de activación no recursiva final de procedimiento o
; función es:
;   (<TAIL-[cl]-PF-CALL> pf-id {(mode par)}*)
; donde [cl] es CREC o NREC (conviene distinguir los dos casos);

(define gen/tail-norec-pf-call
  (lambda (sem-st err-st pf-call)
    (let* ((symb-tab (cod/symb sem-st))
           (pf-class (first pf-call))
           (pf-id (second pf-call))
           ;par-list= (cddr pf-call))
      ; se comprueba la diferencia de tamaños entre uno y otro
      ; subprograma, y se incrementa el local-offset hasta apuntar por
      ; encima de la zona de formales del receptor
      (formal (symb/get-attr symb-tab pf-id 'TYPE-PAR))
      (fmal-size (apply + (map fifth formal)))
      (offsetdif (- fmal-size (symb/local-offset symb-tab)))
      (tmp-mgr (begin
                  (if (> offsetdif 0)
                      (symb/incr-local-offset! symb-tab offsetdif)
                      (tmp/maker symb-tab))))
      ; evalúa los parámetros actuales bajo las restricciones conocidas
      (gen/tail-norec-pf-par sem-st err-st tmp-mgr (cddr pf-call) formal)
      ; añade el enlace de acceso solo si la llamada es NREC (si es CREC
      ; el enlace de acceso no varía)
      (if (eq? pf-class '<TAIL-NREC-PF-CALL>)
          (let ((ea (gen/access-link sem-st err-st pf-id)))
            (cod/gen sem-st "(PS/COPIA (PS/DDIRX BasePila "
                        "PS/ACCS-LINK-LOC
                        ") * ea * PS/DIR)" 'line)))
          (list 'PS/ETQ (symb/get-attr symb-tab pf-id 'ENTRY)
                ")*" 'line)
      ; yastá: esta no era tan fácil, pero queda bien; solo hay que borrar
      ; los temporales y decrementar el offset que se añadió al principio
      (tmp/delete! tmp-mgr)
      (if (> offsetdif 0)
          (symb/decr-local-offset! symb-tab offsetdif))))))

```

Programa 6.5.4 Implementación en SCHEME de la generación de código para una sentencia de activación final no recursiva optimada de procedimiento o función

7. RECURSIVIDAD FINAL Y PROGRAMACIÓN EN PASCAL

7.1 INTRODUCCIÓN

En este capítulo que termina el "cuerpo" del trabajo fijaremos varias ideas relacionadas con la programación en el lenguaje PASCAL cuando se dispone de un procesador que realiza la optimación de las activaciones finales de procedimientos y funciones. Para empezar, analizaremos el alcance de las restricciones de la optimación, es decir, la frecuencia de las activaciones que a pesar de ser finales no pueden resultar optimadas debido a las restricciones descubiertas. Luego veremos que la posibilidad de la optimación puede cambiar algunos hábitos de la programación en PASCAL, o al menos, incorporar algunas técnicas nuevas destinadas a sacar el mayor partido de la optimación.

7.2 ALCANCE DE LAS RESTRICCIONES DE LA OPTIMACIÓN

Las restricciones de la optimación son básicamente de cuatro clases, tal como se clasificaron al comienzo del apartado 6.3.2. Las restricciones de posición final son las condiciones mínimas para que una activación sea optimable. Estadísticamente, puede ocurrir que esta clase de activaciones no represente un porcentaje importante de las llamadas a subprogramas en PASCAL, lo mismo que en cualquier lenguaje de programación, pero eso no invalida la importancia de la optimación, pues se espera que una vez que se

disponga de compiladores que practiquen la misma, los programadores tiendan a utilizar las activaciones finales en mayor medida. Así ocurre en el lenguaje SCHEME, que incluye la necesidad de optimación de las llamadas finales en su definición.

Echaremos un vistazo a las técnicas conocidas para conseguir activaciones finales más adelante, mientras que ahora nos interesa conocer cómo afectan las otras restricciones, primordialmente por comparación con el lenguaje SCHEME, que no las presenta (aunque, salvo las restricciones de anidamiento, ya se concluyó en las secciones 3.3, 4.3 y 5.4 que las restricciones de parámetros y de valor de retorno aparecerían de una manera u otra en este lenguaje: tales restricciones provienen de características semánticas del lenguaje PASCAL que provocan la aparición de "procesos ocultos", que surgirían explícitamente en SCHEME).

En las activaciones finales recursivas, ya sean de procedimientos o de funciones, las únicas restricciones son las de parámetros. Pensamos que estas restricciones no son, en este caso, nada fuertes, y van a existir muy pocos subprogramas recursivos finales que presentan tal característica. En el caso de los parámetros variables o por referencia, la Restricción 3.2.2 prohíbe la optimación si se coloca como parámetro actual un dato local o un parámetro formal por valor del propio subprograma. Téngase en cuenta que los parámetros variables se utilizan básicamente para que un subprograma produzca un valor en el llamador, mediante un efecto lateral y no como resultado de una función, por ejemplo.

Por lo tanto, parece lógico pensar que la activación de un proceso recursivo con parámetros variables tenga el propósito de obtener una serie de valores a su finalización, con lo que la memoria de esos valores se situará en el subprograma que lanza el proceso recursivo, no dentro de este. Además, si este es el esquema básico de uso de los parámetros variables, probablemente el parámetro actual sea exactamente el propio parámetro formal variable, con lo que se podrá llevar a efecto la optimación consistente en no actualizarlo en las diferentes activaciones finales.

Otro uso posible de parámetros variables en subprogramas recursivos aparece en relación con operaciones de manejo de listas ligadas, implementadas en PASCAL, como se sabe, mediante *records* en que uno de los campos es un

apuntador al siguiente elemento. Tomamos como paradigmático el procedimiento de duplicación de una lista ligada escrito por Rohl [1984: 26].

```

procedure CopyList (var destino: listptr; fuente: listptr);
begin
  if fuente = nil then destino := nil
  else
    begin
      new (destino);
      destino^.item := fuente^.item;
      CopyList (destino^.next, fuente^.next)
    end
  end
end ( of procedure "CopyList" )

```

Programa 7.2.1 Un procedimiento recursivo final para duplicar una lista ligada (solo con ligeras variaciones a partir del original de Rohl [1984: 26])

En este ejemplo, el procedimiento recursivo no solo ha de realizar un efecto lateral en la terminación del proceso, sino en cada paso, por lo que necesita el parámetro variable, pero este señala siempre (salvo en la primera activación) a memoria dinámica, por lo que no existe ningún problema, además de que en cualquier caso, la Restricción 3.2.2 no es de aplicación, pues ningún dato local o parámetro por valor local se coloca como parámetro actual variable.

Mirado de otra manera, si en algún momento se colocase un dato local como parámetro actual variable en un momento intermedio de un proceso recursivo directo, sería para obtener un efecto lateral en ese dato a la vuelta del proceso recursivo, con lo que la activación no será, casi con toda seguridad, final, sino que operará a partir del efecto obtenido en uno u otro sentido. Tenemos un ejemplo de esto en la implementación de Wirth [1981: 226-228] del subconjunto PASCAL-S, en que el procedimiento `typ` es activado recursivamente para las declaraciones de *records*, con datos locales como parámetros actuales por referencia, pero porque los efectos laterales que se obtienen son utilizados para la continuación del proceso, es decir, la activación recursiva no es final.

Finalmente, a veces se colocan como parámetros variables objetos sobre los que realmente no se efectúa ningún efecto lateral, debido a una cuestión que ya hemos estudiado anteriormente. Se trata de los objetos de tipo estructurado de gran tamaño, que colocándose como parámetros variables, no necesitan ser copiados enteramente en cada llamada. Pero el propio hecho de la ausencia de efectos laterales sobre tal objeto garantiza prácticamente que el parámetro no va a cambiar en las sucesivas llamadas, por lo que la restricción no será de

aplicación. Si en algún momento se precisase un efecto lateral intermedio, sería necesaria la copia del objeto en uno local, lo que invalidaría la mejora introducida al ser establecido como parámetro variable. En definitiva, es de esperar que las restricciones relativas a los parámetros variables impidan la optimización muy esporádicamente.

El mismo argumento puede servir para la restricciones de parámetros procedimentales o funcionales, en que no puede aparecer como parámetro actual un subprograma local del llamador. El uso de estos parámetros permite que se ejecuten durante cada paso del proceso recursivo o al final del mismo ciertas operaciones que pueden variar entre diferentes activaciones del proceso completo, y por ello se ofrecen como parámetros procedimentales o funcionales. Es lógico pensar que el parámetro actual, en la mayor parte de los casos, no cambiará a lo largo del proceso, y será el subprograma activador de la recursividad el que establezca el valor real de estos parámetros, que por supuesto, no pueden ser locales en el subprograma recursivo final. Hay que tener en cuenta además, como ya dijimos en el capítulo 3, que el uso de parámetros procedimentales y funcionales es muy esporádico en PASCAL, por lo que el impacto de la restricción es mucho menor que en el caso de los parámetros variables.

En el caso de las activaciones no recursivas, la cosa cambia sustancialmente, y no solo porque existen restricciones adicionales. Para empezar, los argumentos anteriores sobre las restricciones de parámetros no se pueden mantener tan sencillamente. Sí sigue siendo cierto que no es probable que se coloque un dato local como parámetro actual variable para obtener un efecto lateral si la activación es final, pues ese efecto lateral no sería observable por el llamador. Pero sí es posible pensar ahora en pasar un objeto local estructurado de gran tamaño como parámetro actual variable (para evitar su copia), cuya información sea procesada por el subprograma receptor sin efectos laterales sobre el objeto. De la misma manera puede ocurrir con la otra clase de parámetros afectada por las restricciones, por lo que concluimos que las situaciones que impiden la optimización de las activaciones finales no recursivas en lo relativo a los parámetros pueden ser más frecuentes que en las activaciones finales recursivas.

A pesar de ello, la restricción que va a producir un efecto más negativo en esta clase de activaciones es la restricción de anidamiento, es decir, el hecho

de que el subprograma receptor no puede ser local en el llamador. Esto es lógico en cierto modo por la propia filosofía de los lenguajes con estructura de bloques: si se definen subprogramas locales a otros subprogramas, son estos últimos los encargados de activar los locales, del mismo modo que es el programa principal el que suele activar los procedimientos y funciones globales, reflejando el árbol de activaciones la estructura de bloques del programa.

Esto no tiene que ser así siempre, y en efecto existen en casi todos los programas determinados bloques que se hacen globales porque han de ser visibles para muchos otros bloques, a cualquier nivel de anidamiento. Pero lo cierto es que existe bastante probabilidad de que las activaciones no recursivas sean de subprogramas locales, sobre todo en comparación con la muy baja frecuencia en que se van a dar los impedimentos de las restricciones de parámetros.

De todos modos, el interés principal de la optimación de las activaciones finales no recursivas no radica en cualquier clase de activaciones, sino primordialmente en aquellas que participan en un proceso recursivo no directo. Es decir, que no resulten optimadas activaciones que no impliquen procesos recursivos no es demasiado importante para nosotros, pues los problemas dinámicos espaciales de los programas ocurren solamente cuando se tiene que dar un comportamiento recursivo, ya sea expresado por recursividad directa o indirecta. Por ejemplo, Rohl [1984: 12ss] propone una técnica denominada "dos niveles" para evitar el trasiego continuo de parámetros que realmente son constantes a lo largo del proceso recursivo. Flegimos entre sus muchos ejemplos el siguiente.

```
function InArray(var a:arraytype; n:sizetype; k:keytype):Boolean;
    function I(j:sizetype):Boolean;
    begin
        if k = a[j].key then I := true
        else if j = n then I := false
        else I := I(j+1)
        end ( of function "I" )
    begin
        InArray := I(1)
    end ( of function "InArray" )
```

Programa 7.2.2 Una función recursiva en "dos niveles" (original de Rohl [1984: 19])

Claramente, la activación final recursiva de la función local *I* es optimable, por lo que el proceso recursivo se ejecutará en espacio constante si se aplica la optimación. Por contra, la activación final no recursiva de la misma función en el cuerpo de *InArray* no es optimable, al ser *I* local en aquella. Pero este caso es justamente el menos interesante para nosotros, porque no produce en sí mismo un proceso recursivo, sino que simplemente lo activa: los problemas de consumo de la pila de control están en la recursividad de la función *I*, no en el lanzamiento del proceso a través de la función base *InArray*.

Ya que estamos con el ejemplo, recalcamos el enorme interés de esta clase de soluciones para, como hemos dicho, dar la máxima eficiencia al proceso recursivo (suponiendo un comportamiento no optimado); lo bueno es que la optimación sale fortalecida si se utiliza esta metodología en dos niveles, pues cualquier disminución del número de parámetros mejorará notablemente el comportamiento optimado, al ser justamente ese uno de los factores más problemáticos del mismo.

La cuestión, por lo tanto, debería reformularse sobre la frecuencia con que los procesos recursivos son expresados mediante subprogramas indirectamente (mutuamente) recursivos pero anidados. Ejemplos típicos de cada caso pueden aducirse: el Programa 4.3.1 mostró un esquema bastante general de recursividad final mutua, concretamente para los procedimientos *par* e *impar*, en que los procedimientos involucrados tienen el mismo anidamiento, y por lo tanto, sus llamadas son optimables; por contra, Wirth [1981: 232-236] y Rohl [1984: 98-101] proponen la creación de árboles sintácticos para expresiones mediante procedimientos anidados correspondientes básicamente a los no-terminales de la gramática independiente del contexto, con lo que la optimación no sería posible (este ejemplo se estudia en la sección que aparece a continuación). En general, puede decirse que este asunto puede provocar algunas modificaciones en los hábitos de programación en PASCAL, justamente para evitar que la restricción de anidamiento tenga lugar, tema que estudiaremos dentro de un momento.

La última clase de restricciones de la optimación afecta a las activaciones finales no recursivas de funciones, y son las restricciones del valor de retorno: no es posible la optimación si no se puede determinar estáticamente en la sentencia de asignación del valor de retorno a la función llamadora que el valor de retorno de la función receptora está contenido entre los valores posibles del

tipo del valor de retorno de la primera. Aquí pueden darse los mismos argumentos que en el caso anterior: si se trata de activaciones que participan en un proceso recursivo no directo, es casi seguro que el tipo de las funciones involucradas sea el mismo, con lo que la restricción no se aplicará.

Concluimos, por tanto, que la restricción más fuerte de la optimación de activaciones finales en PASCAL, a salvo, claro está, de la propia restricción de posición final, es la de anidamiento para las activaciones no recursivas, mientras que es de esperar que las situaciones indicadas por las demás restricciones no se produzcan en los procesos recursivos, ya sean directos o indirectos, que son los de primordial interés a la hora de considerar la inclusión de la optimación en los compiladores del lenguaje.

7.3 RECURSIVIDAD FINAL Y CRITERIOS DE PROGRAMACIÓN

A la luz de esta conclusión, estudiamos ahora la manera en que puede facilitarse la optimación (con la consiguiente mejora del comportamiento dinámico de los programas) mediante técnicas de programación particulares. Desde luego, aquí es donde resulta de importancia fundamental la restricción de posición final, es decir, las técnicas básicas deben mostrar la manera en que se consiguen programas recursivos finales a partir de otros que en principio no lo son. Las investigaciones en la transformación de programas recursivos, recogidas en el capítulo 2, pueden aplicarse aquí. Algunas de estas técnicas son también, lógicamente, bien conocidas entre la comunidad del lenguaje SCHEME [Dybvig 1987: 71-74; Springer y Friedman 1989: 115-125; Friedman, Wand y Haynes 1992: 241-247].

No debemos olvidarnos de la otra perspectiva que introduce la optimación, y que realmente es uno de los aspectos que proporcionan más interés al presente trabajo. Recordemos que la recursividad (o más en general, el mecanismo de llamada a un subprograma, presente en todos los lenguajes) es lo único estrictamente necesario para expresar procesos repetitivos [Abelson, Sussman y Sussman 1985: 22-23]. Las construcciones iterativas de control que proporcionan los lenguajes de programación se ofrecen solo como "azúcar

sintáctico", y el problema es que los procesos recursivos tienen grandes problemas en el comportamiento dinámico, por lo que, tal como se dijo en el capítulo 2, es comprensible que se haya puesto el énfasis en las soluciones iterativas antes que en las recursivas.

Ahora bien, disponemos de una optimación de las activaciones finales recursivas, entre otras cosas, que implica un comportamiento en espacio constante, y una mejora temporal considerable, ya que solo se precisa una actualización de los argumentos y un salto. Realmente, esto es lo que realizan la mayor parte de las construcciones iterativas, por lo que podemos dar la vuelta al asunto: puede usarse una sintaxis recursiva final para expresar sencillamente gran parte de los procesos que habitualmente se programan con ayuda de esas construcciones iterativas. De hecho, la recursividad final no solo posibilita la realización de todos los bucles repetitivos habituales, sino que permite la construcción de estructuras repetitivas muy variadas, como señala Steele [1977a]: como ejemplo están todos los programas recursivos finales que no pueden resolverse por transformación de programas, como vimos en la sección 2.5.

El comportamiento temporal no tiene que ser necesariamente distinto al iterativo, y en muchas ocasiones, la claridad de la solución recursiva puede ser bastante mayor. En estas condiciones, por lo tanto, no podemos estar de acuerdo, sin por ello menoscabar su gran valía, con Wirth cuando afirma [1986a: 138-140] que la recursividad no debe usarse cuando existe una solución iterativa obvia.

Como simple muestra de ello, elegimos el conocido ejemplo de los números de Fibonacci, ya examinado en el apartado 2.2.2. La implementación directa de la definición recursiva de la función es, ciertamente, ineficiente.

```
function Fibonacci (n: integer): integer; ( n >= 0 )
begin
  if n <= 1 then Fibonacci := n
  else Fibonacci := Fibonacci(n-1) + Fibonacci(n-2)
end; ( of function "Fibonacci" )
```

Programa 7.3.1 Implementación recursiva ineficiente de la función que calcula los números de Fibonacci

Una solución recursiva lineal puede darse muy simplemente, usando la estructura en dos niveles mencionada antes y dos argumentos adicionales en la

función interna para acumular los valores previos de la función [Rohl 1984: 88; Dybvig 1987: 73]. Casualmente, esta solución es recursiva final.

```
function Fibonacci (n: integer): integer; { n >= 0 }

function FibAcc (i, acc1, acc2: integer): integer;
{ acc1 = Fibonacci(n-1); acc2 = Fibonacci(n-2) }
begin
  if i = 1 then FibAcc := acc1
  else FibAcc := FibAcc(i-1, acc1+acc2, acc1)
end { of function "FibAcc" }

begin
  if n = 0 then Fibonacci := 0
  else Fibonacci := FibAcc(n, 1, 0)
end; { of function "Fibonacci" }
```

Programa 7.3.2 Implementación recursiva eficiente de la función que calcula los números de Fibonacci

Compárese con la versión sintácticamente iterativa de Wirth [1986a: 139].

```
function Fibonacci (n: integer): integer; { n >= 0 }
var i, acc1, acc2, temp: integer;
begin
  if n <= 1 then Fibonacci := n
  else
    begin
      i := 1; acc1 := 1; acc2 := 0;
      while i < n do
        begin
          temp := acc1;
          acc1 := acc1+acc2;
          acc2 := temp;
          i := i+1
        end;
      Fibonacci := acc1
    end
end; { of function "Fibonacci" }
```

Programa 7.3.3 Implementación iterativa eficiente de la función que calcula los números de Fibonacci (adaptada del original de Wirth [1986a: 139])

Primero, el comportamiento temporal del Programa 7.3.2 (suponiendo que se practica la optimización) no tiene nada que envidiar al Programa 7.3.3. El cuerpo de la sentencia `while` de este último equivale a la actualización de los parámetros en la activación recursiva final (obsérvese la necesidad de una variable temporal, paralelo de la necesidad de evaluar totalmente los parámetros actuales antes de asignarlos a los formales expresada por la Condición 3.2.9). El salto implícito en la sentencia `while` es el paralelo del salto al comienzo del código de la función recursiva final. Y eso es todo: no hay

diferencia apreciable entre el comportamiento recursivo final y el comportamiento iterativo, tanto espacial como temporalmente.

Pero el Programa 7.3.2 representa, a nuestro entender, una solución más inmediata del problema que el Programa 7.3.3, en el esfuerzo de transformación de un proceso recursivo ineficiente en un proceso eficiente. Por supuesto, la transformación de la recursividad final en iteración no es difícil en algunos casos (aunque hay que preocuparse, por ejemplo, de encontrar un orden de actualización de parámetros que sea eficiente). Nuestra postura es que ni siquiera ese esfuerzo tiene necesariamente que ser realizado, manteniendo las bondades del comportamiento dinámico de los programas. Finalmente, creemos que no se puede dudar que el Programa 7.3.2 es, con mucho, más "evidente", expresivo y legible que la alternativa iterativa, como ocurre en general con las soluciones recursivas, los que al final han de ser criterios primordiales de la programación.

Nos preocuparemos ahora de la restricción de anidamiento, en lo relativo a las activaciones finales no recursivas que expresan un proceso recursivo no directo. Como ya hemos mencionado previamente, el proceso completo se ejecutará sin crecimiento de la pila de control si todas las activaciones finales son optimables, y para ello ninguna ha de estar sujeta a la restricción de anidamiento. Por lo tanto, no es posible que ningún subprograma que interviene en el proceso recursivo en posición final esté anidado dentro de otro, lo que es equivalente a decir que todos los subprogramas deben tener el mismo nivel de anidamiento.

Esta condición puede ser natural en algunas ocasiones, como en la recursividad mutua de las funciones par e impar del Programa 4.3.1, y requiere el uso de la directiva *forward* del lenguaje. También ocurre que se encuentran casos en que esa condición no se cumple, y la optimación completa no es posible. Un primer ejemplo de esto es, ya lo hemos dicho, las implementaciones de Wirth [1981] y Rohl [1984] (en principio independientes, pero sorprendentemente similares) de la creación de árboles sintácticos para expresiones. Es muy interesante detenerse en este ejemplo concreto porque pensamos que enseña bastante sobre el uso de la recursividad final (o mejor dicho, sobre su no utilización debido a la falta habitual de la optimación en el compilador).

Tomamos la implementación de Rohl [1984:98-101], por ser más abstracta y simple; la de Wirth [1981: 232-236] tiene básicamente la misma estructura, solo que está enmascarada por las acciones de compilación, pues aparece como parte del compilador del lenguaje PASCAL-S, subconjunto de PASCAL. Mostramos entonces esa estructura básica en el siguiente programa:

```

procedure Expression(var t:treeptr);

  procedure Term(var t:treeptr);

    procedure Factor(var t:treeptr);
    begin
      if input^ in _ then crea-nodo-para-input
      else if input^ = '(' then
        begin
          get(input);
          Expression(t);
          if input^ = ')' then get(input)
          else error (closing parenthesis missing)
        end
      else error (not a valid factor)
    end; { of procedure "Factor" }

    begin { of procedure "Term" }
      Factor(t);
      while input^ in ['*', '/'] do
        begin
          crea-nodo-para-input;
          Factor(t^.right)
        end
      end; { of procedure "Term" }

    begin { of procedure "Expression" }
      Term(t);
      while input^ in ['+', '-'] do
        begin
          crea-nodo-para-input;
          Term(t^.right)
        end
      end; { of procedure "Expression" }
  end;

```

Programa 7.3.4 Un procedimiento para construir árboles de expresiones (con ligeras variaciones a partir del original de Rohl [1984: 99-100])

Observemos que las activaciones de los procedimientos Factor y Term no son optimables solo por ser locales en el bloque en que se llaman, sino realmente porque no están en posición final: las que nos interesan aparecen dentro de una sentencia while, con lo que se pierde la característica de posición final. Tenemos que recordar que estos procedimientos forman parte del compilador de un lenguaje cuya sintaxis para las expresiones sea similar a la de PASCAL; concretamente Wirth las utiliza para el análisis del programa fuente, basado en la técnica descendente recursiva que ya conocemos, y la subsiguiente

generación de código. Si echamos un vistazo a la gramática independiente del contexto que refleja la sintaxis de las expresiones, en la que supuestamente han de basarse estos procedimientos, se entenderá la razón de usar los ciclos `while`.

Como se sabe, la gramática independiente del contexto no ambigua que se utiliza habitualmente para las expresiones es [Díaz Labrador 1994: 192-194]:

$$\begin{aligned}\langle \text{Expression} \rangle &\rightarrow \langle \text{Expression} \rangle + \langle \text{Term} \rangle \\ &\mid \langle \text{Expression} \rangle - \langle \text{Term} \rangle \\ &\mid \langle \text{Term} \rangle\end{aligned}$$

$$\begin{aligned}\langle \text{Term} \rangle &\rightarrow \langle \text{Term} \rangle * \langle \text{Factor} \rangle \\ &\mid \langle \text{Term} \rangle / \langle \text{Factor} \rangle \\ &\mid \langle \text{Factor} \rangle\end{aligned}$$

$$\langle \text{Factor} \rangle \rightarrow n \mid (\langle \text{Expression} \rangle)$$

Esta gramática no es válida para el análisis descendente debido a que es recursiva por la izquierda; eliminar esta característica introduce dos nuevas variables:

$$\langle \text{Expression} \rangle \rightarrow \langle \text{Term} \rangle \langle \text{RExpression} \rangle$$

$$\begin{aligned}\langle \text{RExpression} \rangle &\rightarrow + \langle \text{Term} \rangle \langle \text{RExpression} \rangle \\ &\mid - \langle \text{Term} \rangle \langle \text{RExpression} \rangle \\ &\mid \lambda\end{aligned}$$

$$\langle \text{Term} \rangle \rightarrow \langle \text{Factor} \rangle \langle \text{RTerm} \rangle$$

$$\begin{aligned}\langle \text{RTerm} \rangle &\rightarrow * \langle \text{Factor} \rangle \langle \text{RTerm} \rangle \\ &\mid / \langle \text{Factor} \rangle \langle \text{RTerm} \rangle \\ &\mid \lambda\end{aligned}$$

$$\langle \text{Factor} \rangle \rightarrow n \mid (\langle \text{Expression} \rangle)$$

Los ciclos `while` del Programa 7.3.4 no hacen más que reconocer que el tipo de cadenas generadas por las nuevas variables introducidas para evitar la recursividad por la izquierda es:

$$\langle \text{RExpression} \rangle \stackrel{*}{\Rightarrow} \{ (c \langle \text{Term} \rangle)^n \mid n \geq 0, c = + \vee c = - \}$$

$$\langle \text{RTerm} \rangle \stackrel{*}{\Rightarrow} \{ (c \langle \text{Factor} \rangle)^n \mid n \geq 0, c = * \vee c = / \}$$

Desde luego, esta es una optimación muy conveniente, pero está claro que tiene un precio, y es el de enmascarar la estructura de la gramática subyacente, con todas las consecuencias que esto puede tener (pensando, por ejemplo, en una posible modificación para la inclusión de más operadores).

Si se quiere construir el traductor basado exactamente en esta gramática no recursiva por la izquierda, se obtendría el siguiente programa.

```

procedure Expression(var t:treeptr);

  procedure Term(var t:treeptr);

    procedure Factor(var t:treeptr);
    begin
      if input^ in ... then crea-nodo-para-input
      else if input^ = '(' then
        begin
          get(input);
          Expression(t);
          if input^ = ')' then get(input)
          else error (closing parenthesis missing)
        end
      else error (not a valid factor)
    end; ( of procedure "Factor" )

    procedure RTerm(var t:treeptr);
    begin
      if input^ in ['*', '/'] then
        begin
          crea-nodo-para-input;
          Factor(t^.right);
          RTerm(t) (t)
        end
      end; ( of procedure "RTerm" )

    begin ( of procedure "Term" )
      Factor(t);
      RTerm(t) (t)
    end; ( of procedure "Term" )

    procedure RExpression(var t:treeptr);
    begin
      if input^ in ['+', '-'] then
        begin
          crea-nodo-para-input;
          Term(t^.right);
          RExpression(t) (t)
        end
      end; ( of procedure "RExpression" )

    begin ( of procedure "Expression" )
      Term(t);
      RExpression(t) (t)
    end; ( of procedure "Expression" )

```

Programa 7.3.5 Un procedimiento para construir árboles de expresiones basado exactamente en la gramática subyacente

La comparación entre el Programa 7.3.4 y el Programa 7.3.5 es clara: los ciclos iterativos `while` están representando un proceso recursivo final, justamente el que corresponde a los no terminales $\langle RExpression \rangle$ y $\langle RTerm \rangle$ de la gramática exacta. Las activaciones recursivas de los procedimientos correspondientes, señaladas con $\{+\}$ en el Programa 7.3.5, son perfectamente optimizables, y al ser recursivas, con la mejor razón de optimización (teniendo en cuenta además que no es precisa ninguna actualización del parámetro), por lo que se ejecutarán en todos los aspectos (espacial y temporalmente) de forma similar al ciclo `while` al que sustituyen.

Concretamente, el consumo de pila de control en el Programa 7.3.5 solo incluirá registros de activación adicionales para la activación de los nuevos procedimientos, es decir, un registro de activación de $\langle RExpression \rangle$ por cada activación de $\langle Expression \rangle$ y un registro de activación de $\langle RTerm \rangle$ por cada activación de $\langle Term \rangle$. Y temporalmente, las únicas instrucciones adicionales serán las de la primera activación de cada proceso recursivo directo asociado a los procedimientos $\langle RExpression \rangle$ y $\langle RTerm \rangle$.

Consideramos que el Programa 7.3.5 refleja mucho más claramente el seguimiento de la metodología habitual para la construcción de un analizador sintáctico que el Programa 7.3.4. Evidentemente, si el compilador de PASCAL no aplica la optimización, la solución del Programa 7.3.5 tiene un comportamiento dinámico mucho peor, por lo que se comprende que Wirth y Rohl hayan propuesto la solución que se ha mostrado. Sin embargo, aplicando la optimización, hemos demostrado que el programa recursivo se puede comportar de modo muy parecido al no recursivo, siendo a nuestro entender mucho más expresivo y legible.

Ahora bien, existen en el Programa 7.3.5 activaciones finales no recursivas que no pueden optimarse, debido a que se dan sobre procedimientos locales en el bloque llamador; están marcadas con $\{+\}$. Esto podría solucionarse si todos los procedimientos se definiesen en el mismo anidamiento. Para mantener la característica de ocultación que impone el bloque `Expression`, creamos un bloque genérico (similar a la solución en dos niveles del Programa 7.2.2) en el que se definen internamente los procedimientos correspondientes. Se observa que para ello es preciso una declaración con la directiva `forward`, para el procedimiento `Expression` concretamente.

```

procedure ExpressionProcessing(var t:treeptr);

  procedure Expression(var t:treeptr); forward;

  procedure Factor(var t:treeptr);
  begin
    if input^ in ... then crea-nodo-para-input
    else if input^ = '(' then
      begin
        get(input);
        Expression(t);
        if input^ = ')' then get(input)
        else error (closing parenthesis missing)
      end
    else error (not a valid factor)
  end; ( of procedure "Factor" )

  procedure RTerm(var t:treeptr);
  begin
    if input^ in ['*', '/'] then
      begin
        crea-nodo-para-input;
        Factor(t^.right);
        RTerm(t)
      end
    end; ( of procedure "RTerm" )

  procedure Term(var t:treeptr);
  begin ( of procedure "Term" )
    Factor(t);
    RTerm(t)
  end; ( of procedure "Term" )

  procedure RExpression(var t:treeptr);
  begin
    if input^ in ['+', '-'] then
      begin
        crea-nodo-para-input;
        Term(t^.right);
        RExpression(t)
      end
    end; ( of procedure "RExpression" )

  procedure Expression;
  begin
    Term(t);
    RExpression(t)
  end;

begin ( of procedure "ExpressionProcessing" )
  Expression(t)
end ( of procedure "ExpressionProcessing" )

```

Programa 7.3.6 Otro procedimiento para construir árboles de expresiones basado exactamente en la gramática subyacente, con procedimientos no anidados

Ahora, las activaciones finales no recursivas, salvo la principal, son optimables, y con una buena razón temporal de optimación, pues se realizan sobre procedimientos del mismo nivel de anidamiento; por ello, el comportamiento

será aún más cercano al del programa que utiliza las estructuras iterativas de control.

Sin embargo, la conveniencia de esta solución debe analizarse a la luz de los criterios habituales de programación en PASCAL y lenguajes similares. Concretamente, primero hay que tener en cuenta que la implementación de Wirth está complicada con la existencia de parámetros adicionales para los procedimientos, y datos locales de los mismos, a través de los cuales se establece la compilación; en realidad, Wirth no construye el árbol de la expresión, sino que se va emitiendo el código conforme la expresión se va procesando. Se puede comprobar, sin embargo, que ello no solo no impide la estructura del Programa 7.3.6, sino que las modificaciones a ese respecto son mínimas. Además, las activaciones directamente recursivas no precisan absolutamente ninguna actualización de los dos parámetros de que constan, con lo que el comportamiento temporal del programa recursivo será realmente bueno.

En segundo lugar, la estructura de bloques puede ser necesaria justamente para hacer visibles ciertos datos como no locales a los subprogramas anidados. Incidentalmente, este es a nuestro entender, uno de los aspectos más cuestionables de la estructura de bloques, tal como se define en lenguajes como PASCAL y otros similares: no se puede diferenciar, ante una declaración de un dato en un bloque, si se desea que ese dato sea accesible como no local por los subprogramas anidados en el bloque, o si por el contrario, se trata de un objeto cuya necesidad está estrictamente restringida al procesamiento expresado por la parte de sentencias de ese bloque. Como consecuencia directa de esto, puede ser difícil saber si un subprograma anidado en otro puede "sacarse" de ahí y colocarse al mismo nivel que el que lo contenía. En el caso de la solución de Wirth, por ejemplo, resulta que todos los datos locales de los procedimientos anidados utilizados no se acceden nunca como no locales por los procedimientos interiores, así que todos ellos pueden colocarse con la estructura del Programa 7.3.6.

Visto, por lo tanto, que la alternativa recursiva puede tener un comportamiento dinámico similar a la no recursiva, y que la presencia de parámetros y datos locales no se ve modificada (en este caso concreto), la única diferencia se establece en la utilización del anidamiento. ¿Es conveniente utilizar el anidamiento? ¿Lo es tanto como para desaconsejar la alternativa sin

anidamiento? La postura al respecto de esta cuestión ha de ser la misma que ante otras cuestiones por el estilo: nunca deben "forzarse", creemos, los criterios metodológicos de programación en el lenguaje para obtener una optimación, como la propuesta u otras similares. Debe, en todo caso, mantenerse un equilibrio entre las características de legibilidad, claridad y facilidad de modificación de los programas y la posibilidad de un comportamiento optimado. Lo bueno de la optimación que planteamos es que, en la mayor parte de los casos, mejora justamente esas características deseables de los programas, al permitir soluciones recursivas que tienen, normalmente, mayor claridad que las soluciones iterativas alternativas. Sin embargo, en otros casos, como estamos viendo, puede ser necesaria una separación, relativamente fuerte en algún aspecto, de la estructura esperada, cuyo impacto habrá que analizar a la luz del compromiso que proponemos.

Este problema realmente puede aparecer no solo ante la posibilidad de una optimación, sea cual sea. Por ejemplo, la propia cuestión de la estructura de bloques puede ser cuestionable desde cierto punto de vista, lo mismo que los sistemas de tipos fuertes y tantas características de los lenguajes de programación. Así, si todas las variables son globales y todos los subprogramas también, el comportamiento dinámico se verá mejorado: no hacen falta los enlaces estáticos, el acceso a los datos es prácticamente directo, las llamadas son más sencillas, etc. El lenguaje C, que cumple las anteriores características, se considera habitualmente como un lenguaje que produce uno de los mejores códigos objeto entre los lenguajes de alto nivel. Ahora bien, ¿es realmente C un lenguaje de alto nivel? O lo que es lo mismo, ¿es correcto utilizar C para el desarrollo de grandes sistemas? Evidentemente, las características de abstracción que proporcionan lenguajes como PASCAL, MODULA o ADA tienen un precio, pero pensamos que ese precio está compensado, y con mucho además, por la facilidad que incorporan a la programación, entendiendo este concepto no solo como la preparación de un conjunto de instrucciones que "parecen" ejecutar una cierta tarea.

Por lo tanto, lo que es absolutamente incorrecto es recomendar la escritura de subprogramas globales por el hecho de que la optimación de las activaciones finales no recursivas no puede producirse si se aplican a un subprograma local en el llamador, y lo mismo habría que decir sobre el resto de las restricciones encontradas para la optimación: nunca deberían utilizarse para

recomendar soluciones de programación que vayan en contra de los criterios metodológicos habituales. Otra cosa es que, en cada caso concreto, se analice el impacto de las modificaciones que introduce la posibilidad de la optimización (suponiendo que esta suponga una mejora en algún otro aspecto): si se encuentra que no son demasiado fuertes, el programador puede practicar tales modificaciones, pero si se presume que va a salir gravemente perjudicada alguna de las características deseables de los programas, la posibilidad de optimización debe obviarse.

Así por ejemplo, el Programa 7.3.6 expresa un compromiso intermedio entre las características de ocultación de información propias de la estructura de bloques y la posibilidad de practicar una optimización de las activaciones finales no recursivas al disponer de un grupo de procedimientos con el mismo nivel de anidamiento. Y de todos modos, si esto no se considera suficiente, el Programa 7.3.5 aún admite la optimización de los procesos recursivos directos, con lo que sería válido como solución alternativa que saca partido de la recursividad.

También hay que tener en cuenta en este caso concreto el problema más general en el que se engloba el que hemos considerado: la programación de un analizador sintáctico descendente mediante procedimientos asociados a cada uno de los no terminales de la gramática. En el caso más general, debido a la interdependencia que puede existir entre los no terminales, quizá sea imposible dar una estructura jerárquica o anidada para los mismos. Entonces, la única solución válida sería la del Programa 7.3.7.

Normalmente, sin embargo, para bastantes gramáticas sí es posible obtener un orden jerárquico entre sus no terminales. Por ejemplo, para los lenguajes de programación esto es evidente, pues la propia estructura sintáctica de los programas es jerárquica, y por lo tanto, usar el anidamiento podría enfatizar ese hecho. Sin embargo, si pensamos en una herramienta similar a YACC [Johnson 1975], pero que genere un analizador sintáctico descendente a partir de una gramática independiente del contexto, basado en un programa PASCAL como los examinados, la estructura del Programa 7.3.7 puede ser la única posible a adoptar, pues las maneras habituales de escribir una gramática no incluyen el anidamiento o jerarquía entre sus no terminales. Obsérvese que establecer un grafo de dependencias entre ellos puede no resolver

correctamente la cuestión, pues el orden jerárquico no tiene necesariamente que estar determinado por tal relación.

```

procedure Parser (...);

    procedure <nonterminal1> (...); forward;
    procedure <nonterminal2> (...); forward;
    ...
    procedure <nonterminalk> (...); forward;

    procedure <nonterminal1>;
    ( cuerpo del procedimiento )
    ...
    procedure <nonterminalk>;
    ( cuerpo del procedimiento )

begin { del procedimiento "Parser" }
    <nonterminal1> (...);
end { del procedimiento "Parser" }

```

Programa 7.3.7 Estructura necesaria para los procedimientos de un analizador sintáctico descendente si la interdependencia entre las variables no admite el anidamiento

En definitiva, insistimos en que han de prevalecer los criterios habituales de la metodología de programación en el lenguaje antes que posibles soluciones que puedan facilitar la optimización pero yendo en contra de tales criterios. Lo que ocurrirá a menudo, sin embargo, es que facilitar la optimización no solo no irá en contra, sino que permitirá mejorar el cumplimiento de esa metodología, al ser normalmente las soluciones recursivas más expresivas que las iterativas.

Otra de las conclusiones que puede, incorrectamente, extraerse del análisis realizado a lo largo del capítulo es la conveniencia de utilizar procedimientos en lugar de funciones para la programación. Los análisis de los costes dinámicos temporales de las operaciones que se llevan a cabo en la activación de cada uno de los subprogramas, realizada en la sección 6.2, parecían conducir a la conclusión de que los procedimientos son un poco menos costosos, y ello en cualquiera de los modelos dinámicos que se consideren. Sin embargo, anticipábamos que la conclusión no podía mantenerse solamente a la luz de las cifras obtenidas, pues estas no consideran la influencia de parámetros formales o datos locales, sino que debería realizarse comparando procedimientos y funciones destinados al mismo cálculo.

Desde luego, está claro que las funciones en general, tal como se definen en el lenguaje PASCAL, y sin considerar casos particulares, exigirán mayor coste

temporal, debido a la necesidad, primero, de realizar la indefinición del valor de retorno al entrar en la activación, y segundo, de verificar su definición al salir. Está claro que esta norma es una imposición metodológica del lenguaje, que no solo no discutimos sino que alabamos. Alguien puede, como ya avanzamos, simular el concepto de función mediante un procedimiento, incorporando un parámetro formal por referencia que juegue el papel del valor de retorno. Este es el método que usa, concretamente, Rohl [1984: 171-172] para eliminar la recursividad por transformación de programas (pues no puede aplicar esa técnica a funciones). Tomemos el mismo ejemplo que él: la función de Ackermann; las dos alternativas de programación se muestran en el siguiente programa.

```

program Ackermann (input, output);

var
  resultado: integer;

function Ackermann (m, n: integer): integer;
begin
  if m = 0 then Ackermann := succ(n)
  else if n = 0 then Ackermann := Ackermann (pred(m), 1)
  else Ackermann := Ackermann (pred(m), Ackermann (m, pred(n)))
end;

procedure PAckermann (m, n: integer; var Ack: integer);
  ( el parametro Ack equivale al valor de retorno )
begin
  if m = 0 then Ack := succ(n)
  else if n = 0 then PAckermann (pred(m), 1, Ack)
  else
    begin
      PAckermann (m, pred(n), Ack);
      PAckermann (pred(m), Ack, Ack)
    end
  end;

begin
  ( activación de la función (en una expresión) )
  WriteLn ('El resultado es : ', Ackermann (x, y));

  ( activación del procedimiento (no puede estar en una expresión) )
  PAckermann (resultado, x, y);
  WriteLn ('El resultado es : ', resultado);
end.

```

Programa 7.3.8 La función de Ackermann implementada en PASCAL con una función y con un procedimiento (solo con ligeras variaciones a partir del original de Rohl [1984: 171])

Esta sería la clase de funciones y procedimientos que en todo caso podrían compararse para saber la diferencia en coste temporal. Pues bien, en el modelo dinámico estándar, la evaluación y asignación del parámetro adicional del

procedimiento hace que el coste de llamada iguale o supere el de la función; en cuanto al retorno, los costes son los que se obtuvieron en la sección 6.2, por lo que sigue pesando la verificación del valor de retorno en contra de la función; en conjunto, resulta que no va a haber diferencias entre una y otra versión, o van a ser diferencias inapreciables (téngase en cuenta que la razón entre los costes de las activaciones de funciones y procedimientos, sin considerar parámetros o datos locales, es de 1'5 como mucho, según lo calculado en la sección citada). En el modelo dinámico optimado, sin embargo, hay un aspecto notable a tener en cuenta: el parámetro formal por referencia del procedimiento es optimable, por lo que no es precisa su actualización, y con ello, resulta que el procedimiento sigue manteniendo un mejor comportamiento respecto a la función, a pesar de tener un parámetro adicional. Pero tampoco esta diferencia va a ser muy drástica, en lo referente a las secuencias de llamada y retorno, porque en el modelo dinámico optimado la relación entre los costes de las llamadas recursivas finales de funciones y procedimientos solo sube hasta 2 (o sea, las llamadas resultan muy mejoradas *respecto* al modelo estándar, pero no tienen mucha diferencia entre sí).

Hay otra cuestión a valorar: no tiene el mismo coste el acceso a un dato local (como lo es el valor de retorno de una función, al fin y al cabo), que a un dato no local (como lo es en realidad un parámetro formal variable); en efecto, para la sentencia de la función:

Ackermann := succ(n)

solo hay que usar un acceso directo con desplazamiento a partir del registro BasePila, mientras que para la sentencia del procedimiento:

Ack := succ(n)

se precisa un acceso indirecto con desplazamiento. Desde luego, esta última instrucción suele ser más costosa, al implicar un direccionamiento por memoria, y en el caso concreto de la función de Ackermann se realiza un número significativo de veces (recordemos que, a pesar de ser el caso base de la recursividad, la función de Ackermann no es lineal, ni doble, ni triple, etc., por lo que ese caso se utiliza, como decimos en un gran número de ocasiones para llegar al valor final). Fruto de ello, el código del procedimiento puede ser más lento que el de la función, con lo que se equilibrarían las posibles ventajas que pudiera haber a favor de la solución procedimental en las llamadas y retornos.

Concluimos que las dos clases de procesos se van a mantener prácticamente similares en coste temporal, ya se use una función, ya se use el procedimiento que la simula. No puede, por tanto, extraerse de la Figura 6.2.4 la conclusión de que es mejor, solo por consideraciones de coste temporal, utilizar procedimientos y no funciones en el lenguaje PASCAL, y ello en cualquier modelo dinámico que se considere.

Y, evidentemente, el criterio que debe prevalecer a la hora de elegir procedimientos o funciones para la resolución de problemas es, otra vez, el que dicta la metodología de programación en el lenguaje PASCAL: las funciones han de usarse para expresar cálculos libres de efectos laterales y cuyo objetivo es la obtención de un valor. Ese debe ser *el* criterio, y entiéndasenos bien, lo debería ser incluso aunque se hubiera llegado a la conclusión de que el coste de las activaciones de las funciones es grande comparado con el de los procedimientos equivalentes. Lo que hemos querido demostrar es que ni siquiera eso puede aconsejar el uso de procedimientos en vez de funciones.

Está claro que con la simulación de una función mediante un procedimiento se pueden llevar a cabo los mismos cálculos, pero hay algo que no se puede "simular", como hemos reflejado en el Programa 7.3.8 (esas son justamente las "ligeras variaciones" a las que aludíamos en el epígrafe): la posibilidad de usar la función (su valor de retorno) como operando de una expresión. Para usar el resultado del cálculo con la versión del procedimiento, obligatoriamente debe utilizarse una variable para que sea el parámetro actual en el que quedará acumulado el resultado, y entonces realizar la activación del procedimiento en una sentencia previa a la que usará el resultado. ¿Dónde queda aquí reflejada la claridad del programa, la legibilidad, la naturalidad en su expresión? Desde luego, se pierden en gran medida todas estas propiedades. El código de la función es sencillo, claro, una simple transcripción directa de la expresión matemática de la función. El código del procedimiento hay que mirarlo un par de veces al menos para darse cuenta de lo que hace. Pensamos que estas sí son notables diferencias, y lo que se pierde al utilizar procedimientos para simular funciones no está justificado por una pretendida mejora temporal (que en todo caso, será inapreciable).

Como comentario final, queremos generalizar las conclusiones precedentes, e insistir en la valoración de la recursividad final desde otra perspectiva, que se aplica en general a todos los programas recursivos y a todos

los intentos de conversión en programas iterativos. Las discusiones anteriores han mostrado que cuando se practica una transformación para mejorar la eficiencia de un programa (sin abandonar el lenguaje fuente), hay cierta "información" que "se pierde". Nos referiremos a la claridad o legibilidad de los programas. Parece que las soluciones recursivas para muchos problemas son, o tienden a ser, como hemos dicho en varias ocasiones, "evidentes" en el sentido de que puede extraerse su objetivo a partir de un simple vistazo del programa. Tal propiedad va perdiéndose conforme se realizan sucesivas transformaciones del programa fuente.

Si el programa que se consigue no tiene necesidades de mantenimiento, no hay problema, pero si es un programa inmerso en un gran sistema, que hipotéticamente deba ser modificado, reevaluado o reconstruido, no disponer del proceso completo de transformación puede ser tanto como no disponer del programa, por ejemplo, si es otro programador el que ha de acometer estas tareas. La misma idea es expuesta por Hoogerwoord:

«As a matter of fact, the reason for writing this note was the observation that this simple example —once more— illustrates that there is no point in trying to understand a program without knowledge about *how* it was constructed. Phrased differently, a program without proper documentation is virtually worthless» [Hoogerwoord 1990b: 1-2]

Pensamos, por lo tanto, que la eliminación de la recursividad para conseguir eficiencia en los programas es una tarea necesaria, pero debe hacerse de la manera más *transparente* posible para el programador. Esto es, las herramientas de programación (sean lenguajes o sistemas más complejos) han de proveerse con los mecanismos precisos para la conversión automática de programas posiblemente ineficientes en programas de mejor comportamiento. Entendemos que la optimación de la recursividad final a través del propio compilador es el primer paso que hay que dar en este sentido, y es el primer ejemplo paradigmático de lo que hemos llamado "transparencia". En general, se trata de que exista la menor diferencia entre el programa más sencillo de explicar como solución de un problema y el programa *fuentes* que el programador acaba escribiendo, que siendo equivalente, es más eficiente que el original. Los esfuerzos que se hagan para conseguir que programas recursivos no finales se comporten mejor deberían tener esta idea presente, a nuestro entender. Y en ese búsqueda es dónde ha de ponerse el énfasis ahora.

8. CONCLUSIONES

Las conclusiones del trabajo desarrollado se establecen desde distintas perspectivas. En primer lugar, se puede recoger la proposición que se planteaba como objetivo general en la introducción de la memoria:

"Gran parte de los programas recursivos finales en PASCAL pueden ejecutarse de manera eficiente como resultado de técnicas de compilación alternativas"

y completar los argumentos que allí se avanzaban.

- Se ha comprobado que técnicas simples de compilación, entendidas como una optimización (desarrolladas en las secciones 3.2, 4.2, 5.2 y 5.3), permiten alcanzar el comportamiento iterativo deseado. Las técnicas son simples porque tienen pocas exigencias de análisis estático, dada incluso la estructura habitual de los compiladores del lenguaje, como se ha visto en la sección 6.3. Estas técnicas no realizan una búsqueda específica de la recursividad, sino que se aplican como caso general a cualquier llamada final a un subprograma, es decir, puede afirmarse que, en cierto modo, el compilador no precisa "conocer" que cierta llamada está involucrada en un proceso recursivo.
- El comportamiento de los programas con recursividad final optimados es eficiente porque las necesidades de espacio no crecen durante las llamadas involucradas, y el coste temporal de las mismas disminuye respecto a la implementación clásica, pudiendo asimilarse al coste del programa iterativo correspondiente en muchos casos. En este último sentido, los análisis de la sección 6.2 han reafirmado la conveniencia de distinguir tres clases de llamadas finales, lo que permite conseguir una alta mejora temporal en esos casos particulares.
- La técnica puede aplicarse tanto a la recursividad final de procedimientos como de funciones. Este último caso supone una novedad importante a reseñar, porque hasta ahora, la conversión tradicional basada en la transformación de programas no era capaz de manejar este caso en

PASCAL. Para ello, se ha definido exactamente lo que se entiende por "activación final de función" en este lenguaje (apartado 5.2.1). Por otro lado, la técnica se aplica igualmente a la recursividad final no directa en cualquier grado, aspecto que tampoco se considera en modo alguno en las técnicas transformacionales conocidas.

- Se ha constatado la existencia de restricciones para la optimación de las llamadas finales. Las causas de estas restricciones son de dos clases: en la sección 3.3 se argumentó que la mayoría surge por las características propias de elementos del lenguaje (es decir, son en cierto modo universales, independientes de la técnica particular utilizada para la optimación, como se ha comprobado al observar que tales restricciones surgen igualmente utilizando técnicas alternativas), mientras que en la sección 4.3 se demostró que existe una restricción que se produce directamente como efecto de la estrategia utilizada para la solución del problema, y achacable en último término a la implementación del lenguaje (es decir, la restricción resultará difícil de superar si no se quiere perder la eficiencia tradicionalmente asociada a tal implementación).
- Finalmente, el posible impacto de las restricciones en la optimación (estudiado en el capítulo 7) puede clasificarse exactamente como en el punto anterior. Esto es, las restricciones inherentes al lenguaje representan situaciones que se producen esporádicamente en los programas. Sin embargo, la otra restricción se da con mayor frecuencia en el caso general, aunque se ha argumentado que esta frecuencia es menor en los programas que sean estrictamente recursivos: la restricción afecta en todo caso a algunos programas en los que exista recursividad no directa (mutua), y en la mayoría de ellos, pequeñas modificaciones en los hábitos de programación posibilitarían la optimación, no yendo en contra de las recomendaciones genéricas de uso del lenguaje.

En cuanto a los objetivos más específicos, podrían reseñarse las siguientes aportaciones:

- Se ha comparado la técnica usada en este trabajo con otras estrategias anteriores conocidas para el lenguaje SCHEME, presentadas en la sección 2.4. La que es más adecuada, *a priori*, para PASCAL tiene la ventaja de hacer desaparecer la restricción señalada como propia de la técnica desarrollada en este trabajo. Sin embargo, está basada en una filosofía alejada de la

implementación establecida como clásica para PASCAL, además de exigir un mayor coste estático y dinámico, inconvenientes que la hacen poco atractiva para los diseñadores de compiladores de PASCAL. La estrategia presentada, por contra, puede integrarse con poco esfuerzo en casi cualquier compilador existente.

- La comparación con el lenguaje SCHEME ha permitido descubrir (en la sección 3.3) que una de las restricciones de la optimación inherentes al propio lenguaje PASCAL también existe realmente en SCHEME, si se trata de conseguir un comportamiento iterativo puro. Este extremo no se encuentra reflejado en la definición de este lenguaje, y es opinión del doctorando que debería quedar recogido de alguna manera. La distinción planteada por algunos autores entre "comportamiento iterativo de control" y "comportamiento iterativo puro" es un buen esquema a tal efecto.
- Se ha estudiado en la sección 6.4 la influencia de distintos aspectos de la implementación del lenguaje en la optimación, y se ha concluido que pueden plantear más o menos dificultades, pero nunca impedirla. Son de reseñar dos aspectos concretos: la posibilidad de usar indistintamente las estrategias conocidas de acceso dinámico a los datos no locales (cadena estática o *display*), y la necesidad de elegir una estrategia cuidadosa de paso de parámetros, pues se trata del punto más débil de la técnica desarrollada. De cara a la integración de la optimación en un compilador, es conveniente tener presentes los aspectos que la favorecen a la hora de tomar las decisiones de diseño pertinentes.

La parte aplicada del trabajo doctoral ha consistido en la implementación de la técnica de optimación desarrollada en un compilador de un subconjunto suficientemente significativo de PASCAL, que aparece descrito al final del capítulo 6. El compilador construido es un prototipo, aunque, para mostrar la facilidad de la inclusión de la optimación en otros compiladores existentes del lenguaje, sus características no se alejan de las que son habituales. De esta manera, se han determinado exactamente los requisitos estáticos para practicar la optimación: la única exigencia es la de construir el árbol decorado para la parte de sentencias de cada subprograma.

En cuanto a las aportaciones, en un sentido más genérico, del presente trabajo en relación con el dilema "iteratividad versus recursividad", en particular, y sobre la metodología de programación, en general, pueden apuntarse las siguientes ideas:

- El estudio de la literatura relacionada con el problema (capítulo 2) muestra la existencia aún hoy de una asociación entre "recursividad" e "ineficiencia", al menos en lo que respecta a los lenguajes imperativos: a pesar de que se ensalzan las cualidades de la recursividad como herramienta de programación, es común poner inmediatamente el énfasis en evitar su uso "indiscriminado". Esto es tanto menos adecuado en cuanto que tales argumentos se realizan en obras destinadas a los primeros pasos en el aprendizaje de la programación, fortaleciendo un "rechazo" a la recursividad.
- El trabajo demuestra que la identificación de la recursividad final con iteratividad puede, sin embargo, realizarse (con pequeñas restricciones) en un lenguaje imperativo "tradicional" (pero de uso mayoritario). Además, hasta ahora parecía existir la impresión de que tal identificación solo podía darse para lenguajes con características peculiares alejadas de los lenguajes imperativos (SCHEME, lenguajes funcionales).
- En este sentido, la metodología de programación que actualmente se acepta se basa principalmente en la recursividad, teniendo como meta la obtención de programas recursivos finales, que han de convertirse en iterativos. La presencia de un tratamiento adecuado de la recursividad final en los lenguajes de programación es prácticamente una necesidad bajo este esquema, y se ha mostrado que tal necesidad puede quedar cubierta incluso en un lenguaje "tradicional" como PASCAL con la técnica de optimación que se ha desarrollado.
- La optimación de la recursividad final posibilita hábitos que sacan partido de esta poderosa estrategia de programación sin menoscabar la eficiencia de los programas. El doctorando opina que no se trata de que la recursividad final sustituya a la iteratividad (sintácticamente hablando) como dogma, sino que debe darse una "convivencia pacífica" de ambas estrategias. Disponiendo de la optimación de la recursividad final, se evitan frecuentes modificaciones de los programas que se realizan para mejorar la eficiencia de ciertos procesos recursivos.

- Por otro lado, los criterios habituales de programación en el lenguaje han de prevalecer ante posibles soluciones que faciliten la optimación pero yendo en contra de tales criterios. Sin embargo, lo que ocurrirá a menudo es que facilitar la optimación no solo no irá en contra, sino que posibilitará el cumplimiento pleno de esa metodología: la recursividad final permite la expresión de procesos iterativos más variados que los ofrecidos por las construcciones sintácticas repetitivas habituales en lenguajes como PASCAL.

LÍNEAS FUTURAS DE INVESTIGACIÓN

Lo planteado en estos últimos párrafos da pie, como es obvio, a la presentación de los caminos que pueden seguirse a partir de aquí. Algunos supondrán completar aspectos que no han sido tratados en este trabajo, que indudablemente existen. Otros se refieren al avance en ese objetivo más general de mejora del comportamiento de los programas recursivos.

- Ha de reconocerse que el paso de parámetros en la llamada es el punto más débil de la técnica de optimación desarrollada. Aunque ya se han identificado casos particulares sencillos en los que la actualización de los parámetros no es necesaria (lo que redundaría en un incremento de la *ratio* de mejora temporal del programa optimado), deben realizarse esfuerzos para mejorar el comportamiento en otros casos más generales.
- Es claro que ha de estudiarse la posibilidad de aplicar las técnicas desarrolladas a otros lenguajes imperativos ampliamente extendidos, como C, MODULA o ADA. Se han identificado varias características de estos lenguajes que plantean pequeñas dudas sobre la aplicación de la técnica desarrollada, pero no se ha profundizado en ellas. Sin embargo, no es de esperar que la propia técnica deba modificarse en su filosofía general, por la similitud de la implementación de estos lenguajes con la de PASCAL.
- La mejora del comportamiento de los programas recursivos no finales es el gran paso que ha de darse a partir de aquí. Se conocen diversas heurísticas para la conversión de esquemas recursivos más complejos, pero los intentos de abarcarlos de una forma mecánica no han fructificado desde el

punto de vista práctico. Parece que el tratamiento de otros esquemas recursivos supone un salto cualitativo en las necesidades del conocimiento que ha de aplicarse, de manera que el procesamiento mecánico que es capaz de efectuar un compilador no resulta suficiente, o simplemente es inadecuado. Sin embargo, las estrategias transformacionales no tienen el atractivo necesario, primordialmente por tres razones:

- primero, por resolverse exclusivamente en el dominio del lenguaje fuente, lo que limita en algunos casos las soluciones, si existe falta de expresividad en el lenguaje (como ha ocurrido en este trabajo);
- segundo, por plantearse sobre lenguajes esquemáticos, o en todo caso, subconjuntos muy reducidos de lenguajes existentes;
- y tercero, por el recurso necesario que ha de hacerse en los casos complejos a una pila (lo que tiene el efecto de empeorar a la postre el comportamiento, como demuestran algunos trabajos).

En opinión del doctorando, debería incidirse en un primer momento en los programas que pueden transformarse en recursivos finales sin necesidad de acudir a una pila. La estrategia debería combinar una mezcla de tácticas transformacionales y de compilación, ser transparente para el programador y aplicarse en lenguajes "reales" de uso extendido, como PASCAL (supuesta la optimación de la recursividad final, obviamente).

9. BIBLIOGRAFÍA

- Abelson, H.; Sussman, G.J.; Sussman, J. [1985] *Structure and interpretation of computer programs*. The MIT Press: Cambridge, Massachusetts.
- Adams, S.; Hanson, Ch. [1993] *MIT Scheme user's manual. Edition 1.21beta for Scheme release 7.3*. Massachusetts Institute of Technology.
- Aho, A.; Sethi, R.; Ullman, J.D. [1990] *Compiladores. Principios, técnicas y herramientas*. Addison-Wesley Iberoamericana: México (traducción al castellano de Aho, A.; Sethi, R.; Ullman, J.D. [1986] *Compilers. Principles, techniques, and tools*. Addison-Wesley: Reading, Massachusetts.)
- Allard, J.R.; Hawkinson, L.B. [1991] «Real time programming in Common Lisp», *Communications of the ACM* 34(9) septiembre: 64-69.
- Allen, J. [1978] *Anatomy of LISP*. McGraw-Hill: New York.
- Ammann, U. [1977] «On code generation in a Pascal compiler», *Software-Practice and Experience* 7(3) marzo: 391-423 (*vide también* Barron [1981: 83-123]).
- Ammann, U. [1981] «The Zurich implementation», en Barron [1981: 63-82]).
- Appel, A.W. [1992] *Compiling with continuations*. Cambridge University Press: Cambridge, England.
- Arsac, J.; Kodratoff, Y. [1982] «Some techniques for recursion removal from recursive functions», *ACM Transactions on Programming Languages and Systems* 4(2) abril: 295-322.
- Auslander, M.A.; Strong, H.R. [1978] «Systematic recursion removal», *Communications of the ACM* 21(2) febrero: 127-134.
- Backus, J. [1978] «Can programming be liberated from the von Neumann style? A functional style and its algebra of programs», *Communications of the ACM* 21(8) agosto: 613-641.
- Bailey, R. [1990] *Functional programming with HOPE*. Ellis Horwood: Chichester, England.

- Bakker, J. de [1980] *Mathematical theory of program correctness*. Prentice-Hall International: London.
- Balcázar, J.L. [1993] *Programación metódica*. McGraw-Hill: Madrid.
- Balcázar, J.L. [1994] «A transformation scheme for double recursion», Report LSI-94-8-R, Departament de Llenguatges i Sistemes Informàtics, Universitat Politècnica de Catalunya: Barcelona.
- Barendregt, H.P. [1984] *The lambda calculus. Its syntax and semantics (revised edition)*. North-Holland: Amsterdam.
- Bartley, D.H. [1992] «Optimizing stack frame accesses for processors with restricted addressing modes», *Software-Practice and Experience* 22(2) febrero: 101-110.
- Bennett, J.P. [1990] *Introduction to compiling techniques. A first course using ANSI C, lex and yacc*. McGraw-Hill: London.
- Berlioux, P.; Bizard, Ph. [1986] *Algorithms: the construction, proof, and analysis of programs*. John Wiley & Sons: Chichester, England.
- Betz, D.M. [1989] *XScheme: an object-oriented Scheme. Version 0.17.* .
- Bird, R.S. [1977a] «Notes on recursion elimination», *Communications of the ACM* 20(6) junio: 434-439.
- Bird, R.S. [1977b] «Improving programs by the introduction of recursion», *Communications of the ACM* 20(1) noviembre: 856-863.
- Bobrow, D.G.; Wegbreit, B. [1973] «A model and stack implementation of multiple environments», *Communications of the ACM* 16(10) octubre: 591-603.
- Boehm, C.; Jacopini, G. [1966] «Flow diagrams, Turing machines and languages with only two formation rules», *Communications of the ACM* 9(5) mayo: 366-371.
- Borland International, Inc. [1992a] *Turbo Pascal® versión 7.0. Guía del lenguaje*. Scotts Valley, California.
- Borland International, Inc. [1992b] *Turbo Vision™ versión 2.0. Guía de programación*. Scotts Valley, California.
- Bornat, R. [1979] *Understanding and writing compilers. A do-it-yourself guide*. MacMillan: London.
- Botella, P. [1991] *Compiladors. Anàlisi, lèxica i sintàctica*. Facultat de Informàtica, Universitat Politècnica de Catalunya.

- Botella, P.; Burgués, X.; Franch, X. [1988] «Combining the imperative and the equational programming paradigms in the Excalibur system», *Proceedings of the Workshop on Software Engineering and its Applications*, Toulouse, Francia.
- Boyer, R.S.; Moore, J.S. (eds.) [1981] *The correctness problem in computer science*. Academic Press: London.
- Boyer, R.S.; Moore, J.S. [1984] «A mechanical proof of the unsolvability of the halting problem», *Journal of the ACM* 31(3) julio: 441-458.
- Brinch Hansen, P. [1985] *Brinch Hansen on Pascal compilers*. Prentice-Hall International: London, England.
- Broy, M.; Krieg-Brückner, B. [1980] «Derivation of invariant assertions during program development by transformation», *ACM Transactions on Programming Languages and Systems* 2(3) julio: 321-337.
- Burstall, R.M.; Darlington, J. [1977] «A transformation system for developing recursive programs», *Journal of the ACM* 24(1) enero: 44-67.
- Castro, J.; Cucker, F.; Messeguer, X.; Rubio, A.; Solano, Ll.; Valles, B. [1993] *Curso de programación*. McGraw-Hill: Madrid.
- Clack, Ch.; Myers, C.; Poon, E. [1993] *Programming with Standard ML*. Prentice-Hall: Hemel Hempstead, England.
- Clinger, W. (ed.) [1985] «The revised revised report on Scheme, or an uncommon Lisp», MIT AI Memo 848, MIT AI Laboratory: Cambridge, Massachusetts (también Computer Science Department Technical Report 174, Indiana University).
- Clinger, W. [1984] «The Scheme 311 compiler. An exercise in denotational semantics», *Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming*: 356-364.
- Clinger, W.; Rees, J. (eds.) [1991] «Revised⁴ report on the algorithmic language Scheme», Massachusetts Institute of Technology, Indiana University y University of Oregon.
- Cockshott, P. [1990] *A compiler writer's toolbox. How to implement interactive compilers for PCs using Turbo Pascal*. Ellis Horwood: Chichester, England.
- Cohen, E. [1990] *Programming in the 1990's. An introduction to the calculation of programs*. Springer-Verlag: Berlín.
- Cohen, N.H. [1986] *Ada® as a second language*. McGraw-Hill: New York.

- Colussi, L. [1984] «Recursion as an effective step in program development», *ACM Transactions on Programming Languages and Systems* 6(1) enero: 55-67.
- Cormen, T.H.; Leiserson, C.H.; Rivest, R.L. [1990] *Introduction to algorithms*. The MIT Press: Cambridge, Massachusetts.
- Cortés, U.; Sierra, C. [1987] *Lisp*. Marcombo Boixareu Editores: Barcelona.
- Dahl, O. [1992] *Verifiable programming*. Prentice-Hall International: London.
- Davidson, J.W.; Whalley, D.B. [1991] «Methods for saving and restoring register values across function calls», *Software-Practice and Experience* 21(2) febrero: 149-165.
- Davie, A.J.T. [1992] *An introduction to functional programming systems using Haskell*. Cambridge University Press: Cambridge, England.
- Davis, M. (ed.) [1965] *The undecidable. Basic papers on undecidable propositions, unsolvable problems and computable functions*. Raven Press: Nueva York.
- Deransart, P.; Jourdan, M.; Lorho, B. [1988] *Attribute grammars. Definitions, systems and bibliography (Lecture Notes in Computer Science 323)*. Springer-Verlag: Berlin.
- Díaz de Ilarraza, A.; Lucio, F. [1990] *Verificación de programas y metodología de la programación*. Universidad del País Vasco: Bilbao.
- Díaz Labrador, J.L. [1994] *Teoría de autómatas y lenguajes formales*. La Enseñanza Católica: Bilbao.
- Digital Equipment Corporation [1987] *VAX PASCAL user manual (2nd revised ed.)*.
- Dijkstra, E.W. [1968] «Go to statement considered harmful», *Communications of the ACM* 11(3) marzo: 147-148.
- Dijkstra, E.W. [1976] *A discipline of programming*. Prentice-Hall: Englewood Cliffs, New Jersey.
- Dijkstra, E.W. [1981] «Why correctness must be a mathematical concern», en Boyer y Moore [1981: 1-8].
- Dijkstra, E.W. (ed.) [1990] *Formal development of programs and proofs*. Addison-Wesley: Reading, Massachusetts.
- Diller, A. [1988] *Compiling functional languages*. John Wiley & Sons: Chichester, England.

- Diller, A. [1990] *Z. An introduction to formal methods*. John Wiley & Sons: Chichester, England.
- Dromey, G. [1989] *Program derivation. The development of programs from specifications*. Addison-Wesley: Sidney.
- Dybvig, R.K. [1987] *The SCHEME programming language*. Prentice-Hall: Englewood Cliffs, New Jersey.
- Dybvig, R.K.; Hleb, R.; Bruggeman, C. [1993] «Syntactic abstraction in Scheme», *Lisp and Symbolic Computation* 5: 295-326.
- Eguiluz, A. [1993] *Estructuras de datos y algoritmos* (versión preliminar pendiente de publicación).
- Eguiluz, A. [1994] *Comunicación personal*.
- Eguiluz, A.; Díaz Labrador, J.L. [1995] *Teoría de compiladores* (versión preliminar).
- Eisenberg, M. [1988] *Programming in Scheme*. The MIT Press: Cambridge, Massachusetts.
- Elder, J. [1994] *Compiler construction. A recursive descent model*. Prentice-Hall: Englewood Cliffs, New Jersey.
- Feather, M. [1987] «A survey and classification of some program transformation approaches and techniques», en Meertens, L.G.L.T. [1987: 165-195].
- Felgentreu, K.-U.; Lippe, W.-M.; Simon, F.H. [1987] «Optimizing static scope Lisp by repetitive interpretation of recursive function calls», *IEEE Transactions on Software Engineering* 13(6) junio: 628-635.
- Feyerabend, P. [1986] *Tratado contra el método. Esquema de una teoría anarquista del conocimiento*. Tecnos: Madrid (traducción al castellano de: Feyerabend, P. [1975] *Against Method*. NLB: Londres).
- Fischer, A.E.; Grodzinsky, F.S. [1993] *The anatomy of programming languages*. Prentice-Hall: Englewood Cliffs, New Jersey.
- Fischer, Ch.N.; LeBlanc, R.J., Jr. [1988] *Crafting a compiler*. Benjamin-Cummings: Menlo Park, California.

- Flanagan, C.; Sabry, A.; Duba, B.F.; Felleisen, M. [1993] «The essence of compiling with continuations», *ACM SIGPLAN Notices* 28(6) junio: 237-247.
- Franch, X.; Burgués, X. [1993] «A case study on prototyping with specifications», *Proceedings of the ERCIM Workshop on Development and Transformation of Programs*, Nancy, Francia: anexo sin paginar.
- Franch, X.; Burgués, X.; Botella, P. [1993] «Report on the programming language Merlí», Technical Report LSI-93-T, Universitat Politècnica de Catalunya: Barcelona.
- Friedman, D.P. [1987] «Foreword», en Dybvig [1987: vii-viii].
- Friedman, D.P.; Felleisen, M. [1987] *The little LISPer*. The MIT Press: Cambridge, Massachusetts.
- Friedman, D.P.; Wand, M.; Haynes, Ch.T. [1992] *Essentials of programming languages*. The MIT Press: Cambridge, Massachusetts.
- Fritzson, P.A. (ed.) [1994] *Compiler construction. 5th International Conference CC'94, Edinburgh, U.K., April 1994*. Springer-Verlag: Berlin.
- Gabriel, R.P. [1985] *Performance and evaluation of Lisp systems*. The MIT Press: Cambridge, Massachusetts.
- Gabriel, R.P.; White, J.L.; Bobrow, D.G. [1991] «CLOS: integrating object-oriented and functional programming», *Communications of the ACM* 34(9) septiembre: 28-38.
- Garitagoitia, J.R. [1991] *Estructuras de datos como tipos abstractos* (apuntes sin publicar).
- Gelernter, D.; Jagannathan, S. [1990] *Programming linguistics*. The MIT Press: Cambridge, Massachusetts.
- Ghezzi, C.; Jazayeri, M. [1987] *Programming language concepts (2nd ed.)*. John Wiley & Sons: New York.
- Gimeno, J.M.; Cortés, U. [1989] «UPCLisp: a metacircular definition». Report de Recerca GRIAL 89/7, Centre d'Estudis Avançats de Blanes, CSIC: Blanes, Gerona.
- Give'on, Y.S. [1990] «Is recursion well defined?», *Computers and Education* 14(1): 35-41.

- Gómez, M.; Díaz Labrador, J.L. [1995] *Teoría de la computación (II): calculabilidad y complejidad* (versión preliminar).
- Gordon, M.J.C. [1988] *Programming language theory and its implementation. Applicative and imperative paradigms*. Prentice-Hall International: London.
- Guijarro, D.; Parés, N.; Cortés, U. [1990] *Tècniques de compilació i interpretació per a llenguatges funcionals*. Departament de Llenguatges i Sistemes Informàtics, Universitat Politècnica de Catalunya.
- Hagiya, M.; Mitchell, J.C. (eds.) [1994] *Theoretical aspects of computer software. Proceedings of the International Symposium TACS'94, Sendai, Japan, April 19-22, 1994*. Springer-Verlag: Berlin.
- Hammer, D. (ed.) [1989] *Compiler compilers and high speed compilation. Lecture notes in Computer Science 371*. Springer-Verlag: Berlin.
- Hanson, Ch. [1990] «Efficient stack allocation for tail-recursive languages», *Proc. of the ACM Symposium on Lisp and Functional Programming*: 106-118.
- Hanson, Ch. [1991a] *MIT Scheme reference manual. Edition 1.1 for Scheme release 7.1.2*. Massachusetts Institute of Technology.
- Hanson, Ch. [1991b] *MIT Scheme user's manual. Edition 0.9 for Scheme release 7.1*. Massachusetts Institute of Technology.
- Hanson, Ch. [1993] *MIT Scheme reference manual. Edition 1.41beta for Scheme release 7.3*. Massachusetts Institute of Technology.
- Harbison, S.P. [1992] *Modula-3*. Prentice-Hall: Englewood Cliffs, New Jersey.
- Harbison, S.P.; Steele, G.L., Jr. [1984] *C. A reference manual*. Prentice-Hall: Englewood Cliffs, New Jersey.
- Harel, D. [1987] *Algorithmics: the spirit of computing*. Addison-Wesley: Wokingham, England.
- Hennessy, M. [1990] *The semantics of programming languages. An elementary introduction using structural operational semantics*. John Wiley & Sons: Chichester, England.
- Hewitt, C. [1977] «Viewing control structures as patterns of passing messages», *Journal of Artificial Intelligence* 8(3): 323-369 (también en Winston y Brown [1980: 433-465]).
- Hoare, C.A.R. [1969] «An axiomatic basis for computer programming», *Communications of the ACM* 12(10) octubre: 576-580, 583.

- Hoare, C.A.R.; Shepherdson, J.C. (eds.) [1985] *Mathematical logic and programming languages*. Prentice-Hall International: London.
- Hofstadter, D.R. [1987] *Gödel, Escher, Bach: un Eterno y Grácil Bucle*. Tusquets: Barcelona (traducción al castellano de Hofstadter, D.R. [1979] *Gödel, Escher, Bach: an Eternal Golden Braid*. Basic Books: New York.)
- Holub, A.I. [1990] *Compiler design in C*. Prentice-Hall: Englewood Cliffs, New Jersey.
- Hoogerwoord, R. [1990a] «For the record: a nice and simple program transformation», informe personal RH141.
- Hoogerwoord, R. [1990b] «A sequel to rh141: the Ackermann function», informe personal RH142.
- Hoogerwoord, R. [1992] «A derivation of the tail recursion theorem for lists», informe personal RH165.
- Hopcroft, J.E.; Ullman, J.D. [1979] *Introduction to automata theory, languages, and computation*. Addison-Wesley: Reading, Massachusetts.
- Hudak, P. [1989] «Conception, evolution, and application of functional programming languages», *ACM Computing Surveys* 21(3) septiembre: 359-411.
- Hudak, P.; Fasel, J.H. [1992] «A gentle introduction to Haskell», *ACM SIGPLAN Notices* 27(5) mayo: T/1-T/53.
- Hudak, P.; Peyton Jones, S.; Wadler, Ph. [1992] «Report on the programming language Haskell: A non-strict, purely functional language», *ACM SIGPLAN Notices* 27(5) mayo: R/i-R/164.
- Huet, G. (ed.) [1990] *Logical foundations of functional programming*. Addison-Wesley: Reading, Massachusetts.
- Ichbiah, J.; Barnes, J.; Firth, R.; Woodger, M. [1991] *Rationale for the design of the Ada programming language (corrected ed.)*. Cambridge University Press: Cambridge, England.
- Intel Corporation [1989] 80386. *Guía del programador y manual de referencia*. Anaya: Madrid (traducción al castellano de: Intel Corporation [1986] *80386 Programmer's Reference Manual*).
- Iverson, K.E. [1962] *A programming language*. John Wiley & Sons: New York.

- Jain, R. [1991] *The art of computer systems performance analysis. Techniques for experimental design, measurement, simulation, and modelling*. John Wiley & Sons: New York.
- Jensen, K.; Wirth, N. [1975] *Pascal: user manual and report (2nd ed.)*. Springer-Verlag: New York.
- Jensen, K.; Wirth, N. [1991] *Pascal: user manual and report. ISO Pascal standard (4th ed.)*. Springer-Verlag: New York.
- Johnson, S.C. [1975] «Yacc-yet another compiler compiler», Computing Science Technical Report 32, AT&T Bell Laboratories: Murray Hill, New Jersey.
- Johnson, S.C.; Lesk, M.E. [1978] «Language development tools», *Bell System Technical Journal* 57(6): 2155-2175.
- Jones, N.D.; Gomard, C.K.; Sestoft, P. [1993] *Partial evaluation and automatic program generation*. Prentice-Hall: Hemel Hempstead, England.
- Jones, R.; Maynard, C.; Stewart, I. [1990] *The art of Lisp programming*. Springer-Verlag: London.
- Kamin, S.N. [1990] *Programming languages. An interpreter-based approach*. Addison-Wesley: Reading, Massachusetts.
- Kaser, O.; Ramakrishnan, C.R.; Pawagi, S. [1993] «On the conversion of indirect to direct recursion», *ACM Letters on Programming Languages and Systems* 2(1-4) marzo: 151-164.
- Kelley, A.; Pohl, I. [1984] *A book on C. An introduction to programming in C*. Benjamin/Cummings: Redwood City, California.
- Kelley, A.; Pohl, I. [1990] *A book on C. Programming in C (2nd ed.)*. Benjamin/Cummings: Redwood City, California.
- Kernighan, B.W.; Plauger, P.J. [1976] *Software tools*. Addison-Wesley: Reading, Massachusetts.
- Kernighan, B.W.; Ritchie, D.M. [1988] *The C programming language (2nd ed.)*. Prentice-Hall: Englewood Cliffs, New Jersey.
- Kessler, R.R. [1988] *LISP, objects, and symbolic programming*. Scott, Foresman and Co.: Glenview, Illinois.
- Kogge, P.M. [1991] *The architecture of symbolic computers*. McGraw-Hill: New York.

- Kranz, D.; Kelsey, R.; Rees, J.; Hudak, P.; Philbin, J. y Adams, N. [1986] «ORBIT: an optimizing compiler for Scheme», *ACM SIGPLAN Notices* 21(7) julio: 219-233.
- Landin, P.J. [1966] «The next 700 programming languages», *Communications of the ACM* 9(3) marzo: 157-166.
- Layer, D.K.; Richardson, Ch. [1991] «Lisp systems in the 1990s», *Communications of the ACM* 34(9) septiembre: 48-57.
- Lee, P. [1989] *Realistic compiler generation*. The MIT Press: Cambridge, Massachusetts.
- Lee, P. (ed.) [1991] *Topics in advanced language implementation*. The MIT Press: Cambridge, Massachusetts.
- Leinbach, L.C. [1986] «On classifying recursive algorithms», *ACM SIGCSE Bulletin* 18(1) febrero: 186-190.
- Lesk, M.E. [1975] «Lex-a lexical analyzer generator», Computing Science Technical Report 39, AT&T Bell Laboratories: Murray Hill, New Jersey.
- Levine, J.R.; Mason, T.; Brown, D. [1992] *lex & yacc (2nd ed.)*. O'Reilly & Associates: Sebastopol, California.
- Lewi, J.; Vlamincx, K. de; Steegmans, E.; Horebeek, I. van [1992] *Software development by LL(1) syntax description*. John Wiley & Sons: Chichester, England.
- Lightship Software [1990] *MacScheme. Manual and software*. The MIT Press: Cambridge, Massachusetts.
- Louden, K. [1987] «Recursion versus non-recursion in Pascal: recursion can be faster», *ACM SIGPLAN Notices* 22(2) febrero: 62-67.
- MacLennan, B.J. [1987] *Principles of programming languages. Design, evaluation, and implementation (2nd ed.)*. Holt, Rinehart and Winston: New York.
- MacLennan, B.J. [1990] *Functional programming. Practice and theory*. Addison-Wesley: Reading, Massachusetts.
- Mak, R. [1991] *Writing compilers and interpreters. An applied approach*. John Wiley & Sons: New York.
- Manber, U. [1989] *Introduction to algorithms. A creative approach*. Addison-Wesley: Reading, Massachusetts.

- McCarthy, J. [1960] «Recursive functions of symbolic expressions and their computation by machine», *Communications of the ACM* 3(4) abril: 184-195.
- Meertens, L.G.L.T. (ed.) [1987] *Program specification and transformation*. Elsevier: Amsterdam..
- Meyer, B. [1990] *Introduction to the theory of programming languages*. Prentice-Hall: London.
- Michaelson, G. [1989] *An introduction to functional programming through lambda calculus*. Addison-Wesley: Wokingham, England.
- Milner, R.; Tofte, M.; Harper, R. [1990] *The definition of Standard ML*. The MIT Press: Cambridge, Massachusetts.
- Milner, W.L. [1988] *Common Lisp. A tutorial*. Prentice-Hall: Englewood Cliffs, New Jersey.
- Moreno, A.; Cortés, U. [1989] «Streams: what kind of objects they are (I)», Report de Recerca GRIAL 89/5, Centre d'Estudis Avançats de Blanes, CSIC: Blanes, Gerona.
- Morgan, C. [1990] *Programming from specifications*. Prentice-Hall: New York.
- Motorola, Inc. [1985] *MC68020 32-bit microprocessor user's manual (2nd ed.)*. Prentice-Hall: Englewood Cliffs, New Jersey.
- Moynihán, V.D.; Wallis, P.J.L. [1991] «The design and implementation of a high-level language converter», *Software-Practice and Experience* 21(4) abril: 391-400.
- Mueller, R.A.; Page, R.L. [1988] *Symbolic computing with Lisp and Prolog*. John Wiley & Sons: New York.
- Naur, P. (ed.) [1960] «Report on the algorithmic language Algol 60», *Communications of the ACM* 3(5) mayo: 299-314.
- Naur, P. (ed.) [1963] «Revised report on the algorithmic language Algol 60», *Communications of the ACM* 6(1) enero: 1-17.
- Neuhold, E.J.; Paul, M. (eds.) [1991] *Formal description of programming concepts*. Springer-Verlag: New York.
- Niekerk, P.A.W. van [1990] *XLC: a compiler for XLisp*. Faculty of Electrical Engineering, Technical University Delft.
- Norvig, P. [1992] *Paradigms of artificial intelligence programming. Case studies in Common Lisp*. Morgan Kaufmann: San Mateo, California.

Orbaek, P. [1994] «OASIS: an optimizing action-based compiler generator», en Fritzson, P.A. [1994: 1-15].

Partsch, H.; Pepper, P. [1986] «Program transformations expressed by algebraic type manipulations», *Recherche* 5(3) : 197-212.

Partsch, H.; Steinbrüggen, R. [1983] «Program transformation systems», *ACM Computing Surveys* 15(3) septiembre: 199-236.

Paulson, L.C. [1991] *ML for the working programmer*. Cambridge University Press: Cambridge, England.

Perrott, R.H.; Zarea-Aliabadi, A. [1986] «Supercomputer languages», *ACM Computing Surveys* 18(1) marzo: 5-22.

Péter, R. [1981] *Recursive functions in computer theory*. Ellis Horwood: Chichester, England.

Peyton Jones, S.L. [1987] *The implementation of functional programming languages*. Prentice-Hall International: London.

Peyton Jones, S.L.; Lester, D.R. [1992] *Implementing functional languages*. Prentice-Hall International: London.

Pittman, T.; Peters, J. [1992] *The art of compiler design. Theory and practice*. Prentice-Hall International: Englewood Cliffs, New Jersey.

Pleban, U. [1989] «Foreword», en Lee [1989].

Pratt, T.W. [1984] *Programming languages. Design and implementation (2nd ed.)*. Prentice-Hall: Englewood Cliffs, New Jersey.

Queinnec, Ch. [1992] «Compiling syntactically recursive programs», *Lisp Pointers* V(4) octubre: 2-10.

Queinnec, Ch. [1993] «Continuation conscious compilation», *Lisp Pointers* VI(1) enero: 2-14.

Quine, W.V.O. [1968] *Palabra y objeto*. Labor: Barcelona (traducción al castellano de: Quine, W.V.O. [1960] *Word and object*. Technology Press: Cambridge, Massachusetts).

- Rees, J. [1994] «The Scheme of things: implementing lexically scoped macros», *LISP Pointers* VI(1) enero: 33-37.
- Rees, J.; Clinger, W. (eds.) [1986] «Revised³ report on the algorithmic language Scheme», AI Memo 848a, Massachusetts Institute of Technology: Cambridge, Massachusetts (también en *ACM SIGPLAN Notices* 21(12) diciembre: 37-79).
- Rees, M.; Robson, D. [1988] *Practical compiling with Pascal-S*. Addison-Wesley: Wokingham. England.
- Rees, M.J.; Goodson, J.I.; Reynolds, J.J.M.; Zell, H.J. [1981] «Pascal on an advanced architecture», en Barron [1981: 261-276]).
- Reiser, M. [1991] *The Oberon system. User guide and programmers manual*. ACM Press: New York.
- Reiser, M.; Wirth, N. [1992] *Programming in Oberon. Steps beyond Pascal and Modula*. ACM Press: New York.
- Roberts, E. [1986] *Thinking recursively*. John Wiley & Sons: New York.
- Rohl, J.S. [1984] *Recursion via Pascal*. Cambridge University Press: Cambridge, Massachusetts.
- Rozas, G.J. [1984] «LIAR, an ALGOL-like compiler for SCHEME», S.B. Thesis, MIT Department of Electrical Engineering and Computer Science: Cambridge, Massachusetts.
- Sanchís, F.J.; Galán, C. [1986] *Compiladores. Teoría y construcción*. Paraninfo: Madrid.
- Sannella, D. (ed.) [1994] *Programming languages and systems-ESOP'94. Proceedings of the 5th European Symposium on Programming, Edinburgh. UK. April 11-13, 1994*. Springer-Verlag: Berlin.
- Sebesta, R.W. [1989] *Concepts of programming languages*. Benjamin/Cummings: Redwood City, California.
- Sedgewick, R. [1988] *Algorithms (2nd ed.)*. Addison-Wesley: Reading, Massachusetts.
- Sesa i Nogueras, E.; Cortés, U. [1989] «Implementing continuations», Report de Recerca GRIAL 89/6, Centre d'Estudis Avançats de Blanes, CSIC: Blanes, Gerona.

- Sethi, R. [1992] *Lenguajes de programación. Conceptos y constructores*. Addison-Wesley Iberoamericana: México (traducción al castellano de Sethi, R. [1989] *Programming languages. Concepts and constructs*. Addison-Wesley: Reading, Massachusetts.)
- Shimasaki, M.; Fukaya, S.; Ikeda, K.; Kiyono, T. [1980] «An analysis of Pascal programs in compiler writing», *Software-Practice and Experience* 10(2) febrero: 149-157.
- Sinclair, K.H.; Moon, D.A. [1991] «The philosophy of Lisp», *Communications of the ACM* 34(9) septiembre: 40-47.
- Smith, J.D. [1988] *An Introduction to Scheme*. Prentice-Hall: Englewood Cliffs, New Jersey.
- Souquière, J. (ed.) [1993] *Proceedings of the ERCIM Workshop on Development and Transformation of Programs*, Nancy, France, November 2-4.
- Springer, G.; Friedman, D.P. [1989] *Scheme and the art of programming*. The MIT Press: Cambridge, Massachusetts.
- Stark, W.R. [1990] *LISP, Lore, and Logic. An algebraic view of LISP programming, foundations, and applications*. Springer-Verlag: New York.
- Steel, T.B., Jr. (ed.) [1966] *Formal language description languages for computer programming. Proceedings of the IFIP Working Conference on Formal Language Description Languages, Vienna, Austria, September 1964*. North-Holland: London.
- Steele, G.L., Jr. [1977a] «Debunking the "Expensive procedure call" myth or, procedure call implementations considered harmful or, LAMBDA: the ultimate GOTO», *ACM Conference Proceedings*: 153-162.
- Steele, G.L., Jr. [1977b] «Macaroni is better than spaghetti», *Proceedings of the Symposium on Artificial Intelligence and Programming Languages, ACM SIGPLAN Notices* 12(8) agosto: 60-66.
- Steele, G.L., Jr. [1977c] «Compiler optimization based on viewing LAMBDA as RENAME plus GOTO», S.M. Thesis, MIT AI Laboratory: Cambridge, Massachusetts (también en Winston y Brown [1980: 399-431]).
- Steele, G.L., Jr. [1978] «RABBIT: a compiler for SCHEME», AI Memo 474, MIT AI Laboratory: Cambridge, Massachusetts.
- Steele, G.L., Jr. [1990] *Common Lisp. The language* (2nd ed.). Digital Press.
- Stepney, S. [1993] *High integrity compilation*. Prentice-Hall: New York.

- Stroustrup, B. [1984] «The UNIX system: Data abstraction in C», *AT&T Bell Laboratories Technical Journal* 63(8) octubre: 1701-1732.
- Stroustrup, B. [1986] *The C++ programming language*. Addison-Wesley: Reading, Massachusetts.
- Sussman, G.J.; Steele, G.L., Jr. [1975] «SCHEME: an interpreter for extended lambda calculus», *AI Memo 349*, MIT AI Laboratory: Cambridge, Massachusetts.
- Swan, T. [1994] «Fractal rulers», *Dr. Dobbs's Journal* 19(6) junio: 117-121.
- Tanenbaum, A.S.; Staveren, H. van; Keizer, E.G.; Stevenson, J.W. [1983] «A practical tool kit for making portable compilers», *Communications of the ACM* 26(9) septiembre: 654-660.
- Tennent, R.D. [1991] *Semantics of programming languages*. Prentice-Hall International: London.
- Terry, P.D. [1986] *Programming language translation. A practical approach*. Addison-Wesley: Wokingham, England.
- Texas Instruments [1987a] *PC Scheme. A simple, modern LISP. User's guide (rev. B)*. Austin, Texas.
- Texas Instruments [1987b] *TI Scheme language reference manual (rev. B)*. Austin, Texas.
- Tofte, M. [1990] *Compiler generators. What they can do, what they might do, and what they will probably never do*. Springer-Verlag: Berlin.
- Turner, D.A. (ed.) [1990] *Research topics in functional programming*. Addison-Wesley: Reading, Massachusetts.
- Ullmann, J.R. [1994] *Compiling in Modula-2. A first introduction to classical recursive descent dcompiling*. Prentice-Hall: Hemel Hempstead, UK.
- Waite, W.M.; Goos, G. [1984] *Compiler construction*. Springer-Verlag: New York.
- Watson, D. [1989] *High-level languages and their compilers*. Addison-Wesley: Wokingham, England.
- Watt, D.A. [1990] *Programming language concepts and paradigms*. Prentice-Hall: New York.

- Watt, D.A. [1991] *Programming language syntax and semantics*. Prentice-Hall International: London.
- Watt, D.A. [1993] *Programming language processors. Compilers and interpreters*. Prentice-Hall International: London.
- Watt, D.A.; Wichmann, B.A.; Findlay, W. [1987] *ADA. Language and methodology*. Prentice-Hall: Hemel Hempstead, UK.
- Weiser, L. [1991] *Comparative programming languages. Generalizing the programming function*. Prentice-Hall International: Englewood Cliffs, New Jersey.
- Welsh, J. [1981] «Two 1900 compilers», en Barron [1981: 171-180]).
- Welsh, J.; Sneeringer, W.J.; Hoare, C.A.R. [1977] «Ambiguities and insecurities in Pascal», *Software-Practice and Experience* 7(6) junio: 685-696 (vide también Barron [1981: 5-19]).
- Wexelblat, R.L. (ed.) [1981] *ACM SIGPLAN History of Programming Languages Conference, June 1-3, 1978*. Academic Press: London.
- Wichmann, B.A. [1977] «How to call procedures, or second thoughts on Ackermann's function», *Software-Practice and Experience* 7(3) marzo: 317-329.
- Wilson, L.B.; Clark, R.G. [1988] *Comparative programming languages*. Addison-Wesley: Wokingham, England.
- Winograd, T. [1991] «Máquinas pensantes: ¿son posibles?, ¿lo somos?», *Revista de Occidente* 119, abril: 113-150.
- Winskel, G. [1993] *The formal semantics of programming languages. An introduction*. The MIT Press: Cambridge, Massachusetts.
- Winston, P.H.; Brown, R.H. (eds.) [1979] *Artificial intelligence: an MIT perspective. Volume I: expert problem solving, natural language understanding, intelligent computer coaches, representation and learning*. The MIT Press: Cambridge, Massachusetts.
- Winston, P.H.; Brown, R.H. (eds.) [1980] *Artificial intelligence: an MIT perspective. Volume II: understanding vision, manipulation, computer design, symbol manipulation*. The MIT Press: Cambridge, Massachusetts.
- Winston, P.H.; Horn, B.K.P. [1991] *Lisp (3ª ed.)*. Addison-Wesley Iberoamericana: México (traducción al castellano de Winston, P.H.; Horn, B.K.P. [1989] *Lisp (3rd ed.)*. Addison-Wesley: Reading, Massachusetts.)

- Wirth, N. [1971a] «Program development by stepwise refinement», *Communications of the ACM* 14(4) abril: 221-227.
- Wirth, N. [1971b] «The design of a Pascal compiler», *Software-Practice and Experience* 1(4) abril: 309-334.
- Wirth, N. [1981] «Pascal-S: a subset and its implementation», en Barron [1981: 199-260]).
- Wirth, N. [1985] «From programming language design to computer construction» (ACM Turing Award lecture), *Communications of the ACM* 28(2) febrero: 160-164.
- Wirth, N. [1986a] *Algorithms and data structures*. Prentice-Hall International: London.
- Wirth, N. [1986b] «Microprocessor architectures: a comparison based on code generation by compiler», *Communications of the ACM* 29(10) octubre: 978-990.
- Wirth, N. [1988] *Programming in Modula-2*. Springer-Verlag: Berlin.
- Wirth, N. [1990] *Compiladores*. Rueda: Alcorcón (Madrid).
- Wirth, N.; Gutknecht, J. [1992] *Project Oberon. The design of an operating system and compiler*. ACM Press: New York.