Departamento de Arquitectura de Computadores
Universidad de Málaga

TESIS DOCTORAL

Detección Automática de Estructuras de Datos Basadas en Punteros

Francisco Javier Corbera Peña

Málaga, Septiembre de 2001

Dr. D. Rafael Asenjo Plaza
Profesor Titular del Departamento
de Arquitectura de Computadores
de la Universidad de Málaga

Dr. D. Emilio López Zapata
Catedrático del Departamento de
Arquitectura de Computadores de
la Universidad de Málaga

CERTIFICAN:

Que la memoria titulada *"Detección Automática de Estructuras de Datos Basadas en Punteros"*, ha sido realizada por D. Francisco Javier Corbera Peña bajo nuestra dirección en el Departamento de Arquitectura de Computadores de la Universidad de Málaga y concluye la Tesis que presenta para optar al grado de Doctor en Ingeniería Informática.

Málaga, 25 de Julio de 2001

Fdo: Dr. D. Rafael Asenjo Plaza
Codirector de la Tesis Doctoral

Fdo: Dr. D. Emilio López Zapata
Codirector de la Tesis Doctoral
Director del Departamento de
Arquitectura de Computadores

A Ana

Agradecimientos

El trabajo que culmina con la realización de esta memoria, ha sido posible gracias a la ayuda y el apoyo de muy diversas formas, de muchas personas a las cuales doy mi más sincero agradecimiento.

En primer lugar, agradecer a mis directores de tesis, Emilio L. Zapata y Rafael Asenjo Plaza, la confianza depositada en mi y los esfuerzos realizados para guiarme a lo largo del desarrollo de esta tesis.

A los compañeros del departamento de Arquitectura de Computadores de Málaga, Andrés Rodriguez, Mario González, Julián Ramos, Eladio Gutiérrez, Sergio Romero, Mª Antonia Trenas, Eva Martínez, Gerardo Bandera, Javier Hormigo, José Mª González, Manolo Ujaldón, Mª Ángeles González, Felipe Romero, Oswaldo Trelles, Nicolás Guil, Manuel Sánchez, Óscar Plata, Julio Villalba, Juan López, Guillermo Pérez, Pablo Pérez, Olga Pérez, Sonia González, Mª Carmen Solís, Magdalena Álvarez y Filo Artacho, he de agradecerles que me hayan dado su apoyo y brindado su amistad, al igual que Ezequiel Herruzo y Gonzalo Cerruela, actualmente en la universidad de Córdoba.

A mis padres Francisco y Mercedes quiero agradecerles su sacrificio para que haya podido llegar hasta aquí, y sobre todo por su cariño y comprensión. Es totalmente imposible pagarles todo aquello que han hecho por mi, sólo quiero decirles, muchas gracias.

Al resto de mi familia: Isa, Adri, Santi, Francisco, María, Santi jr., Juan, Ana, Agustín, Rocío, Juan Eugenio, Inma, ..., quiero expresarles mi gratitud y cariño por hacerme sentir querido y arropado. Quiero hacer una mención especial para mis abuelos María y Paco, y a la memoria de Encarna y Antonio.

Agradecemos el soporte y la financiación de este trabajo a los proyectos TIC96-1125-C03 de la CICYT, BRITE-EURAM III BE95-1564 de la Unión Europea, y APART (Automatic Performance Analysis: Resources and Tools) del EU Esprit IV Working Group No. 29488.

Por último, sobre todo quiero agradecer a Ana su amor, cariño y apoyo, que han sido en muchos momentos mi único salvavidas. Por soportar siempre con una sonrisa tantas horas bajas y hacer más especiales los buenos momentos. Realmente sin ella no hubiera sido posible este trabajo.

Índice

Índice de Figuras

Índice de Tablas

Prefacio

Debido al claro incremento del coste computacional de las aplicaciones actuales, existe una gran demanda de arquitecturas cada día más rápidas y con mejores prestaciones. En esta evolución de los computadores, las arquitecturas paralelas son un paso inevitable para la consecución de mayores velocidades de procesamiento. Como consecuencia, los modelos y técnicas de programación también han evolucionado para cubrir el desarrollo de algoritmos para estas nuevas plataformas paralelas. Sin embargo, es un hecho conocido, que la evolución de los compiladores para el desarrollo de algoritmos paralelos sigue un menor ritmo que el de estas arquitecturas. Esta situación impide en gran medida el uso generalizado de las máquinas paralelas existentes.

Un compilador no sólo simplifica la tarea de escribir y leer programas, sino que además proporciona portabilidad entre distintas plataformas. Sin embargo estas ventajas no deben suponer una significativa disminución en la eficiencia del código ejecutable. La coexistencia de estos tres aspectos dio lugar a que desapareciese progresivamente la programación en ensamblador, en una evolución hacia el uso intensivo de los lenguajes de alto nivel. Actualmente, la programación a mano de códigos paralelos es el equivalente de lo que en aquellos tiempos era la programación en ensamblador. Los paralelizadores automáticos intentan ser aquellos compiladores que poco a poco eliminen la paralelización manual, de forma que los usuarios puedan seguir escribiendo sus programas en algún lenguaje de alto nivel y se genere un ejecutable eficiente para cualquier plataforma independientemente de su arquitectura.

Básicamente un paralelizador automático es un traductor, que partiendo de un código secuencial, es capaz de extraer el paralelismo y producir un nuevo código, para una arquitectura paralela determinada. La transformación consta de dos etapas. En la primera se identifican las partes del código que son susceptibles de ser ejecutadas en paralelo. La segunda, tras la identificación de las zonas paralelas, consiste en la generación del código que se ejecutará sobre el multiprocesador. En la primera etapa es fundamental el uso de buenas técnicas de análisis de dependencias, que permitan discernir si las acciones pueden o no ejecutarse concurrentemente. Se han desarrollado técnicas bastante potentes para el análisis de dependencias en códigos regulares, que utilizan principalmente arrays como estructuras de datos. Sin embargo no existen técnicas eficaces para el análisis de dependencias de códigos que utilizan estructuras de datos dinámicas, por lo que los paralelizadores automáticos actuales no las soportan.

Este tipo de estructuras ya no pueden dejarse de lado, debido a la gran aceptación que en los últimos años están teniendo lenguajes de programación como C, C++ o Java, que las utilizan asiduamente, así como su importancia en el desarrollo y eficiencia de aplicaciones irregulares y simbólicas. Un primer paso para el soporte de dichas estructuras por los paralelizadores automáticos es el desarrollo de análisis de dependencias eficaces para los accesos a los datos mediante punteros. El análisis de estos accesos es bastante más complicado que el de los accesos a estructuras tipo array, en las que se sabe en tiempo de compilación la forma en que los datos son almacenados en la estructura. Para la estructuras de datos dinámicas

se desconoce la forma que tomarán dichas estructuras puesto que son creadas y modificadas en tiempo de ejecución. Los análisis de forma de estructuras dinámicas pretenden descubrir en tiempo de compilación las características fundamentales de dichas estructuras para que sean tenidas en cuenta a la hora de hacer el análisis de dependencias. El trabajo de esta tesis se centra en el desarrollo de nuevas técnicas para el análisis de forma de las estructuras de datos dinámicas. La memoria de este trabajo se organiza en cinco capítulos cuyo contenido se resume a continuación.

En el capítulo 1 se presentan las distintas técnicas existentes actualmente en el área del análisis de los accesos a estructuras dinámicas de datos. Presentamos aproximaciones basadas en anotaciones explícitas del programador describiendo las estructuras, otras basadas en calcular las relaciones de las variables puntero entre sí, y por último las basadas en tratar de describir de algún modo la forma de las estructuras de datos apuntadas por los punteros. De entre estas, destacaremos las basadas en *grafos* por ser las que presentan un marco más adecuado para el análisis de forma.

La mejora de una de estas aproximaciones basadas en grafos es presentada en el capítulo 2. En concreto, hemos extendido el método de Sagiv [67] que presenta unas características muy buenas para el análisis de forma, de modo que pueda soportar estructuras dinámicas más complejas, como por ejemplo la utilizada en la descomposición LU de una matriz dispersa basada en listas doblemente enlazadas. Dichas mejoras han sido implementadas y los resultados del análisis del tipo de estructura anterior son mostrados al final de este capítulo. Los trabajos publicados en relación con este capítulo se encuentran en [13, 17].

Un método nuevo basado en grafos es presentado en el capítulo 3, en vista de que los resultados obtenidos para códigos reales por el método anterior y por los otros métodos actuales son muy conservativos. Dicho método es implementado en un *pseudocompilador* que devuelve un conjunto de grafos describiendo las posibles formas de las estructuras de datos para cada sentencia del código. Con dicho *pseudocompilador* se han analizado una serie de códigos reales basados en complejas estructuras de datos dinámicas, obteniéndose descripciones muy precisas sobre las mismas. En [14, 15, 16, 18] encontramos la evolución del trabajo desarrollado en este capítulo.

En el capítulo 4 se presenta una extensión de nuestro método para el soporte de arrays de punteros dentro de las estructuras de datos. Este tipo de estructuras son muy comunes y ningún método hasta la fecha las soportaba. Una vez modificado el *pseudocompilador* para el soporte de estas nuevas estructuras, se han analizado de nuevo varios códigos reales basados en ellas, obteniéndose de nuevo resultados satisfactorios. Hasta donde nosotros conocemos, ninguna de las técnicas desarrolladas hasta la fecha para el análisis de la forma de las estructuras de datos dinámicas, es capaz de obtener resultados tan precisos para los códigos analizados en este trabajo. Los resultados relacionados con este capítulo serán enviados al ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation (PLDI).

Finalmente, en el capítulo de conclusiones, sintetizamos las principales aportaciones de esta tesis, así como enumeramos las principales líneas en las que seguiremos trabajando en el futuro.

1 Introducción

Los compiladores son herramientas que simplifican la traducción de programas escritos en lenguajes de alto nivel al lenguaje máquina usado por el computador. Los compiladores deben generar códigos eficientes y ocultar al programador los detalles de la arquitectura, por supuesto contando con la colaboración del sistema operativo. Estas herramientas permitieron la evolución de la programación en ensamblador hacia lenguajes de alto nivel mucho más amables con el programador e independientes de la máquina programada. Esta abstracción aleja a los programas de la eficacia que de ellos se espera, por lo que los compiladores no pueden ser meros traductores de código sino que deben obtener una traducción óptima, aplicando agresivas técnicas de optimización. Los compiladores para máquinas secuenciales han llegado a un grado de madurez bastante elevado, pero no se puede decir lo mismo de los compiladores paralelizadores que tratan de generar código paralelo a partir de uno secuencial.

Esta traducción de código secuencial a paralelo implica básicamente dos grandes tareas. La primera es identificar aquellas partes del código secuencial que tienen cualidades para poder ser ejecutadas en paralelo, y la segunda es la generación del código paralelo eficiente que se ejecute sobre un multiprocesador. Encuadrado en la primera tarea, está un buen análisis de dependencias que determine qué operaciones pueden realizarse concurrentemente y cuales no. Básicamente el análisis de dependencias trata de determinar si dos accesos distintos a los datos manejados por el código son independientes y por lo tanto pueden realizarse concurrentemente. Para poder obtener esta información del cuerpo del código, es fundamental conocer cómo se accede a los datos en el programa, y la forma de las estructuras que almacenan dichos datos. Históricamente los esfuerzos se han concentrado en determinar como se accede a los datos, ya que la forma en que son almacenados era conocida, puesto que asiduamente se han utilizado estructuras basadas en arrays, la estructura por excelencia de FORTRAN.

Pero el panorama ha cambiado debido a la amplia aceptación de lenguajes de programación como C, C++, Fortran90 o Java, donde encontramos, como uno de los elementos principales de acceso a los datos, al *puntero*. Es en este tipo de lenguajes donde los códigos irregulares y simbólicos han encontrado una herramienta fundamental, puesto que les permite la creación de todo tipo de estructuras de datos complejas. Estas estructuras ayudan a un rápido desarrollo de este tipo de códigos asi como a un aumento de rendimiento en su ejecución. A las estructuras basadas en punteros se las suele conocer con el nombre de *estructuras dinámicas* puesto que la forma de la estructura es desconocida en tiempo de compilación al ser creada y modificada en tiempo de ejecución. Para poder determinar cómo se accede a los

datos de este tipo de estructuras, es necesario un conocimiento previo de la manera en que son almacenados. En definitiva, el análisis necesita de una nueva fase que determine la *forma* de las estructuras de datos apuntadas por los punteros, para a partir de ella determinar si dos accesos distintos son independientes o no.

En la siguiente sección presentamos los problemas asociados a la utilización de punteros en los códigos, asi como diversas técnicas para intentar solucionarlos.

1.1 Dependencias de datos y punteros

Las tareas de optimización y paralelización llevadas a cabo por los compiladores, se basan en un conocimiento seguro sobre las referencias a memoria, determinando en tiempo de compilación si dos referencias dadas, siempre acceden a posiciones de memoria diferentes. Desafortunadamente, la presencia de alias en los programas hace de esta tarea un problema no-trivial. Un alias aparece en un programa cuando hay dos o más modos distintos de hacer referencia a una misma posición de memoria. Los elementos que introducen alias en los programas son los arrays, las variables puntero y las estructuras dinámicas basadas en punteros. Por ejemplo, las referencias a un array $a[i+2*j]$ y $a[j+2*i]$, las referencias con punteros $*q$ y $*p$, y los accesos a estructuras $p \rightarrow item$ y $q \rightarrow next \rightarrow item$, pueden acceder a la misma posición de memoria.

Se han desarrollado, con bastante éxito, potentes análisis de dependencias para resolver el problema de los alias en arrays [2, 74, 76], incluso en presencia de funciones no lineales en los accesos a los mismos [42] o accesos irregulares [36]. Este tipo de técnicas son el núcleo de los paralelizadores actuales como Polaris [7, 8], MIPSpro^{TM} Auto-Parallelizing [43] heredero de PFA [35, 34], Parafrase [32, 63], SUIF [37], etc. Sin embargo para punteros no existen resultados tan alentadores como los obtenidos para los códigos basados en arrays.

El análisis de punteros puede ser dividido en dos subproblemas distintos. El primero se centra en los punteros apuntando a objetos cuya memoria se ha reservado estáticamente, típicamente almacenados en la pila (*stack*), llamados *punteros al stack*. Por otro lado, está el problema de los punteros que apuntan a objetos cuya memoria se ha obtenido dinámicamente, denominados *punteros al heap*. Un puntero apunta a un objeto si contiene la dirección de memoria de dicho objeto. Así por ejemplo en la figura 1.1, el puntero p apunta al objeto x, y el puntero q al objeto y. Puesto que x e y están almacenados en el *stack*, los punteros p y q son *punteros al stack*. Por otro lado, los punteros r y s son *punteros al heap* puesto que apuntan a objetos creados dinámicamente durante la ejecución del programa.

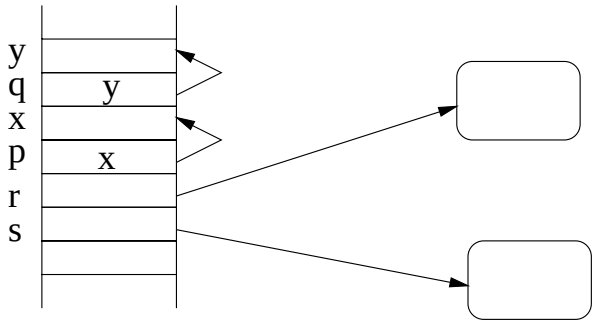


Figura 1.1: Ejemplo de *punteros al stack* y *punteros al heap*.

Los primeros métodos de análisis de punteros al *stack* se centraron en el cálculo de las relaciones de alias generadas por la asociación de los parámetros actuales con los parámetros pasados por referencia en las llamadas a procedimientos. Éste es un caso restringido de alias de punteros al *stack* típico de lenguajes como FORTRAN, en el que tan solo los puntos de llamada a procedimiento contribuyen a la creación de alias y las relaciones a la entrada de un procedimiento se mantienen durante todo su cuerpo. Técnicas para este tipo de alias han sido presentadas en [6, 3, 61, 11, 12], en donde en la mayoría de ellos, la información sobre los alias se obtiene en dos pasadas sobre el programa. Un primer paso calcula los alias obvios generados en los puntos de llamada, y un siguiente paso de propagación, pasa los alias a través del grafo de llamadas del programa.

Los punteros al *stack* presentan la propiedad, muy importante, de que sus destinos siempre poseen un nombre, ya que todos los objetos almacenados en el *stack* tienen nombre en tiempo de compilación. Usando esta propiedad, la información de alias para estos punteros puede ser descrita por medio de pares *apunta-a*. Por ejemplo, en la figura 1.1, existen los pares (p, x) y (q, y), denotando que la variable p apunta al objeto x y la variable q al objeto y. Esta aproximación fue propuesta inicialmente por Emami y col [26] y es usada por otros algoritmos de análisis de punteros más recientes [64, 73, 71, 69].

Sin embargo, esta propiedad no se cumple para los objetos creados dinámicamente y almacenados en el *heap*. A dichos objetos tan solo se puede acceder a través de referencias como por ejemplo $*r$, $r \rightarrow item$, etc., donde r es un *puntero al heap*. No se puede usar un esquema de nombres simple para nombrar estos objetos puesto que puede ser creado un número potencialmente infinito de ellos.

Existen dos aproximaciones al problema del análisis de los *punteros al heap*. La primera aproximación consiste en asociar a cada objeto del *heap* algún conjunto de nombres estáticos, y usar el mismo *framework* que el usado para el análisis de *punteros al stack*. Nos podemos encontrar con esquemas de nombres tan simples como usar un único nombre, *heap*, para nombrar a todos los objetos almacenados en el *heap* [26]. Otros más precisos usan como nombres de los objetos los puntos del código donde fue reservada la memoria [55], o estos mismos puntos concatenados con un nombre de procedimiento, representando el contexto actual de llamada a procedimiento [10, 73]. Para la figura 1.1 se podrían obtener los pares *apunta-a* $(r, heap)$ y $(s, heap)$ para el primer caso y $(r, malloc_U)$ y $(s, malloc_V)$ para el segundo, si la memoria del objeto apuntado por r hubiera sido reservada en la sentencia U y la de s en la V.

Sin embargo, tratar el *heap* como un conjunto de posiciones de memoria nombradas estáticamente hace que el análisis sea muy impreciso, ya que un simple nombre va a representar varios objetos del *heap* completamente distintos y sin relación alguna.

Por este motivo, la otra aproximación al análisis de *punteros al heap* ha sido desarrollar un análisis especial para este tipo de punteros distinto al del *stack*. En esta nueva aproximación, a diferencia de la anterior en la que se intentan identificar los posibles destinos de un puntero dado (*apunta-a*), se pretende descubrir las relaciones abstractas entre los propios punteros dirigidos al *heap*.

El objetivo del análisis del *heap* es determinar estáticamente, en tiempo de compilación, la *estructura del heap en cada punto del programa*. Este tipo de análisis debe de ser capaz de responder a la pregunta de si dos referencias del programa pueden apuntar a una misma posición de memoria en el *heap*. A la luz de los resultados teóricos que establecen que el cálculo de relaciones de alias precisos por medio de un análisis estático es indecidible [54], no

se espera que el análisis obtenga información totalmente precisa, sino que se producirán errores conservativos indicando que dos referencias pueden acceder a la misma posición de memoria cuando realmente pueden hacerlo o no en tiempo de ejecución. Por tanto la información útil que se obtendrá de estos análisis será la de carácter "negativo", en el sentido que si indican que dos referencias *no pueden* apuntar al mismo sitio, dichas referencias en tiempo de ejecución apuntarán a posiciones de memoria distintas.

Tenemos que comentar que existen otra serie de análisis de objetos en el *heap* más centrados en otros problemas como la cuenta de referencias de cada posición o el tiempo de "vida" de las posiciones de memoria [45, 50, 65, 20, 72], enfocados al problema de la "recolección de basura" (*garbage collect*).

1.1.1 Análisis de punteros al HEAP

Como ya hemos comentado, este tipo de análisis debe determinar en tiempo de compilación la estructura del *heap* en cada punto del programa, para poder establecer si dos referencias distintas pueden o no acceder a la misma posición de memoria. Se han propuesto diferentes métodos para resolver este problema.

Anotaciones explícitas

Estás técnicas no se pueden encuadrar dentro de las técnicas de análisis automáticas, puesto que se basan en la obtención de información sobre las estructuras por medio del programador. En ellas, el programador, de alguna forma, va a describir explícitamente las propiedades de las estructuras dinámicas que serán usadas en el código, de manera que posteriormente el compilador sea capaz de hacer un análisis de alias preciso.

Hendren, Hummel y Nicolau [41, 46], presentan un mecanismo llamado *ADDS* (Abstract Description of Data Structures), para expresar explícitamente las propiedades de *dimensión* y *dirección* de las estructuras de datos. Esto implica añadir alguna información semántica a las declaraciones de tipo de las estructuras de datos. Por ejemplo, esta sería la declaración de tipo de una lista doblemente enlazada utilizando *ADDS*:

```
type ListaD [X]
{
  int data;
  ListaD *nxt is uniquely forward along X;
  ListaD *prv is uniquely backward along X;
};
```

Esta descripción informa al compilador que si la lista es recorrida usando tan solo enlaces por *nxt*, nunca se visitará dos veces el mismo elemento de la lista.

Hummel, Hendren y Nicolau [47, 48], presentan una aproximación más formal para representar las propiedades de alias de las estructuras de datos. Proponen un lenguaje, *ASAP* (Abstract Specification of Aliasing Properties), basado en *expresiones regulares*, que captura las propiedades de alias en forma de axiomas aplicables a las definiciones de tipo de las estructuras de datos. El lenguaje de descripción *ADDS* es intuitivo, pero no puede representar de forma precisa un importante número de estructuras de datos y no soporta fácilmente un método directo de análisis de dependencias. Con esta nueva aproximación, las relaciones de alias de una lista doblemente enlazada pueden expresarse como:

$\forall$ p, p.nxt <> p.ϵ

$\forall$ p, p.prv <> p.ϵ
$\forall$ p, p.nxt.prv = p.ϵ

que indica que siguiendo el enlace *nxt* o *prv* se llega a otro elemento de la lista distinto del actual, pero siguiendo consecutivamente los enlaces *nxt* y *prv* se alcanza de nuevo el elemento de partida.

Las descripciones en *ADDS* pueden ser traducidas a axiomas fácilmente. También presentan un test de dependencias de propósito general para estructuras de datos dinámicas, que se basa en un *demostrador de teoremas* que usa la información proporcionada por los axiomas de las estructuras de datos.

Este tipo de técnicas van a ser descartadas puesto que nuestro objetivo es la extracción automática de las características de las estructuras de datos dinámicas, y estos métodos necesitan de la intervención explícita del programador.

Relaciones entre variables puntero al HEAP

Estos métodos se basan en describir de alguna forma las relaciones existentes entre las variables puntero *al heap* en cada punto del programa.

Deutsch [21, 22] calcula las relaciones de alias entre las variables que apuntan al *heap* en forma de pares de *accesos simbólicos*. Un *camino de acceso simbólico* (*SAP*, Simbolic Access Path) es un camino de acceso que puede contener expresiones simbólicas de la forma B^k, donde B es un conjunto de caminos de acceso, llamado *base* y k es una variable. Así, por ejemplo, el $SAP\ X \rightarrow (nxt)^i \rightarrow hd$, tiene como base nxt. Este SAP, parametrizado en i, representa de forma finita un infinito número de caminos de acceso desde la cabeza de una lista hasta el campo hd de sus nodos.

Un par alias, en este caso, consistirá en un par de caminos de acceso simbólico relacionados por una ecuación. Así, una dependencia del tipo "el elemento i-ésimo de la lista X es alias con el elemento $2i+1$-ésimo de la lista Y", se expresaría como $(< X \rightarrow (nxt)^i, Y \rightarrow (nxt)^j >, j = 2i+1)$.

Aunque es un *framework* bastante completo, el manejo de las expresiones simbólicas es muy complejo y no está claro si es suficientemente práctico para su implementación en un compilador real.

Matsumoto y col. [59] calculan las relaciones de alias entre variables puntero por medio de *cadenas normalizadas de acceso*. Por ejemplo, una referencia del tipo $P \rightarrow nxt \rightarrow nxt \rightarrow ptr \rightarrow nxt \rightarrow nxt$ quedaría representada por la cadena normalizada $P \rightarrow ptr \rightarrow nxt$. Para construir las cadenas normalizadas, de la referencia original se eliminan las referencias consecutivas del mismo tipo. Así en el ejemplo, el puntero P y nxt son de un mismo tipo mientras que ptr es de otro, por lo que $P \rightarrow nxt \rightarrow nxt$ se queda tan solo representada por P.

Proponen un algoritmo de análisis de alias basado en estas cadenas, además de mejoras sobre el mismo mediante la detección de listas enlazadas y estructuras tipo árbol. Este método es menos potente que el anterior al utilizar una simplificación para los accesos al *heap* mientras que el primero utiliza expresiones simbólicas que se adecuan bastante a la descripción de estructuras recursivas.

Ghiya y Hendren [28] proponen un método para determinar si dos punteros apuntan a la misma estructura enlazada o a regiones totalmente disjuntas del *heap* no conectadas.

Para ello utilizan la abstracción *matriz de conexión*, C, que es una matriz booleana con una fila y columna por cada variable puntero, que almacena la información de conexión entre ellas. Así, si p y q son dos variables puntero, $C[p,q] = 1$ indica que los punteros p y q posiblemente apunten a objetos del *heap* pertenecientes a la misma estructura, es decir p y q están conectados. Por otro lado si $C[p,q] = 0$, los objetos apuntados por p y q definitivamente pertenecen a diferentes estructuras de datos (no están conectados). Esta información es útil para distinguir accesos a estructuras completamente distintas.

Este último método es mucho más simple que los anteriores, ya que tan solo está enfocado a distinguir rápidamente accesos a estructuras de datos disjuntas. Si $C[p,q] = 1$, hay que aplicar otros métodos más complejos para determinar relaciones más precisas entre p y q.

Descripciones de las estructuras apuntadas por los punteros

Por último, estos métodos van a tratar de describir, de diversas formas, el tipo de estructuras a las que apuntan las variables puntero, para poder deducir sus características principales y utilizarlas para determinar si distintos accesos a las mismas pueden conducir a una misma posición de memoria.

Hendren y Nicolau [38, 39] abstraen las propiedades de las estructuras de datos de los programas, identificando propiedades regulares en las mismas, como si son estructuras tipo árbol o grafos dirigidos acíclicos (DAG, Directed Acyclic Graph). Utilizan lo que denominan, *path matrix analysis*, donde una *path matrix*, P, es una matriz de punteros *al heap*. Cada entrada de dicha matriz, $P[r,s]$ contiene una expresión regular a base de enlaces, denominada *path expression* que sintetiza todos los caminos de acceso desde el objeto apuntado por r al apuntado por s. Si el valor de $P[r,s] = \{right^1 left^+\}$, el objeto apuntador por s puede ser alcanzado desde el objeto apuntado por r, tomando su campo puntero $righ$, seguido de un número indeterminado de enlaces por punteros $left$.

Con la información del *path matrix* determinan si la estructura apuntada por cada puntero es un árbol o un DAG. En caso de que las estructuras sean de tipo árbol, aplican un análisis de interferencias basado en esta información.

Este método es efectivo para árboles y de forma muy limitada para algún tipo de DAG, pero sin embargo no puede trabajar con estructuras cíclicas.

Ghiya y Hendren [29] se centran en estimar la forma de la estructura de datos accesible desde un puntero dado, determinando si es una estructura tipo árbol, DAG o un grafo general conteniendo ciclos. Para ello utilizan tres abstracciones simples: La *Matriz de Dirección*, D, es una matriz booleana que informa de la posible existencia de un camino en la estructura de datos entre el objeto apuntado por un puntero y el apuntado por otro. Así, $D[p,q] = 1$ indica que es posible la existencia de un camino entre el objeto apuntado por p y el apuntado por q. Si $D[p,q] = 0$ no existe ningún camino entre los objetos apuntados por p y q. La *Matriz de Interferencias*, I, es otra matriz booleana que indica si una posición del *heap* puede ser visitada desde dos punteros distintos. Si $I[p,q] = 1$, puede existir un objeto en el *heap* accesible desde caminos que comienzan en los punteros p y q. Si vale 0, no hay ningún objeto accesible desde ambos punteros. A cada variable puntero *al heap* se le asocia un *atributo de forma* que indica la posible forma de la estructura de datos accesible desde dicho puntero. Este atributo de forma puede tomar tres valores, *TREE* si el puntero apunta a una estructura tipo árbol, *DAG* si lo hace a un DAG, y *CYCLE* si apunta a un grafo general.

Las relaciones de dirección son usadas para estimar los atributos de forma de los punteros,

mientras que las relaciones de interferencia se usan para determinar de forma segura las relaciones de dirección.

Posteriormente, **Ghiya y Hendren** [27, 30] integran todo un conjunto de métodos para implementar un análisis de alias jerárquico (tanto al *stack* como al *heap*) en el compilador de código C, *McCAT* [40]. Así, si el análisis de alias *al stack* que utilizan [25, 26] indica que no hay punteros que apunten *al heap*, no se realiza el análisis dirigido al *heap*. En otro caso, llevan a cabo un análisis simple y rápido, el análisis de conexión [28], que indica si dos punteros pueden apuntar o no a una misma estructura. Si los resultados obtenidos siguen siendo conservativos, se aplica el análisis más avanzado y costoso sobre la forma de las estructuras [29], para determinar si tiene forma de árbol, de *DAG* o contiene ciclos.

Estos análisis están muy enfocados a su implementación en un compilador real (*McCAT*), de manera que aún el análisis más avanzado que llevan a cabo, [29], basado en las relaciones de dirección y de interferencia, es bastante simple, con el objeto de ser realizado de una manera rápida por el compilador. Esto hace que el método se comporte de manera muy conservativa, informando que estructuras doblemente enlazadas muy comunes son cíclicas, en cuyo caso es incapaz de manejarlas con exactitud.

Existen otros métodos que proponen *frameworks* parametrizables para llevar a cabo distintos análisis de programas con estructuras de datos recursivas.

Así, **Jones y Muchnick** [52] proponen un *framework* flexible, donde representan los puntos en los cuales se crean o modifican estructuras dinámicas, mediante *tokens*. Los *tokens* pueden considerarse como representaciones locales de las estructuras de datos en dichos puntos del programa. Definen entonces una función que representa de forma finita las relaciones entre los *tokens* y los valores de los datos. La definición de esta función se basa en la simulación de las sentencias del programa, usando una interpretación abstracta [19]. El análisis es parametrizable por la elección del conjunto de *tokens*.

Sagiv y col. [68] presentan una familia de interpretaciones abstractas que son capaces de determinar las invariantes de forma de las estructuras. Ellos representan los objetos en el *heap* mediante estructuras utilizando una lógica tri-valuada. Asocian a cada objeto de las estructuras un conjunto de *predicados lógicos*, de manera que las cuestiones sobre los objetos almacenados se pueden contestar evaluando los predicados usando una lógica tri-valuada con la semántica de Kleene [53]. Si la fórmula toma el valor *true*, entonces se cumple para todos los elementos representados por la estructura. Si toma el valor *false*, no se cumple para ninguno de dichos elementos y si toma el valor *unknown*, no se sabe si se cumple para todos, para alguno o para ninguno de los elementos representados por la estructura. Así, por ejemplo, $x(v)$ indica que la variable x apunta al objeto v. La fórmula que expresa la propiedad de que las variables x e y no son alias sería: $\forall v : \neg(x(v) \wedge y(v))$.

Por medio de una interpretación abstracta del programa van calculando las estructuras tri-valuadas que representan a las estructuras de datos dinámicas que pueden aparecer en cada punto de la ejecución del programa. Es un *framework* en el cual se pueden "instanciar" otros métodos de análisis, eligiendo y definiendo de forma adecuada los predicados que definen las estructuras. Sin embargo, los predicados que proponen y que implementan en [58], son útiles para estructuras de datos y códigos mucho más simples de los que nosotros presentaremos en esta memoria. En [58] presentan una implementación del método presentado en [68], mostrando algunos resultados obtenidos en búsquedas, inserciones, borrados e intercambio de elementos en listas, pero no de ningún algoritmo que siga la estructura de algún problema real. En [57] presentan nuevos predicados para determinar automáticamente la corrección parcial

de programas y poder detectar errores de programación. Para ello introducen predicados "*en orden*" que informan si los objetos accesibles desde uno dado por un determinado enlace siguen un orden determinado. En base a estos predicados prueban la corrección de un procedimiento que manipula una lista ordenada, viendo si dicho predicado se mantiene tras la ejecución del procedimiento. No proponen ningún predicado nuevo en cuanto a la detección de nuevas estructuras de datos con respecto a los trabajos anteriores.

Por último, dentro de este tipo de aproximaciones que tratan de describir la forma de las estructuras de datos apuntadas por los punteros de un programa, nos encontramos con una familia de métodos que tienen en común la utilización de grafos para describir la forma de las estructuras de datos.

Métodos basados en grafos

Como hemos comentado, estos métodos utilizan grafos para describir las estructuras de datos que pueden aparecer en cada sentencia de un programa. Los nodos de los grafos representarán de alguna manera a los objetos almacenados en el *heap* y las aristas de los grafos representarán las referencias entre los objetos mediante punteros.

Un primer ejemplo de este tipo de técnicas fue presentado por **Jones y Muchnick** [51] en el que analizan las estructuras dinámicas similares a las creadas por LISP, creadas por un lenguaje simple sin procedimientos. Abstraen las estructuras del *heap* en cada punto del programa por medio de un conjunto de grafos. Los nodos de dichos grafos representan objetos del *heap* mientras que las aristas representan los enlaces entre ellos. Los nodos apuntados por variables son etiquetados por el nombre de las variables. Los nodos que pueden tener más de otro nodo apuntándole o pueden ser parte de un ciclo son etiquetados como *shared* o *cyclic*. Como cada sentencia del programa va a transformar los grafos de sus sentencias predecesoras, aquellas sentencias en que confluyen varios flujos utilizan la unión de conjuntos para generar la representación de las estructuras a la entrada de dichas sentencias. Puesto que los grafos abstraen estructuras recursivas, pueden tener un número ilimitado de nodos. Para evitar la construcción de grafos infinitos usan la noción de *k-limitación*, por la cual, todos los nodos del grafo accesibles desde una variable después de recorrer k o más enlaces, son colapsados en un nodo *sumario*. En la figura 1.2 podemos ver un ejemplo de una lista enlazada con una *2-limitación*.

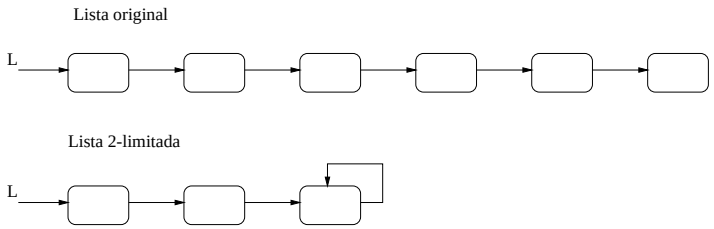


Figura 1.2: Ejemplo de lista *2-limitada*.

Las variables que aparecen en los grafos pueden apuntar a nodos que no son *shared* ni *cyclic*, a nodos que pueden ser *shared* pero no *cyclic* o a nodos *cyclic*. La memoria de los objetos del *heap* accesibles por medio de las primeras puede ser liberada cuando los punteros que las referencian son destruidos, las accesibles por las segundas necesitan un contador de referencias para determinar cuando pueden ser liberadas y los de las terceras necesitan de una

fase de *recogida de basura (garbage collect)*.

El problema de esté método es la pérdida de información de los objetos más allá del límite k, lo que provoca que el análisis sea bastante conservativo más allá de dicho límite, además de estar muy limitado por el tipo de estructuras que soporta el lenguaje analizado.

Por otro lado, **Larus y Hilfinger** [56] usa una variación de los grafos *k-limitados* llamada *grafos de alias*, para analizar programas en LISP. Las aristas del grafo son etiquetadas con el nombre de los campos punteros por los que comienzan los enlaces representados en la arista. Además etiquetan los nodos o con un nombre de variable o por medio de un *path expression*. Estos *path expression* son expresiones regulares que resumen los posibles caminos de acceso desde una variable al nodo. A diferencia de Jones y Muchnick [51], tan solo mantienen un *grafo de alias* para cada punto del programa en vez de un conjunto de grafos. Definen un operador unión que combina dos grafos de alias en uno nuevo que contendrá todos los alias de los grafos unidos.

De nuevo, para mantener el tamaño finito de los grafos, introducen nodos sumarios utilizando una *s-l-limitación*. En este tipo de limitación, ningún nodo tiene más de s aristas de salida, y ninguno tiene una etiqueta mayor que l (longitud del *path expression*). Una vez construidos los grafos, existe un potencial conflicto si los caminos de acceso en una determinada sentencia pueden conducir a un nodo con la misma etiqueta en sus respectivos grafos de alias.

Este método es eficaz pero sólo para estructuras simples como árboles y listas, pero su operación de unión, la sumarización de nodos y la operación de etiquetado de los nodos son bastante complejas.

Horwitz, Pfeiffer y Reps [44] presentan otra variación de grafos *k-limitados*, llamados *storage graphs*. El objetivo del análisis es determinar las dependencias entre sentencias del programa, por lo que en cada grafo, los nodos son etiquetados usando la última sentencia del programa que estableció un valor en los mismos. Así, fácilmente una sentencia S es dependiente de una sentencia T, si S lee una posición cuya abstracción en el *storage graph* está etiquetada con T. Para cada punto del programa mantienen un conjunto de grafos *k-limitados*. Este método también se presenta ineficaz a la hora de analizar estructuras un poco complejas como estructuras doblemente enlazadas.

Otra aproximación basada en grafos es la presentada por **Chase, Wegman y Zadeck** [9]. Su abstracción se basa en lo que denominan *Storage Shape Graphs*, que contienen un nodo por cada variable puntero y otro por cada punto del programa donde se reserva memoria. Ahora no existe una *k-limitación*, sino que el número de nodos está limitado por los factores descritos anteriormente. Por tanto, si todos los elementos de una lista son creados en un mismo punto, todos ellos serán representados por un nodo con un arco hacia él mismo. Para evitar interpretar este arco como una estructura cíclica, amplían su abstracción con una *cuenta de referencias* para cada nodo, donde nodos con valor menor o igual que uno representan estructuras tipo lista o árbol. En determinadas ocasiones, este método permite lo que se denomina *strong update*, que implica reemplazar todos los enlaces de un determinado nodo por otros nuevos, si se tiene la certeza de que las posiciones accedidas por una sentencia son exactamente las representadas por dicho nodo. Esta operación de *strong update* es muy deseable puesto que proporciona una modificación mucho más exacta de los grafos. Esta operación tan solo puede ser aplicada sobre nodos que representan a posiciones individuales del *heap*.

Este método utiliza un único *Storage Shape Graph* para representar el *heap* de cada senten-

cia del programa, por lo que definen una operación de unión de grafos, que resulta bastante compleja, pues intenta minimizar la creación de nodos *sumario* y tan solo une nodos que representan el mismo punto de reserva de memoria del programa.

Posteriormente, **Plevyak, Chien y Karamcheti** [62] extienden el modelo de Chase y col. [9] para poder manejar estructuras cíclicas regulares como listas doblemente enlazadas, con mayor precisión, creando los *Abstract Storage Graphs* (ASG). Introducen un tipo de nodos nuevos en los grafos llamados *nodos elección*, que representan a dos determinados enlaces apuntando a un nodo sumario los cuales no pueden existir al mismo tiempo. Además anotan cada nodo con *identity paths*, para indicar combinaciones de enlaces que pueden crear ciclos en la estructura. Mantienen la complejidad en la operación de unión de grafos como en [9] y de nuevo tan solo pueden llevar a cabo el *strong update* bajo unas determinadas condiciones.

El método presentado por **Sagiv, Reps y Wilhelm** [66, 67] se basa en lo que ellos denominan *Static Shape Graphs* (SSG), que al igual que los anteriores representa las posibles configuraciones del *heap* en un determinado punto de la ejecución del programa, con tan solo un grafo. La diferencia fundamental con los anteriores es el esquema de nombres que utiliza para nombrar los nodos. Nombran cada nodo con un conjunto (posiblemente vacío) de variables puntero que en ese punto en concreto del programa apuntan todas al mismo objeto del *heap*. Este esquema de nombres le proporciona una serie de características muy interesantes como son: i) una pista rápida sobre las relaciones de alias, ii) siempre realizan lo que denominan *strong nullification* que es la operación de *strong update* presentada en [9] y que sólo podían realizar bajo determinadas circunstancias. Al aplicar siempre esta operación se consigue mantener un mayor determinismo dentro de la representación (los enlaces existentes siempre son sustituidos por otros nuevos, como realmente ocurre en la ejecución de un programa). Esto es posible gracias a una operación de *materialización* por la cual, del nodo apuntado por un arco en un SSG, se crea un nuevo nodo que representa los objetos *realmente* apuntados por el enlace que representa el arco en una determinada sentencia.

Además, los nodos del grafo llevan asociada una información adicional *shared*, parecida a las *cuentas de referencias* de [9], informando si los objetos representados por un nodo pueden ser referenciados a la vez o no por otro objetos del *heap*. Gracias a la utilización permanente de la operación *strong nullification*, hay nodos que pueden pasar de ser *shared* a no serlo. Esta vuelta atrás no ocurre en el método de Chase [9], donde un nodo que se convierte en *shared* permanece así hasta el final, aunque los objetos que represente dejen de serlo.

De nuevo, puesto que se mantiene tan solo un grafo por cada sentencia, en la interpretación abstracta del programa, se va a necesitar la operación de unión de grafos. En este método, gracias al esquema de nombres utilizado, la operación de unión es bastante simple a diferencia de lo que ocurría en los métodos anteriores.

El método de Sagiv presenta unas características muy deseables en relación a los anteriores, sobre todo gracias a que su simple y a la vez robusto esquema de nombres y su operación de *materialización*, permiten que siempre se pueda llevar a cabo la eliminación de los enlaces existentes en un nodo para ser sustituidos por otros nuevos, preservando en buena medida la precisión de la representación. Aún así, dicho método no puede soportar estructuras muy comunes como son las doblemente enlazadas (listas doblemente enlazadas, árboles con punteros al padre, etc.).

Presentamos a continuación algunos conceptos, ya mencionados, relacionados con los métodos basados en grafos, para una mejor comprensión de cómo los grafos representan las estructuras dinámicas de datos que pueden aparecer en un programa.

1.2 Estructuras dinámicas y su representación mediante Grafos

Como ya hemos comentado, los métodos basados en grafos pretenden representar el estado de la memoria de cada sentencia de un programa en tiempo de ejecución, mediante grafos. El objetivo principal es determinar las relaciones que pueden existir entre distintos accesos a las estructuras de datos, para ver si distintas referencias podrán apuntar a una misma posición de memoria o no. El éxito de estos métodos depende de la precisión con la que sean capaces de capturar las propiedades de los objetos enlazados almacenados en el *heap*, ya que una pérdida de precisión hace que el análisis se comporte conservativamente, reportando muchas "posibles" relaciones.

Entre los distintos métodos basados en grafos presentados anteriormente, algunos de ellos utilizan tan solo un grafo para representar todas las configuraciones de la memoria para una determinada sentencia [56, 9, 62, 66, 67], mientras que otros utilizan múltiples grafos para la misma tarea [51, 44]. Los primeros tienen la ventaja de una representación más compacta y por tanto necesitan menos requerimientos a la hora de la implementación del método que los basados en múltiples grafos. Por el contrario, los segundos, al mantener más de un grafo, mezclan menos información sobre objetos del *heap*, lo que en un principio les capacita para una mayor precisión en la representación. Lo que pasa es que la precisión depende en un alto grado de la exactitud con la que los nodos representen los objetos del *heap*.

A partir de ahora, nos referiremos con **porción de memoria**, a un objeto en el *heap*, que no es más que un trozo de memoria de un determinado tipo reservado por una instrucción del tipo *malloc()* en C. Dentro de estas porciones de memoria hay sitio para campos que almacenan datos, y otros campos que son punteros a otras porciones de memoria. A este tipo de campos los denominaremos **selectores**.

Una determinada configuración de la memoria será un conjunto de porciones de memoria y los enlaces entre ellas por medio de selectores. Una posible configuración sería una simple lista enlazada como la mostrada en la figura 1.3(a), donde existe una variable puntero *list* que apunta a una porción de memoria que es la primera de una lista de cuatro porciones enlazadas por su selector *nxt*. Los métodos basados en grafos tendrán que representar cada configuración de memoria de tamaño desconocido en tiempo de compilación, mediante un grafo de tamaño finito, como por ejemplo el mostrado en la figura 1.3(b).

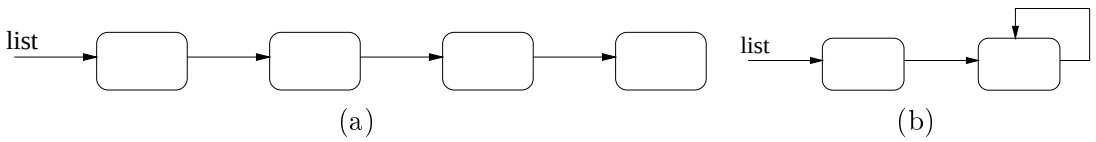


Figura 1.3: Lista enlazada (a) y un posible grafo que la representa (b).

Al tener que representar mediante un grafo finito configuraciones de memoria de tamaño desconocido, habrá nodos en el grafo que representen más de una porción de memoria de una misma configuración. Así, en el grafo de la figura 1.3(b), el segundo nodo está representando a las porciones segunda, tercera y cuarta de la lista de la figura 1.3(a). A este tipo de nodos los vamos a denominar nodos **sumario** puesto que representan a varias porciones de memoria a la vez. Los distintos métodos basados en grafos se distinguen en cómo se representan las porciones de memoria por los nodos. Así, están los métodos *k-limitados* y sus variantes [51, 56, 44], que sumarizan todas aquellas porciones de memoria alejadas de una

variable puntero en un mismo nodo. Otros sumarizan las porciones de memoria según en que punto del programa hayan sido creadas [9, 62], o según el conjunto de variables puntero que referencian a las porciones de memoria, sumarizando aquellas que son referenciadas por las mismas variables [66, 67]. Vemos que cada método tiene su propio sistema de **sumarización**, pero esta operación de sumarización es necesaria en todos para mantener el tamaño finito de los grafos, uniendo diversos nodos en uno solo que represente todas las porciones de memoria representadas por los nodos sumarizados.

Aparte de la forma en que las porciones de memoria son asociadas a los nodos, también es importante la información adicional que se asocia a los nodos para una mejor descripción y manejo de las porciones de memoria representadas en los nodos. Así, por ejemplo, en el grafo de la figura 1.3(b) se podría pensar que el segundo nodo, que posee un enlace sobre él mismo, está representando una estructura cíclica. Algunos métodos asocian algún tipo de información para resolver dudas como la anterior. Así en [51] se asociaban a cada nodo atributos de *shared* y *cyclic* para indicar si las porciones representadas en el nodo pueden ser apuntadas desde más de una porción o si forman parte de un ciclo. En [9] llevan una especie de cuenta de las referencias de entrada de las porciones de memoria representadas por un nodo con su atributo *reference counts*. En [66, 67] se añade la información *shared* a cada nodo que indica si las porciones de memoria representadas por el mismo pueden o no ser referenciadas más de una vez desde el *heap*. Este tipo de información sería la encargada de indicar al método, que el segundo nodo del grafo de la figura 1.3(b) representa o no una estructura con ciclos.

La elección del modo en que las porciones de memoria son representadas en los nodos y en definitiva, cómo se sumarizan los nodos, junto con la información adicional asociada a cada nodo, van a determinar la precisión del análisis. Aparte, el modo en que son utilizadas estas propiedades a la hora de interpretar los programas, modificando los grafos, será determinante para que la precisión se mantenga y el análisis no tenga que ser demasiado conservativo. En este sentido, el método de Sagiv y col. [66, 67] es el que mejor se comporta puesto que permite que, en cualquier caso, siempre se pueda llevar a cabo lo que denominan *strong nullification*. Esto significa que cuando se crea un enlace nuevo en una porción de memoria, se elimine el enlace existente en el nodo que la representa y sea sustituido por el nuevo. ¿Donde está pues el problema en esta operación? El problema es que para eliminar un enlace de un nodo y sustituirlo por uno nuevo, el método tiene que estar seguro de que el nodo representa única y exclusivamente a las porciones de memoria en las que la sentencia cambia el selector, y no está representando otras sumarizadas con las anteriores. Si esto no se puede determinar, el análisis tiene que ser conservativo, y debe mantener el enlace antiguo además del nuevo, ya que el nodo puede estar representando porciones de memoria no modificadas por la sentencia. Así por ejemplo, el efecto de la sentencia $list \rightarrow nxt = aux$ sobre el grafo de la figura 1.4(a) es distinto si se realiza la operación *strong nullification* (figura 1.4(b)) del obtenido si no se realiza (figura 1.4(c)) .

Como vemos con la operación *strong nullification* el enlace existente en el nodo apuntado por $list$ es sustituido por uno nuevo hacia el nodo apuntado por aux. Si no se puede llevar a cabo esta operación, el análisis tiene que ser conservativo y mantener en el nodo apuntado por $list$ tanto el enlace antiguo como el nuevo al nodo apuntado por aux. Como se ve es una operación muy deseable, y el método de Sagiv puede aplicarla siempre.

Esta característica del método de Sagiv, es posible gracias al esquema de nombres que utiliza, descrito anteriormente, y a que realiza una operación de **materialización** de nodos nuevos a partir de nodos sumarios. La materialización consiste en la creación de un nuevo

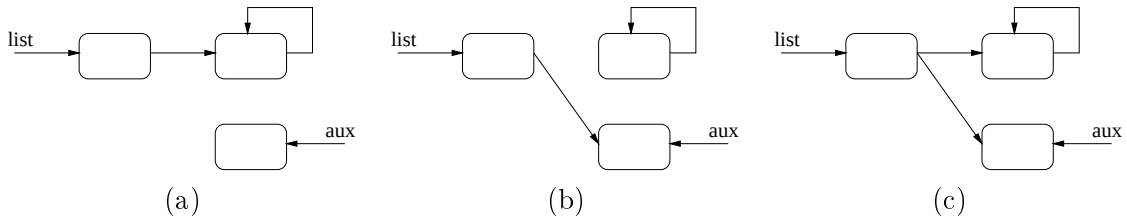


Figura 1.4: Efecto de la sentencia $List \rightarrow nxt = aux$ sobre un grafo (a), con *strong nullification* (b) y sin ella (c).

nodo a partir de un nodo existente, de forma que el nuevo nodo represente única y exclusivamente a determinadas porciones de memoria (referenciadas por un determinado enlace) separándolas del resto de porciones de memoria representadas en un nodo. El nuevo nodo tendrá características y enlaces similares a las del nodo del cual se materializa. La información adicional asociada a cada nodo es fundamental a la hora de que la materialización sea lo más precisa posible. Por ejemplo, la sentencia $aux = list \rightarrow nxt$ sobre el grafo de la figura 1.5(a) produce una materialización para separar del nodo sumario, que es apuntado desde el nodo referenciado por $list$, aquellas porciones de memoria que realmente son referenciadas por $list \rightarrow nxt$ de aquellas que están representadas por el mismo nodo pero no lo son. La figura 1.5(b) muestra el efecto de la sentencia usando la materialización mientras que en la figura 1.5(c) se muestra el efecto sin utilizarla.

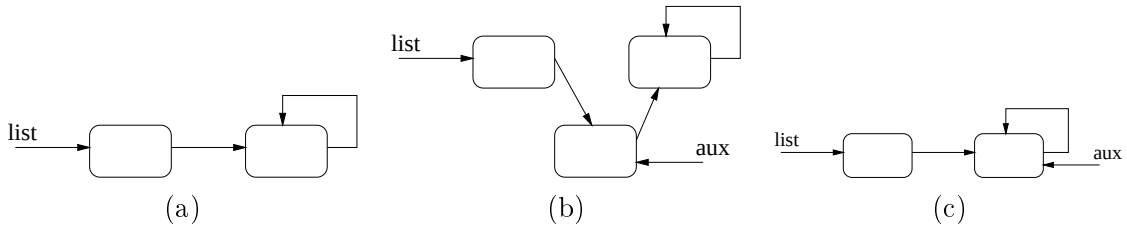


Figura 1.5: Efecto de la sentencia $aux = List \rightarrow nxt$ sobre un grafo (a), con *materialización* (b) y sin ella (c).

Vemos como la operación de materialización va a permitir un análisis más preciso puesto que se van a aislar en nodos distintos aquellas porciones de memoria realmente implicadas en una sentencia, de aquellas que no lo están, pero estaban representadas en un mismo nodo. Esto permitirá que las modificaciones de los enlaces del grafo y de la información adicional de cada nodo se ajusten mejor a la operación real sobre las porciones de memoria.

Bueno, una vez visto como se pueden representar un número indeterminado de posiciones de memoria por un grafo finito, estos métodos de alguna manera van a llevar a cabo una *ejecución simbólica* del código sobre los grafos para obtener el grafo o los grafos que representan las configuraciones de memoria que pueden aparecer en cada sentencia. Debido a que este análisis es estático, en tiempo de compilación se tienen que considerar todos los posibles caminos de ejecución según el grafo de flujo de control. Por este motivo, los grafos asociados a cada una de las sentencias van a tener que representar distintas configuraciones de memoria pertenecientes a distintos caminos del flujo de control. Por ejemplo, el siguiente trozo de código muestra la creación de una lista enlazada por el selector nxt y apuntada por la variable $list$:

...

```
1     list  = malloc();
2     ant =  list ;
3     while (cond)
      {
4       new = malloc();
5       new→ data = value;
6       new→ nxt = NULL;
7       ant→ nxt = new;
8       ant = new;
9       new = NULL;
      }
...
```

En tiempo de compilación, sin ninguna información sobre la condición del bucle de la sentencia 3 (*cond*), las posibles configuraciones que pueden aparecer en la sentencia 9 serían listas enlazadas apuntadas por *list* de longitud indefinida. Algunas de estas posibles configuraciones se presentan en la figura 1.6.

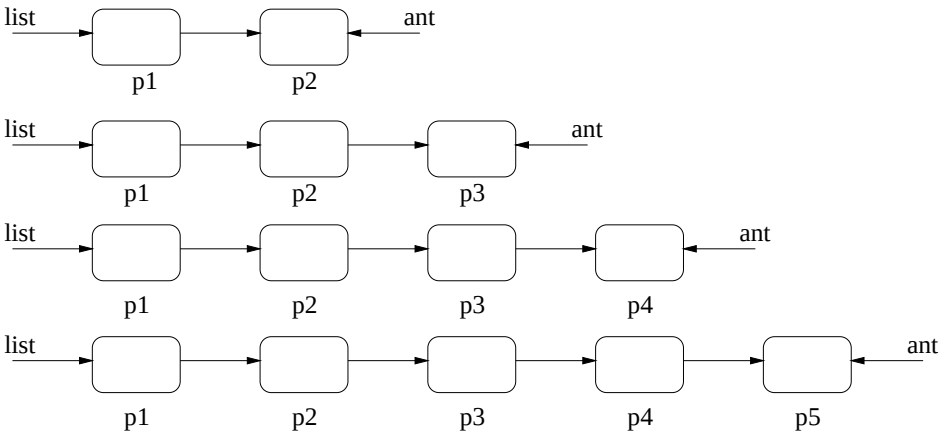


Figura 1.6: Posibles configuraciones de memoria para sentencia 9.

Estas distintas configuraciones de memoria tendrán que ser representadas por el grafo o los grafos asociados a la sentencia 9. En la figura 1.7 se presenta un posible grafo que representa todas esas distintas configuraciones de memoria.

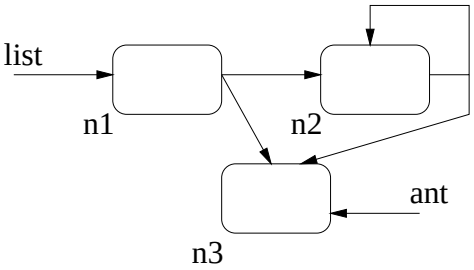


Figura 1.7: Grafo que representa las configuraciones de memoria para la sentencia 9.

En la figura 1.7 el nodo n_1 está representando a las porciones de memoria p_1 de las distintas configuraciones de memoria, por tanto representa distintas porciones de memoria pero todas pertenecientes a distintas configuraciones de memoria. Por tanto, varias porciones representadas por n_1 nunca pueden coexistir. El nodo n_3 representa a los últimos elementos

de cada lista, representando a las porciones p_2, p_3, p_4 y p_5 de las distintas configuraciones de memoria. Es similar a n_1 en el sentido de que en cada configuración de memoria tan solo representa a una porción. Sin embargo, el nodo n_2, está representando a distintas porciones de memoria pertenecientes a distintas configuraciones de memoria, pero a la vez está representando a distintas porciones de memoria pertenecientes a una misma configuración de memoria (*nodo sumario*). Así representa a p_2 de la segunda lista, a p_2 y p_3 de la tercera lista y a p_2, p_3 y p_4 de la última lista.

Por tanto, vemos que se van a mezclar informaciones de distintas configuraciones de memoria, de manera que la precisión del método será mayor, cuanto mejor sea capaz de aislar las características de las distintas configuraciones de memoria que representa. No va a bastar con que un método pueda representar mediante sus grafos, cierto tipo de estructuras dinámicas (listas doblemente enlazadas, árboles, etc..), sino que los resultados de la representación obtenida de un grafo dependerán de la complejidad del código analizado. Nos estamos refiriendo a que puede que existan dos códigos distintos que creen una lista doblemente enlazada, uno de una forma natural y sencilla, y el otro por medio de complicados bucles dentro de estructuras condicionales. Aunque en tiempo de ejecución, ambos códigos generen la misma estructura de datos, en tiempo de compilación, el análisis del segundo será mucho más complejo al tener que observar muchas más posibilidades debidas a los muchos caminos distintos en el grafo de flujo de control. Así un método que soporte listas doblemente enlazadas puede que sea capaz de obtener información satisfactoria para el primer código y fallar para el segundo.

En los siguientes capítulos veremos en detalle las ideas que proponemos para que un compilador, no sólo pueda obtener eficientemente el grafo de la figura 1.7 a partir del código mostrado anteriormente, sino que sea capaz de analizar códigos reales que manejan estructuras significativamente más complejas que una lista enlazada.

2 Grafos de forma estáticos: SSG

En este capítulo presentamos una serie de modificaciones importantes hechas sobre el método de análisis desarrollado por Sagiv y col. [67] basado en grafos de forma. Las modificaciones introducidas permiten que el método sea capaz de analizar con éxito códigos que manejan estructuras de datos dinámicas mucho más complejas y comunes que las soportadas en el método original. Estas modificaciones son presentadas en la siguiente sección. En la sección 2.2 se presentan los resultados obtenidos tras analizar un código que realiza las operaciones principales usadas en la descomposición LU de una matriz dispersa.

2.1 Mejora del análisis de forma

Como hemos visto, el método basado en SSGs presenta unas características muy buenas para el análisis de la forma de las estructuras dinámicas de datos usadas en los códigos. Aun así, el método se queda corto a la hora de poder analizar y obtener representaciones adecuadas de estructuras de datos usadas muy a menudo en códigos C. Es por este motivo que hemos propuesto e implementado una serie de mejoras sobre dicho método que le permiten aumentar la exactitud a la hora de analizar este tipo de estructuras.

Pero antes de mostrar las mejoras añadidas al método, presentaremos brevemente algunos conceptos sobre los SSGs, junto con la notación usada en [67].

2.1.1 Notación y conceptos

El algoritmo de análisis de forma se basa en una abstracción de la memoria llamada *static shape graph* (SSG). Un SSG es un grafo dirigido finito que aproxima la forma de las estructuras dinámicas de datos que pueden aparecer durante la ejecución de un programa. El algoritmo de análisis de forma es un procedimiento iterativo que calcula un SSG para cada punto del programa.

Antes de continuar, tenemos que decir que durante este capítulo vamos a utilizar una notación tipo $PASCAL$ cuando presentemos las operaciones que implican punteros, para mantener la misma notación utilizada en [67].

Los SSGs están formados por dos tipos de nodos, *variable-nodes* ($PVar$) y *shape-nodes*, y dos tipos de aristas *variable-edges* y *selector-edges*. Los *variable-nodes* representan a las varia-

bles puntero declaradas en el programa, mientras que los *shape-nodes* representan porciones de memoria reservadas por el programa, de algún tipo declarado en el mismo. Dentro de cada una de estas porciones hay espacio para datos y para punteros a otras porciones de memoria. A estos punteros se les va a denominar *selectores*. Las aristas *variable-edges* representan enlaces entre variables puntero (*variable-nodes*) y porciones de memoria (*shape-nodes*), mientras que los *selector-edges* representan enlaces entre distintas porciones de memoria por medio de algún *selector*.

Cada $SSG^{\#}$ está representado por un par de conjuntos de aristas, $<E_v^{\#}, E_s^{\#}>$, donde:

- $E_v^{\#}$ es el conjunto de *variable-edges* del grafo, cada uno de los cuales se expresa como $[x, n]$, donde $x \in PVar$ y n es un *shape-node*, representando que la variable x apunta al nodo n.

- $E_s^{\#}$ es el conjunto de *selector-edges* del grafo, denotados por $<s, sel, t>$ donde s y t son *shape-nodes* y sel es un selector, representando que el nodo s apunta al nodo t por medio del selector sel.

Como hemos comentado anteriormente, una de las principales características de este método es el esquema de nombres usado para los nodos. Los *shape-nodes* se nombran usando un conjunto de variables puntero (que puede ser el conjunto vacío). El conjunto de *shape-nodes* de $SSG^{\#}$ ($shape\_nodes(SSG^{\#})$ es un subconjunto de $\{n_X \mid X \subseteq PVar\}$.

De este modo, un *shape-node* n_Z, donde $Z \neq \emptyset$ representa aquellas porciones de memoria que son referenciadas única y exclusivamente por el conjunto de variables puntero que aparecen en el conjunto Z. Así, por ejemplo $n_{\{x\}}$ representa a todas aquellas porciones de memoria referenciadas por la variable x en el punto determinado del programa al que pertenece el SSG en el cual aparece $n_{\{x\}}$. Este nodo puede representar distintas porciones pertenecientes a los distintos estados de la memoria que pueden aparecer en un punto determinado de un programa, debidas a los distintos caminos en el flujo de control que pasan por dicho punto.

El *shape-node* n_ϕ representa aquellas porciones de memoria reservadas por el programa y que en un punto determinado del mismo no son referenciadas por ninguna variable puntero. El nodo n_ϕ, por tanto, puede representar múltiples porciones de memoria correspondientes a un "mismo estado de la memoria".

En la figura 2.1 (a) presentamos un ejemplo de una lista de cuatro elementos, representada por cuatro porciones de memoria (l_1, l_2, l_3 y l_4) enlazadas mediante el selector nxt. Vemos como l_1 es referenciada por la variable x. En la figura 2.1 (b) se puede observar como quedaría la lista tras la eliminación del primer elemento (porción l_1). Si ambas configuraciones de memoria pudieran aparecer en un mismo punto del programa, por ejemplo dentro de un bucle, el SSG que las representaría sería el mostrado en la figura 2.1 (c).

El nodo $n_{\{x\}}$ de la figura 2.1 (c) representa las porciones de memoria l_1 de (a) y l_2 de (b) que son las referenciadas por la variable x. Como podemos ver, está representando distintas porciones de memoria pertenecientes a dos estados de la memoria de caminos distintos del flujo de control que pasan por la misma sentencia. Sin embargo el nodo $n_\emptyset$ representa distintas porciones de memoria que pueden pertenecer a un mismo estado de la memoria. Para el caso (a) representaría las porciones de memoria l_2, l_3 y l_4 y para el caso (b) la l_3 y l_4.

En el SSG de la figura 2.1 (c) podemos ver como el nodo $n_\emptyset$ tiene un enlace por el selector nxt hacia él mismo. Este selector esta representando los enlaces por nxt entre l_2 y l_3, y

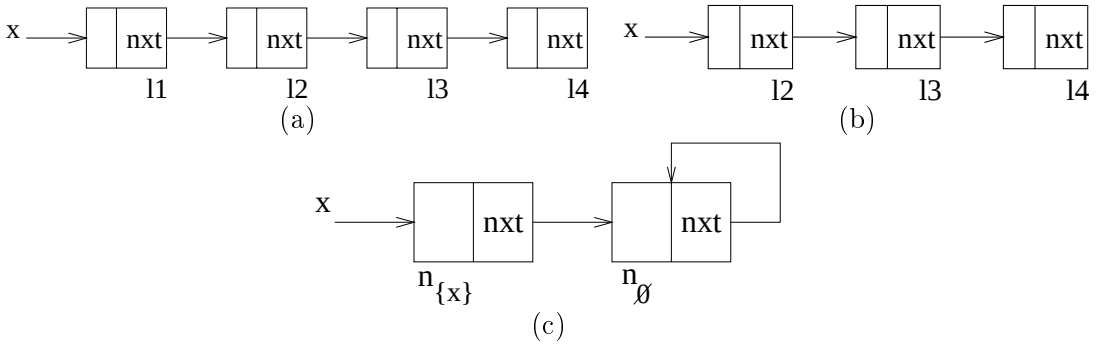


Figura 2.1: Ejemplo de SSG para lista enlazada.

entre l_3 y l_4 del caso (a), y el de l_3 a l_4 del caso (b) (puesto que todas esas porciones de memoria están siendo representadas por el mismo nodo, $n_\emptyset$). Sin embargo este enlace podría estar representando algún ciclo dentro de la estructura apuntada por x. Para poder distinguir cuándo un selector que va desde un nodo a él mismo, representa o no algún tipo de ciclo dentro de la estructura de datos, se introduce información adicional a cada nodo.

Así, cada *shape-node* n en un SSG tiene asociado un flag booleano, denotado como $is^\#(n)$ (is shared), de manera que cuando tiene el valor *true* indica que las porciones de memoria representadas por n pueden ser referenciadas por punteros desde dos o más porciones de memoria distintas a la vez. Cuando tiene el valor *false*, si el nodo n es referenciado por varios selectores en un SSG, dichos selectores están representando enlaces que nunca apuntan a la misma porción de memoria representada por n. Este sería el caso del SSG presentado en la figura 2.1 (c) ($is^\#(n_\emptyset)$ =*false*).

Compatibilidad entre nodos

Un concepto muy importante es el de *compatibilidad* entre nodos, debido al hecho de que un único nodo n puede representar, como hemos visto, distintas porciones de memoria pertenecientes al mismo o a distintos estados de la memoria. De este modo, dos *shape-nodes* distintos n_X y n_Y, con $X \neq Y$ y $X \cap Y \neq \emptyset$, representan configuraciones de variables incompatibles, es decir, representan porciones de memoria pertenecientes a distintos estados de la memoria. Esto es así puesto que n_X representa porciones de memoria apuntadas por todas (y sólo) las variables pertenecientes a X, de manera que si $X \neq Y$ y $X \cap Y \neq \emptyset$, significa que al menos hay una variable que aparece en X y en Y. Como se sabe que en un estado concreto de la memoria una variable apunta como mucho a una sola porción de memoria, las porciones de memoria representadas por n_X y n_Y no pueden pertenecer a un mismo estado de la memoria.

Una consecuencia de esto es que para todo *selector-edge* de la forma $<n_X, sel, n_Y> \in E_s^\#$, se tiene que cumplir que o $X = Y$ o bien $X \cap Y = \emptyset$. Existe una función de compatibilidad entre nodos que es usada para determinar si un conjunto de nodos de un SSG son compatibles entre si. Con lo expuesto anteriormente es fácil deducir que la función $compatible^\#(n_{Z_1}, ..., n_{Z_k})$ indica que $\forall i, j : Z_i = Z_j \vee Z_i \cap Z_j = \emptyset$, es decir todos esos nodos son compatibles (pueden pertenecer a un mismo estado de la memoria) si el conjunto de variables que los referencian es el mismo, o no tienen ninguna variable en común.

Sumarización y materialización

Hay dos operaciones muy importantes que son la *sumarización* y la *materialización*, que se aplican a los nodos a medida que los SSG van siendo transformados por el análisis de las sentencias. Como hemos visto, en un SSG existen una serie de nodos que se diferencian en su nombre, formado por el conjunto de variables puntero que referencian las porciones de memoria representadas por el nodo. El caso es que a medida que se opera con las variables puntero en el programa, el nombre de estos nodos va cambiando conforme varía el conjunto de variables. A menudo ocurre que durante el análisis, dos nodos distintos n_X y n_Y $(X \neq Y)$, por medio de las transformaciones que sufre el grafo durante el análisis, pasan a tener el mismo nombre. Es decir, las transformaciones han provocado que el conjunto de variables puntero X sea igual a Y y por tanto ambos nodos representan ahora las mismas porciones de memoria. Por tanto, ambos nodos deben ser unidos en uno único que cubra de forma conservativa todas las porciones de memoria antes representadas por n_X y n_Y. A esta unión de nodos se denomina *sumarización*. Asi pues el nuevo nodo *sumario* debe poseer todos los selectores que tenían los nodos originales, y su información *shared* ($is^{\#}$) debe ser conservativa con respecto a la de los nodos originales (será *true* si lo era en n_X o n_Y). Como se puede ver, la operación de *sumarización* introduce inexactitud en la representación, en el sentido de que se va uniendo información de distintas porciones de memoria, siempre de forma conservativa.

La operación de *materialización* es la contraria a la *sumarización*. Así, por ejemplo, dado el SSG de la figura 2.1 (c) para una sentencia del tipo $y := x.nxt$, nos encontramos que la variable y debe apuntar a aquellas porciones de memoria que son referenciadas desde el nodo $n_{\{x\}}$ por el selector nxt. Es por tanto necesario *materializar* un nuevo nodo $n_{\{y\}}$ desde el nodo $n_{\emptyset}$ que es el nodo apuntado por nxt desde $n_{\{x\}}$. Esta *materialización*, separa de las porciones de memoria representadas por $n_{\emptyset}$, aquellas que pueden ser referenciadas por nxt desde $n_{\{x\}}$, de aquellas que no lo pueden ser. La exactitud de esta *materialización* en referencia a los selectores del nuevo nodo y a su información *shared*, dependen de la exactitud con que las distintas porciones de memoria son representadas en el nodo sumario $n_{\emptyset}$.

Construcción de los *SSG*s: Semántica Abstracta

Una vez vistas las características más importantes de un SSG, vamos a centrarnos ahora en su construcción. El método de análisis de forma, como hemos dicho, es un algoritmo iterativo que construye un SSG para cada punto del programa a analizar. El algoritmo opera en el dominio de los SSGs, donde cada sentencia del programa tiene asociado una función de transformación de un SSG a otro SSG. Estas transformaciones quedan reflejadas en la *semántica abstracta* asociada con cada tipo de sentencia. Por tanto se define una *semántica abstracta* para los seis tipos de sentencias que manejan punteros como son $x := nil$, $x.sel := nil$, $x := new$, $x := y$, $x.sel := y$ y $x := y.sel$ (cualquier otra sentencia más compleja que utilice punteros, puede ser construida con varias de estas más simples usando variables temporales). En definitiva la *semántica abstracta* de una sentencia, st, especifica las modificaciones que hay que llevar a cabo sobre un SSG de entrada que representa los posibles estados de la memoria antes de la ejecución de dicho st. Estas transformaciones van a obtener un SSG de salida que debe representar los nuevos estados de memoria que se obtendrán tras ejecutar la sentencia sobre los representados por el SSG de entrada. En cuanto a notación se utilizará $E_v^{\#\prime}$, $E_s^{\#\prime}$ y $is^{\#\prime}$ para representar a $E_v^{\#}$, $E_s^{\#}$ y $is^{\#}$ tras la ejecución de una sentencia.

En la figura 2.2 se puede observar cómo a cada sentencia del código se le asocia un grafo,

y que dicho grafo es construido aplicando la semántica abstracta de la sentencia sobre el grafo de la sentencia predecesora.

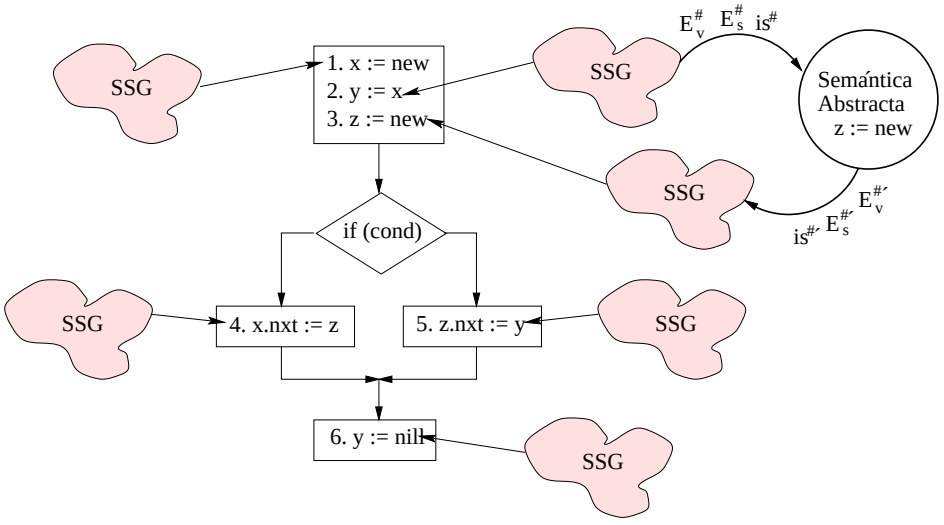


Figura 2.2: Visión general del *framework*.

El proceso iterativo de análisis de un código consiste en la *interpretación abstracta* de cada una de las sentencias del código, aplicando la *semántica abstracta* de la sentencia al SSG que representa todos los posibles estados de la memoria obtenidos por los diversos caminos del flujo de control que han llegado a dicho punto. Tras la *interpretación abstracta* de la sentencia, se obtiene el SSG que representa la memoria tras la ejecución de la misma y que será el grafo de entrada para la siguiente sentencia que aparezca en el flujo de control tras la actual. En el flujo de control nos podemos encontrar con puntos en los que hay bifurcaciones del flujo (if, case, for, ...), y otros donde confluyen varios flujos (final de if, case, for, ...). En las bifurcaciones, se pasa el SSG de la cabecera a cada una de las sentencias en las que se bifurca el flujo. En la confluencia de flujos, el SSG que pasa a la siguiente sentencia, es la *unión* de todos los SSGs de las últimas sentencias en cada una de las ramas de la bifurcación. Por tanto existe una operación de *unión*, que dados varios SSGs, crea un nuevo SSG que representa todas los estados posibles de memoria representados en cada una de las ramas.

Esta operación de *unión* es usada además en cada sentencia del programa para calcular el SSG asociado a la misma. Puesto que a una sentencia se puede llegar siguiendo distintos caminos del flujo de control y puesto que el SSG asociado con la sentencia debe representar todos los posibles estados de memoria que se pueden dar tras la ejecución de la sentencia, dicho SSG debe crearse uniendo todos los grafos que se producen tras la interpretación abstracta de dicha sentencia. Esto nos sirve para explicar como son analizados los bucles, puesto que en estos la longitud del flujo de control no esta limitada en tiempo de compilación ya que no se conocen las condiciones de parada. Por tanto hay que determinar cuando dejar de analizar las sentencias del cuerpo de un bucle. La condición de parada es que el SSG asociado con la cabecera del bucle no cambie de un paso al siguiente, con lo que se ha alcanzado un *punto fijo*. Eso quiere decir que dicho SSG está representando todos los posibles estados de memoria que se pueden alcanzar tras la ejecución, cualquier número de veces, del cuerpo del bucle. Por más veces que se analizara el bucle, el SSG no va a cambiar más. Para localizar este punto fijo es necesaria una operación de *comparación* de grafos, que comprueba si todos los nodos y selectores de ambos grafos son iguales.

Una vez visto por encima el funcionamiento del método, vamos a pasar a describir las mejoras implementadas para aumentar el número de tipos de estructuras que es capaz de analizar con éxito. Como hemos comentado anteriormente, la exactitud del método viene marcada por cómo los nodos son *sumarizados*, puesto que lleva consigo una mezcla de información conservativa de distintas porciones de memoria. Aparte, la operación *materialización* que extrae nodos de los nodos sumario, será más exacta en la representación resultante cuanto mayor información se tenga acerca de las distintas porciones de memoria representadas dentro del nodo sumario. Es por tanto que nuestras mejoras van encaminadas a aumentar la exactitud en las representaciones tras la ejecución de dichas operaciones.

En concreto nuestras mejoras van en dos direcciones:

- Permitir la existencia de más de un nodo sumario $n_\emptyset$ en cada SSG. Como se ha visto anteriormente, el nodo $n_\emptyset$ representa todas aquellas porciones de memoria no apuntadas directamente por ninguna variable puntero, por lo tanto es único en cada SSG. Nosotros hemos añadido cierta información adicional a cada nodo para permitir la existencia de varios nodos $n_\emptyset$ en cada grafo, y así reducir la mezcla de información que se produce en la *sumarización*.

- Añadir información adicional a los nodos, que aunque no va a evitar la *sumarización* como en el caso anterior, si que va a proporcionar información muy útil a la hora de la *materialización* de nuevos nodos, para que la representación de las porciones de memoria tras dicha *materialización* sea mucho más fiel a la realidad.

2.1.2 Más de un nodo sumario por SSG

Como hemos expuesto anteriormente, el método original tan solo permite la existencia de un nodo $n_\emptyset$ por grafo. Nosotros vamos a extender el número permitido de nodos $n_\emptyset$, añadiendo dos nuevos atributos a los *shape-nodes*. Estos dos atributos van a hacer referencia al *tipo* de estructura al que pertenecen las porciones de memoria representadas por un nodo, y el otro va a hacer referencia a la conexión existente entre las diversas estructuras que son construidas dinámicamente.

Un nodo sumario por cada "tipo" de puntero.

Cuando sólo puede existir un único nodo $n_\emptyset$, este representa todas las porciones de memoria que no son referenciadas directamente por ninguna variable puntero. Dentro de estas porciones de memoria nos encontramos distintos *tipos*, entendiendo por *tipo* de una porción de memoria, el tipo declarado del puntero usado para reservar dicha porción de memoria. Como ya hemos comentado, al sumarizar varios nodos en uno único, la información *shared* ($is^\#$) es común para todas las porciones de memoria de los nodos originales, y si alguna tiene $is^\# = true$, entonces permanece así para el nodo $n_\emptyset$.

Por ejemplo, permitiendo la existencia de un único $n_\emptyset$, todos las porciones de memoria de dos estructuras de distinto tipo, como las mostradas en la figura 2.3 (a), serán representadas por el único nodo $n_\emptyset$. El SSG que representaría dichas estructuras se puede ver en la figura 2.3 (b).

Puesto que $n_\emptyset$ representa todas aquellas porciones de memoria en la figura 2.3 (a) no apuntadas por *List* ni por *Graph*, $is^\#(n_\emptyset) = true$, ya que hay porciones de memoria que son

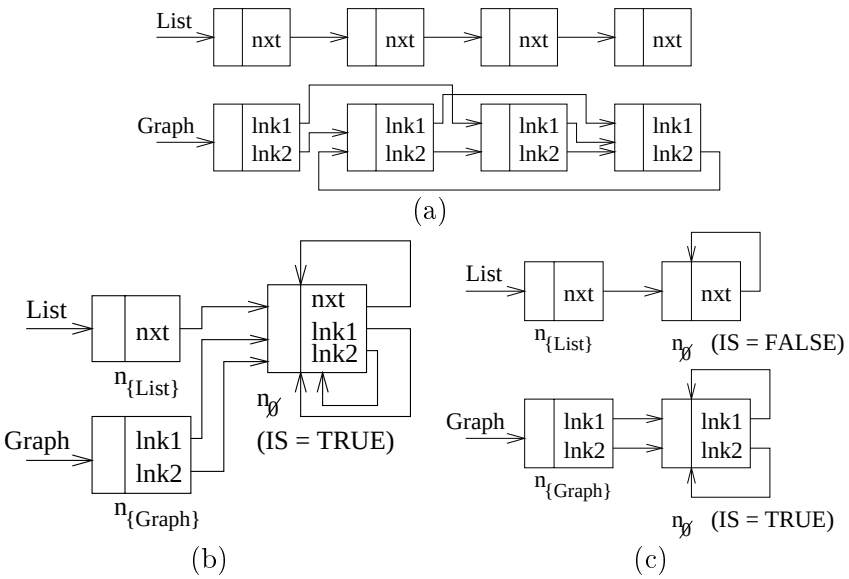


Figura 2.3: (a) Estructuras reales. (b) SSG sin información del *tipo*. (c) SSG con información *tipo*. En (b) la *List* y *Graph* pueden compartir elementos, y *List* puede ser cíclica.

referenciadas desde más de una porción, en concreto ocurre con las de *Graph*. Esto hace que aunque los elementos de *List* no son referenciados más que desde otro elemento, el SSG de la figura 2.3 (b) no pueda proporcionarnos esta información, al haber mezclado las porciones de memoria de distintos tipos de estructuras. Esto se puede resolver permitiendo la existencia de un nodo $n_\emptyset$ para cada *tipo* diferente de estructura declarada en el programa. En la figura 2.3 (c) vemos el SSG que resultaría permitiendo la existencia de un nodo sumario por cada tipo de estructura. Como se puede observar, ahora existen dos nodos $n_\emptyset$, uno para *List* y otro para *Graph*, de forma que para *List*, $is^\#(n_\emptyset) = false$, puesto que ahora sólo representa las porciones de memoria apuntadas desde *List*, y no existe ninguna que sea referenciada más de una vez. Esto ha sido posible al mantener separadas, en nodos distintos, las porciones de memoria de la lista y del grafo, evitando así que se mezcle la información *shared* de ambos tipos de estructuras.

Para obtener este comportamiento, además del atributo $is^\#$, hemos asociado a cada nodo información sobre el *tipo* de porciones de memoria que está representando. Este nuevo atributo se denomina $type^\#$. Para el manejo de dicha información es necesario saber el *tipo* de cada variable puntero del programa ($type\_var$), que es tomado de la parte de declaración de variables del código.

En el apéndice A.1 se presentan las modificaciones sobre la semántica abstracta de las sentencias para dar soporte a este nuevo atributo.

Un nodo sumario por cada "componente conexa".

Con la inclusión del atributo $type^\#$ hemos conseguido mantener distintos nodos sumarios $n_\emptyset$ para los distintos *tipos* de estructuras declaradas en el código del programa. Sin embargo, cuando en un mismo programa se manejan varias estructuras de un mismo *tipo* que no comparten ningún elemento, habrá un único nodo $n_\emptyset$ para ese tipo de estructura, que representará todas las porciones de memoria no referenciadas por variables pertenecientes a ambas

estructuras. Sería interesante poder distinguir explícitamente esas dos estructuras no conexas, incluso siendo del mismo *tipo*.

Por ejemplo, en la figura 2.4 (a) podemos ver dos estructuras del mismo *tipo* que no comparten ningún elemento. Como se puede ver, en la estructura apuntada por *Graph1* todos los elementos son referenciados como máximo por otro elemento, mientras que en la estructura referenciada por *Graph2* existe un elemento (el tercero) que es apuntado por más de un elemento de dicha estructura. Esto provoca que en el SSG de la figura 2.4 (b), obtenido con el método original, que representa ambas estructuras, el nodo $n_{\emptyset}$ sea *shared* ($is^{\#}(n_{\emptyset}) = true$). La información que se puede deducir de dicho SSG, es que las estructuras apuntadas tanto por *Graph1* como por *Graph2* pueden contener ciclos internos, así como que es posible que ambas estructuras compartan elementos. Esta información evitaría cualquier posibilidad de paralelización de un recorrido sobre la estructura apuntada por *Graph1*.

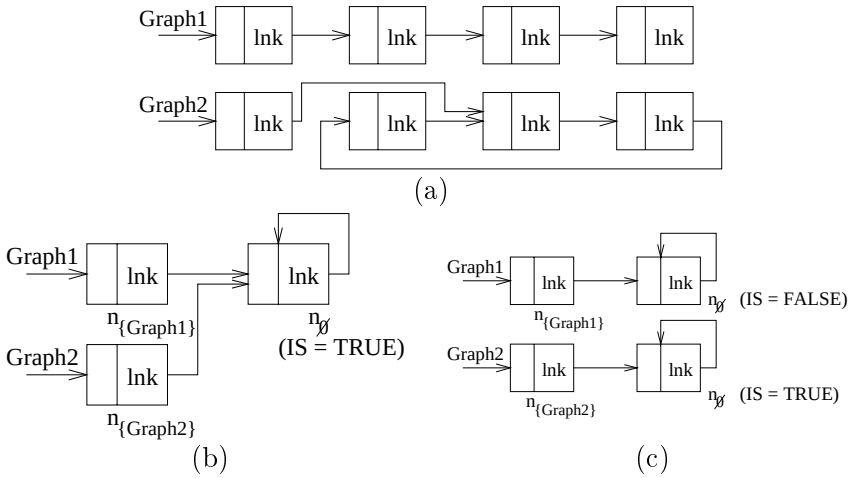


Figura 2.4: (a) Estructuras reales. (b) SSG sin atributo *structure*. (c) SSG con *structure*. En (b) Graph1 y Graph2 pueden compartir nodos y Graph1 puede contener ciclos.

Para evitar en la medida de lo posible esta pérdida de información acerca de la estructuras, se ha introducido un nuevo atributo en cada nodo que mantiene información sobre la estructura a la que pertenece. Hemos denominado a dicho atributo $structure^{\#}$. En la figura 2.4 (c) se presenta el SSG que representa a las estructuras de las figura 2.4 (a), utilizando este nuevo atributo. Como se puede observar, existen dos nodos $n_{\emptyset}$, uno para la estructura apuntada por *Graph1* y otro para la de *Graph2*. De nuevo, al no tener que mezclar la información de las porciones de memoria de las distintas estructuras, es posible mantener $is^{\#}(n_{\emptyset}) = false$ para *Graph1*. De este SSG se puede deducir que la estructura apuntada por *Graph1* es una estructura acíclica que no comparte ningún elemento con la estructura referenciada por *Graph2*.

Este nuevo atributo $structure^{\#}$ tomará el mismo valor para todos los nodos entre los que existe un "camino" (sucesión de nodos conectados por medio de *selector-edges*). En concreto, definimos el conjunto de nodos "conectados" a un nodo dado n, $C[Es^{\#}](n)$, como el conjunto de nodos de cualquiera de los caminos que empiezan en cualquier nodo que hay en cualquier camino hacia o desde el nodo n:

$$C[Es^{\#}](n) = \{n_j \mid \exists n_1, .., n_i, n_1\prime, .., n_i\prime, n_a \ ((< n_a, sel_1, n_1 >, < n_1, sel_2, n_2 >, .., < n_i, sel_{i+1}, n >) \in Es^{\#} \wedge (< n_a, sel_1\prime, n_1\prime >, < n_1\prime, sel_2\prime, n_2\prime >, .., < n_i\prime, sel_{i+1}\prime, n_j >) \in Es^{\#})\}$$

Las modificaciones sobre la semántica abstracta de las sentencias para soportar el atributo *structure* se presentan en el apéndice A.2.

2.1.3 Información adicional en cada nodo

La otra modificación introducida al método original, aparte de la posibilidad de que exista más de un nodo sumario $n_\emptyset$ en cada grafo, es la introducción de información adicional a *shared*, para describir de una manera más precisa las porciones de memoria que son representadas por los nodos.

El proceso de *sumarización*, necesario para mantener el tamaño de los grafos finito, lleva consigo la mezcla de las características de distintas porciones de memoria en un único nodo. Si cuando al analizar una sentencia se pretende modificar el grafo de una forma precisa, es necesario extraer de los nodos *sumario* aquellas porciones de memoria que realmente van a ser actualizadas por la sentencia. Esta tarea, como comentábamos anteriormente, la lleva a cabo la operación denominada *materialización*. El problema es que cuando se materializa un nodo desde otro, el método tiene que ser conservativo a la hora de determinar los enlaces que va a mantener el nuevo nodo, de todos los enlaces que posee el nodo desde el cual se ha materializado.

Por ejemplo en la figura 2.5 (a), se muestra un SSG que representa una lista enlazada por el selector nxt apuntada desde la variable x. Vemos que el nodo $n_\emptyset$ representa todas aquellas porciones de memoria de la lista no apuntadas por la variable x.

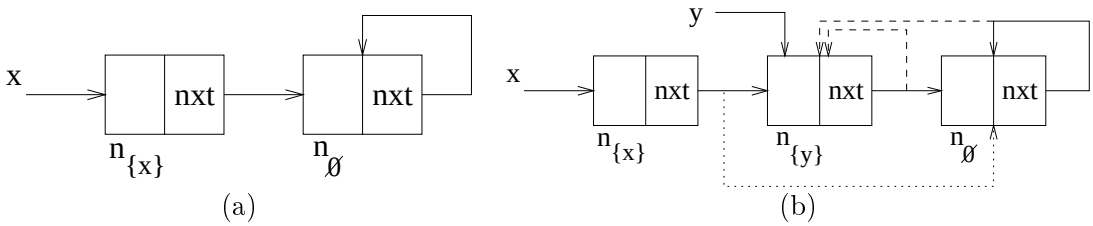


Figura 2.5: (a) SSG representando una lista. (b) Materialización del nodo apuntado por $x.nxt$.

Si se quiere analizar la sentencia $y := x.nxt$ sobre dicho SSG, es necesaria la materialización de un nuevo nodo ($n_{\{y\}}$ (ver figura 2.5 (b)), ahora referenciado desde y) desde el nodo sumario $n_\emptyset$. El nuevo nodo, hereda todos los enlaces del nodo $n_\emptyset$. Así, puesto que $n_{\{x\}}$ apuntaba por nxt a $n_\emptyset$ ahora también lo hace sobre $n_{\{y\}}$. Además, como existía un enlace entre $n_\emptyset$ y él mismo, en principio pueden existir los enlaces $<n_{\{y\}}, nxt, n_\emptyset>$, $<n_{\{y\}}, nxt, n_{\{y\}}>$, y $<n_\emptyset, nxt, n_{\{y\}}>$. Algunos de estos enlaces pueden ser eliminados. Por ejemplo, el enlace desde $n_{\{x\}}$ a $n_\emptyset$ (línea de puntos) desaparece, puesto que es seguro que tras la ejecución de la sentencia $y := x.nxt$, las porciones de memoria referenciadas por $x.nxt$ tienen que ser también apuntadas directamente por y. Por tanto, como $n_{\{y\}}$ representa todas aquellas porciones de memoria referenciadas por y, el nodo $n_{\{x\}}$ sólo puede apuntar a $n_{\{y\}}$ por nxt.

Sin embargo existen otros enlaces que no se corresponden con una lista enlazada, que son los que quedan entre $n_\emptyset$ y el nuevo nodo $n_{\{y\}}$ y entre $n_{\{y\}}$ y él mismo (líneas discontinuas). Si no se pudieran eliminar estos enlaces, la representación de la lista hecha por el grafo, informa de la posible existencia de un enlace desde algunas de las porciones de memoria no apuntadas ni por x ni por y, a la referenciada por y, con lo cual podrían existir ciclos en la estructura

(se podrían alcanzar las porciones de memoria referenciadas por y varias veces haciendo un recorrido de la estructura por nxt). Incluso, cabe la posibilidad de que la porción de memoria apuntada por y se referencie a sí misma por nxt. En definitiva, al no eliminar esos enlaces, la representación de la estructura se aleja bastante de la realidad, reduciendo la utilidad de la información proporcionada por el grafo sobre la estructura de datos.

Es por este motivo que el método original introduce la información $is^{\#}$ en cada nodo, indicando si las porciones de memoria representadas por el mismo pueden ser o no referenciadas más de una vez desde otras. En caso de que $is^{\#}(n_{\emptyset})$ en la figura 2.5 (a) fuera *false*, se sabe que las porciones de memoria representadas por el nodo pueden ser referenciadas como máximo por otra. Esta información es muy útil, puesto que permite la eliminación de los enlaces representados en la figura 2.5 con líneas discontinuas. Como $n_{\{y\}}$ se ha materializado por la sentencia $y := x.nxt$, se sabe que el enlace entre $n_{\{x\}}$ y $n_{\{y\}}$ por nxt debe existir. Como la $is^{\#}$ informa que las porciones de memoria de dicho nodo tan solo pueden ser referenciadas desde otra (en este caso $n_{\{x\}}$), se puede eliminar cualquier otro selector que apunte a $n_{\{y\}}$ que no provenga de $n_{\{x\}}$ (el que proviene de $n_{\{y\}}$ y de $n_{\emptyset}$). Como se puede observar, eliminando dichos enlaces, la representación obtenida de la estructura de datos en el grafo es mucho más fiel a la realidad.

Como podemos ver de lo expuesto, la información adicional sobre la estructura interna de las porciones de memoria representadas por un nodo, es muy importante a la hora de manejar los grafos para obtener una representación lo más fiel posible a la realidad. Hemos comprobado que la información proporcionada por $is^{\#}$ es insuficiente a la hora de manejar estructuras más complejas muy usadas en códigos C. Por este motivo, hemos dotado a cada nodo con más información acerca da las porciones de memoria que representa. En concreto, los nuevos atributos añadidos a cada nodo son:

1. Información *shared* ($is\_sel^{\#}$) por cada tipo de selector.
2. Información sobre ciclos en la estructura ($cyclelinks^{\#}$).

En las dos próximas secciones describimos detalladamente cada una de estas dos nuevas informaciones introducidas en cada nodo.

Información *SHARED* por selector

En el método original tan solo se mantiene la información $is^{\#}$ que indica cuando las porciones de memoria representadas por un nodo pueden ser referenciadas más de una vez desde otras porciones de memoria. Sin embargo, esta información no tiene en cuenta los selectores por los que el nodo es referenciado, lo que provoca una pérdida de exactitud durante el análisis.

Así, por ejemplo, en la figura 2.6 (a) presentamos una lista doblemente enlazada en la cual cada elemento posee un enlace por nxt al siguiente elemento de la lista, y otro enlace por prv al elemento anterior. El SSG que representa dicha estructura se presenta en la figura 2.6 (b). Como se puede observar, la información $is^{\#}(n_{\emptyset})$ es *true* puesto que dicho nodo representa todos los elementos de la lista no referenciados por ninguna variable y hay un elemento (el central) que es referenciado desde otros dos (por nxt desde el primero y por prv desde el tercero). Si sólo tuviéramos en cuenta la información $is^{\#}$, al materializar un nodo desde $n_{\emptyset}$, puesto que es *shared*, heredaría todos los enlaces que hay sobre $n_{\emptyset}$, apareciendo ciclos en la estructura tanto por enlaces nxt como por prv ante la imposibilidad de eliminarlos puesto que $is^{\#}$ es *true*.

Como se observa, se pierde una información muy interesante sobre la estructura de datos y es que, si dicha estructura es recorrida tan solo utilizando un tipo de selector (ya sea *nxt* o *prv*), nunca se visitará dos veces el mismo elemento. Esta información puede ser deducida del SSG presentado en la figura 2.6 (c), obtenido utilizando un atributo *shared* para cada tipo de selector.

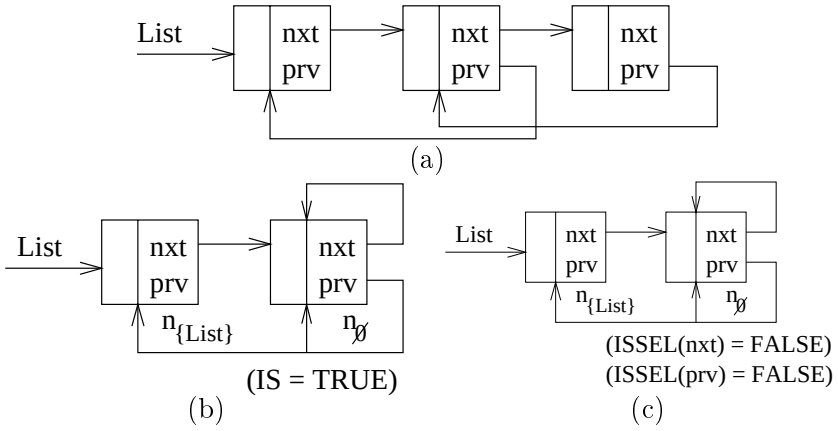


Figura 2.6: (a) Lista doblemente enlazada. (b) SSG sin is_sel. (c) SSG con is_sel.

Hemos añadido a cada nodo un nuevo atributo aparte de $is^{\#}(n)$, de la forma:

$$is\_sel^{\#}(n, sel) \rightarrow \{false,\ true\}$$

que indica si alguna de las porciones de memoria representadas por el nodo n es referenciada desde otras porciones de memoria más de una vez por el selector sel. Como se puede observar en la figura 2.6 (c), el nodo $n_{\emptyset}$ tiene un valor del atributo $is\_sel^{\#}$ para cada tipo de selector (nxt y prv) que tiene el valor *false*. De este grafo sí que es posible extraer la información de que un recorrido, sobre la estructura apuntada por *List*, que tan solo utilice un tipo de selector (ya sea nxt o prv), no visitará dos veces el mismo elemento. La información extraída de este nuevo grafo, se ajusta mucho mejor a la forma de la estructura real. La semántica abstracta modificada que es capaz de manejar la información proporcionada por este nuevo atributo, es presentada en el apéndice A.3.

Vamos a ver sobre los grafos de la figura 2.6 (b) y (c) la utilidad de la información $is\_sel^{\#}$. En la figura 2.7 (a) se presenta el grafo obtenido tras la materialización de un nuevo nodo $n_{\{X\}}$ desde el nodo $n_{\emptyset}$ debida a la sentencia $X := List.nxt$ para el grafo presentado en la figura 2.6 (b), en el cual no existe la información "shared" por selector. En la figura 2.7 (b) se presenta el obtenido para el de la figura 2.6 (c) utilizando en este caso la información $is\_sel^{\#}$.

Como se puede observar, en la figura 2.7 (a), el nuevo nodo materializado $n_{\{X\}}$ es referenciado por nxt desde $n_{\{List\}}$, $n_{\emptyset}$ y desde el mismo $n_{\{X\}}$, puesto que estos eran los nodos que apuntaban a $n_{\emptyset}$ antes de materializarlo. Claramente se ve que realmente tan solo debería aparecer el enlace desde el nodo $n_{\{List\}}$, pues la materialización se ha llevado a cabo provocada por la sentencia $X := List.nxt$. Sin embargo los demás enlaces por nxt no pueden ser eliminados puesto que el nodo es "shared" ($is^{\#}(n_{\{X\}})$ =*true*). La información obtenida de este grafo es demasiado conservativa, permitiendo la existencia de ciclos en la estructura por nxt ($n_{\{X\}} \rightarrow n_{\{X\}}$, $n_{\{X\}} \rightarrow n_{\emptyset} \rightarrow n_{\{X\}}$).

En la figura 2.7 (b) vemos como queda el grafo tras la materialización utilizando la información "shared" por selector. Como podemos observar, ahora no existen los enlaces

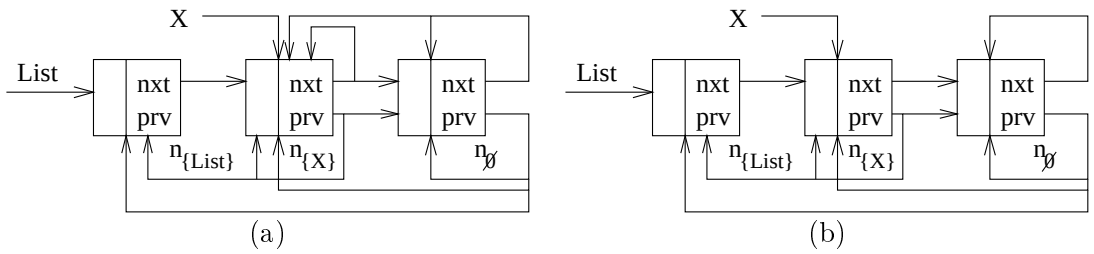


Figura 2.7: (a) Materialización sin $is\_sel^{\#}$ (b) Materialización con $is\_sel^{\#}$

$<n_{\{X\}}, nxt, n_{\{X\}}>$ ni $<n_{\emptyset}, nxt, n_{\{X\}}>$, evitando la posibilidad de existencia de ciclos en la estructura si se utiliza sólo el selector nxt. Estos dos enlaces son eliminados puesto que no son "compatibles" con el enlace $<n_{\{List\}}, nxt, n_{\{X\}}>$. Se sabe que esta referencia definitivamente existe puesto que el nodo $n_{\{X\}}$ es materializado precisamente para representar aquellas porciones de memoria referenciadas por nxt desde las porciones apuntadas por *List*. No son compatibles puesto que como $is\_sel^{\#}(n_{\{X\}}, nxt)$ =*false* no puede haber ninguna otra referencia sobre $n_{\{X\}}$ por nxt que no sea $<n_{\{List\}}, nxt, n_{\{X\}}>$.

Hemos visto como con la utilización de la información "shared" por selector, los grafos obtenidos tras las materializaciones representan con mayor exactitud los posibles estados de la memoria, evitando la inclusión de una serie de enlaces que se sabe que no se corresponden con las estructuras representadas. Aun así, como se puede ver en la figura 2.7 (b), siguen apareciendo enlaces que no se corresponden con las estructuras reales. Podemos ver como existen los enlaces $<n_{\emptyset}, prv, n_{\{List\}}>$, $<n_{\{X\}}, prv, n_{\{X\}}>$ y $<n_{\{X\}}, prv, n_{\emptyset}>$ que no deberían aparecer. Estos enlaces no desaparecen puesto que el nodo $n_{\{X\}}$ sigue siendo "shared" ($is^{\#}(n_{\{X\}})$ =*true*) y además el selector prv no es usado en la sentencia (por tanto no se conoce la certeza de la existencia de algún enlace prv sobre $n_{\{X\}}$). En la sección siguiente veremos como se solucionan estos problemas con la inclusión de la propiedad *cycle links*.

2.1.4 Cycle links

Como hemos visto, con la información proporcionada por el atributo "shared" por selector podemos reconstruir de una manera más precisa el conjunto de enlaces de un nodo materializado. Sin embargo, hemos visto que aún siguen apareciendo enlaces que no existían en el nodo antes de la sumarización. Para cierto tipo de estructuras existe una propiedad que puede ser utilizada para depurar aún más los enlaces que se crean para un nodo materializado. Nos referimos a las estructuras denominadas "doblemente enlazadas" en las que se cumple la propiedad de que si una porción de memoria referencia a otra por un selector sel_1, esta última siempre referenciará a la primera por otro selector sel_2. Ejemplos muy comunes de este tipo de estructuras son las listas doblemente enlazadas, en las que cada elemento apunta por un selector al siguiente elemento y por otro al elemento anterior en la lista, o por ejemplo árboles binarios donde cada elemento apunta a su hijo izquierdo y este a su vez referencia al padre.

La idea es por tanto detectar este tipo de situaciones y poder introducir este tipo de información dentro de los nodos de manera que pueda ser utilizada a la hora de la materialización para eliminar enlaces que realmente no se dan en las estructuras reales. Esta idea fue introducida por Plevyak y col. en [62], y nosotros la hemos adaptado para dar mayor exactitud a los SSGs, introduciendo en cada nodo un nuevo atributo denominado *cycle links*.

Por tanto, vamos a introducir en cada nodo n el atributo *cycle links* que va a mantener información sobre pares de selectores sel_1, sel_2 que cumplan la siguiente propiedad: cualquier porción de memoria referenciada por sel_1 desde cualquier porción de memoria representada por el nodo n, deberá a su vez referenciar a esta última por el selector sel_2. De este modo definimos el atributo $cyclelinks^{\#}(n) = \{< sel_1, sel_2 >\}$ como un conjunto de pares de selectores sel_1 y sel_2 que cumplen la propiedad comentada anteriormente para todas las porciones de memoria representadas por n.

Ahora nos centramos en el verdadero potencial de la información *cycle links*, al utilizarla para eliminar enlaces del nuevo nodo materializado, obteniendo de esa manera una representación más exacta de la forma real de las estructuras representadas por el nodo sumario. En concreto los *cycle links* de un nodo nos dan información referente a los enlaces de las porciones de memoria representadas por el nodo. Al materializar un nodo, se le asignan los mismos enlaces que tiene el nodo del cual se ha materializado, pero si alguno de ellos no cumple las relaciones impuestas por la propiedad *cycle link*, puede ser eliminado. Esto es así, puesto que la propiedad impone unas restricciones a los selectores de un nodo, y como esta sentencia no modifica la conectividad del grafo (tan solo lee un selector), tras su ejecución las restricciones se deben cumplir también.

En la figura 2.8 (a) vemos como quedaría el SSG que representa una lista doblemente enlazada como la presentada en la figura 2.6 (a) anotando cada nodo con su conjunto de *cycle links*. Como se puede observar el nodo $n_{\{List\}}$ posee el *cycle link* $<nxt, prv>$ puesto que representa al primer elemento de la lista. El nodo $n_{\emptyset}$ posee dos, $<nxt, prv>$ y $<prv, nxt>$ puesto que está representando todos los demás elementos de la lista, los cuales tiene un sucesor al que apuntan por nxt y el cual les apunta por prv y un predecesor al que referencian por prv y este a su vez los referencia por nxt.

En la figura 2.8 (b) presentamos de nuevo el SSG obtenido tras materializar un nuevo nodo debido a la interpretación de la sentencia $X := List.nxt$, ya mostrado en la figura 2.7 (b), utilizando la información proporcionada por el atributo $is\_sel^{\#}$ pero sin utilizar los *cycle links*. Como ya comentamos anteriormente gracias al atributo $is\_sel^{\#}$, se podían eliminar una serie de enlaces en el nodo $n_{\{X\}}$, pero aún seguían apareciendo enlaces por el selector prv que no se corresponden con las estructuras que realmente se están representando en le grafo. La figura 2.8 (c) muestra el SSG obtenido utilizando la información *cycle links* a la hora de materializar $n_{\{X\}}$. Como podemos observar, dicha figura si que representa de forma fiel los enlaces existentes en una lista doblemente enlazada cuyo primer elemento es apuntado por *List* y cuyo segundo elemento lo es por X.

Se han podido eliminar tres enlaces que aparecen en la figura 2.8 (b) que no se corresponden con la estructura real, que son:

- El enlace $<n_{\{X\}}, prv, n_{\{X\}}>$ se puede eliminar puesto que el nodo $n_{\{X\}}$ posee el *cycle link* $<prv, nxt>$ lo que implicaría la existencia también el enlace $<n_{\{X\}}, nxt, n_{\{X\}}>$, y como dicho enlace no existe, el primero tampoco puede existir.

- El enlace $<n_{\{X\}}, prv, n_{\emptyset}>$ tampoco puede existir, debido al mismo *cycle link* de $n_{\{X\}}$, $<prv, nxt>$, ya que $n_{\emptyset}$ debería apuntar a $n_{\{X\}}$ por nxt y no lo hace.

- Por último, el enlace $<n_{\emptyset}, prv, n_{\{List\}}>$ es eliminado debido a que no satisface el *cycle link* $<prv, nxt>$ del nodo $n_{\emptyset}$, ya que $n_{\{List\}}$ no referencia a $n_{\emptyset}$ con el selector nxt.

En el apéndice A.4 se presenta cómo se utiliza esta información para poder llevar a cabo

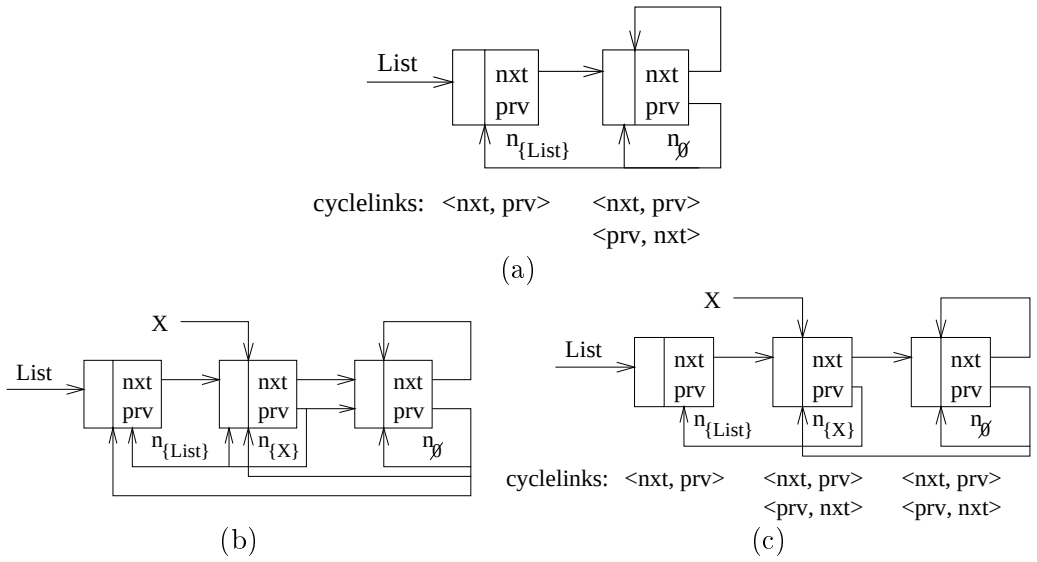


Figura 2.8: (a) SSG de una lista doblemente enlazada con la propiedad *cyclelinks*$^{\#}$ (b) Materialización sin *cyclelinks*$^{\#}$ (c) Materialización con *cyclelinks*$^{\#}$

esta eliminación de enlaces.

2.2 Resultados experimentales

El método original junto con las modificaciones propuestas en este capítulo han sido implementadas en un *pseudo compilador* simple, el cual lee código C y devuelve para cada sentencia del programa el SSG que describe todos los posibles estados de la memoria tras la ejecución del programa.

Se ha usado dicho compilador para analizar varios códigos simples que crean, modifican y borran listas, árboles binarios, etc. Para presentar los resultados obtenidos por el compilador, usaremos un código sintético, el cual reproduce las principales operaciones que se llevan a cabo en el algoritmo de factorización LU dispersa, basado en estructuras dinámicas de datos. Antes de presentar las características de este código sintético, de su estructura de datos y de los resultados obtenidos, vamos presentar algunos detalles de como el compilador genera los SSGs en base al código C.

2.2.1 Detalles del análisis

Lo primero que tenemos que decir es que hemos desarrollado el compilador para que sea capaz de interpretar los seis tipos de instrucciones básicas sobre punteros descritos al comienzo de este capítulo. Hemos trasladado la sintaxis original del método, que expresa las sentencias en *PASCAL*, a la sintaxis de C para describir dichas sentencias, puesto que pensamos que C es el lenguaje en el que más fuertemente se hace uso de estructuras dinámicas basadas en punteros. Este cambio no supone ningún tipo de modificación del método puesto que la semántica de las sentencias es exactamente la misma, tan solo varía la forma de expresarlas. Esto no afecta ni al compilador ni al análisis puesto que tan solo se modifica el "nombre" (sintaxis) de las sentencias pero no su "significado" (semántica).

Así, los seis tipos de instrucciones analizados por el compilador son:

- $x = NULL$ en sustitución de $x := nil$.
- $x = malloc()$ en sustitución de $x := new$.
- $x = y$ en vez de $x := y$.
- $x \rightarrow sel = NULL$ sustituyendo a $x.sel := nil$.
- $x \rightarrow sel = y$ sustituye a $x.sel := y$.
- $x = y \rightarrow sel$ en vez de $x := y.sel$.

Por lo tanto, nuestro compilador tan solo entiende esos seis tipos de sentencias con punteros además de las estructuras de control *if* y *while*. Cualquier otro tipo de sentencia que no utilice punteros no será analizada por el compilador puesto que no va a generar variación alguna en las estructuras dinámicas de datos ni en las variable puntero.

Debido a que el compilador sólo es capaz de analizar esas seis sentencias relacionadas con punteros, cualquier otra más compleja debe ser descompuesta a mano o mediante un preprocesador, en una serie de sentencias de esos seis tipos, usando variables temporales. Además, una misma variable puntero no puede aparecer en ambos lados de la sentencia, como en $x = x \rightarrow nxt$. Por ejemplo, las siguientes dos líneas de código sacan un elemento de una lista doblemente enlazada, apuntado por x, enlazando entre sí los elementos anterior y posterior al mismo:

```
...
x→nxt→prv = x→prv;
x→prv→nxt = x→nxt;
...
```

deberían ser reemplazadas por el siguiente código:

```
...
/* x→nxt→prv = x→prv; */
_tmp1 = x→nxt;
_tmp2 = x→prv;
_tmp1→prv = _tmp2;
_tmp1 = NULL;
_tmp2 = NULL;

/* x→prv→nxt = x→nxt; */
_tmp1 = x→prv;
_tmp2 = x→nxt;
_tmp1→nxt = _tmp2;
_tmp1 = NULL;
_tmp2 = NULL;
...
```

En este código se hace exactamente la misma operación que con las dos sentencias originales, pero utilizando únicamente sentencias pertenecientes a los seis tipos de instrucciones descritos anteriormente. Como podemos observar se hace uso de variables temporales no declaradas en el código original ($\_tmp1$, $\_tmp2$). Dichas variables temporales deben ser eliminadas inmediatamente después de su utilización.

En definitiva, en cada una de las sentencias tan solo puede aparecer una referencia por un selector. Si aparece más de una, se parte la sentencia en varias en las que sólo aparezca una referencia, usando para ello variables puntero temporales.

Además de las transformaciones anteriores, la implementación de este compilador impone que cualquier sentencia de la forma $lhs = rhs$ en la que $rhs \neq NULL$ debe ser inmediatamente precedida por una sentencia de asignación del tipo $lhs = NULL$. Así cada sentencia del tipo $x = malloc()$, $x = y$ y $x = y \rightarrow sel$ debe ser precedida por $x = NULL$, y la sentencia $x \rightarrow sel = y$ debe ser precedida por $x \rightarrow sel = NULL$. Esta transformación la lleva a cabo automáticamente el compilador, ejecutando la semántica abstracta de $x = NULL$ o $x \rightarrow sel = NULL$ dependiendo de la sentencia actual. Esto simplifica la semántica abstracta de las sentencias, aislando en sentencias distintas lo que es la generación de aristas en los grafos (ya sean *variable-edges* o *selector-edges*) producida por las sentencias $x = malloc()$, $x = y$, $x = y \rightarrow sel$ y $x \rightarrow sel = y$, de la eliminación de las mismas producida por las sentencias $x = NULL$ y $x \rightarrow sel = NULL$.

De esta forma, la semántica abstracta de la sentencia $x \rightarrow sel = y$ se ve simplificada. Ahora no tiene que tener en cuenta las modificaciones que son necesarias para eliminar el enlace por sel desde la porción de memoria apuntada por x.

Interpretación abstracta

Una vez realizadas todas las transformaciones sobre el código expuestas anteriormente, dicho código puede ser representado como un grafo de flujo de control $G = (V, A)$, donde V es el conjunto de vértices que representan a las sentencias del programa y $A \subseteq V \times V$ es el conjunto de arcos dirigidos del grafo que representan las transferencias de control entre las sentencias del código. El grafo G tiene un único vértice inicial el cual no tiene predecesores al cual denominamos $ROOT$.

El algoritmo de análisis de forma es un proceso iterativo de análisis de flujo de datos [60], que calcula un SSG, $SSG_v^\#$, para cada vértice v del grafo de flujo de control, como el punto fijo donde del grafo $SSG_v^\#$ no cambia más, como:

$$SSG_v^\# = \begin{cases} SSG_\emptyset^\# \text{ si } v = ROOT \\ \cup_{<u,v>\in A}[st(v)]^\#(SSG_u^\#) \text{ en otro caso.} \end{cases}$$

donde $SSG_\emptyset^\#$ es un SSG vacío (sin nodos ni aristas) representando el estado de la memoria inicial y $[st(v)]^\#(SSG_u^\#)$ representa la aplicación de la semántica abstracta de la sentencia $(st(v))$ del vértice $v \in V$ sobre los SSGs de los vértices predecesores $(SSG_u^\#)$. Como se puede observar, el SSG que se obtiene para la sentencia del vértice v, $SSG_v^\#$, se construye uniendo todos los SSGs resultantes de aplicar la semántica abstracta de la sentencia sobre los SSGs de las sentencias predecesoras (vértices que tienen un arco hacia v).

El compilador se ha implementado de manera que hay una estructura ST_i asociada a cada sentencia del programa en la cual se mantiene información del tipo de sentencia junto con las variables y selectores usados en la misma $(Sentencia(ST_i))$. Además se mantiene también el SSG obtenido para dicha sentencia $(SSG(ST_i))$. Todas las estructuras que representan sentencias del programa están enlazadas de manera que se puede conocer para cualquier sentencia ST_i, cuales son las sentencias predecesoras $(Predecesores(ST_i))$ y sucesoras $(Sucesores(ST_i))$ de la misma.

El núcleo del compilador consiste en un bucle que va analizando iterativamente las sentencias almacenadas en una lista de trabajo $(ListaT)$. Esta lista de trabajo se inicializa con todas las sentencias del código. La primera sentencia del código (ST_1), tiene como predecesora una

sentencia vacía (ST_0) que lo único que hace es introducir el SSG vacío como grafo inicial. A continuación presentamos un *pseudocódigo* que representa el proceso de análisis de un código llevado a cabo por nuestro compilador:

```
forall (ST_i)
  Insertar (ST_i, ListaT);
while (ListaT != NULL)
 {
   Extraer(ST_i, ListaT);
   SSG_i = SSG(ST_i);
   forall (ST_j ∈ Predecesores(ST_i))
     {
       SSG_j = Interpretacion_Abstracta(ST_i, SSG(ST_j));
       SSG_i = Union(SSG_i, SSG_j);
     }
   if (SSG_i != SSG(ST_i))
     {
       SSG(ST_i) = Union(SSG(ST_i), SSG_i);
       forall (ST_j ∈ Sucesores(ST_i))
         Insertar (ST_j, ListaT);
     }
 }
```

Como se puede observar, en cada paso del bucle se extrae una sentencia ST_i de la lista de trabajo la cual debe ser analizada. Dicho análisis consiste en la aplicación de la semántica abstracta asociada con dicha sentencia sobre todos los SSGs de las sentencias predecesoras de ST_i. Se hace la unión de todos grafos obtenidos tras la interpretación abstracta de la sentencia, y si el nuevo grafo obtenido es distinto del que existía en ST_i, se actualiza $SSG(ST_i)$ y se introduce todos los sucesores de dicha sentencia en la lista de trabajo. Esto hace que mientras el SSG asociado con una sentencia cambie, sus sucesores serán de nuevo interpretados sobre el nuevo grafo que representa estados de memoria no representados anteriormente. El proceso terminará cuando el SSG asociado con todas y cada una de las sentencias del código no cambia y por tanto representa todos los posibles estados de la memoria para cada sentencia.

2.2.2 Código ejemplo

Se ha escogido como base para nuestro código ejemplo, el algoritmo de factorización LU de matrices dispersas, ya que es un buen caso de estudio al ser utilizado en otros muchos problemas irregulares para resolver sistemas lineales no simétricos dispersos.

Normalmente, para obtener un ahorro en tiempo y memoria, las entradas cero de las matrices dispersas no son explícitamente almacenadas. Hay una amplia gama de métodos para almacenar los elementos distintos de cero de las matrices dispersas ([5][23]). Nosotros sólo hemos considerado los que utilizan listas enlazadas para representar las matrices dispersas.

La descomposición LU puede almacenar los coeficientes de la matriz dispersa en listas doblemente enlazadas que representan las columnas de la matriz, en las que cada elemento de la lista contiene una entrada no cero junto con el índice de la fila a la que pertenece dicha entrada. Junto con estos datos, existen dos punteros a los elementos no cero anterior y posterior en la columna. Existe una lista cabecera, también doblemente enlazada, en la cual hay un nodo por cada columna de la matriz que apunta al primer elemento de la lista doblemente enlazada que representa dicha columna. En la figura 2.9 podemos ver una representación gráfica de la estructura dinámica que mantiene los elementos de la matriz dispersa. Esta estructura se denomina *LLCS* (Linked List Column Storage) y es presentada en [1].

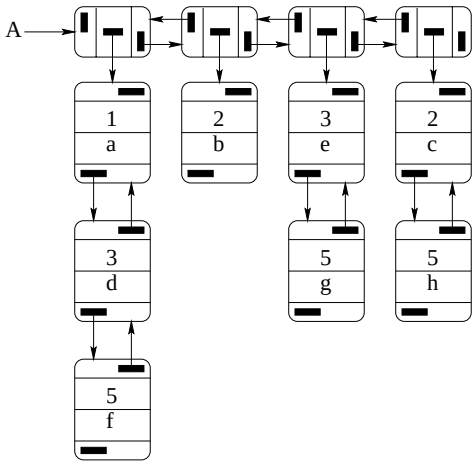


Figura 2.9: Representación de una matriz dispersa.

El usar listas enlazadas donde sólo se mantienen los elementos no ceros de la matriz permite un acceso muy rápido a dichos elementos por columnas, además de permitir hacer permutación de columnas fácilmente. Aparte, al ser listas doblemente enlazadas, la inserción y borrado de elementos de la matriz también es inmediata.

Analizando las principales operaciones realizadas por el algoritmo de descomposición LU de una matriz dispersa representada por litas doblemente enlazadas, hemos creado un código que reproduce la creación de la estructura que representa la matriz, junto con las operaciones más relevantes del código, como son el recorrido de la lista cabecera en busca de una columna, el recorrido de una columna, la inserción y el borrado de un elemento en una columna.

A continuación presentamos el código analizado:

```
 1:    A = malloc();
 2:    aux1col = A;
       /* creacion de la matriz */
 3:    while (condicion1)
         {
 4:        new = malloc();
 5:        aux1 = new;
           /* creacion de una columna */
 6:        while (condicion2)
             {
 7:            aux2 = malloc();
 8:            aux1→nxt = aux2;
 9:            aux2→prv = aux1;
10:            aux1 = aux2;
             }
11:        aux1col→col = new;
12:        new = NULL;
13:        aux1 = NULL;
14:        aux2 = NULL;
15:        aux2col = malloc();
16:        aux1col→nxt = aux2col;
17:        aux2col→prv = aux1col;
18:        aux1col = aux2col;
         }
19:    aux2col = NULL;
20:    aux1col = NULL;
```

```
        /* recorrido sobre la lista cabecera apuntada por A, con puntero aux1col */
21:     aux1col = A;
22:     while (condicion3)
          {
23:         aux2col = aux1col→nxt;
24:         aux1col = aux2col;
          }
25:     aux2col = NULL;
        /* recorrido de la columna apuntada por aux1col, con puntero aux1 */
26:     aux1 = aux1col→col;
27:     while (condicion4)
          {
28:         aux2 = aux1→nxt;
29:         aux1 = aux2;
          }
        /* insercion de un nuevo elemento depues de aux1*/
30:     new = malloc();
31:     aux2 = aux1→nxt;
32:     new→nxt = aux2;
33:     aux1→nxt = new;
34:     new→prv = aux1;
35:     aux2→prv = new;
36:     aux2 = NULL;
37:     new = NULL;
        /* continua recorrido de la columna apuntada por aux1col, con puntero aux1 */
38:     while (condicion4)
          {
39:         aux2 = aux1→nxt;
40:         aux1 = aux2;
          }
        /* borrado del elemento apuntado por aux1*/
41:     aux2 = aux1→nxt;
42:     new = aux1→prv;
43:     aux1→nxt = NULL;
44:     aux1→prv = NULL;
45:     aux1 = NULL;
46:     aux2→prv = new;
47:     new→nxt = aux2;
48:     aux2 = NULL;
49:     new = NULL;
50:     aux1col = NULL;
```

Como se puede observar, en el código anterior tan solo aparecen los seis tipos de instrucciones con punteros que reconoce el compilador. Las asignaciones a campos dentro de las estructuras que no son puntero (como por ejemplo el valor de una entrada o la fila a la que pertenece) no son tenidas en cuenta puesto que no van a cambiar la conectividad de la estructura de datos. Además, el código anterior no es exactamente el analizado puesto que falta insertar antes de cualquier asignación del tipo $pvar = ...$ y $pvar \rightarrow sel = ...$ las correspondientes $pvar = NULL$ y $pvar \rightarrow sel = NULL$ respectivamente. Dichas sentencias son insertadas automáticamente por el compilador durante el análisis.

Brevemente podemos comentar que el bucle de la sentencia 3 crea la estructura completa apuntada por la variable A. En cada paso de dicho bucle se crea una nueva lista doblemente enlazada que representa una nueva columna, apuntada por la variable *new*. Dicha creación se lleva a cabo en el bucle de la sentencia 6. Una vez creada la columna, se crea una nueva entrada de la lista cabecera y se hace que apunte por el selector *col* a la nueva columna creada. Al terminar el bucle 3, la variable A apunta a una estructura similar a la mostrada en la figura

2.9.

A continuación, en el bucle 22 se recorre la lista cabecera, dejando la variable $aux1col$ apuntando a uno de sus elementos. El bucle de la sentencia 27 recorre la columna apuntada por $aux1col \rightarrow col$ quedando la variable $aux1$ apuntando a una de sus entradas. Entre las sentencias 30 y 37 se inserta un nuevo elemento justo después de $aux1$.

Por último, se sigue recorriendo la columna apuntada por $aux1col \rightarrow col$ con el puntero $aux1$ en el bucle de la sentencia 38. La entrada apuntada por dicha variable al finalizar el recorrido es eliminada de la lista doblemente enlazada entre las sentencias 41 y 50.

Antes de mostrar los resultados obtenidos por el compilador tras analizar el código vamos a mostrar un ejemplo paso a paso, de la creación de una columna en el bucle de la sentencia 6. En la tabla 2.1 mostramos los grafos obtenidos para las sentencias 6, 9 y 10 en distintas iteraciones del bucle.

Como se puede observar, el grafo que representa la configuración de memoria para la sentencia 6 la primera vez que se entra (iteración 1), consta tan solo de dos nodos. El nodo $n_{\{A,aux1col\}}$ representa la porción de memoria creada en la sentencia 1 y el nodo $n_{\{aux1,new\}}$ la creada en la sentencia 4. Tras analizar las sentencias 7, 8 y 9 sobre este grafo de entrada, se obtiene el mostrado en la fila 1 de la tabla para la sentencia 9. En el podemos observar que aparece un nuevo nodo, $n_{\{aux2\}}$, apuntado por $aux2$ que representa la porción de memoria reservada en la sentencia 7. Vemos que está doblemente enlazado con el nodo $n_{\{aux1,new\}}$ por los selectores prv y nxt. La creación de estos enlaces se ha llevado a cabo en las sentencias 8 y 9. Tras el análisis de esta última se han creado un *cycle link* en cada nodo, $<nxt, prv>$ en $n_{\{aux1,new\}}$ y $<prv, nxt>$ en $n_{\{aux2\}}$. Esta información recoge la relación cíclica entre dichos enlaces y las porciones de memoria representadas por los nodos. En el grafo correspondiente a la sentencia 10 en la iteración 1 podemos ver que tan solo se diferencia del de la sentencia 9 en que la variable $aux1$ ahora apunta al mismo nodo que $aux2$.

En la segunda iteración, el grafo obtenido para la sentencia 6 es la unión de los de sus sentencias predecesoras (5 y 10). En este grafo vemos como las variables $aux1$ y $new1$ apuntan a dos nodos distintos. El nodo $n_{\{aux1,new\}}$ representa la porción de memoria apuntada por ambas variables correspondiente al estado de memoria que proviene de 5. Por otro lado los nodos $n_{\{new\}}$ y $n_{\{aux1,aux2\}}$ representan a las correspondientes de la sentencia 10 (donde ahora $aux1$ apunta al siguiente nodo apuntado por new). Tras ejecutar de nuevo las sentencias $7-9$ sobre el nuevo grafo, obtenemos el mostrado en la fila 2 para la sentencia 9, en el que se han enlazado los nodos apuntados por $aux1$ y $aux2$ como indican dichas sentencias. Ahora el nodo $n_{\{aux1\}}$ es anotado con dos *cycle links*, $<nxt, prv>$ y $<prv, nxt>$, puesto que ahora representa el elemento central de una lista de tres elementos. Además la información $is^{\#}(n_{\{aux1\}}) = true$ puesto que dicho nodo es referenciado por dos nodos distintos ($n_{\{new\}}$ y $n_{\{aux2\}}$). Sin embargo $is\_sel^{\#}(n_{\{aux1\}}, nxt) = false$ y $is\_sel^{\#}(n_{\{aux1\}}, prv) = false$, puesto que no hay dos nodos que lo referencien por el mismo selector. Hay que decir que esta información $is\_sel^{\#}$ se va a mantener *false* para todos los nodos y selectores de los grafos presentados.

Tras el análisis de la sentencia 10 vemos que aparece el nodo $n_{\emptyset}$ que representa ese elemento central que ahora no es apuntado por ninguna variable.

En la iteración tercera, los cambios sobre los SSGs de cada sentencia son similares a los descritos anteriormente. Cabe destacar que tras el análisis de la sentencia 10, el nodo $n_{\emptyset}$ ahora representa a los dos elementos centrales de la lista, de ahi que existan enlaces por nxt y prv hacia él mismo, representando los enlaces existentes entre dichos elementos centrales. Aunque el nodo $n_{\emptyset}$ es *shared*, se puede deducir que cualquier recorrido por las porciones de memoria

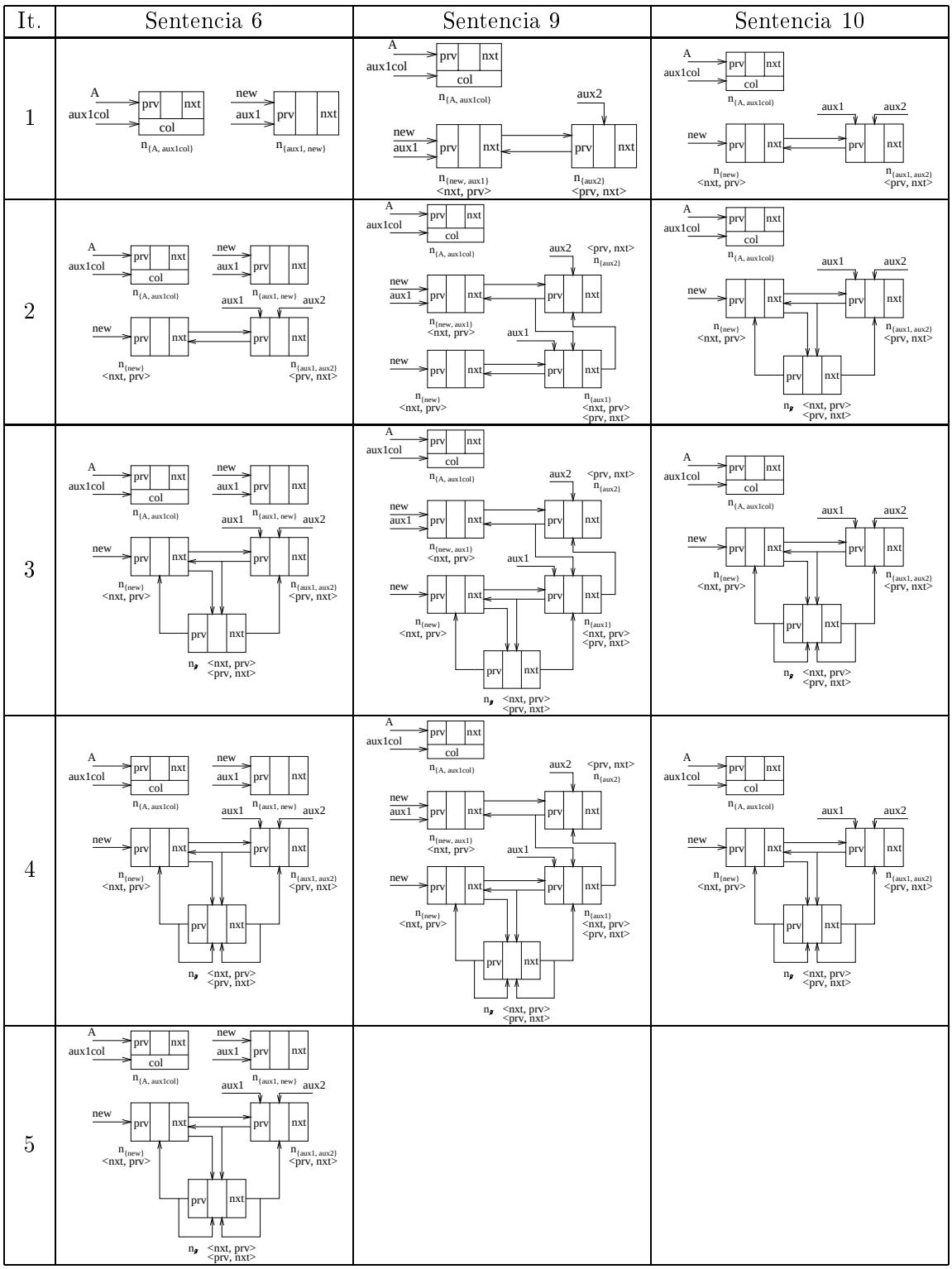


Tabla 2.1: SSGs obtenidos para las sentencias 6, 9 y 10 en las sucesivas iteraciones del bucle.

representadas por dicho nodo no visitaría dos veces la misma puesto que la información *shared por selector* nos dice que ninguna de dichas porciones es referenciada dos veces por el mismo selector.

La cuarta iteración nos lleva a que en la sentencia 10 se obtenga el mismo grafo que para la iteración anterior. Esto es debido a que ahora el nodo $n_{\emptyset}$ representa todas las porciones de memoria intermedias de la lista, ya sean dos o tres. Como se puede observar, ya se ha alcanzado un punto fijo, de manera que si seguimos iterando siempre se obtendrá el mismo SSG, puesto que las nuevas porciones reservadas en la sentencia 7 irán siendo representadas por el nodo $n_{\emptyset}$. Por tanto el grafo obtenido representa una lista doblemente enlazada de longitud indeterminada.

El grafo para la sentencia 6 que se pasa a la sentencia 11 es el mostrado en la fila 5 de la tabla. Como se puede observar es exactamente el mismo que el de la fila 4. De este grafo cabe destacar que el nodo $n_{\emptyset}$ es *shared* pero la información *shared por selector* es *false* para todos los nodos y selectores.

Por último en la figura 2.10 podemos ver los grafos obtenidos para las sentencias 11 y 14 obtenidos a partir del grafo anterior.

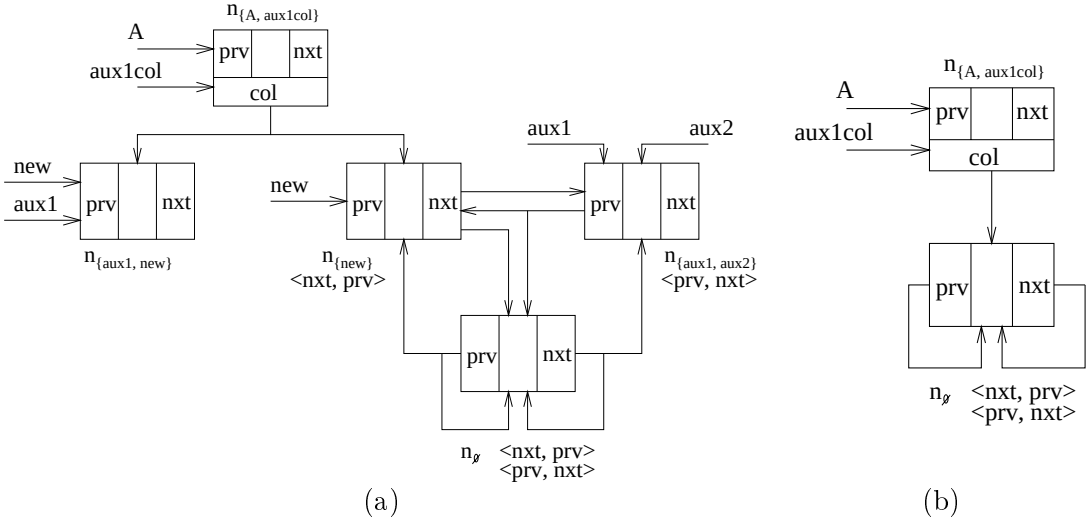


Figura 2.10: (a) SSG de la sentencia 11. (b) SSG de la sentencia 14.

Como podemos observar en el grafo de la sentencia 11 (fig. 2.10 (a)), se ha creado un enlace por el selector *col* desde el nodo $n_{\{A,aux1col\}}$ a los nodos $n_{\{aux1,new\}}$ y $n_{\{new\}}$ como indica la semántica abstracta de dicha sentencia. En el SSG correspondiente a la sentencia 14 mostrado en la figura 2.10 (b) podemos ver que tan solo hay dos nodos, $n_{\{A,aux1col\}}$ y $n_{\emptyset}$. Esto es debido a que tras la *nulificación* de las variables $aux1$, $aux2$ y new, todos los nodos son sumarizados en $n_{\emptyset}$ puesto que son del mismo *tipo* y pertenecen a la misma *estructura*. Por tanto el nodo $n_{\emptyset}$ está representando todos los elementos de la lista doblemente enlazada que forma la columna apuntada desde la porción de memoria referenciada por A. De la información mantenida para este nodo podemos deducir que un recorrido de esta columna por un selector único (nxt o prv) nunca visitará dos veces el mismo elemento.

Tras el análisis del bucle de la sentencia 3, en la sentencia 20 obtenemos el SSG de la figura 2.11 (a) que representa la estructura que almacena la matriz dispersa (ver figura 2.9). Este mismo grafo es obtenido tras la ejecución de la última sentencia del código, tras haber

seleccionado una columna cualquiera, haber insertado un elemento en la misma y posteriormente haber borrado otro. Como podemos observar el método es capaz de determinar que la forma de la estructura tras estas operaciones es exactamente igual a la inicialmente generada. El grafo obtenido por el método original tras el bucle de la sentencia 3 es el mostrado en la figura 2.11 (b).

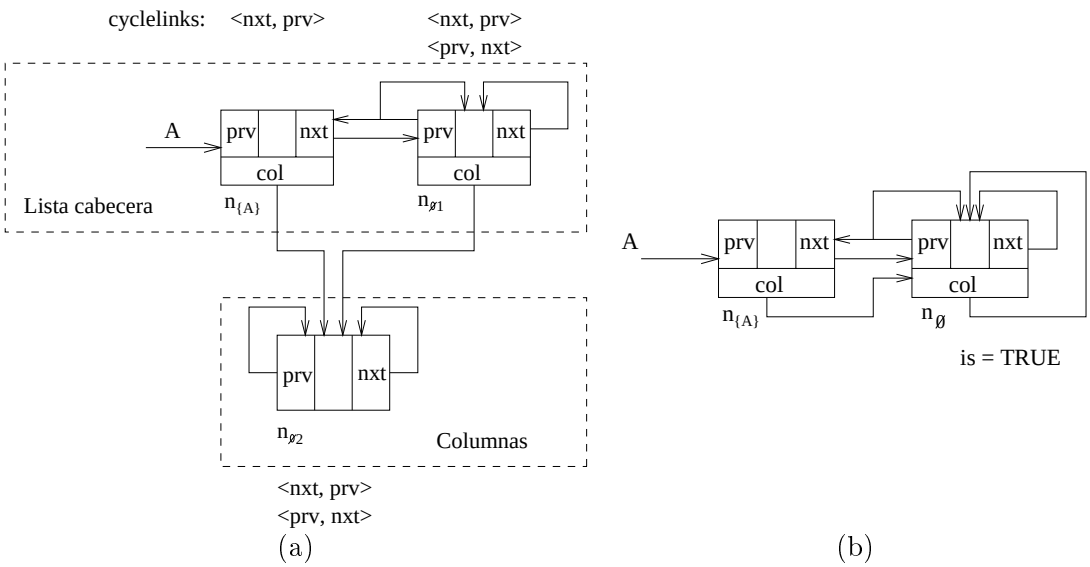


Figura 2.11: SSG para la matriz dispersa: (a) método modificado (b) método original.

Como se puede observar en el SSG del método original (fig. 2.11 (b)), hay tan solo un nodo sumario $n_\emptyset$ que representa a todos los elementos de la lista cabecera y de las listas que representan las columnas. Además dicho nodo es *shared*, lo que implica que:

- Si recorro la estructura apuntada por A por el selector nxt es posible visitar dos veces el mismo nodo (puesto que al ser *shared* una porción de memoria podría ser referenciada más de una vez por el selector nxt).
- Por el selector *col* desde el nodo apuntado por A se puede llegar a nodos de la lista cabecera, puesto que es *shared* y se mantiene un solo nodo sumario para los dos tipos de porciones de memoria (los de la lista cabecera y los de las columnas). Es decir, por *col* se podría acceder a nodos de la lista cabecera.
- Las listas que representan las columnas, al estar representadas todas por $n_\emptyset$, podrían compartir elementos entre si, es decir un mismo elemento podría pertenecer a más de una columna.

Como podemos ver, las características deducidas del SSG obtenido por el método original no reflejan las peculiaridades de la estructura de datos, evitando así poder hacer cualquier tipo de optimización o paralelización en base a la posible forma de la estructura de datos.

La cosa cambia si nos fijamos en el SSG obtenido por el método modificado, mostrado en la figura 2.11 (a). Como podemos observar en dicho grafo, existen tres nodos. El que representa la porción de memoria referenciada por A, un nodo sumario $n_{\emptyset 1}$ que representa las porciones pertenecientes a la lista cabecera y otro, $n_{\emptyset 2}$, representando los elementos de las listas que mantienen las columnas. Aunque ambos nodos sumarios son *shared*, la información

shared por selector es *false* para todos los nodos y selectores. En base a dicho grafo se puede deducir la siguiente información sobre la forma de la estructura de datos apuntada por A:

- La lista cabecera representada por los nodos $n_{\{A\}}$ y $n_{\emptyset 1}$ es acíclica si se recorre utilizando tan solo uno de los selectores nxt o prv. Esto es así ya que al ser $is\_sel^{\#}(n_{\emptyset 1}, nxt)$ =*false* las porciones representadas por el nodo tan solo son referenciadas como máximo por otra con el selector nxt. Esta información indica que ninguna de dichas porciones de memoria puede ser visitada dos veces si sólo se utiliza el selector nxt para hacer el recorrido. La misma deducción se puede hacer para el selector prv.

- Los elementos de la lista cabecera apuntan a distintos elementos de las listas que representan las columnas, puesto que $is\_sel^{\#}(n_{\emptyset 2}, col)$ =*false*. Al no existir ninguna porción de memoria en $n_{\emptyset 2}$ apuntada más de una vez por col es imposible que dos elementos de la lista cabecera apunten a la misma columna.

Como podemos observar la información proporcionada por el nuevo método si nos da una idea más real de la forma de la estructura de datos, información que podría ser tenida en cuenta para posibles optimizaciones o paralelizaciones de recorridos sobre la estructura de datos creada por el código.

3 Conjunto reducido de grafos de forma de referencias: RSRSG

Las modificaciones presentadas en el capítulo anterior sobre el método de Sagiv ([67]) nos han permitido analizar con éxito estructuras mucho más complejas que las soportadas por el método original. Sin embargo, esta ampliación en el soporte de estructuras más complejas no es suficiente para analizar con éxito núcleos de códigos reales que hacen uso de este tipo de estructuras.

Así, por ejemplo, hemos visto que el método modificado es capaz de analizar con éxito un código que crea la estructura usada en el algoritmo de factorización LU dispersa basado en listas doblemente enlazadas y la somete a distintas operaciones que tienen lugar en dicho algoritmo. Sin embargo, es incapaz de analizar con éxito el núcleo del algoritmo que lleva a cabo la factorización LU en el cual nos encontramos con muchos bucles anidados en los que se realizan distintas acciones dependiendo de diversas condiciones por medio de estructuras de control condicionales (*IF*).

El problema fundamental cuando nos enfrentamos con códigos reales es que una misma sentencia es alcanzada siguiendo muy distintos caminos en el flujo de control. El método basado en los SSGs, asocia a cada sentencia tan solo un grafo en el cual se mezclan todos los estados de memoria que pueden aparecer en dicha sentencia provenientes de muy diversos caminos del flujo de control. Esto nos hace mezclar la información de muy diversas porciones de memoria, siempre siendo conservativos, por lo que al final la información es demasiado general. Además de esta mezcla de muy diversos estados de memoria que pueden alcanzar una sentencia, está el hecho de que la información almacenada en cada nodo además del esquema de nombres utilizado para sumarizar varios de ellos, hacen que se mezclen nodos que representan porciones de memoria con una serie de características singulares que si se mantuvieran por separado evitarían la pérdida de información que se produce en el método basado en los SSGs.

Para obtener un procedimiento capaz de analizar con cierta exactitud las estructura de datos dinámicas usadas en códigos reales, hemos desarrollado un nuevo método. Este nuevo método se basa también en representar los estados de memoria que pueden aparecer tras la ejecución de una sentencia mediante grafos de forma que describen las estructuras dinámicas de datos apuntadas desde las variables puntero del código. El análisis consistirá de nuevo en la interpretación abstracta de cada sentencia del código para obtener la representación del estado de la memoria tras la ejecución de la sentencia. La diferencia es que en este nuevo

método vamos a mantener más de un grafo para cada sentencia del programa evitando así que se mezcle la información de estados muy distintos de la memoria. Además vamos a permitir la existencia de muchos más nodos en cada grafo, evitando de esta forma que se mezcle la información de porciones de memoria con características diferentes. Por último, a la hora de manipular los grafos, se ha añadido una fase que *enfoca* los nodos para que de este modo se modifiquen con mayor precisión.

3.1 Descripción del método

Básicamente, nuestro método se basa en aproximar todas las posibles configuraciones de memoria que pueden aparecer tras la ejecución de una sentencia en el código. Una sentencia del código puede ser alcanzada siguiendo muy diversos caminos del grafo de flujo de control. Cada uno de estos caminos del grafo de flujo de control tiene asociada una configuración de memoria que es modificada por cada una de las sentencias atravesadas por dicho camino. Por lo tanto, una sentencia del código modificará todas las configuraciones de memoria asociadas con todos los caminos del flujo de control que pasan a través de ella.

Cada una de estas configuraciones de memoria será representada por un grafo finito al que denominaremos *Grafo de Forma de Referencias* (Reference Shape Graph) o RSG. El nombre hace referencia a que las propiedades fundamentales que van a determinar la composición de los RSGs están muy relacionadas con las *referencias* de las que forma parte toda porción de memoria.

Los RSGs son grafos en los que los nodos representan porciones de memoria que tienen *patrones de referenciado* similares, y las aristas representan los enlaces entre las porciones de memoria representadas por los nodos. Como veremos, un nodo va a representar varias porciones de memoria si dichas porciones de memoria referencian a otras y son referenciadas de forma similar, con lo que se conseguirá mantener las características fundamentales de los enlaces entre porciones de memoria de una forma bastante exacta, al evitar que se mezclen la información de porciones de memoria con patrones de referenciado muy distintos. Pero no sólo se van a tener en cuenta estos patrones a la hora de decidir si dos porciones de memoria pueden o no ser representadas por un mismo nodo, sino que vamos a asociar una serie diversa de propiedades a cada porción de memoria, que mantendrán información de distintos aspectos relacionados con la porción de memoria. Esta serie de propiedades son las que determinan la exactitud con la que los grafos van a representar las estructuras de datos ya que van a determinar cuando dos porciones de memoria serán representadas por un mismo nodo y por tanto van a condicionar de alguna forma el número de nodos que van a aparecer en cada grafo. Para determinar si dos porciones de memoria serán representadas por un mismo nodo, basta con ver si el conjunto de propiedades de ambas porciones es similar. De igual manera, veremos que dos nodos podrán ser *sumarizados* en uno sólo si representan porciones de memoria con propiedades similares. Es por este motivo que se puede representar una configuración de memoria de tamaño indeterminado por medio de un grafo finito, ya que el número de nodos que pueden aparecer en el mismo está limitado por las propiedades asociadas a cada nodo.

Si tenemos en cuenta lo expuesto anteriormente, podemos ver que cada sentencia del código va a tener asociado un conjunto de RSGs que van a describir todas las formas que pueden tomar las estructuras dinámicas de datos referenciadas desde las variables puntero del programa. Lo ideal sería mantener todos los RSGs distintos que se generan para una determinada sentencia, puesto que así se mantendrían por separado todas las configuraciones de memoria

distintas. El problema es que el número de grafos asociados a cada sentencia es demasiado elevado y hace impracticable el análisis, tanto por la cantidad de memoria necesaria para su almacenamiento como por el tiempo necesario para su análisis. Es por este motivo que nuestro método asocia a cada sentencia un *Conjunto Reducido de Grafos de Forma de Referencias* (Reduced Set of Reference Shape Graphs) o RSRSG. Es decir, no todos los RSGs distintos que se obtengan para una sentencia se van a mantener individualmente, sino que varios de estos RSGs, serán fundidos en uno solo que representará las configuraciones de memoria representadas por los grafos originales. Sólo se unirán grafos que representen configuraciones de memoria similares, para que la pérdida de precisión sobre las configuraciones representadas, tras la unión sea mínima. Por tanto, como veremos, para que dos grafos sean unidos tendrán que satisfacer una serie de condiciones. Al igual que las propiedades de los nodos limitan el número de los mismos en un RSG, las condiciones para la unión de los grafos también ponen un límite el número máximo de RSGs que pueden estar asociados a una sentencia. Esta unión de grafos reduce drásticamente el número de RSGs asociados a cada sentencia haciendo que el análisis sea factible, reduciendo los requerimientos de memoria y tiempo durante el análisis.

En la figura 3.1 podemos ver de forma esquemática todo lo comentado anteriormente. Vemos como cada sentencia tiene asociado un RSRSG que en puede estar formado por un sólo grafo RSG (sentencias 1, 2, 3, 4, 5 y 6), o por varios (sentencia 7), provenientes de distintos caminos del flujo de control y que no representan configuraciones de memoria similares. Como se puede apreciar, el RSRSG de la sentencia 7 está formado por dos RSGs, uno proviene del grafo del RSRSG de la sentencia 6, y el otro de la unión de los grafos pertenecientes a las sentencias 4 y 5, que según nuestros criterios, que se verán en este capítulo, son compatibles y pueden ser fundidos.

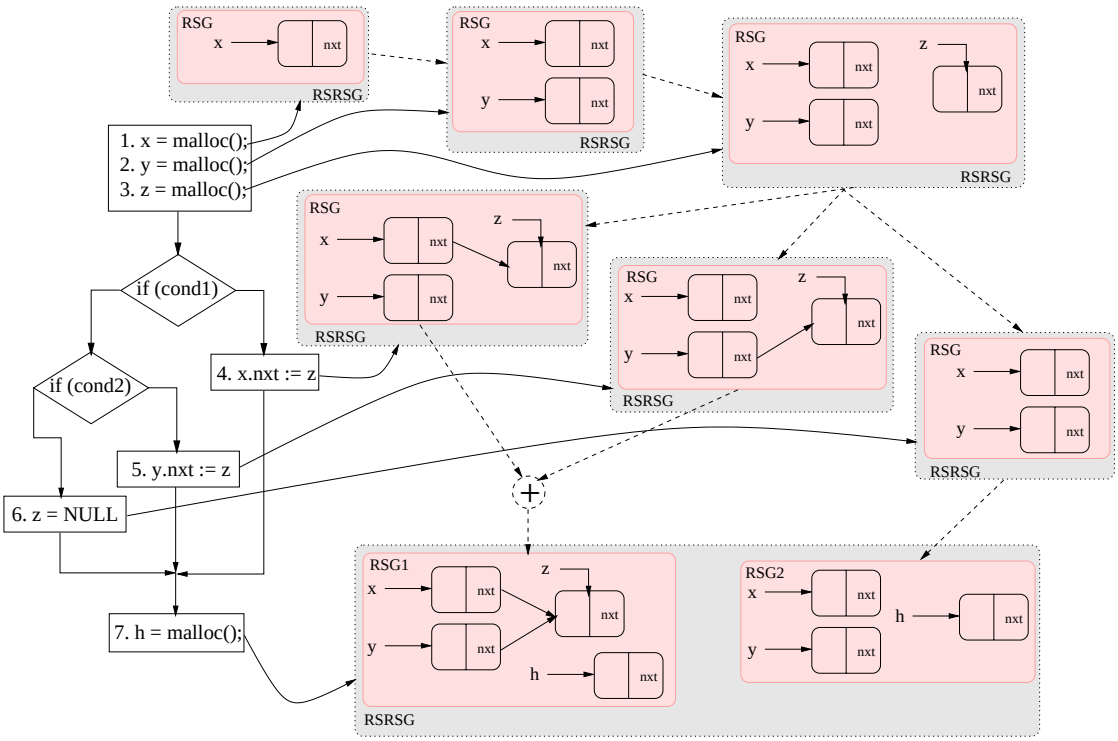


Figura 3.1: Visión esquemática de los RSRSGs.

El cálculo del RSRSG asociado a cada sentencia se lleva a cabo como con los SSGs, por

medio de la *ejecución simbólica* del código sobre los grafos. Es decir, cada sentencia modifica los RSRSGs de sus sentencias predecesoras, para reflejar los cambios que se producen por la ejecución de dicha sentencia sobre las configuraciones de memoria representadas por los mismos. Por tanto, la *ejecución simbólica* del código consiste en la interpretación abstracta de cada sentencia del código, iterativamente hasta que se alcanza un punto fijo en el que el RSRSG asociado a cada una de las sentencias del código no cambia más. Esto nos lleva a tener que definir una *semántica abstracta* para cada sentencia, que especifica las modificaciones que hay que llevar a cabo sobre un RSRSG, de manera que el RSRSG modificado represente de una forma lo más exacta posible, las nuevas configuraciones de memoria tras la ejecución de la sentencia.

Al igual que con el método basado en los SSGs, tan solo son analizadas las seis sentencias básicas en C que utilizan punteros:

- $x = NULL$
- $x = malloc()$
- $x = y$
- $x \rightarrow sel = NULL$
- $x \rightarrow sel = y$
- $y = x \rightarrow sel$

Sentencias más complejas que utilicen punteros deberán ser descompuestas en una serie de sentencias de esos seis tipos como ya se expuso en la sección 2.2.1.

Para aumentar la exactitud con la que los RSRSGs representan las estructuras reales utilizadas por el código, hemos visto que se puede obtener una información muy valiosa de las estructuras condicionales utilizadas en el código. Por ejemplo, es muy frecuente encontrar en códigos C basados en estructuras dinámicas enlazadas, patrones del tipo: $while(x! = NULL)...$, donde x es una variable puntero que recorre cierta estructura enlazada. En este caso, se puede asegurar que en el punto de entrada al cuerpo del bucle se cumple la condición de que la variable $x \neq NULL$. Del mismo modo, para una terminación normal del bucle, se cumple que justo después del cuerpo del bucle la variable $x = NULL$. Hemos usado esta información para reducir el número de RSGs y la complejidad de los mismos en dichas situaciones. Más concretamente, hemos creado una serie de *pseudoinstrucciones* que hemos denominado *FORCE* que son insertadas en el código antes del análisis para ser interpretadas como una instrucción más. Las *pseudoinstrucciones* creadas son:

- $FORCE(x == NULL)$
- $FORCE(x! = NULL)$
- $FORCE(x == y)$
- $FORCE(x! = y)$
- $FORCE(x \rightarrow sel == NULL)$

Al igual que para los seis tipos básicos de instrucciones con punteros, hemos tenido que definir la *semántica abstracta* de estas *pseudoinstrucciones*. En el apéndice B.7 se presentan más en profundidad dichas *pseudoinstrucciones*.

3.2 Grafos de Forma de Referencias: RSG

Antes de definir los RSGs necesitamos presentar la notación usada para describir las diferentes configuraciones de memoria que pueden aparecer en la ejecución de un programa.

Definición 3.2.1 Llamamos *configuración de memoria* a una colección de estructuras dinámicas. Estas estructuras de memoria están formadas por *porciones de memoria* que están enlazadas entre sí mediante referencias. Dentro de las porciones de memoria hay sitio para datos y para punteros a otras porciones de memoria. Estos punteros son denominados selectores.

Representamos una configuración de memoria mediante la tupla $M = (L, P, S, PS, LS)$, donde:

- **L** es el conjunto de porciones de memoria.
- **P** es el conjunto de variables puntero (**pvars**) declaradas en el programa.
- **S** es el conjunto de selectores declarados en las estructuras de datos del programa.
- **PS** es el conjunto de referencias desde *pvars* a porciones de memoria, de la forma $<pvar, l>$, donde la variable puntero $pvar \in P$ apunta a la porción de memoria $l \in L$.
- **LS** es el conjunto de enlaces entre porciones de memoria, de la forma $<l_1, sel, l_2>$ donde la porción de memoria $l_1 \in L$ referencia a la porción $l_2 \in L$ por medio del selector $sel \in S$.

Usaremos $L(m)$, $P(m)$, $S(m)$, $PS(m)$ y $LS(m)$ para referirnos a cada uno de esos conjuntos para una configuración de memoria m determinada. □

En la figura 3.2 presentamos un ejemplo de una configuración de memoria M, (en concreto se trata de una lista doblemente enlazada apuntada por la variable x). En este caso, los conjuntos definidos anteriormente quedan de la siguiente manera: $L(M) = \{l_1, l_2, l_3, l_4\}$, $P(M) = \{x\}$, $S(M) = \{nxt, prv\}$, $PS(M) = \{< x, l_1 >\}$ y $LS(M) = \{< l_1, nxt, l_2 >, < l_2, prv, l_1 >, < l_2, nxt, l_3 >, < l_3, prv, l_2 >, < l_3, nxt, l_4 >, < l_4, prv, l_3 >, \}$. Debemos indicar que cuando presentemos en alguna figura una configuración de memoria, sus porciones de memoria serán representadas con rectángulos con sus esquinas rectas.

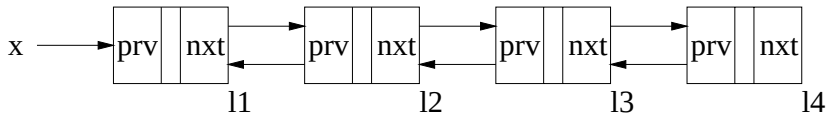


Figura 3.2: Ejemplo de configuración de memoria.

Por tanto, el RSRSG asociado a una sentencia de un programa será una aproximación de todas las configuraciones de memoria M que pueden aparecer tras la ejecución de dicha sentencia.

Una vez conocida la definición de una configuración de memoria, vamos a describir ahora los RSGs.

Definición 3.2.2 Un RSG es un grafo representado por la tupla $RSG = (N, P, S, PL, NL)$ donde:

- **N** es el conjunto de nodos del grafo. Cada nodo representa varias porciones de memoria si tienen una serie de propiedades en común.
- **P** es el conjunto de variables puntero (**pvars**) declaradas en el programa.
- **S** es el conjunto de selectores declarados en las estructuras de datos.
- **PL** es el conjunto de referencias desde pvars a nodos, de la forma $<pvar, n>$, donde la variable $pvar \in P$ apunta al nodo $n \in N$.
- **NL** es el conjunto de enlaces entre nodos, de la forma $<n_1, sel, n_2>$, donde el nodo $n_1 \in N$ referencia al nodo $n_2 \in N$ por el selector $sel \in S$.

Usaremos la notación $N(rsg)$, $P(rsg)$, $S(rsg)$, $PL(rsg)$ y $NL(rsg)$ para hacer referencia a dichos conjuntos para un grafo rsg dado. □

En la figura 3.3 presentamos un ejemplo de un RSG, rsg. Los conjuntos que definen este grafo son: $N(rsg) = \{n_1, n_2, n_3\}$, $P(rsg) = \{x\}$, $S(rsg) = \{nxt, prv\}$, $PL(rsg) = \{< x, n_1 >\}$ y $NL(rsg) = \{< n_1, nxt, n_2 >, < n_2, prv, n_1 >, < n_2, nxt, n_2 >, < n_2, prv, n_2 >, < n_2, nxt, n_3 >, < n_3, prv, n_2 >, \}$. Cuando en alguna figura aparezca la representanción de un RSG, los nodos se presentarán con rectángulos con sus esquinas redondeadas.

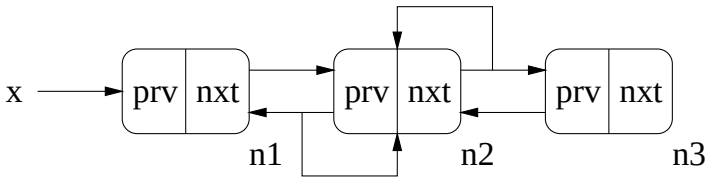


Figura 3.3: Ejemplo de grafo RSG.

Tenemos que decir que los conjuntos $P(m)$ y $P(rsg)$ al igual que $S(m)$ y $S(rsg)$ son los mismos en ambos dominios, el de las configuraciones de memoria y el de los RSGs, puesto que hacen referencia a variables puntero y selectores declarados en el código. Por este motivo nos referiremos a ellos únicamente como P y S respectivamente.

Para obtener el RSG que representa una configuración de memoria M dada, usamos una *función de abstracción* $F : M \rightarrow RSG$. Esta función es la encargada de mapear porciones de memoria en nodos, referencias entre pvars y porciones de memoria en referencias entre pvars y nodos, y por último referencias entre porciones de memoria en referencias entre los nodos que las representan. Es decir, la función F pasa del dominio de las configuraciones de memoria al dominio de los grafos. Esta función F en realidad está formada por tres funciones:

- $F_n : L \rightarrow N$ es la encargada de mapear las porciones de memoria en sus correspondientes nodos.
- $F_p : PS \rightarrow PL$ por su parte se encarga de transformar las referencias desde pvars a las porciones de memoria en referencias de pvars a los correspondientes nodos.
- $F_l : LS \rightarrow NL$ pasa los enlaces entre porciones de memoria a enlaces entre nodos.

Lema 3.2.1 Podemos ver que:

- $F_p(< pvar, l >) =< pvar, n >$ si $F_n(l) = n$
- $F_l(< l_1, sel, l_2 >) =< n_1, sel, n_2 >$ si $F_n(l_1) = n_1 \wedge F_n(l_2) = n_2$

Se puede deducir que trasladar referencias a porciones de memoria en referencias a nodos es trivial una vez que dichas porciones de memoria han sido mapeadas a nodos. Esto supone trasladar toda la complejidad de la función F a la función F_n que mapea las porciones de memoria en nodos. □

En la figura 3.4 mostramos de forma gráfica como actúan las funciones F_n, F_p y F_l para obtener el RSG de la figura 3.3 que representa a la configuración de memoria presentada en la figura 3.2.

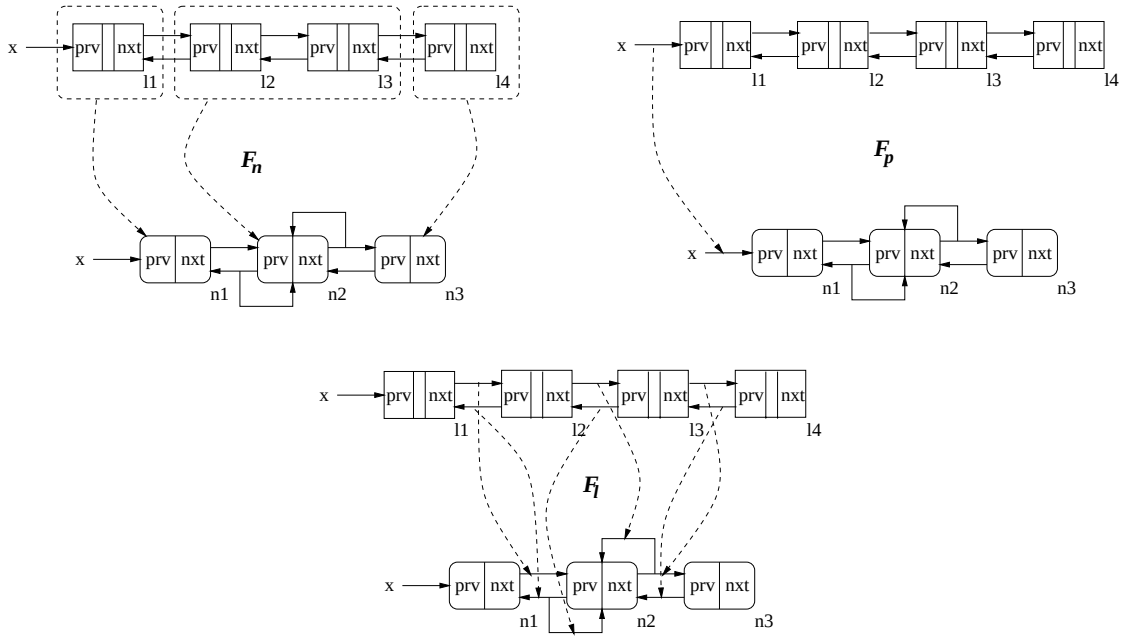


Figura 3.4: Generación de un grafo RSG a partir de una configuración de memoria por medio de la función F.

Nos vamos a centrar pues en definir la función F_n. Esta función extrae una serie de propiedades importantes de las porciones de memoria y, dependiendo de ellas, dichas porciones de memoria son mapeadas en nodos (de manera que dichos nodos van a representar a dichas porciones de memoria en los RSGs). Por lo tanto, si varias porciones de memoria comparten las mismas propiedades, entonces la función F_n las mapea todas en el mismo nodo del RSG. Es por este motivo que el número de nodos de un RSG, como ya hemos dicho, es finito, puesto que porciones con similares propiedades no son distinguibles por el método y serán representadas por el mismo nodo. Por tanto, de las propiedades va a depender la exactitud con la que un RSG va a representar una serie de configuraciones de memoria, puesto que son ellas las que determinan cuando distintas porciones de memoria son representadas por un mismo nodo.

Con todo lo dicho, podemos ver la gran dependencia que tiene la función F_n de las propiedades asociadas a las porciones de memoria. Por este motivo, la función F_n se definirá una vez se definan dichas propiedades. Las propiedades son las siguientes:

- **Type**: mantiene información sobre el *tipo* de estructura a la que pertenece la porción de memoria.
- **Structure**: hace referencia a porciones de memoria que pertenecen a una misma componente conexa de un tipo de estructura.
- **Información Shared**: informa sobre referencias múltiples sobre las porciones de memoria ya sea por un mismo selector o por varios.
- **Cycle links**: recoge información sobre pares de porciones de memoria que se referencian mutuamente por distintos selectores.
- **Simple paths**: contendrá caminos de acceso simple desde las pvars a las porciones de memoria.
- **Reference patterns**: mantendrá información sobre los patrones de referencias de las porciones de memoria (selectores por los que son apuntados y por los que apuntan dichas porciones de memoria).
- **Información Touch**: tiene en cuenta aspectos temporales, en el sentido de que mantiene información sobre pvars que han apuntado directamente a las porciones de memoria, alguna vez en la ejecución del código.

Antes de describir en profundidad cada una de estas propiedades, vamos a presentar una estructura ejemplo a la que haremos referencia en la descripción de las mismas.

3.2.1 Estructura ejemplo

Hemos creado un código C que genera, recorre y modifica la estructura de datos presentada en la figura 3.5. El código es presentado en el apéndice E.1.1. Como podemos observar se trata de una estructura de datos dinámica bastante compleja. Se trata de una lista doblemente enlazada en la cual cada elemento posee un puntero a un árbol binario. Además, las hojas de estos árboles a su vez poseen punteros a otras listas doblemente enlazadas. La principal entrada a la estructura de datos se realiza a través de la variable puntero S que apunta al primer elemento de la lista cabecera. Los elementos de esta lista poseen tres punteros: nxt apunta al siguiente elemento de la lista, prv al anterior y $tree$ a la raíz del árbol correspondiente a dicho elemento. Los elementos de dicho árbol poseen a su vez otros tres selectores: rgh a su hijo derecho, lft a su hijo izquierdo y $list$ a la lista doblemente enlazada si el elemento es una hoja del árbol. Los elementos de estas listas tan solo poseen dos selectores: nxt y prv.

Tenemos que decir que tal y como es construida la estructura, se cumplen las siguientes condiciones, que por otra parte son las características fundamentales de dicha estructura de datos:

- Los elementos de la lista cabecera siempre apuntan a árboles binarios distintos. Es decir, no existen dos elementos de la lista que apunten a la misma raíz de un árbol.
- Los distintos árboles binarios no comparten elementos entre sí.
- Las hojas de los árboles apuntan a distintas listas, por lo que dos hojas de dos árboles distintos o de un mismo árbol no puede apuntar nunca a la misma porción de memoria.

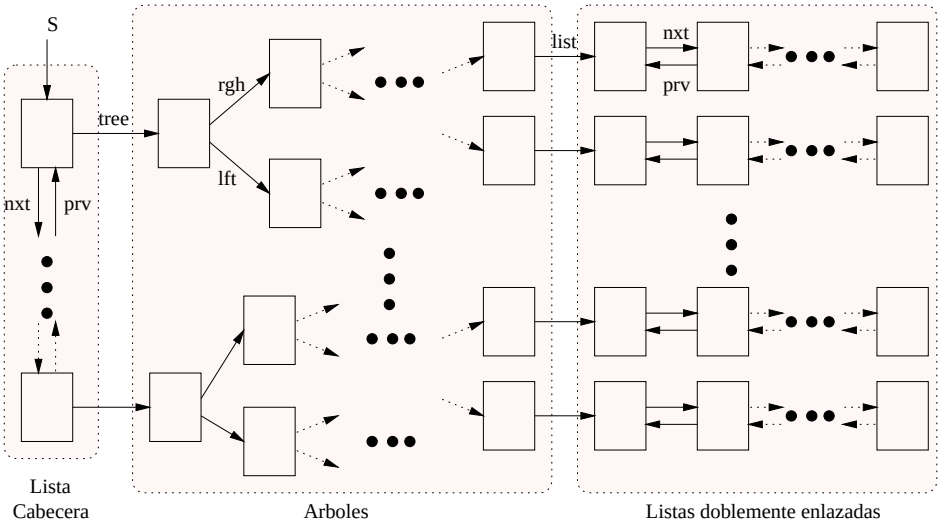


Figura 3.5: Ejemplo de estructura de datos dinámica compleja.

- Las listas doblemente enlazadas apuntadas desde las hojas de los árboles no comparten ningún elemento (son totalmente independientes).

Lo que pretendemos es que tras analizar el código que genera esta estructura de datos, los grafos devueltos para cada sentencia del código aproximen de una manera lo más fiel posible la estructura real. En concreto, para la última sentencia del código, en la que la estructura de datos apuntada por la variable S es la presentada en la figura 3.5, queremos obtener un RSRSG del cual podamos determinar las características descritas anteriormente para la estructura de datos.

En efecto, en la figura 3.6 presentamos una representación simplificada del RSRSG, obtenido con el compilador que proponemos, para la última sentencia del código que genera la estructura de datos de la figura 3.5. Decimos que el grafo de la figura 3.6 es una simplificación puesto que el número de nodos del RSGs es bastante elevado, por lo que se han juntado para su presentación en la figura 3.6 aquellos nodos cuya unión no desvirtúa la información proporcionada por el RSRSG. En el apéndice D se muestra el RSG perteneciente al RSRSG obtenido tras el análisis del código por nuestro compilador.

Como veremos a lo largo de este capítulo, del RSRSG obtenido y de las propiedades de los nodos podemos deducir las características fundamentales de la estructura apuntada desde la variable S. Por ahora sólo comentaremos que la lista cabecera está representada por los nodos n_1, n_2 y n_3, representando el primer elemento, los elementos centrales y el último elemento de la lista respectivamente. Los nodos n_4, n_5 y n_6 representan los elementos de los árboles binarios, donde n_4 representa las raíces de los mismos, n_5 los elementos centrales y n_6 las hojas. Por último los nodos n_7, n_8 y n_9 representan todos los elementos de las listas doblemente enlazadas apuntadas desde las hojas de los árboles.

De las aristas del grafo podemos ver los enlaces que existen en la estructura de datos. Así por ejemplo, podemos ver como las porciones de memoria representadas por n_2 apuntan por el selector nxt o a otra porción representada por n_2 ($<n_2, nxt, n_2>$) o al último elemento de la lista representado por n_3 ($<n_2, nxt, n_3>$). Además, como podemos ver, cada elemento de la lista cabecera tiene un enlace por el selector $tree$ con un nodo raíz de un árbol ($<n_1, tree, n_4>$, $<n_2, tree, n_4>$ y $<n_3, tree, n_4>$). Algo similar se puede deducir de las hojas

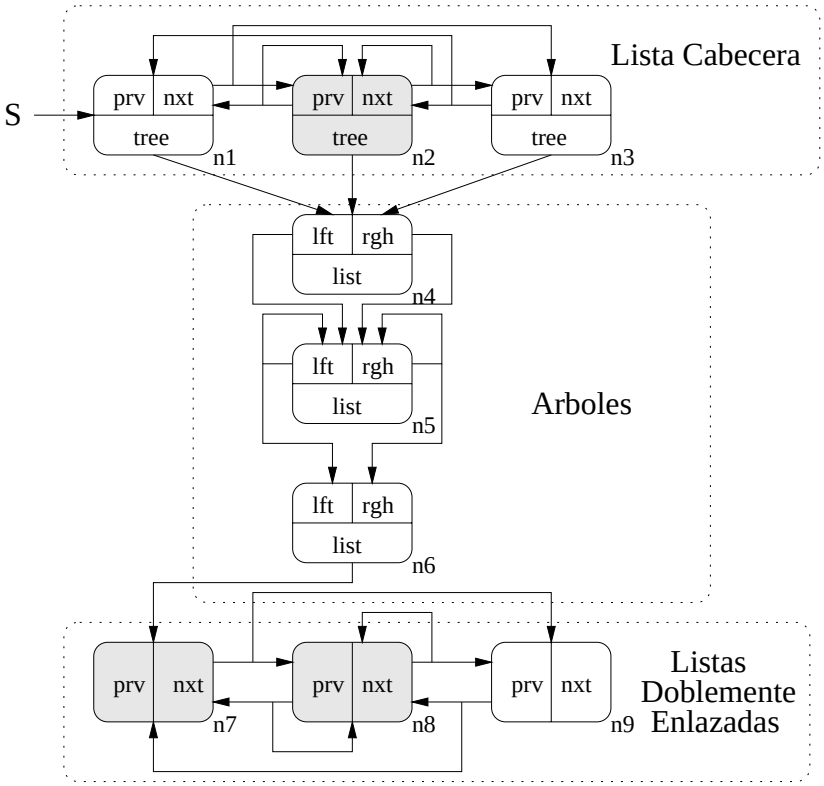


Figura 3.6: Representación simplificada del RSRSG obtenido para la estructura ejemplo.

de los árboles representadas por n_6, que referencian al inicio de alguna lista doblemente enlazada ($<n_6, list, n_7>$). Como se ve las conexiones en el grafo nos revelan los enlaces reales en la estructura de datos. En base a las diversas propiedades asociadas con cada nodo veremos como también se pueden deducir las características de independencia entre las distintas partes de la estructura (listas no comparten elementos, árboles tampoco comparten elementos entre sí, las listas son acíclicas si se recorren utilizando sólo un tipo de selector, el árbol es acíclico, etc.).

A continuación se presentan en profundidad cada una de las propiedades asociadas a los nodos.

3.2.2 Propiedad TYPE

Esta propiedad extrae información del texto del código, anotando para cada porción de memoria reservada, la definición de tipo de puntero utilizado para dicha reserva. La idea es que dos punteros de diferentes tipos no van a apuntar a la misma porción de memoria. Esto supone una restricción sobre el manejo de punteros que se hace en C. Nuestro método no permite el *type casting* por el cual se puede hacer que un puntero de cualquier tipo apunte a cualquier porción de memoria.

Esta restricción hace que una porción de memoria apuntada por un puntero de tipo *List* será considerada de tipo *List*. Por tanto, el método asigna a la propiedad *TYPE* de una porción de memoria l el tipo de la variable puntero usada cuando la porción de memoria es reservada.

De este modo, dos porciones de memoria distintas l_1 y l_2 podrán ser mapeadas al mismo nodo n si tienen el mismo valor para la propiedad $TYPE$. La función $TYPE(l)$ devuelve el valor del atributo $TYPE$ para la porción de memoria l.

Vamos a usar el mismo nombre para referirnos a la propiedad y a la función tanto para porciones de memoria como para nodos. De forma que $TYPE(n) = TYPE(l)$ si $F_n(l) = n$, es decir la propiedad $TYPE$ de un nodo tiene el mismo valor que toma para las porciones de memoria que representa dicho nodo, que tiene que ser igual en todas ya que si no, no serían mapeadas al mismo nodo por la función F_n.

Esta propiedad permite que en el grafo de la figura 3.6 los nodos que representan porciones de memoria pertenecientes a la lista cabecera (n_1, n_2 y n_3) no se mezclen con los que representan los elementos de los árboles (nodos n_4, n_5 y n_6) ni con los que representan las listas apuntadas desde las hojas de los árboles (n_7, n_8 y n_9). No se mezclan dichos nodos puesto que su propiedad $TYPE$ es distinta.

3.2.3 Propiedad STRUCTURE

Con esta propiedad se pretende mantener en nodos distintos porciones de memoria que aún siendo del mismo tipo (igual $TYPE$) pertenecen a componentes no conexas de la estructura de datos. Se asocia a cada porción de memoria la propiedad $STRUCTURE$ que tomará el mismo valor para todas las porciones de memoria pertenecientes a una misma componente conexa de una estructura.

En cuanto a la función F_n podemos decir que dos porciones de memoria podrán ser representadas por el mismo nodo si tienen el mismo valor en la propiedad $STRUCTURE$, lo que indicará que pertenecen a una misma estructura.

La condición para que dos porciones de memoria tengan el mismo valor para la propiedad $STRUCTURE$ es la siguiente:

$$\begin{gathered} STRUCTURE(l_a) = STRUCTURE(l_b) = val \text{ si } \exists l_n, l_1, ..., l_i, l'_1, ..., l'_i \in L \mid \\ (< l_n, sel_1, l_1 >, < l_1, sel_2, l_2 >, ... < l_i, sel_{i+1}, l_a > \in LS) \wedge \\ (< l_n, sel'_1, l'_1 >, < l'_1, sel'_2, l'_2 >, ... < l'_i, sel'_{i+1}, l_b > \in LS) \end{gathered}$$

es decir, tendrán el mismo valor en esta propiedad si existe alguna porción de memoria l_n desde la cual existe un camino hacia l_a y hacia l_b, donde l_n puede ser uno de los propios l_a o l_b. Se entiende por camino una secuencia de porciones de memoria y selectores de manera que existen enlaces entre ellas por medio de esos selectores.

De forma similar se define $STRUCTURE(n)$ en el dominio de los grafos:

$$\begin{gathered} STRUCTURE(n_a) = STRUCTURE(n_b) = val \text{ si } \forall l_a, F_n(l_a) = n_a, \forall l_b, F_n(l_b) = n_b, \\ \exists l_n, l_1, ..., l_i, l'_1, ..., l'_i \in L \mid \\ (< l_n, sel_1, l_1 >, < l_1, sel_2, l_2 >, ... < l_i, sel_{i+1}, l_a > \in LS) \wedge \\ (< l_n, sel'_1, l'_1 >, < l'_1, sel'_2, l'_2 >, ... < l'_i, sel'_{i+1}, l_b > \in LS) \end{gathered}$$

donde los dos nodos, n_a y n_b, pertenecen a la misma *estructura* si para todas las porciones de memoria que representan, existe alguna otra desde la cual hay siempre un camino hacia estas.

3.2.4 Información SHARED

Como ya vimos en los SSGs, la información *shared* es muy importante a la hora de determinar si se puede acceder a una misma porción de memoria desde diversos sitios, información por otro lado, determinante para decidir si es posible que haya dependencias entre distintos accesos a la estructura de datos.

La información *shared* indica si una porción de memoria puede ser referenciada más de una vez desde otras porciones. Para un nodo, la información *shared* indicará si alguna de las porciones de memoria que representa puede ser referenciada más de una vez desde otras. Este tipo de información servirá por ejemplo para determinar si determinados ciclos en un RSG representan o no verdaderos ciclos en las estructuras de datos que representa. Así, por ejemplo, un nodo de un RSG con un enlace hacia él mismo, nunca podrá representar un ciclo en la estructura que representa si dicho nodo no es *shared*.

Para mantener esta información, al igual que hacíamos en los SSGs, asociamos dos atributos a cada porción de memoria y nodo.

- $SHARED(l)$ con $l \in L$, es una función booleana que devuelve *true* si existen al menos otras dos porciones l_1 y l_2 que referencian a l por selectores distintos:

$$SHARED(l) = \begin{cases} 0 \text{ si } \nexists l_1, l_2 \in L \mid (< l_1, sel_1, l >, < l_2, sel_2, l > \in LS) \wedge \\ \qquad (sel_1 \neq sel_2) \\ 1 \text{ en caso contrario.} \end{cases}$$

  Por tanto si trasladamos esta función a los nodos de un RSG ($SHARED(n)$ con $n \in N(rsg)$) tenemos que:

$$SHARED(n) = \begin{cases} 0 \text{ si } \nexists l, l_1, l_2 \in L \mid (F_n(l) = n) \wedge \\ \qquad (< l_1, sel_1, l >, < l_2, sel_2, l > \in LS) \wedge (sel_1 \neq sel_2) \\ 1 \text{ en caso contrario (puede haber una porción de} \\ \qquad \text{memoria representada por } n \text{ que sea referenciada} \\ \qquad \text{por diferentes selectores)} \end{cases}$$

  Esta función puede ser usada para determinar si puede existir un ciclo en la estructura de datos representada o no. Así si $SHARED(n)$ es 0, se sabe que aunque se alcance el nodo n desde dos sitios distintos por selectores sel_1 y sel_2, realmente se están visitando dos porciones de memoria distintas representadas por el mismo nodo n.

- $SHSEL(l, sel)$ con $l \in L$ y $sel \in S$, es otra función booleana que indica si la porción de memoria l es referenciada más de una vez por el selector sel desde otras porciones de memoria:

$$SHSEL(l, sel) = \begin{cases} 0 \text{ si } \nexists l_1, l_2 \in L \mid (< l_1, sel, l >, < l_2, sel, l > \in LS) \wedge \\ \qquad (l_1 \neq l_2) \\ 1 \text{ en caso contrario.} \end{cases}$$

  Para el caso se los nodos, la función $SHSEL(n, sel)$ con $n \in N(rsg)$ y $sel \in S$, toma el valor *true* si si alguna de las porciones de memoria representadas por n puede ser referenciada más de una vez por el selector sel desde otras porciones de memoria:

$$SHSEL(n, sel) = \begin{cases} 0 \text{ si } \nexists l, l_1, l_2 \in L \mid (F_n(l) = n) \wedge \\ \qquad (< l_1, sel, l >, < l_2, sel, l > \in LS) \wedge (l_1 \neq l_2) \\ 1 \text{ en caso contrario (puede haber una porción de} \\ \qquad \text{memoria en } n \text{ referenciada por el selector } sel \\ \qquad \text{varias veces).} \end{cases}$$

Gracias a esta propiedad podemos deducir una serie de características muy importantes de la estructura presentada en la figura 3.5 a partir de grafo de la figura 3.6. Antes de presentar dichas características tenemos que proporcionar más información de la que aparece en el grafo de la figura 3.6. Hay que decir que los nodos sombreados poseen su atributo $SHARED(n)$ =*true*. En cuanto al atributo $SHSEL(n, sel)$ tenemos que decir que es *false* para todos los nodos y para todos los selectores. En base a esta información podemos deducir lo siguiente:

- El nodo n_2 es *shared* puesto que representa porciones de memoria que son referenciadas a la vez desde otras dos porciones (la siguiente y la anterior). Vemos como los nodos que representan al primer elemento (n_1) y al último (n_3) no lo son puesto que no son apuntadas más que desde una porción de memoria (la posterior y la anterior respectivamente).

- El nodo n_8 es *shared* por los mismos motivos expuestos para el nodo n_2. Sin embargo, el nodo n_7 que representa los primeros elementos de estas listas, en este caso si que es *shared* a diferencia de n_1. Esto es debido a que las porciones de memoria que representa también son referenciadas desde otras dos, la siguiente en la lista y una hoja de algún árbol.

- Todas las listas doblemente enlazadas (tanto la lista cabecera como las otras) son acíclicas si se recorren utilizando sólo un selector (nxt o prv). Esto se puede deducir puesto que los nodos que las representan tienen el valor *false* para el atributo *SHSEL* en todos sus selectores, lo que indica que ninguna porción de memoria representada por dichos nodos puede ser apuntada más de una vez por el mismo selector. Por tanto, a cualquiera de las porciones representadas en n_2 tan solo le llega un enlace por nxt, lo que impide la existencia de ciclos en la estructura si sólo se utiliza el selector nxt para recorrerla. Lo mismo ocurre para el selector prv.

- Los elementos de la lista cabecera apuntan a distintos árboles binarios, ya que *SHSEL* $(n_4, tree)$ = *false*. Esta información indica que las porciones de memoria representadas por n_4 no pueden ser referenciadas más de una vez por el selector $tree$, y por tanto una misma raíz de árbol no puede ser apuntada desde varios elementos de la lista.

- Cualquier recorrido por los selectores lft y rgh sobre los elementos de los árboles, nunca pasará dos veces por una misma porción de memoria. Esta información se deduce puesto que los nodos n_4, n_5 y n_6 que representan los elementos del árbol tienen $SHARED(n)$ = *false* y $SHSEL(n, sel)$ =*false* para cualquier selector.

- Dos hojas distintas de un mismo árbol o de distintos árboles no pueden apuntar a la misma lista doblemente enlazada, puesto que $SHSEL(n_7, list)$ =*false*. Esto implica que las porciones de memoria representadas por n_7 no pueden ser referenciadas más de una vez por el selector $list$ y por tanto dos hojas distintas del árbol forzosamente tienen que referenciar a dos porciones distintas representadas por n_7.

3.2.5 Propiedad CYCLELINKS

Esta propiedad introducida para los SSGs, es incorporada también para los RSGs, puesto que incrementa la exactitud con que los grafos representan las estructuras dinámicas de datos. Básicamente evita la aparición de enlaces que no existen en la estructura real, manteniendo información sobre ciclos simples dentro de dichas estructuras, como se vio en el capitulo anterior.

Podemos decir que la propiedad *cyclelinks* de un nodo, n, es un conjunto de pares de selectores $<sel_i, sel_j>$, de manera que si se siguen consecutivamente los selectores sel_i y sel_j desde el nodo n, dicho nodo es alcanzado de nuevo. Un ejemplo claro de *cyclelinks* serían los selectores *siguiente* y *anterior* en una lista doblemente enlazada. Así, los elementos centrales de dicha lista, poseerían dos *cyclelinks*, $<siguiente, anterior>$ y $<anterior, siguiente>$, ya que si desde ese elemento se avanza por los selectores *siguiente* y después por *anterior* o primero por *anterior* y después por *siguiente*, siempre se direcciona el elemento original.

Formalmente, para una porción de memoria $l \in L$ definimos:

$$CYCLELINKS(l) = \{< sel_i, sel_j > \mid sel_i, sel_j \in S \wedge (\exists l_i \in L, \\ < l, sel_i, l_i > \in LS \wedge < l_i, sel_j, l > \in LS)\}$$

Para un nodo, los *cyclelinks* deben representar aquellos ciclos simples que se cumplen en todas las porciones de memoria representadas por el nodo. De esta forma, para el nodo $n \in N(rsg)$ tenemos que:

$$CYCLELINKS(n) = \{< sel_i, sel_j > \mid sel_i, sel_j \in S \ \wedge \ (\forall l_i, \ F_n(l_i) = n, \\ \text{si } \exists l_j \in L, < l_i, sel_i, l_j > \in LS \text{ entonces } \exists < l_j, sel_j, l_i > \in LS)\}$$

en definitiva vemos que un *cyclelink* $<sel_i, sel_j>$ puede pertenecer al conjunto de *cyclelinks* de un nodo n si todas las porciones l_i representadas por el nodo, que tienen un enlace por el selector sel_i a otra porción l_j, son referenciadas nuevamente desde la porción l_j por el selector sel_j.

Como se verá más adelante, esta propiedad no es tenida en cuenta a la hora de decidir cuando dos porciones de memoria o dos nodos pueden ser sumarizados. Sólo nos proporciona información interesante sobre la estructura interna de las porciones representadas por un nodo, posteriormente utilizada a la hora de materializar un nodo nuevo a partir de otro, evitando la aparición de enlaces que realmente no existen. Esto implica que cuando dos nodos sean sumarizados, el conjunto de *cyclelinks* resultantes del nuevo nodo se construirá con los *cyclelinks* compatibles de ambos nodos.

3.2.6 Propiedad SIMPLE PATHS

Esta nueva propiedad tiene en cuenta la "cercanía" de las porciones de memoria a las variables puntero a la hora de representar dichas porciones de memoria en un mismo nodo. La idea en la que nos hemos basado a la hora de crear esta propiedad es la siguiente: los puntos de entrada a las estructuras dinámicas de datos usadas en un código son las variables puntero utilizadas en el mismo. Es decir, para hacer cualquier acceso a dichas estructuras forzosamente hay que utilizar las variables puntero (pvars) y por tanto las porciones de memoria a las que se accede primero son las directamente apuntadas por dichas pvars. Con esta propiedad lo que pretendemos es mantener las "cercanías" de las porciones de memoria apuntadas por las

distintas pvars de la manera más exacta posible, ya que es ahí donde se producen los accesos a las estructuras (y por tanto las modificaciones sobre las mismas).

Básicamente lo que pretendemos es evitar que se mezclen porciones y nodos cercanos a los distintos puntos de acceso de la estructura, preservando por separado sus propiedades.

Los *simple paths* de una porción de memoria l están formados por parejas del tipo $<pv, sel>$ o $<pv, \emptyset>$, donde pv es una pvar y sel un selector. Dichos *simple paths* son los puntos de acceso a la porción l desde pvars más "cercanos". Con cercanos nos referimos a que el camino desde la pvar a la porción l sea mínimo en cuanto al número de porciones intermedias. Si $<pv, \emptyset>$ pertenece a los *simple path* de una porción es porque dicha porción es apuntada directamente por la variable puntero pv. Si por el contrario aparece $<pv, sel>$ con $sel \neq \emptyset$, es porque la entrada más cercana a dicha porción desde variables puntero es por la variable pv siguiendo el selector sel.

Por ejemplo, en la figura 3.7 podemos ver una lista con cinco elementos enlazados. El conjunto de *simple paths* de la porción l_1 sería $\{< x, \emptyset >\}$ puesto que la variable x apunta directamente a dicha porción. Lo mismo sucede para la porción l_3 y la pvar y, por tanto su conjunto sería $\{< y, \emptyset >\}$. Para la porción l_2 el acceso más cercano es por la pvar x y el selector nxt por lo que su *simple paths* sería $\{< x, nxt >\}$. Por último las porciones de memoria l_4 y l_5 comparten el mismo conjunto puesto que la entrada más cercana para ambas es por la pvar y y el selector nxt ($\{< y, nxt >\}$).

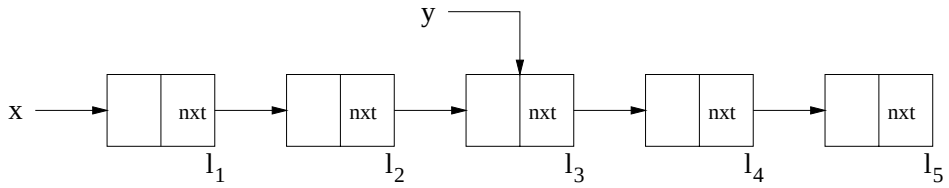


Figura 3.7: Lista simple de cinco elementos.

Formalmente, para una porción de memoria $l \in L(m)$ definimos:

$$SPATH(l) = \{sp_1, ..., sp_n\} \text{ donde } sp_i =< pv, sel^* > \text{ con } pv \in P,$$

$$sel^* = \begin{cases} \emptyset \text{ si } < pv, l > \in PS(m) \\ sel \in S \text{ si } \begin{cases} \cdot\ \exists l_1, ..., l_i \in L(m), < pv, l_1 > \in PS(m) \wedge (< l_1, sel, l_2 >, \\ \quad < l_2, sel_2, l_3 >, ..., < l_i, sel_i, l >) \in LS(m) \\ \cdot\ \nexists l'_1, ..., l'_j \in L(m)\ \nexists pv2 \in P, < pv_2, l'_1 > \in PS(m) \wedge \\ \quad (< l'_1, sel'_1, l'_2 >, < l'_2, sel'_2, l'_3 >, ..., < l'_j, sel'_j, l >) \\ \quad \in LS(m) \\ \cdot\ j < i \\ \cdot\ \nexists pv_2 \in P, < pv_2, l > \in PS(m) \end{cases} \end{cases}$$

Como se ve, si la porción de memoria es directamente referenciada por una pvar pv, en sus *simple paths* tan solo aparecen pares del tipo $<pv_i, \emptyset>$ donde las distintas $pv_i \in P$ son las variables puntero que referencian directamente la porción l. Esto es así ya que si pv apunta a l, el camino más corto a l no utiliza ningún selector, por lo tanto ninguno de los *simple paths* de l puede contener un selector distinto de $\emptyset$. Si por el contrario l no es apuntada directamente por una pvar, entonces formarán parte de los *simple paths* aquellas pvar y selectores a partir

de los cuales se puede acceder a l pasando por el mínimo número de porciones de memoria intermedias.

A la hora de definir los *simple paths* para los nodos de un RSGs hemos hecho una simplificación puesto que los *simple paths* descritos para las porciones de memoria dependen de la longitud del camino, y cuando estamos en el dominio de los RSGs la longitud de los caminos no se puede determinar. No se puede determinar puesto que, como hemos comentado anteriormente, al representar una configuración de memoria de tamaño indeterminado por medio de un grafo de tamaño finito, tenemos que representar varias porciones de memoria en un mismo nodo, perdiendo las dimensiones reales de los caminos. Así un nodo con un selector sel sobre el mismo puede representar dos o más porciones de memoria enlazadas por sel, no sabiendo el número exacto de las mismas que representa.

La simplificación consiste en insertar en los *simple paths* de un nodo n, $<pv, \emptyset>$ si el nodo es directamente apuntado por la variable pv, y $<pv, sel>$ si el nodo n no es directamente apuntado por ninguna variable puntero, pero si es referenciado por el selector sel desde un nodo directamente apuntado por pv.

De este modo, para $n \in N(rsg)$ definimos:

$$SPATH(n) = \{sp_1, ..., sp_n\} \text{ donde } sp_i =< pv, sel^* > \text{ con } pv \in P,$$
$$sel^* = \begin{cases} \emptyset \text{ si } < pv, n > \in PL(rsg) \\ sel \in S \text{ si } \exists n_i \in N(rsg), < pv, n_i > \in PL(rsg) \wedge \\ \quad < n_i, sel, n > \in NL(rsg) \ \wedge \ (\nexists pv_i \in P, < pv_i, n > \in PL(rsg)) \end{cases}$$

Con esta restricción en realidad en los *simple paths* de un nodo aparecerán entradas por las que se llega a las porciones representadas por el nodo pero que no tiene por que ser por un camino de longitud mínima desde una variable puntero. En el dominio de los grafos (RSGs), un camino desde una variable pv aparecerá en los *simple paths* de un nodo, si en dicho camino como máximo aparece un enlace por un selector. Es decir, el nodo o es apuntado directamente por pv o es apuntado directamente por un nodo que es a su vez apuntado por pv.

En la figura 3.8 vemos dos RSGs que representan una lista apuntada por dos variables x e y. La variable x apunta al principio de la lista, y la variable y a algún elemento intermedio de la misma. El RSG de la figura 3.8 (a) se ha obtenido utilizando la propiedad *SPATH* en cada nodo, mientras que el RSG de la figura 3.8 (b) se ha obtenido sin la utilización de la propiedad *simple paths*.

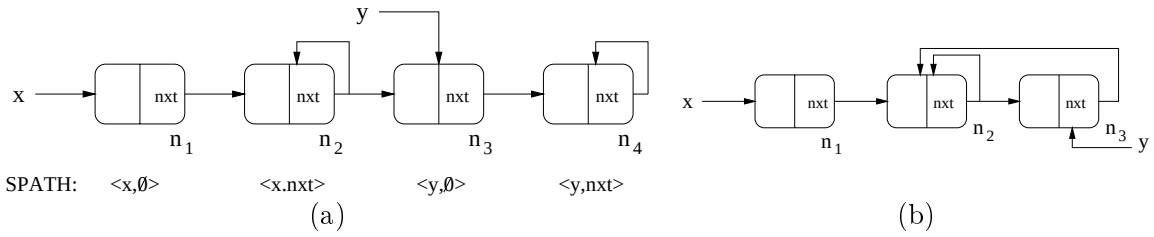


Figura 3.8: Lista doblemente enlazada: (a) RSG con SPATH. (b) RSG sin SPATH.

Como podemos observar, el uso de la propiedad *SPATH* en el RSG (a) hace que las porciones de memoria de la lista que hay entre la apuntada por x y la apuntada por y no se mezclen con las porciones de la lista que van después de la porción apuntada por y. Esto

es así porque los *simple paths* de los nodos n_2 y n_4 son distintos ($\{< x, nxt >\}$ para n_2 y $\{< y, nxt >\}$ para n_4). Sin embargo en el RSG (b), al no utilizar la propiedad *simple paths* el nodo n_2 representa tanto a las porciones que hay entre las apuntadas por x e y, como a las que hay después de la apuntada por y. Esta "sumarización" de nodos hace que se mezclen las propiedades de esos conjuntos de porciones de memoria, haciendo que se pueda perder información sobre alguno de los dos grupos de porciones lo que provocará que la representación se aleje de la estructura real. Veremos un ejemplo de este problema cuando presentemos los resultados obtenidos tras analizar varios códigos con nuestro método.

Antes de continuar con las siguiente propiedades, vamos a presentar una serie de conceptos y funciones relacionados con los *simple paths* necesarios cuando definamos la *semántica abstracta* de las sentencias.

En primer lugar vamos a definir la longitud de un *simple path*, $sp_i =< pv, sel_i >$, como:

$$LEN(sp_i) = \begin{cases} 0 \text{ si } sel_i = \emptyset \\ 1 \text{ si } sel_i = sel \in S \end{cases}$$

Basándonos en la definición de la longitud de un *simple path*, podemos dividir el conjunto de *simple paths* de un nodo $n \in N(rsg)$, basándonos en la longitud de los mismos, de la siguiente manera:

$$SPATH(n) = SPATH0(n) \cup SPATH1(n)$$

donde:

- $SPATH0(n) = \{sp_i \mid sp_i \in SPATH(n) \ \wedge \ LEN(sp_i) = 0\}$
- $SPATH1(n) = \{sp_i \mid sp_i \in SPATH(n) \ \wedge \ LEN(sp_i) = 1\}$

Esta división de los *simple paths* de un nodo será utilizada cuando se definan los criterios de compatibilidad entre nodos, que permitirán o no que dos nodos sean sumarizados.

3.2.7 Propiedad REFERENCE PATTERNS

Esta nueva propiedad es una de las más importantes ya que permite representar en distintos nodos porciones de memoria con patrones de referenciado diferentes. Entendemos por patrones de referenciado los tipos de selectores de los enlaces que apuntan a una porción de memoria y por los que dicha porción de memoria apunta a otras.

Cuando tratamos con estructuras de datos dinámicas, vemos que lo realmente importante para determinar la forma de dichas estructuras son los enlaces que poseen las porciones de memoria. Con la inclusión de esta propiedad pretendemos representar en nodos distintos aquella porciones de memoria singulares. Entendemos por porción de memoria singular, aquella que posee una combinación de enlaces apuntándole o por los que ella apunta, diferente de las demás. Por ejemplo, podemos considerar como porciones de memoria singulares la cabeza y la cola de una lista puesto que la primera no es apuntada por ninguna otra porción de la lista, y la última no apunta a ninguna otra. Con esta propiedad obtendríamos un nodo distinto para representar la cabeza de la lista, otro para los elementos centrales y otro para la cola de la lista. En cuanto la estructura de datos se va complicando, el número de porciones de memoria singulares va aumentando, y el uso de la propiedad *reference patterns* hace que la representación de la estructura por parte del RSG sea más exacta.

Gracias a esta propiedad, en el grafo de la figura 3.6 existe un nodo que representa el primer elemento de la lista cabecera (n_1) tan solo apuntado por selector prv, otro para representar los elementos centrales (n_2) apuntados por selectores nxt y prv y otro para el último elemento (n_3) sólo apuntado por selector nxt. Lo mismo ocurre con las raíces de los árboles representadas por n_4, los elementos centrales por n_5 y las hojas por n_6. Los nodos n_7, n_8 y n_9 representan los correspondientes primeros elementos, elementos centrales y últimos elementos respectivamente de las listas apuntadas desde los árboles.

Para obtener este comportamiento en el que las porciones de memoria son agrupadas por afinidad en la manera en que son referenciada y en la que referencian, hemos definido dos conjuntos de selectores para cada una. Los conjuntos son: $SELINset$ que contendrá los selectores por los que una porción de memoria es referenciada y $SELOUTset$ contendrá los selectores por los que esta porción de memoria referencia a otras. Así por ejemplo, para un elemento central en una lista doblemente enlazada, sus conjuntos serían: $SELINset(l) = \{nxt, prv\}$ y $SELOUTset(l) = \{nxt, prv\}$, puesto que es referenciado desde otras porciones por los selectores nxt y prv y a su vez referencia a otras por estos dos mismos selectores.

Formalmente definimos dichos conjuntos para una porción $l \in L(m)$ como:

$$\cdot\ SELINset(l) = \{sel_i \in S \mid \exists l_i \in L(m), < l_i, sel_i, l > \in LS(m)\}$$
$$\cdot\ SELOUTset(l) = \{sel_i \in S \mid \exists l_i \in L(m), < l, sel_i, l_i > \in LS(m)\}$$

como podemos ver un selector sel_i formará parte del conjunto $SELINset(l)$ si existe otra porción de memoria l_i que referencia a l por el selector sel_i. Formará parte de $SELOUTset(l)$ si l apunta a alguna otra porción de memoria por dicho selector.

Sobre la base de estos conjuntos definimos dos funciones booleanas $SELIN$ y $SELOUT$ que posteriormente utilizaremos para definir la compatibilidad entre los patrones de referenciado de dos porciones de memoria.

$$SELIN(l, sel) = \begin{cases} 1 \text{ si } sel \in SELINset(l) \\ 0 \text{ en caso contrario.} \end{cases}$$

$$SELOUT(l, sel) = \begin{cases} 1 \text{ si } sel \in SELOUTset(l) \\ 0 \text{ en caso contrario.} \end{cases}$$

La función $SELIN(l, sel)$ será *true* sólo si la porción l es referenciada por el selector sel, mientras que $SELOUT(l, sel)$ lo será cuando l referencie a otra porción por el selector sel.

Una vez presentada la propiedad *reference pattern* en el dominio de la memoria, pasamos al dominio de los grafos. En principio hay que pasar los conjuntos de selectores de "entrada" y de "salida" a los nodos, de manera que representen tipos de enlaces de entrada y salida a las porciones de memoria que representa cada nodo. Pero esto no basta, ya que en el dominio de los RSGs, hay veces en las que no es posible determinar si un determinado selector sel apunta o no a todos las porciones de memoria representadas por un determinado nodo. Cuando presentemos la semántica abstracta de las sentencias veremos que hay veces que al insertar o borrar enlaces en un nodo y debido a las propiedades de dicho nodo (en concreto a la información *shared*), no es posible determinar con exactitud si el selector sel aparece como selector de entrada/salida de todas las porciones representadas por el nodo. Por este motivo, en el dominio de los RSGs tenemos que introducir "información posible", en el sentido de que un selector sel pueda ser selector de entrada a determinadas porciones representadas por el

nodo y pueda no serlo para otras determinadas porciones representadas por el mismo nodo.

Por este motivo, en cada nodo n, además de los conjuntos $SELINset(n)$ y $SELOUTset(n)$ que contiene los selectores de entrada y salida de todas las porciones representadas por n, añadimos los conjuntos $PosSELINset(n)$ y $PosSELOUTset(n)$ que recogen la "información posible". Para un nodo $n \in N(rsg)$ los cuatro conjuntos quedan definidos de la siguiente manera:

$$\begin{aligned}
&\cdot\ SELINset(n) = \{sel_i \in S \mid \forall l, F_n(l) = n,\ \exists l_i \in L(m), \\
&\qquad < l_i, sel_i, l > \in LS(m)\} \\
&\cdot\ SELOUTset(n) = \{sel_i \in S \mid \forall l, F_n(l) = n,\ \exists l_i \in L(m), \\
&\qquad < l, sel_i, l_i > \in LS(m)\} \\
&\cdot\ PosSELINset(n) = \{sel_i \in S \mid \exists l, F_n(l) = n,\ \exists l_i \in L(m), \\
&\qquad < l_i, sel_i, l > \in LS(m)\} \\
&\cdot\ PosSELOUTset(l) = \{sel_i \in S \mid \exists l, F_n(l) = n,\ \exists l_i \in L(m), \\
&\qquad < l, sel_i, l_i > \in LS(m)\}
\end{aligned}$$

Las funciones booleanas definidas para las porciones de memoria, hay que volverlas a definir para los nodos. Por lo tanto $SELIN$ y $SELOUT$ para un nodo $n \in N(rsg)$ y un selector $sel \in S$ quedan definidas de la siguiente manera:

$$SELIN(n, sel) = \begin{cases} 2 \text{ si } sel \in PosSELINset(n) \\ 1 \text{ si } sel \in SELINset(n) \wedge sel \notin PosSELINset(n) \\ 0 \text{ en otro caso.} \end{cases}$$

$$SELOUT(n, sel) = \begin{cases} 2 \text{ si } sel \in PosSELOUTset(n) \\ 1 \text{ si } sel \in SELOUTset(n) \wedge sel \notin PosSELOUTset(n) \\ 0 \text{ en otro caso.} \end{cases}$$

Vemos como ahora estas funciones no son booleanas puesto que además de los valores *true* (1) y *false* (0), pueden devolver un valor de incertidumbre en la información (2). Estas funciones serán utilizadas para definir la compatibilidad entre la información proporcionada por la propiedad *reference patterns* de dos nodos y así poder o no ser sumarizados en uno solo.

3.2.8 Información TOUCH

Hasta el momento, todas las propiedades asociadas tanto a las porciones de memoria como a los nodos, hacen referencia a aspectos "espaciales" de las configuraciones de memoria. Con aspectos "espaciales" nos referimos a que las propiedades tiene en cuenta aspectos relacionados con los enlaces que poseen las porciones de memoria de una determinada configuración en un momento dado de la ejecución del programa. Para una configuración de memoria correspondiente a una sentencia del programa en un determinado momento en la ejecución del mismo las propiedades *structure, simple paths, shared, reference patterns y cyclelinks* dependen de la conectividad que existe entre las porciones de memoria en ese instante. La propiedad *type* tan solo informa del tipo de dato al que pertenece cada porción en dicho instante.

Sin embargo vamos a ver que a la hora del análisis es interesante, algunas veces, mantener para cada porción de memoria, algo de información referente a la "historia" de dicha porción

a lo largo de la ejecución de un trozo de código. Para ver la utilidad de mantener información "temporal" en las porciones de memoria vamos a presentar un ejemplo simple.

El código de la figura 3.9 recorre los elementos de una matriz M almacenada como una lista cabecera donde cada elemento contiene un puntero *col* a las columnas de la matriz representadas a su vez por otras listas enlazadas por el selector *nxt* (es decir, nos referimos a una estructura estructura similar a la *LLCS* de la figura 2.9 presentada en el capitulo 2).

```
1:    x = M;
2:    while (condicion1)
        {
3:        z = x→col;
4:        while (condicion2)
            {
              /* procesar elemento apuntado por z */
5:            k = z→nxt;
6:            z = k;
7:            k = NULL;
            }
8:        y = x→nxt;
9:        x = y;
10:       y = NULL;
        }
```

Figura 3.9: Código ejemplo.

Como se puede observar hay dos bucles anidados. El primero (línea 2) recorre la lista cabecera con la variable x y el segundo (línea 4) recorre la columna correspondiente a la posición donde apunta x por medio de variable z. En la figura 3.10 podemos observar una representación compacta del RSG que representa la matriz en la sentencia 1.

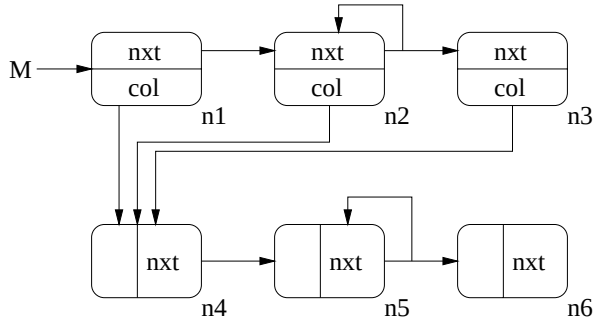


Figura 3.10: Grafo que representa una matriz con almacenamiento LLCS.

En el grafo de la figura los nodos n_1, n_2 y n_3 representan el primer elemento, los elementos centrales y el último elemento respectivamente, de la lista cabecera, mientras que los nodos n_4, n_5 y n_6 son los correspondientes de las listas que representan a las columnas. Como podemos ver de cada elemento de la lista cabecera puede haber un enlace a una de estas últimas listas por el selector *col*. Tenemos que decir que la información *SHARED* y *SHSEL* es falsa para todos los nodos y todos los selectores. Por tanto, todas las listas son acíclicas y además distintos elementos de la lista cabecera apuntan a distintas listas por el selector *col*, como ya vimos al presentar la propiedad *shared*. Eso quiere decir que la variable z nunca va ha visitar dos veces el mismo elemento en ninguno de los recorridos que realiza el bucle de la sentencia 4.

En la figura 3.11 podemos ver un RSG obtenido para la sentencia 3 del código. Vemos como la variable x apunta a algún elemento central de la lista cabecera, y la variable z a su vez, al primer elemento de la lista apuntada desde la porción referenciada por x.

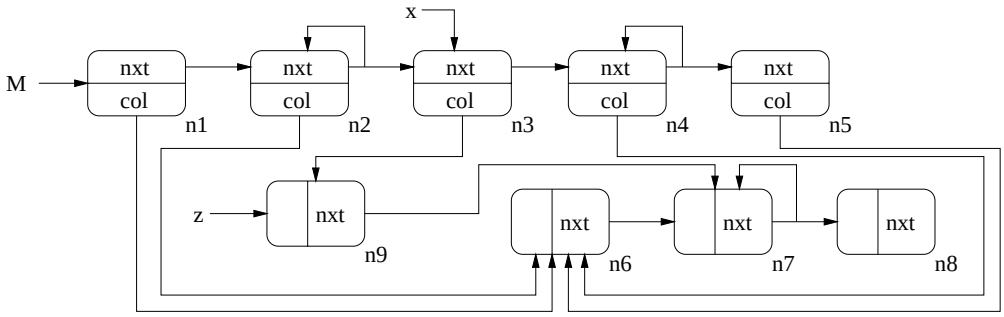


Figura 3.11: RSG en la sentencia 3.

Como se puede observar en el grafo, vemos que tanto las listas ya visitadas por la variable z como las que aún no lo han sido, están todas representadas por los mismos tres nodos (n_6, n_7 y n_8), puesto que sus propiedades son indistinguibles. Esto en principio no supone ninguna restricción, puesto que aunque estén representadas por los mismos nodos, sus propiedades nos están diciendo que no hay dos elementos de la lista cabecera que apunten a la misma columna. El problema puede surgir de la no diferenciación de los nodos que representan porciones de memoria ya visitadas de las que aún no lo han sido. Así, conforme el análisis progrese, habrá nodos que representen porciones de memoria ya visitadas por z que volverán a ser visitadas. Si por ejemplo se insertan selectores sobre dichas porciones de memoria, el análisis erróneamente insertará varios selectores sobre una misma porción, cuando esto en realidad no sucede.

Vemos que el problema radica en que, aunque se puede saber que las listas apuntadas por los elementos de la lista cabecera son independientes, esta información no puede ser utilizada puesto que el análisis no es capaz de distinguir las porciones visitadas por z de las que aún no lo han sido.

La idea por tanto es intentar evitar representar porciones de memoria visitadas por una determinada variable en un recorrido, en el mismo nodo que porciones que no lo han sido todavía. Si esto se consigue, se evitarán el tipo de problemas descritos anteriormente (aplicar varias veces la misma modificación a una porción de memoria). Como se verá al presentar los resultados experimentales del método, no siempre es posible conseguir esto.

Siguiendo con el ejemplo, en la figura 3.12 podemos ver el RSG que se obtendría para la misma sentencia 3, pero marcando cada nodo con la variable puntero que lo ha "visitado".

Como podemos observar ahora, las porciones de memoria de las columnas ya visitadas por z están representadas por los nodos n_6, n_7 y n_8 (marcadas con **z**), y las no visitadas aún, por los nodos n_{10}, n_{11} y n_{12}. En este caso no se mezclan los nodos pues hay una nueva propiedad que los diferencia: *haber sido visitado por z*. Ahora, conforme avance el análisis, z no volverá a apuntar a un nodo que represente porciones que ya ha visitado antes, por lo que no aplicará varias veces la misma modificación a una de ellas.

Para poder dotar al análisis de este comportamiento, hemos introducido en cada porción de memoria y en los nodos de los RSGs, una nueva propiedad que hemos denominado *touch*. Esta propiedad va a mantener para cada porción de memoria el conjunto de variables puntero que han visitado dicha porción de memoria. Por "visitado" entendemos variables puntero que

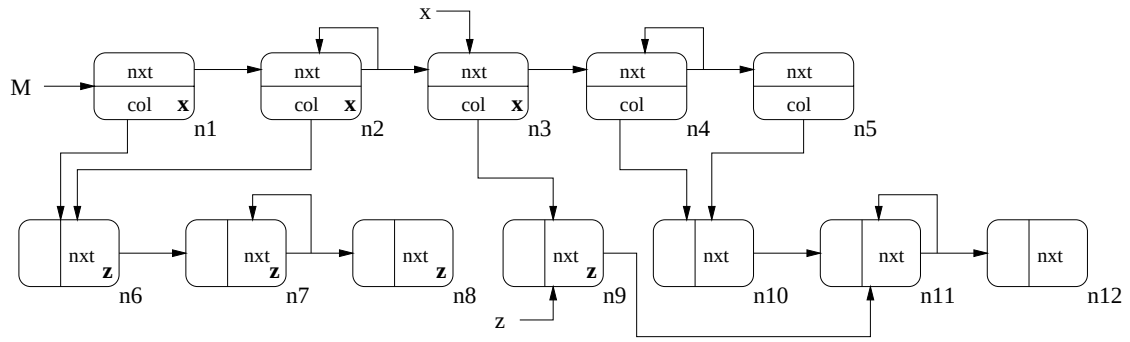


Figura 3.12: RSG en la sentencia 3 con marcas de "visitas".

en algún momento han apuntado a dicha porción de memoria. Hemos impuesto algunas restricciones sobre la información mantenida por *touch* de manera que sea lo más útil posible pero minimizando el coste de implementación. A continuación presentamos algunas matizaciones sobre la información que va a mantener *touch*:

- **Duración de la información TOUCH**. Como hemos visto esta información tan solo es de utilidad dentro de los cuerpos de los bucles que recorren estructuras dinámicas, por lo que este "recordatorio" de que pvars han visitado una porción no se va ha mantener para toda la ejecución del código, sino sólo dentro del cuerpo de dichos bucles.

- **Variables puntero en TOUCH**. Las variables puntero que van a poder aparecer en la propiedad *touch* de una porción de memoria o nodo, no serán todas las que han apuntado a dicha porción en algún momento. En realidad las únicas que nos interesan son aquellas variables puntero que son utilizadas por los bucles para recorrer estructuras dinámicas enlazadas. Este tipo de variables puntero son denominadas *punteros de inducción* por Yuan-Shin Hwang [49] o *navegadores* por Rakesh Ghiya [31]. Nosotros hemos utilizado el método de Hwang [49] para detectar que variables puntero recorren estructuras enlazadas en cada bucle, además de obtener una expresión, denominada *path expression*, que relaciona cualquier otra variable puntero con los *punteros de inducción*.

  Un *path expression* es una tupla asociada con cada variable puntero del código de la forma:

  $$< p, e, f, r >$$

  donde p es la variable puntero en cuestión, e es el punto de entrada a la estructura dinámica referenciada por p, f es el selector o selectores por los que avanza p si es un *puntero de inducción* y r es el camino relativo desde un *puntero de inducción* hasta el elemento que referencia p.

  Como podemos observar, obteniendo el *path expression* de cada variable puntero, podemos determinar si es un *puntero de inducción* de algún bucle (si $f \neq \emptyset$) y si no lo es, se sabe el camino relativo desde un *puntero de inducción* a p (indicado por r).

Para el código de la figura 3.9 se obtendrían los siguientes *path expressions*:

1. $< x, M, nxt, \emptyset >$
2. $< z, x, nxt, col >$
3. $< k, z, \emptyset, nxt >$
4. $< y, x, \emptyset, nxt >$

De la expresión (1) podemos deducir que la variable x es un *puntero de inducción* que recorre la estructura apuntada por M por el selector nxt. En (2) vemos que z es otro *puntero de inducción* pero que recorre la estructura apuntada por $x \rightarrow col$ mediante el selector nxt. La expresión (3) nos informa que k visita cada vez el elemento apuntado por $z \rightarrow nxt$ y (4) que y hace lo propio pero con $x \rightarrow nxt$.

A partir de ahora nos referiremos al *path expression* de una variable $pv \in P$ como $PE(pv) =< p, e, f, r >$. Para referirnos a cada uno de los cuatro elementos de la tupla lo haremos de la siguiente manera: $PE(pv).p$, $PE(pv).e$, $PE(pv).f$ y $PE(pv).r$ respectivamente.

Una vez que hemos determinado que las únicas variables puntero, pv, que pueden aparecer en *touch* son los *punteros de inducción* ($PE(pv).f \neq \emptyset$), tenemos que ver que ocurre cuando una porción de memoria no es visitada directamente por un *puntero de inducción*, sino por otra variable puntero pv_2. En este caso, anotaremos en *touch* de dicha porción de memoria, el *puntero de inducción* que es el punto de entrada a la estructura que apunta pv_2 y que se obtiene de los *path expressions*. Con esto lo que hacemos es anotar de la misma manera todos aquellos accesos a porciones de memoria que dependan de un mismo recorrido (*puntero de inducción*).

Una vez vistas las peculiaridades de la información proporcionada por *touch* vamos a describir formalmente dicha propiedad para las porciones de memoria $l \in L(m)$.

Antes de definirla, necesitamos presentar una serie de conceptos relacionados con la ejecución de un programa. En primer lugar, recordamos, que el *grafo de flujo de control* de un programa $G = (V, A)$, está formado por un conjunto de vértices V que representan las sentencias del programa y un conjunto de arcos A representando las transferencias de control entre sentencias. Definimos un *camino de ejecución* (CE) en el grafo G, entre dos vértices $v_a, v_b \in V(G)$ como:

$$CE(v_a, v_b, G) = (v_a, v_1, v_2, ..., v_{i-1}, v_i, v_b) \mid v_1, ..., v_i \in V(G) \wedge$$
$$\exists < v_a, v_1 >, < v_1, v_2 >, ... < v_{i-1}, v_i >, < v_i, v_b > \in A(G)$$

Un *camino de ejecución* entre dos vértices de G, v_a, v_b (sentencias), consiste en cualquier sucesión de vértices de G empezando en v_a y terminando en v_b, de manera que entre dos vértices consecutivos hay un arco dirigido en G que los une.

Si notamos como $m(v_i)$ la configuración de memoria del vértice v_i perteneciente a un *camino de ejecución*, y $PE(pv, v_i)$ el *path expression* obtenido para la variable puntero pv en el vértice v_i de un *camino de ejecución* dado, podemos definir la propiedad *touch* de una porción de memoria $l \in L(m(v_b))$ con respecto a un vértice en una ejecución, v_a como:

$$TOUCH(l, v_a, v_b) = \{pv \mid pv \in P, \exists v_i \in CE(v_a, v_b, G) \wedge$$
$$((PE(pv, v_b).f \neq \emptyset \wedge < pv, l > \in PS(m(v_i))) \vee$$
$$(\exists pv_1, .., pv_n, PE(pv_1, v_b).e = pv_2 \wedge PE(pv_2, v_b).e = pv_3 \wedge ... \wedge$$
$$PE(pv_n, v_b).e = pv \wedge PE(pv, v_b).f \neq \emptyset \wedge < pv_1, l > \in PS(m(v_i))))\}$$

El significado de este atributo es el siguiente: el contenido de la propiedad *touch* de una porción de memoria l correspondiente a una configuración de memoria m en un momento de la ejecución (vértice del grafo de flujo de control v_b), está relacionado con una ejecución concreta de una serie de sentencias comprendidas entre los vértices v_a y v_b (como hemos visto, *touch* tiene sentido dentro de los bucles que recorren estructuras dinámicas enlazadas. El vértice v_a representaría el punto de entrada y salida del bucle, es decir su cabecera.). Como se puede ver, $TOUCH$ está formado por un conjunto de variables puntero pv. Pero para que una pvar, pv, pueda pertenecer al $TOUCH$ de una porción de memoria para el vértice v_b tiene que pertenecer a uno de estos grupos:

- pv ha apuntado directamente a la porción l en algún momento entre la ejecución de v_a y la de v_b y además pv debe ser un puntero de inducción del bucle actual ($PE(pv, v_b).f \neq \emptyset$).

- deben existir una serie de variables puntero de manera que cada una de ellas es el punto de entrada de la anterior en su *path expression* ($PE(pv_i, v_b).e = pv_{i+1}$) y la primera pv_1 ha apuntado directamente a l en algún punto intermedio de la ejecución entre v_a y v_b. Además la última de todas ellas pv, es decir el primer punto de entrada, es un *puntero de inducción* para el bucle actual ($PE(pv, v_b).f \neq \emptyset$), y es ella la que aparece en $TOUCH$.

Como se puede apreciar, la propiedad $TOUCH$ no especifica más que un conjunto de variables puntero, pero la potencia de la misma radica en el tipo de variables puntero que pueden formar parte de este conjunto y de su dependencia de la "historia" de la porción de memoria correspondiente.

Si pasamos al dominio de los grafos, definimos de forma similar $TOUCH$ para un nodo $n \in N(rsg)$ para un determinado punto de la ejecución del programa v_b con respecto a otro punto v_a como:

$$TOUCH(n, v_a, v_b) = \{pv \mid pv \in P, \forall l_i, F_n(l_i) = n, \exists v_i \in CE(v_a, v_b, G) \wedge$$
$$((PE(pv, v_b).f \neq \emptyset \wedge < pv, l_i > \in PS(m(v_i))) \vee$$
$$(\exists pv_1, .., pv_n, PE(pv_1, v_b).e = pv_2 \wedge PE(pv_2, v_b).e = pv_3 \wedge ... \wedge$$
$$PE(pv_n, v_b).e = pv \wedge PE(pv, v_b).f \neq \emptyset \wedge < pv_1, l_i > \in PS(m(v_i))))\}$$

Podemos observar que la definición de *touch* para un nodo es igual a la de las porciones de memoria, excepto que ahora las condiciones expuestas anteriormente se deben cumplir para todas las porciones de memoria l_i representadas por el nodo n.

Cuando presentemos la semántica abstracta de las sentencias en el apéndice B, prestaremos especial atención a los aspectos relacionados con el manejo del atributo $TOUCH$ de los nodos, puesto que como hemos visto debe estar estrechamente relacionado con los bucles que recorren las estructuras dinámicas. Allí veremos como es generada la información y como es eliminada cuando ya no tiene sentido mantenerla (final de los bucles).

Con la presentación de esta última propiedad hemos terminado la descripción de la información asocia a cada porción de memoria y a cada nodo, y estamos en condiciones de describir la función F_n.

3.2.9 Función de abstracción

Como se presentó al comienzo de esta sección, para pasar del dominio de la memoria al dominio de los grafos, es decir para obtener el grafo RSG que representa una configuración concreta de memoria, hacemos uso de una función de abstracción $F : M \rightarrow RSG$, que dividíamos en tres funciones encargadas de mapear porciones de memoria en nodos (F_n), referencias de variables puntero a porciones de memoria a referencias de variables puntero a nodos (F_p) y por último referencias entre porciones de memoria en referencias entre nodos (F_l).

Ya vimos que F_p y F_l se construyen a partir de F_n, es decir una vez que las porciones de memoria han sido mapeadas en los nodos del RSG. Como hemos dicho, para mapear las porciones de memoria de una configuración de memoria de tamaño desconocido en los nodos de un grafo RSG de tamaño finito, era necesario que los nodos del grafo pudieran representar a la vez a varias porciones de memoria. Para que esta unión no implicara la pérdida de información relevante sobre la estructura, hemos definido las propiedades descritas anteriormente, que van a permitir la unión en un mismo nodo de sólo aquellas porciones similares.

Teniendo esto en cuenta, definimos la función $F_n : L \rightarrow N$ de la siguiente manera:

$$F_n(l) = n \text{ si } \forall l_i, F_n(l_i) = n \begin{cases} TYPE(l) = TYPE(l_i) \wedge \\ STRUCTURE(l) = STRUCTURE(l_i) \wedge \\ SHARED(l) = SHARED(l_i) \wedge \\ SHSEL(l, sel) = SHSEL(l_i, sel) \forall sel \in S \wedge \\ C\_SPATH(l, l_i, k) = 1 \wedge \\ SELINset(l) = SELINset(l_i) \wedge \\ SELOUTset(l) = SELOUTset(l_i) \wedge \\ TOUCH(l, v_a, v_b) = TOUCH(l_i, v_a, v_b) \end{cases}$$

Como podemos apreciar, varias porciones de memoria son mapeadas en el mismo nodo, si la información proporcionada por sus propiedades es la misma. Por tanto dos porciones de memoria con los mismos valores en sus propiedades son "indistinguibles" para el método y serán representadas por el mismo nodo. Esto lleva consigo una consecuencia inmediata: el número de nodos máximo que pueden aparecer en un RSG está limitado por el número distinto de combinaciones de valores para las propiedades que puedan existir. Esto es así puesto que si dos porciones de memoria distintas tienen el mismo valor para todas sus propiedades serán representadas por el mismo nodo. Cuantos más valores puedan tomar, más combinaciones distintas de propiedades tendremos y mayor será el número máximo de nodos que se pueden obtener para un grafo.

Hay una excepción a la hora de comparar las propiedades de las porciones de memoria. En concreto, la propiedad *SPATH* no se ha comparado de la misma forma que las demás. Para su comparación se ha utilizado una función booleana *C_ SPATH* que devuelve 1 si los *simple paths* de ambas porciones de memoria son "compatibles". Lo que pretendemos con esto, es reducir el número posible de nodos distintos del grafo.

Con la función *C_ SPATH* hemos reducido el número distinto de combinaciones de *simple paths* que dan lugar a nodos distintos, permitiendo que dos porciones se puedan representar en

un mismo nodo sin tener que compartir exactamente todos sus *simple paths*, como presentamos a continuación.

$$C\_SPATH(l_1, l_2, k) = 1 \text{ con } \begin{cases} \cdot\ SPATH0(l_1) = SPATH0(l_2) \text{ si} \\ \quad (SPATH0(l_1) \neq \emptyset\ \wedge\ SPATH0(l_2) \neq \emptyset)\ \vee \\ \cdot\ |SPATH(l_1) \cap SPATH(l_2)| \geq k \text{ con} \\ \quad (SPATH0(l_1) = \emptyset\ \wedge\ SPATH0(l_2) = \emptyset) \end{cases}$$

lo que implica que dos porciones de memoria pueden ser representadas por el mismo nodo si se da una de las siguientes condiciones. La primera condición se aplica si al menos una de las porciones de memoria es directamente apuntada por una variable puntero. En este caso, únicamente se tiene en cuenta que los conjuntos $SPATH0$ de ambas porciones sean iguales, es decir, si una porción es directamente apuntada por una variable puntero sólo se podrá "sumarizar" con porciones que sean apuntadas por exactamente las mismas variables puntero. Por otro lado, la otra condición sólo se aplica cuando las porciones de memoria no sean apuntadas directamente por variables puntero, en cuyo caso, los conjuntos $SPATH1$ deben tener en común al menos k elementos. Es decir, en principio no se impone que coincidan todos los $SPAHT1$, sino al menos k. Con este parámetro podemos afinar la representación con respecto a los *simple paths*. Con $k = 0$ se despreciarían todos los *simple paths* de longitud uno, mientras que para un k igual al número máximo de combinaciones de variables puntero con selectores tendríamos la condición de que los *simple paths* de ambas porciones deberían coincidir en su totalidad.

Una vez visto como se pasa del dominio de las configuraciones de memoria al de los grafos RSGs, a partir de ahora nos moveremos en dicho dominio, viendo como dichos grafos son modificados conforme las sentencias son analizadas para ir reflejando las nuevas configuraciones de memoria que se pueden obtener para cada sentencia del código.

Antes de pasar a la descripción de los RSRSG asociados a cada sentencia vamos a presentar una operación fundamental sobre los RSGs: *la compresión de un RSG*.

3.2.10 Compresión de un RSG

Como hemos indicado, los RSGs representan varias de las configuraciones de memoria que se pueden dar en una sentencia. Dichos grafos son modificados, según indica la semántica abstracta de la sentencia, generando un nuevo RSG donde se reflejan las modificaciones impuestas por la sentencia sobre las configuraciones de memoria. Dichas modificaciones pueden traer consigo la generación o eliminación de enlaces en el grafo, tanto entre nodos como entre variables y nodos, también pueden aparecer o desaparecer nodos, etc. Esto hace que se modifiquen las *propiedades* asociadas con ciertos nodos, y nodos que antes de la modificación tenían valores distintos en sus propiedades y por tanto no podían ser sumarizados, ahora puede que compartan las mismas y si puedan ser representados por un único nodo.

Por tanto, después de aplicar la semántica abstracta de la sentencia sobre los RSGs, estos deben ser "comprimidos" de manera que los nodos que comparten las mismas propiedades sean representados por un único nodo (puesto que representan porciones de memoria indistinguibles). Para llevar a cabo esta operación hemos creado la función $COMPRESS\_RSG$.

Antes de describir dicha función, tenemos que ver cuando dos nodos representan porciones de memoria indistinguibles, teniendo en cuenta sus propiedades. La función de comparación

$C\_NODES\_RSG(n_i, n_j)$, es una función booleana que devuelve *true* si los nodos n_i y n_j representan porciones de memoria similares y por tanto pueden ser sumarizados. Para determinar esa "compatibilidad" entre nodos, la función booleana se comporta de forma similar a F_n, en el sentido de que compara las propiedades de los nodos. Existen una serie de diferencias en la comparación de algunas propiedades, como veremos a continuación, debido a la incertidumbre de algunas propiedades asociadas a los nodos. Por tanto definimos $C\_NODES\_RSG(n_i, n_j)$, con $n_i, n_j \in N(rsg)$ como:

$$
\begin{array}{l}
C\_NODES\_RSG(n_i, n_j) = \\
\left\{ \begin{array}{ll}
1 \text{ si } & \left\{ \begin{array}{l}
TYPE(n_i) = TYPE(n_j) \wedge \\
STRUCTURE(n_i) = STRUCTURE(n_j) \wedge \\
SHARED(n_i) = SHARED(n_j) \wedge \\
SHSEL(n_i, sel) = SHSEL(n_j, sel) \forall sel \in S \wedge \\
C\_SPATH(n_i, n_j, k) = 1 \wedge \\
C\_REFPAT(n_i, n_j) = 1 \wedge \\
TOUCH(n_i, v_a, v_b) = TOUCH(n_j, v_a, v_b)
\end{array} \right. \\
0 \text{ en otro caso.} &
\end{array} \right.
\end{array}
$$

Como podemos ver, la compatibilidad entre dos nodos consiste en comparar la igualdad de sus propiedades, a excepción de sus *simple paths* y de sus *reference patterns*. Para comparar estas dos propiedades, no exigimos la igualdad, puesto que daría lugar a la aparición de muchos más nodos en el grafo sin aumento significativo de la exactitud en la representación. Para llevar esto a cabo se han definido dos funciones booleanas que comparan ambas propiedades.

Para comparar los *simple paths*, se utiliza la función $C\_SPATH(n_i, n_j, k)$ similar a la presentada para la comparación de esta misma propiedad sobre las porciones de memoria. Para comparar los nodos de un RSG queda como sigue:

$$
C\_SPATH(n_i, n_j, k) = 1 \text{ con } \left\{ \begin{array}{l}
\cdot\ SPATH0(n_i) = SPATH0(n_j) \text{ si} \\
\quad (SPATH0(n_i) \neq \emptyset \ \wedge \ SPATH0(n_j) \neq \emptyset) \vee \\
\cdot\ |SPATH(n_i) \cap SPATH(n_j)| \geq k \text{ con} \\
\quad (SPATH0(n_i) = \emptyset \ \wedge \ SPATH0(n_j) = \emptyset)
\end{array} \right.
$$

Como se puede observar es exactamente igual a la definida para las porciones de memoria, cambiándolas ahora por nodos. Como en aquel caso, serán "compatibles" los *simple paths* de dos nodos, si son apuntados por el mismo conjunto de variables puntero ($SPATH0$) o, si no son apuntados por ninguna variable puntero, entonces deben tener en común al menos k *simple paths* de longitud 1 ($SPATH1$). Como se verá en los resultados experimentales, vamos a tener varios niveles de exactitud en el análisis dependiendo de la complejidad del código analizado, de manera que la exactitud del análisis vaya creciendo (más espacio y memoria) conforme se necesite. tan solo se van a utilizar dos valores para el parámetro k: 0 y 1. Con el valor 0 tan solo se tienen en cuenta el conjunto $SPATH0$ y con 1, basta con que coincidan los $SPAHT1$ en un *simple path*. Valores por encima de 1 no producen mejora en la representación pero si un aumento en el coste del análisis.

En cuanto a la comparación de los *reference patterns*, hemos definido la función booleana $C\_REFPAT(n_i, n_j)$ de la siguiente manera:

$$C\_REFPAT(n_i, n_j) =$$
$$\begin{cases} 1 \text{ si } \forall sel_i \in S, (SELIN(n_i, sel_i) = SELIN(n_j, sel_i) \ \vee \ SELIN(n_i, sel_i) = 2 \\ \quad \vee \ SELIN(n_j, sel_i) = 2) \ \wedge \ (SELOUT(n_i, sel_i) = SELOUT(n_j, sel_i) \\ \quad \vee \ SELOUT(n_i, sel_i) = 2 \ \vee \ SELOUT(n_j, sel_i) = 2) \\ 0 \text{ en otro caso.} \end{cases}$$

En este caso, la restricción impuesta sobre los *reference patterns* para la compatibilidad de dos nodos, es que para todos los selectores, la información de si es o no selector de "entrada" o "salida" coincida (valores 0 o 1 en *SELIN* y *SELOUT*), o bien no se sabe si el selector es o no un selector de "entrada" o "salida" (valor 2 en *SELIN* y *SELOUT*). Como podemos observar, hemos suavizado la restricción de "compatibilidad" entre dos nodos con respecto a sus *reference patterns*, a que sólo coincida la información sin "incertidumbre". Con esto reducimos el número de combinaciones distintas de *reference patterns* que pueden aparecer en los nodos de un RSG, y por tanto se reduce el número máximo de nodos permitidos en los RSGs.

Una vez definida la función booleana $C\_NODES\_RSG(n_i, n_j)$ que indica si los nodos n_i y n_j, pertenecientes a un grafo rsg, son compatibles y pueden ser sumarizados en un único nodo, vamos a presentar la función de *compresión de grafos*, $COMPRESS\_RSG(rsg)$ que genera un nuevo grafo a partir de rsg uniendo aquellos nodos "compatibles" del mismo, operación que mantiene finito el tamaño de los RSGs.

La función $COMPRESS\_RSG(rsg) = rsg_c$, genera un nuevo grafo rsg_c a partir del grafo original rsg de la siguiente manera:

- El conjunto de variables puntero P y el de selectores S no varían para el nuevo grafo rsg_c.
- El conjunto de nodos del grafo comprimido, $N(rsg_c)$, contendrá los nodos que no pueden ser sumarizados con otros (no hay ningún otro nodo en rsg compatible con ellos), más los nodos resultantes de la sumarización de nodos compatibles. Usamos la función $MERGE\_COMP\_NODES$ para generar un nodo sumario como la unión de una serie de nodos compatibles, como veremos más tarde. Formalmente:

$$N(rsg_c) = \{n \mid (n \in N(rsg) \ \wedge \ \nexists n_i \in N(rsg), C\_NODES\_RSG(n, n_i) = 1) \ \vee \ (n = MERGE\_COMP\_NODES(n_1, ..., n_k), n_1, ..., n_k \in N(rsg) \wedge \forall i = 1..k-1, C\_NODES\_RSG(n_i, n_{i+1}) = 1)\}$$

- El nuevo conjunto de referencias desde variables puntero, $PL(rsg_c)$, es básicamente el mismo que el del grafo original, $PL(rsg)$, sólo que mapeando todos lo nodos sobre los nuevos. Esto es llevado a cabo por la función $MAP\_RSG(n) = n_c$ que mapea el nodo $n \in N(rsg)$ en el nuevo nodo $n_c \in N(rsg_c)$:

$$PL(rsg_c) = \{< pv, MAP\_RSG(n) > \ | \ < pvar, n > \in PL(rsg)\}$$

- La misma idea se aplica al conjunto de referencias entre nodos del nuevo grafo, $NL(rsg_c)$:

$$NL(rsg_c) = \{< MAP\_RSG(n_i), sel, MAP\_RSG(n_j) > \ | \ < n_i, sel, n_j > \in NL(rsg)\}$$

Definimos la función $MERGE\_COMP\_NODES$, usada para la sumarización de los nodos compatibles de un grafo, como:

$$MERGE\_COMP\_NODES(n_1, ..., n_k) = MERGE\_NODES(n_1, \\ MERGE\_NODES(n_2, ..., MERGE\_NODES(n_{k-1}, n_k)...))$$

la cual traslada la sumarización de varios nodos en la sumarización recursiva de parejas de nodos usando la función $MERGE\_NODES(n_1, n_2)$, que es la encargada de generar un nuevo nodo n que represente las porciones de memoria anteriormente representadas por los nodos n_1 y n_2. Las propiedades del nuevo nodo n, se establecen de manera que preserven la descripción de las porciones de memoria representadas por los nodos originales.

De esta forma, $MERGE\_NODES(n_1, n_2) = n$, donde:

- $TYPE(n) = TYPE(n_1) = TYPE(n_2)$
- $STRUCTURE(n) = STRUCTURE(n_1) = STRUCTURE(n_2)$
- $SHARED(n) = SHARED(n_1) = SHARED(n_2)$
- $SHSEL(n, sel_i) = SHSEL(n_1, sel_i) = SHSEL(n_2, sel_i)\ \forall sel_i \in S$
- $TOUCH(n) = TOUCH(n_1) = TOUCH(n_2)$
- $SPATH(n) = SPATH(n_1) \cup SPATH(n_2)$
- $SELINset(n) = SELINset(n_1) \cap SELINset(n_2)$
- $SELOUTset(n) = SELOUTset(n_1) \cap SELOUTset(n_2)$
- $PosSELINset(n) = PosSELINset(n_1) \cup PosSELINset(n_2) \cup \\ (SELINset(n_1) \cup SELINset(n_2)) \setminus SELINset(n)$
- $PosSELOUTset(n) = PosSELOUTset(n_1) \cup PosSELOUTset(n_2) \cup \\ (SELOUTset(n_1) \cup SELOUTset(n_2)) \setminus SELOUTset(n)$
- $CYCLELINKS(n) = \{< sel_i, sel_j > \ | \\ < sel_i, sel_j > \in (CYCLELINKS(n_1) \cap CYCLELINKS(n_2)) \vee \\ < sel_i, sel_j > \in CYCLELINKS(n_1) \ \wedge \ \nexists n_k \in N(rsg), \\ \quad < n_2, sel_i, n_k > \in NL(rsg) \vee \\ < sel_i, sel_j > \in CYCLELINKS(n_2) \ \wedge \ \nexists n_k \in N(rsg), \\ \quad < n_1, sel_i, n_k > \in NL(rsg)\}$

Como podemos observar, el nuevo nodo tendrá exactamente los mismos valores para las propiedades *TYPE*, *STRUCTURE*, *SHARED*, *SHSEL*, y *TOUCH* que tenían los nodos originales. Esto es fácil de comprender puesto que para que dos nodos de un grafo puedan ser sumarizados (*C_NODES_RSG*) estas propiedades en ambos nodos deben coincidir. Por tanto el nuevo nodo que va a representar a las porciones de memoria de ambos, debe tener el mismo valor que los nodos originales en dichas propiedades.

En cuanto al conjunto de *SPATH* del nuevo nodo, puesto que se permite la unión de nodos cuyos conjuntos *SPATH* no sean los mismos, debe ser la unión de los conjuntos *SPATH* de ambos nodos. De esta manera los *simple paths* de las porciones de memoria representadas en ambos nodos, siguen apareciendo en el nuevo nodo, que es el que ahora las representa.

La propiedad *reference patterns* se comporta conservativamente, puesto que vamos a permitir la sumarización de dos nodos que no tengan exactamente los mismos *reference patterns* (C_NODES_RSG). Para cada selector, sel_i, si aparecía en ambos como un selector de entrada ($SELINset$) o de salida ($SELOUTset$), aparecerá de igual forma en el nuevo nodo. Si por el contrario sel_i no aparece ni en los selectores de entrada ni en los de salida de ambos nodos, tampoco lo hará para el nuevo nodo. Pero, ¿que pasa si sel_i en uno de ellos no se sabe con certeza si pertenece o no a los selectores de entrada o salida? En ese caso se va a mantener la incertidumbre en el nodo resultante, independientemente de lo que suceda con sel_i en el otro nodo. Así, por ejemplo, si sel_i está en $PosSELINset(n_1)$ indica que no se sabe con certeza si las porciones de memoria representadas por n_1 son referenciadas desde otras por sel_i. Por otro lado, si sel_i está en $SELINset(n_2)$ indicaría que todas las porciones de memoria representadas por n_2 son referenciadas desde otras por el selector sel_i. Si ambos nodos (n_1 y n_2) son unidos, el nuevo nodo n va a representar a todas las porciones de memoria antes representadas por n_1 y n_2, y por tanto habrá porciones de memoria en n que sean apuntadas por sel_i (las de n_2 y posiblemente algunas de n_1) y otras que no lo sean (algunas otras de n_1). Por lo tanto, sel_i debe pasar a $PosSELINset(n)$ del nuevo nodo, manteniendo la incertidumbre.

Por último tenemos la propiedad *cyclelinks*. Como ya comentamos cuando presentamos las propiedades, esta propiedad no va a determinar cuando dos nodos pueden o no ser sumarizados, de ahí que no aparezca en la función C_NODES_RSG. La información mantenida por dicha propiedad, es usada a la hora de transformar los RSGs por medio de la semántica abstracta, para mantener lo más exacto posible el conjunto de enlaces de cada nodo (cuando se presente la semántica abstracta se verá como la propiedad $CYCLELINKS$ permitirá la eliminación de ciertos enlaces que no existen en la estructura representada). Por tanto, cuando sumarizamos dos nodos, n_1 y n_2 en uno nuevo n, los *cyclelinks* del nuevo nodo deben ser los que existieran a la vez en los nodos originales, que eran los que poseían las porciones de memoria representadas por ambos y que ahora lo son por n. Es claro entonces que el conjunto de $CYCLELINKS$ del nuevo nodo debería ser la intersección de los conjuntos de $CYCLELINKS$ de los nodos originales. Sin embargo, se pueden añadir más *cycle links* de uno de los nodos que no aparezca en el otro. Por ejemplo, supongamos que el *cycle link* $<sel_i, sel_j>$ pertenece a $CYCLELINKS(n_1)$, lo que implica que las porciones de memoria representadas por n_1 que referencian a otras por el selector sel_i, son a su vez referenciadas por estas otras mediante el selector sel_j. Si dicho *cycle link* no pertenece a $CYCLELINKS(n_2)$, la propiedad descrita anteriormente, no se da para las porciones de memoria representadas por n_2. Pero, ¿que pasa si ninguna porción de memoria de n_2 apunta a alguna otra por sel_i? Si esto ocurre, al unir las porciones de ambos nodos en el nuevo nodo n, se sigue cumpliendo que las porciones de memoria representadas por n que apuntan a alguna otra por el selector sel_i (sólo las que pertenecían a n_1), son de nuevo referenciadas desde estas otras por el selector sel_j, y por tanto $<sel_i, sel_j>$ debe pertenecer a los $CYCLELINKS(n)$.

Con todo lo descrito, ya hemos visto como un RSG representa una configuración de memoria cualquiera de una manera finita y preservando las características fundamentales de dicha configuración de memoria. A continuación vamos a ver como representamos todas las posibles configuraciones de memoria que pueden aparecer tras la ejecución de una sentencia de un código.

3.3 Conjunto Reducido de RSGs: RSRSG

Como ya comentamos al principio de este capítulo, el método va a tratar de aproximar las posibles configuraciones de memoria que se pueden dar tras la ejecución de cada una de las sentencias del programa. Ya hemos visto como un RSG puede describir una configuración de memoria de una forma finita. Debido a los distintos caminos del flujo de control que pasan por una misma sentencia, dicha sentencia debe modificar las distintas configuraciones de memoria que alcanzan dicha sentencia siguiendo los distintos caminos del flujo. Es por tanto que puede haber más de un RSG asociado con la sentencia, representando las distintas configuraciones de memoria.

Mantener un grafo por cada configuración de memoria posible, implicaría la existencia de un número demasiado elevado de grafos por sentencia para códigos un poco complejos (con bastantes estructuras de control condicionales), lo que haría el método inviable en la práctica. Es por este motivo que nuestro método no mantiene por separado todos los grafos que representan configuraciones de memoria individuales, sino que va a mantener un *Conjunto Reducido de RSGs (RSRSG)* para cada sentencia. Cada RSRSG describirá de una forma aproximada todas las posibles configuraciones de memoria que pueden aparecer tras la ejecución de una sentencia. Estos RSRSGs, como su propio nombre indica, estarán formados por un conjunto de RSGs distintos, donde cada uno va a representar un subconjunto del total de configuraciones de memoria asociadas a la sentencia. Por tanto cada RSG va a representar más de una configuración de memoria. La pregunta ahora es la siguiente: de todas las configuraciones de memoria, ¿cuales de ellas pueden ser representadas por el mismo RSG? Para ello vamos a definir la "compatibilidad" entre grafos, de manera que dos grafos pertenecientes a un mismo RSRSG que sean "compatibles" serán sumarizados en uno único que represente las configuraciones de memoria de ambos. A continuación presentamos como hemos resuelto el problema de la "compatibilidad" entre grafos, para después presentar como los RSRSG son mantenidos de un tamaño finito por medio de la unión de grafos "compatibles".

3.3.1 Compatibilidad entre distintos RSGs

Determinar la compatibilidad o similitud entre distintos grafos es un problema similar al que se tenía con los nodos y las porciones de memoria. Los nodos de los RSGs representan varias porciones de memoria distintas a la vez, ¿como se sabe que porciones pueden ser representadas por el mismo nodo? La respuesta la dimos, asociando una serie de propiedades a cada porción, y en función de los valores de dichas propiedades, las porciones de memoria eran mapeadas en nodos, donde las porciones con propiedades muy similares son representadas todas por un mismo nodo. Con esto podíamos representar una configuración de memoria de tamaño desconocido con un grafo de tamaño finito. También hemos visto que dos nodos se pueden sumarizar en uno sólo si representan porciones de memoria "similares", comparando sus propiedades (*C_NODES_RSG*).

Algo similar se va ha hacer con los RSGs, vamos a imponer una serie de condiciones sobre los mismos que determinarán si dos RSGs pueden sumarizarse en uno que represente todas las configuraciones de memoria que anteriormente representaban los grafos originales. Esta unión de grafos "compatibles" es la que limitará el número distinto de RSGs que pueden aparecer para cada sentencia, por lo que para cada una se va a mantener un *conjunto reducido de RSGs*.

La idea principal para determinar si dos RSGs son similares y pueden ser sumarizados, es

representar en un mismo RSG, aquellos grafos que representan configuraciones de memoria "accesibles" de forma similar. De nuevo, la idea es que las estructuras de datos sólo pueden ser manejadas por medio de las variables puntero del código, por lo que vamos a permitir la unión de grafos donde las variables puntero apunten a "porciones" de memoria similares.

Hemos definido una función booleana $COMPATIBLE(rsg_1, rsg_2)$ que determina si los grafos rsg_1 y rsg_2 son lo suficientemente similares como para poder ser sumarizados en un único grafo. Esta función es la encargada de ver si las propiedades relacionadas con los accesos a las estructuras que representan son similares en ambos casos, de forma similar a lo que hacía la función *C_NODES_RSG* comparando las propiedades de dos nodos para determinar si podían o no ser sumarizados. En este caso definimos $COMPATIBLE$ como:

$$COMPATIBLE(rsg_1, rsg_2) = \begin{cases} 1 \text{ si } (ALIAS(rsg_1) = ALIAS(rsg_2)) \wedge \\ \qquad (COMP\_NODES(rsg_1, rsg_2) = 1) \\ 0 \text{ en otro caso.} \end{cases}$$

Podemos apreciar que para que dos grafos rsg_1 y rsg_2 sean compatibles, es decir, puedan ser sumarizados en un único grafo, se deben cumplir dos condiciones: i) las relaciones de alias entre las variables puntero en ambos grafos deben ser idénticas; y ii) ciertos nodos en ambos grafos deben ser "compatibles". Antes de ver como comprueba estas dos condiciones la función $COMPATIBLE$, vamos a arrojar algo más de luz sobre ambas condiciones.

Como hemos dicho anteriormente, pretendemos que grafos que representan configuraciones de memoria con accesos similares desde las variables puntero, se puedan representar en un único grafo. La primera condición que hemos impuesto es que las relaciones de alias entre las variable puntero en ambos grafos sean las mismas. Las relaciones de alias nos informan que variables puntero apuntan a una misma porción de memoria. Así, si por ejemplo en un grafo la variable pv_1 es alias con pv_2 (ambas apuntan a la misma porción de memoria) y en el otro grafo, pv_1 no es alias con ninguna otra variable, está claro que los puntos de acceso a las estructuras representadas en ambos grafos son distintos, por lo que no vamos a permitir su unión. Sólo se permitirá si, como hemos dicho, las relaciones de alias entre las variables en ambos grafos son exactamente iguales.

En cuanto a la segunda condición, trata de ir un paso más allá, y permitir sólo la unión de grafos con relaciones de alias idénticas (condición primera), pero que además, los nodos a los que apuntan las variables puntero son similares en ambos grafos. Es decir, aunque las variables puntero en ambos grafos se relacionen entre sí de la misma forma, puede que una misma variable pv_1 apunte a porciones de características muy distintas en ambos grafos, en cuyo caso no conviene unir los grafos. Sólo en el caso en el que las variables puntero en cada grafo apunten a porciones de memoria de características similares se permitirá la sumarización de ambos grafos.

Pasamos a continuación a definir las relaciones de alias entre las variables puntero y la compatibilidad entre los nodos apuntados por dichas variables, usados por la función $COMPATIBLE$:

- $ALIAS(rsg)$ es el conjunto de relaciones de alias, alr_i, del grafo rsg, donde cada alr_i identifica todas las variables puntero que apuntan al mismo nodo n_i:

$$ALIAS(rsg) = \{alr_1, ..., alr_n\} \text{ donde } alr_i = \{pv_1, ..., pv_m \in P \mid \\ \exists n_i \in N(rsg), < pv_1, n_i >, ..., < pv_m, n_i > \in PL(rsg)\}$$

- $COMP\_NODES(rsg_1, rsg_2)$ es una función booleana que devuelve *true* si los nodos apuntados directamente por la misma variable puntero son "compatibles", es decir tienen propiedades similares, y por tanto dichos nodos pueden ser sumarizados. Esta función se construye en dos pasos: primero se identifican los nodos de ambos grafos, $n_j \in N(rsg_1)$ y $n_k \in N(rsg_2)$, que son apuntados por el mismo conjunto de variables puntero; el segundo paso determina si los nodos tienen propiedades similares, usando para ello las funciones booleanas C_NODES y $C\_STRUCTURES$:

$$COMP\_NODES(rsg_1, rsg_2) = \begin{cases} 1 \text{ si } C\_STRUCTURES(rsg1, rsg_2) \wedge \\ \quad \forall pv_i \in P, < pv_i, n_j > \in PL(rsg_1) \wedge \\ \quad < pv_i, n_k > \in PL(rsg_2) \wedge \\ \quad C\_NODES(n_j, n_k) = 1 \\ 0 \text{ en caso contrario.} \end{cases}$$

donde las funciones booleanas C_NODES y $C\_STRUCTURES$ se describen como:

$$C\_NODES(n_i, n_j) = \begin{cases} 1 \text{ si } \begin{cases} TYPE(n_i) = TYPE(n_j) \wedge \\ SHARED(n_i) = SHARED(n_j) \wedge \\ SHSEL(n_i, sel) = SHSEL(n_j, sel) \forall sel \in S \wedge \\ C\_SPATH(n_i, n_j, k) = 1 \wedge \\ C\_REFPAT(n_i, n_j) = 1 \wedge \\ TOUCH(n_i, v_a, v_b) = TOUCH(n_j, v_a, v_b) \end{cases} \\ 0 \text{ en otro caso.} \end{cases}$$

$$C\_STRUCTURES(rsg_1, rsg_2) = \begin{cases} 1 \text{ si } \forall pv_i \in P, < pv_i, n_1 > \in PL(rsg_1), < pv_i, n_2 > \in PL(rsg_2), \\ \quad \nexists pv_j \in P, < pv_j, n_1' > \in PL(rsg_1), < pv_j, n_2' > \in PL(rsg_2), \\ \quad STRUCTURE(n_1) = STRUCTURE(n_1') \wedge \\ \quad STRUCTURE(n_2) \neq STRUCTURE(n_2') \\ 0 \text{ en otro caso.} \end{cases}$$

Como se puede ver, una vez localizados los nodos que son apuntados por las mismas variables puntero, utilizamos la función C_NODES para ver si tienen propiedades similares. En realidad basta con comparar las propiedades de los nodos de igual forma a como lo hace la función C_NODES_RSG presentada en la sección anterior para definir la función que comprime los RSGs. La diferencia entre ambas funciones es que mientras que C_NODES_RSG comparaba dos nodos del *mismo* RSG, la función C_NODES lo hace con nodos de *distintos* grafos. Para la mayoría de las propiedades, da igual que los nodos pertenezcan al mismo grafo o no, pero para la propiedad *structure* esto no es así. El problema con esta propiedad, es que su valor es un identificador común para los nodos entre los que hay un camino, por lo que no se puede comparar este atributo para dos nodos de distintos grafos. Es por esto que la única diferencia entre C_NODES_RSG y C_NODES es que la última no tiene en cuenta $STRUCTURE$ a la hora de comparar los nodos, y que si era tenido en cuenta por C_NODES_RSG. Por este motivo, la comparación de las propiedades *structure* se lo dejamos a otra función aparte, $C\_STRUCTURES$.

La función $C\_STRUCTURES$ tiene que buscar una equivalencia entre las *estructuras* de ambos grafos. La manera de hacerlo es hacer relaciones de equivalencia entre las estructuras de los nodos apuntados por las variables puntero. Por ejemplo, si la variable pv_1 apunta al nodo n_1 en el grafo rsg_1 y al nodo n_2 en el grafo rsg_2, las estructuras de dichos nodos ($STRUCTURE(n_1) = s1, STRUCTURE(n_2) = s2$) deben ser equivalentes ($s1 \equiv s2$). Supongamos que existe otra variable puntero pv_2 que apunta a los nodos n_1' y n_2' pertenecientes a los grafos rsg_1 y rsg_2 respectivamente. Si la estructura de n_1' es igual a la de n_1 ($STRUCTURE(n_1') = s1$), entonces para que se mantenga la relación de equivalencia $s1 \equiv s2$, la estructura de n_2' debe ser igual a la de n_2 ($STRUCTURE(n_2') = s2$). Si esto ocurre para todos los nodos apuntados por variables puntero en ambos grafos, entonces las propiedades *structure* de los nodos de ambos grafos son equivalentes. Si por el contrario no se cumple para algún nodo, entonces las *structure* no son equivalentes, con lo cual los nodos de ambos grafos no son "compatibles" y se evitará la sumarización de los grafos.

En resumen, para que dos RSGs sean compatibles y por tanto se puedan sumarizar, las relaciones de alias en ambos grafos deben ser las mismas y además los nodos apuntados por las variables puntero en ambos grafos deben de ser muy similares para que cuando sean unidos no se pierda información relevante de las estructuras de datos. Una consecuencia importante de esta manera de ver la compatibilidad entre los grafos, es que en cada grafo, *cada variable puntero puede apuntar como máximo a un nodo*. Esto es así, ya que como sólo se unen grafos con alias iguales, los nodos apuntados por una variable pv se sumarizarán en uno sólo, puesto que deben ser compatibles (segunda condición de compatibilidad de grafos). Si no se sumarizan en uno sólo, indicaría que los nodos apuntados por pv no son compatibles, pero si esto ocurre los grafos no serían unidos, precisamente por la segunda condición.

Una vez que se puede determinar cuando dos grafos son compatibles y pueden ser unidos, vamos a presentar cómo se mantiene el RSRSG de cada sentencia de un tamaño finito.

3.3.2 Sumarización de los grafos compatibles de un RSRSG

Como ya hemos comentado anteriormente, cuando se aplica la semántica abstracta de una sentencia sobre un RSG, para ver las modificaciones producidas por dicha sentencia sobre las configuraciones de memoria representadas por el RSG, las propiedades de los nodos cambian o aparecen nuevos nodos y/o enlaces. Para mantener el tamaño finito del RSG, tras estas modificaciones el grafo será simplificado aplicando la función $COMPRESS\_RSG$ descrita en la sección 3.2.10. De forma similar, cuando se aplica la semántica abstracta de una sentencia sobre un RSRSG, es decir, sobre todas las posibles configuraciones de memoria que pueden llegar a dicha sentencia, veremos que esto va a provocar cambios en los grafos pertenecientes al RSRSG asi como la aparición de nuevos RSGs. Por tanto, vamos a crear una función $COMPRESS\_RSRSG$ que es la encargada de mantener el tamaño finito del número de RSGs que aparecen en cada RSRSG.

De igual manera que la función $COMPRESS\_RSG$ transformaba el RSG uniendo aquellos nodos similares (compatibles) y por tanto reduciendo su tamaño, $COMPRESS\_RSRSG$ va a sumarizar aquellos RSGs del RSRSG similares (compatibles) para reducir el número de grafos que contiene dicho RSRSG.

Por tanto definimos $COMPRESS\_RSRSG(rsrsg) = rsrsg_c$ como:

$$COMPRESS\_RSRSG(rsrsg) = \{rsg_i \mid (rsg_i \in rsrsg \wedge \nexists rsg_j \in rsrsg, COMPATIBLE(rsg_i, rsg_j) = 1) \vee (rsg_i = MERGE\_COMP\_RSG(rsg_{k_1}, ..., rsg_{k_n}), rsg_{k_1}, ..., rsg_{k_n} \in rsrsg, \forall i = 1..(n-1), COMPATIBLE(rsg_{k_i}, rsg_{k_{i+1}}) = 1\}$$

donde el nuevo conjunto de grafos $rsrsg_c$ esta formado por aquellos grafos rsg_i que pertenecían a $rsrsg$ y no son compatibles con ningún otro grafo del conjunto, junto con los nuevos grafos resultantes de unir todos los grafos compatibles de $rsrsg$. Para comprobar la compatibilidad entre grafos se utiliza la función booleana $COMPATIBLE$ descrita en el apartado anterior. Para la unión de los grafos compatibles se ha utilizado una nueva función, $MERGE\_COMP\_RSG$, que toma una serie de grafos compatibles y devuelve un único grafo que es la unión de todos ellos de manera que representa todas las configuraciones de memoria que antes representaban el conjunto de grafos.

Esta función consiste en la unión recursiva de parejas de grafos llevada a cabo por la función $MERGE\_RSG(rsg_1, rsg_2)$ de la siguiente manera:

$$MERGE\_COMP\_RSG(rsg_1, ..., rsg_n) = MERGE\_RSG(rsg_1, MERGE\_RSG(rsg2, ..., MERGE\_RSG(rsg_{n-1}, rsg_n)...))$$

Al final, la reducción del número de grafos de un RSRSG se basa en la unión de grafos compatibles llevada a cabo por la función $MERGE\_RSG(rsg_1, rsg_2) = rsg$, la cual representa en un solo grafo, rsg, la información de las configuraciones de memoria representadas por los dos grafos rsg_1 y rsg_2. Esta función, no modifica ni el conjunto de variables puntero, P, ni el de selectores S de los grafos, puesto que estos conjuntos dependen únicamente del código analizado. Sin embargo los conjuntos de nodos $N(rsg)$, de enlaces desde variables puntero $PL(rsg)$ y de enlaces entre nodos $NL(rsg)$ del nuevo grafo, deben ser construidos a partir de los correspondientes conjuntos de los grafos rsg_1 y rsg_2.

En concreto, algunos de los nodos de rsg_1 y de rsg_2 serán sumarizados si son compatibles. Para determinar si son compatibles utilizaremos la función C_NODES descrita en la sección 3.3.1, y para la unión de los mismos la función $MERGE\_NODES$ presentada en la sección 3.2.10.

Por tanto podemos definir los nuevos conjuntos del grafo devuelto por la nueva función $MERGE\_RSG(rsg_1, rsg_2) = rsg$, como:

- El conjunto de nodos del nuevo grafo $N(rsg)$, está formado por tres subconjuntos: los nodos de rsg_1 no compatibles con ningún otro de rsg_2; los nodos de rsg_2 no compatibles con ninguno de rsg_1; y por último, los nodos resultantes de la unión de nodos compatibles de ambos grafos ($MERGE\_NODES$):

$$N(rsg) = \{n_i \in N(rsg_1) \mid \nexists n_j \in N(rsg_2), C\_NODES(n_i, n_j) = 1\} \cup \{n_i \in N(rsg_2) \mid \nexists n_j \in N(rsg_1), C\_NODES(n_i, n_j) = 1\} \cup \{n = MERGE\_NODES(n_i, n_j), \forall n_i \in N(rsg_1), \forall n_j \in N(rsg_2) \mid C\_NODES(n_i, n_j) = 1\}$$

  Teniendo en cuenta como son generados los nodos del nuevo grafo, podemos definir una función $MAP(n_i), n_i \in rsg_1$ que indica que nodo del nuevo grafo rsg es el que representa al nodo n_i del grafo rsg_1. Podemos ver que:

$$MAP(n_i) = \begin{cases} n \in N(rsg) \text{ si } \exists n_j \in N(rsg_2) \mid (C\_NODES(n_i, n_j) = 1) \wedge \\ \qquad (MERGE\_NODES(n_i, n_j) = n) \\ n_i \in N(rsg) \text{ en otro caso.} \end{cases}$$

El nodo del nuevo grafo rsg que representa al nodo n_i del grafo rsg_1, será o bien un nuevo nodo n resultante de la unión de n_i con algún nodo n_j de rsg_2 compatible con el primero, o si no es compatible con ningún nodo de rsg_2, n_i estará representado por él mismo en el nuevo grafo rsg. La función MAP para los nodos de rsg_2 es igual, pero cambiando $n_j \in N(rsg_2)$ por $n_j \in N(rsg_1)$. Esta función MAP es usada a continuación para construir los nuevos conjuntos $PL(rsg)$ y $NL(rsg)$.

- El conjunto de referencias desde variables puntero a nodos, $PL(rsg)$, se obtiene transformando las referencias que hay en rsg_1 y rsg_2 a las nuevas referencias sobre los nodos de rsg, utilizando la función MAP:

$$PL(rsg) = \{< pvar, MAP(n_i) \mid \forall < pvar, n_i > \in PL(rsg_1)\} \cup \\ \{< pvar, MAP(n_j) \mid \forall < pvar, n_j > \in PL(rsg_2)\}$$

  Si la variable pv_1 apuntaba al nodo n_1 en el grafo rsg_1, en el grafo rsg apuntará al nuevo nodo que representa a n_1 (función MAP).

- De forma similar los enlaces entre nodos de los grafos rsg_1 y rsg_2 es trasladada a enlaces entre los nuevos nodos de rsg *(NL(rsg))* tan solo con saber qué nodos de rsg representan a los de los grafos originales con la función MAP:

$$NL(rsg) = \{< MAP(n_i), sel_j, MAP(n_k) > \mid \forall < n_i, sel_j, n_k > \in NL(rsg_1)\} \\ \cup \{< MAP(n_i), sel_j, MAP(n_k) > \mid \forall < n_i, sel_j, n_k > \in NL(rsg_2)\}$$

  En el nuevo grafo, rsg, se mantienen todos los enlaces existentes en los grafos rsg_1 y rsg_2, tan solo se cambia la fuente y el destino de los enlaces, a los correspondientes nodos de rsg que representan ahora a los antiguos nodos de rsg_1 y rsg_2 (función MAP).

Ya hemos visto la manera en que se mantiene de tamaño finito el conjunto de grafos, RSRSG, asociado a cada sentencia. Como hemos descrito, aquellos grafos que representan estructuras de datos de acceso similar (función $COMPATIBLE$) no van a mantenerse por separado, sino que se van a sumarizar en un nuevo grafo (función $MERGE\_RSG$) que va a representar en un solo grafo todas aquellas configuraciones de memoria anteriormente representadas por los grafos compatibles.

Una vez definidos los RSGs y los RSRSGs, y visto como mantienen la información de las configuraciones de memoria de una forma finita, manteniendo las características principales de las mismas, vamos a presentar la semántica abstracta de las sentencias. Dicha semántica abstracta será la encargada de modificar los RSRSGs para que reflejen la nuevas configuraciones de memoria tras la ejecución de las sentencias. Antes de pasar a ver la semántica abstracta, en la tabla 3.1 presentamos un cuadro resumen de los conceptos más importantes de los RSGs y los RSRSGs.

RSG	RSRSG
Representa de una manera finita configuraciones de memoria, capturando las propiedades fundamentales de las porciones de memoria en los nodos.	Representa con un conjunto finito de RSGs, todas las posibles configuraciones de memoria que pueden aparecer para cada sentencia de un código.
Para mantener el tamaño finito del grafo, los nodos que representan porciones de memoria similares (similares propiedades) son sumarizados en un único nodo, que representa todas las porciones de memoria representadas por los originales: Función $COMPRESS\_RSG$	Para mantener limitado el número de RSGs en un RSRSG, los grafos similares son sumarizados en uno único, de manera que este representa todas las configuraciones de memoria antes representadas por los originales: Función $COMPRESS\_RSRSG$
Para determinar si dos nodos son similares y pueden ser sumarizados en uno, se comparan sus propiedades, que mantienen las características fundamentales de las porciones de memoria que representan: Función C_NODES_RSG	Para determinar si dos grafos son similares y pueden ser sumarizados en uno, se comparan las propiedades de acceso a los grafos que determina si los grafos representan configuraciones de memoria similares: Función $COMPATIBLE$

Tabla 3.1: Cuadro resumen sobre los RSGs y RSRSGs.

3.4 Semántica Abstracta

En esta sección vamos a describir como las sentencias modifican los grafos que representan configuraciones de memoria de algún camino del flujo de control que pasa por dicha sentencia, para obtener los nuevos grafos que representan las configuraciones de memoria ya modificadas por la sentencia. Estos grafos modificados formarán el RSRSG asociado a la sentencia que describe todas las posibles configuraciones de memoria que se pueden dar tras la ejecución de la sentencia y que será uno de los conjuntos de grafos de entrada para las sentencias sucesoras de la actual.

Definiremos la semántica abstracta para cada una de las sentencias básicas que manejan punteros descritas anteriormente, de forma que dado un RSG devolverá un conjunto de RSGs como resultado de la modificación del RSG original. Como podremos ir viendo, puesto que la semántica de cada sentencia se define para un solo RSG, el análisis de la sentencia sobre un RSRSG, consistirá en aplicar la semántica abstracta de la sentencia sobre todos los grafos pertenecientes a dicho RSRSG.

A continuación presentamos los pasos en los que se divide la interpretación de cada sentencia sobre los grafos de un RSRSG dado, dejando la descripción formal de la semántica abstracta de cada sentencia para el apéndice B.

3.4.1 Interpretación abstracta de las sentencias

Antes de que un RSG dado sea modificado por una sentencia, dicho grafo tiene que ser modificado para *enfocar* en aquellas regiones del grafo que serán directamente modificadas por dicha sentencia, de manera que los cambios sean lo menos conservativos posible.

Como hemos visto, una de las características fundamentales de los métodos basados en grafos, es que para representar configuraciones de memoria de tamaño desconocido de forma finita, deben *sumarizar* la información de distintas porciones de memoria en un solo nodo. Nosotros asociamos una serie de propiedades a cada nodo, precisamente para poder distinguir las características fundamentales de las porciones de memoria representadas por el mismo. La cuestión es ahora, que cuando un grafo va a ser modificado por la acción de una sentencia que utiliza punteros, debemos estar seguros de que los nodos modificados representan solo porciones de memoria apuntadas por los punteros de la sentencia y no a otras porciones *compatibles* con estas y que fueron *sumarizadas* en un mismo nodo, haciendo así menos conservativa la aplicación de la sentencia. Por ejemplo, en la figura 3.13 podemos observar un RSG que representa una de las listas doblemente enlazadas apuntadas desde una hoja de un árbol, de la estructura de datos ejemplo presentada en la figura 3.5 y cuyo RSRSG reducido presentamos en la figura 3.6. El primer elemento de dicha lista está a su vez apuntado por la variable x.

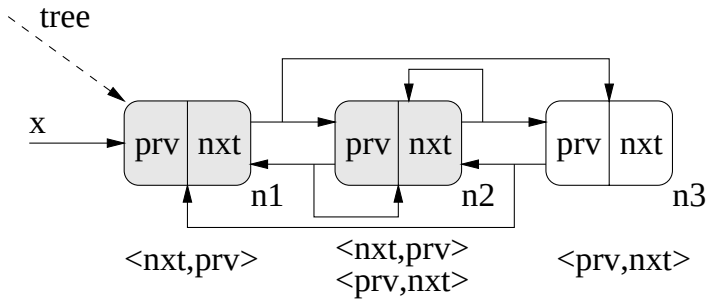


Figura 3.13: Lista doblemente enlazada apuntada desde una hoja de un árbol.

En dicho grafo, el nodo n_2 representa todas aquellas porciones de memoria que no son ni la primera ni la última de la lista doblemente enlazada. Si este grafo es uno de los que llegan a una sentencia en la que se hace referencia a $x \rightarrow nxt$ ya sea en la parte derecha (rhs) o izquierda (lhs) de la misma, las modificaciones que implica dicha sentencia se tendrán que llevar a cabo sobre las porciones de memoria realmente apuntadas por $x \rightarrow nxt$. Si se modificara directamente el nodo n_2 estaríamos cambiando propiedades y/o enlaces de todas las porciones representadas por n_2, cuando hay algunas de ellas que no son referenciadas por $x \rightarrow nxt$ (tan solo el segundo elemento de la lista lo es, mientras que los demás elementos centrales no lo son), debiendo ser en este caso muy conservativos.

Por este motivo, antes de aplicar la semántica abstracta de una sentencia sobre un grafo, es necesario *enfocar* en el grafo ambas partes de la sentencia (rhs y lhs) para asegurarnos de que los nodos implicados en las referencias realmente representen sólo a las porciones de memoria referenciadas desde la sentencia. Fijándonos en las sentencias simples con punteros analizadas por el método, podemos ver que tanto rhs como lhs pueden ser de dos tipos:

- **pv**: Una simple variable puntero. En este caso, las porciones de memoria referenciadas ya están enfocadas, puesto que como ya sabemos, una variable puntero cualquiera, sólo puede referenciar a un nodo en un RSG. Dicho nodo representa todas las porciones de

memoria que son referenciadas directamente por la variable y ninguna más (si alguna no fuera apuntada por la variable puntero, no sería compatible con las demás, $SPATH0$, y por tanto no sería sumarizada en el mismo nodo). Por este motivo, este tipo de rhs o lhs no tienen que ser enfocados, puesto que ya lo están.

- **pv→sel**: Este caso, como hemos visto en el ejemplo anterior, es más problemático, puesto que el nodo apuntado por $pv \rightarrow sel$ puede representar a más porciones de memoria que las realmente referenciadas por la expresión. El proceso de *enfoque* para este caso se divide en dos pasos: *materialización de grafos* y *materialización de nodos*.

  1. La *materialización de grafos* consiste en dividir el grafo original, de manera que en cada uno de ellos el nodo apuntado por pv tan solo tenga un destino por el selector sel. Ya sabemos que un nodo de un RSG representa muchas porciones de memoria, en este caso, todas ellas apuntadas por pv. Puede que cada una de ellas referencie a porciones distintas por el selector sel y por tanto a nodos distintos. La idea pues de esta división, es dejar en cada grafo un solo nodo destino para este selector, centrándonos en cada caso en un enlace distinto de $pv \rightarrow sel$. Esta división es llevada a cabo por una función creada para tal efecto, denominada $DIVIDE$. Antes de pasar al siguiente paso de *materialización de nodos* en cada uno de los grafos obtenidos, el proceso de *enfoque* del grafo lleva consigo una operación adicional. Con la *división* se ha determinado la certeza de que el nodo apuntado por pv referencia por sel a un único nodo determinado. Esta certeza, junto con la eliminación de los otros destinos por sel del nodo apuntado por pv, hace que existan nodos y enlaces en cada grafo que realmente representan porciones de memoria y enlaces representadas por otro grafo obtenido en la división, por lo que se pueden eliminar del grafo. Esta *poda* de nodos y enlaces es llevada a cabo por la función $PRUNE$ que será presentada más adelante.
  2. Una vez materializados los grafos, en cada uno de ellos tenemos que *materializar el nodo* que realmente es el que representa las porciones de memoria referenciadas por $pv \rightarrow sel$, como hemos presentado en el ejemplo de la figura 3.13.

Cuando ya se han obtenido los diferentes grafos enfocados para las referencias que aparecen en la sentencia, dichos grafos son ya modificados por la semántica abstracta de las sentencias, que transforman el grafo para reflejar la ejecución de la sentencia. Formalmente, vamos a definir $ST_{[st]}(rsg)$ como las modificaciones sobre el grafo rsg producidas por la interpretación abstracta de la sentencia $[st]$ sobre dicho grafo, que darán lugar a un conjunto de grafos (el *enfoque* de nodos y grafos se incluye dentro de esta interpretación abstracta para aquellas sentencias que realmente lo necesiten), de la forma:

$$ST_{[st]}(rsg) = \{rsg_1, ..., rsg_n\}$$

Estos grafos obtenidos, son modificaciones sobre el grafo original rsg, que pueden cambiar las propiedades y los enlaces de los nodos. Estos cambios, provocan que nodos que antes no eran compatibles, ahora si lo sean y puedan ser sumarizados, reduciendo asi el tamaño de dichos grafos y manteniéndolo de tamaño finito. Para llevar a cabo estas sumarizaciones, utilizamos la función $COMPRESS\_RSG$ creada para tal efecto y presentada en la sección 3.2.10.

Por último, hay que recordar que el objetivo de la interpretación abstracta de una sentencia es obtener el conjunto de grafos, RSRSG, que representan todas las configuraciones de

memoria que se pueden obtener tras la ejecución de una sentencia $[st]$. Por tanto, de todos los grafos obtenidos tras aplicar la semántica abstracta de la sentencia ya comprimidos, tenemos que construir el conjunto reducido de los mismos, uniendo aquellos que son muy similares (compatibles). Esta idea ya se presentó en la sección 3.3, y para ello utilizamos la función $COMPRESS\_RSRSG$, que aparece en dicha sección.

Por tanto, ya podemos definir de forma general la interpretación abstracta de cualquier sentencia $[st]$ sobre un RSRSG de entrada ($RSRSG_{in}$), definiendo el RSRSG que genera ($RSRSG_{out}$) tras la aplicación de las modificaciones de dicha $[st]$ sobre los RSGs de $RSRSG_{in}$.

Sea $B([st], RSRSG_{in}) = \{rsg_i \mid rsg_i = COMPRESS\_RSG(rsg'_i), \forall rsg'_i \in ST_{[st]}(rsg_j), \forall rsg_j \in RSRSG_{in}\}$ el conjunto de todos los rsg_i individuales generados a partir de la aplicación de la semántica abstracta de $[st]$ sobre todos los grafos de $RSRSG_{in}([st])$, una vez comprimidos. Entonces definimos

$$RSRSG_{out}([st], RSRSG_{in}) = COMPRESS\_RSRSG(B([st], RSRSG_{in}))$$

que es el conjunto reducido de RSGs que representan las configuraciones de memoria tras la ejecución de la sentencia $[st]$ sobre las configuraciones de memoria que alcanzan la sentencia y que están representadas por $RSRSG_{in}$.

La interpretación de un programa completo consistirá, como en el método basado en los SSGs, en un proceso iterativo de aplicación de la semántica abstracta de cada sentencia sobre los RSRSGs de sus predecesores en el grafo de flujo de control, para generar el RSRSG asociado a la sentencia actual, hasta que se alcanza un punto fijo en el que el RSRSG asociado a cada sentencia no cambia más. El siguiente pseudocódigo muestra el proceso iterativo de creación del conjunto RSRSG de cada sentencia del código (RSRSG(ST) es el conjunto de grafos asociados a la sentencia ST).

```
forall (ST_i)
  Insertar (ST_i, ListaT);
while (ListaT != NULL)
 {
   Extraer(ST_i, ListaT);
   RSRSG_st = RSRSG(ST_i);
   forall (ST_j ∈ Predecesores(ST_i))
     {
       RSRSG_a = RSRSG_out([ST_i], RSRSG(ST_j));
       RSRSG_st = COMPRESS_RSRSG(RSRSG_st ∪ RSRSG_a);
     }
   if (RSRSG_st != RSRSG(ST_i))
     {
       RSRSG(ST_i) = RSRSG_st;
       forall (ST_j ∈ Sucesores(ST_i))
         Insertar (ST_j, ListaT);
     }
 }
```

Es decir, se introducen todas las sentencias en una lista de trabajo. Mientras que haya alguna sentencia en la lista, se extrae una, se aplica la semántica abstracta de dicha sentencia al conjunto RSRSG de sus predecesores, creando un RSRSG nuevo para la sentencia actual. Si el nuevo RSRSG es distinto del que tenía asociado la sentencia, hay nueva información que tiene que ser analizada por las sentencias sucesoras de la actual. Por esto, estas sentencias sucesoras son introducidas en la lista de trabajo para ser analizadas posteriormente. El proceso termina cuando la lista de trabajo no tiene más sentencias por analizar, lo que implica que

ninguna sentencia a modificado el RSRSG que tiene asociado, y por tanto se ha alcanzado un punto fijo.

Para una mejor comprensión de la ejecución simbólica de una sentencia, en la figura 3.14 presentamos una representación a alto nivel del proceso de generación del conjunto de grafos de salida $RSRSG_{out}([st])$ a partir del conjunto de grafos de entrada $RSRSG_{in}([st])$.

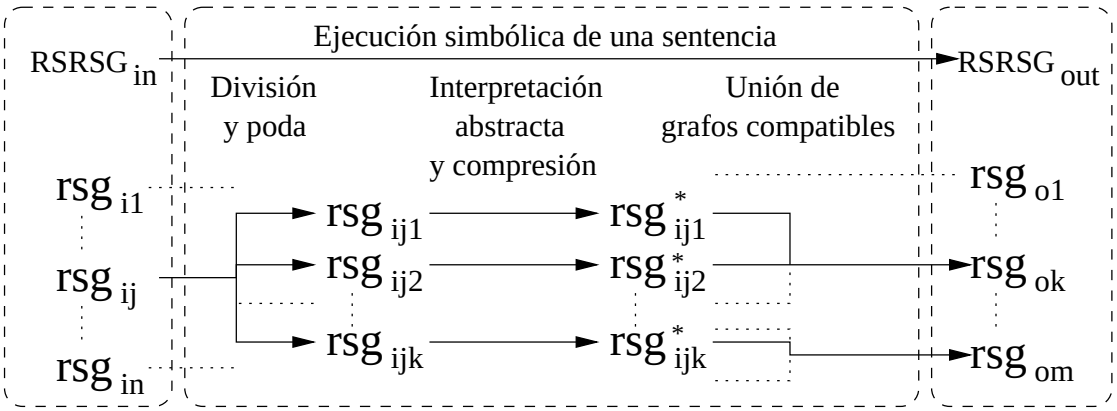


Figura 3.14: Descripción esquemática de la ejecución simbólica de una sentencia.

Como se puede apreciar en la figura 3.14, los grafos pertenecientes a $RSRSG_{in}$, antes de pasar a ser modificados por la semántica abstracta de la sentencia, son *divididos* y cada uno de los grafos resultantes de la división son *podados*, de manera que se consigue enfocar las referencias de la sentencia. Posteriormente, cada uno de estos grafos es modificado aplicando la semántica abstracta de la sentencia y posteriormente es *comprimido* ($COMPRESS\_RSG$) para mantener de tamaño finito todos los grafos. De entre todos los grafos obtenidos en los pasos anteriores, se *unen* los grafos compatibles para formar el conjunto reducido de grafos, $RSRSG_{out}$ ($COMPRESS\_RSRSG$) que representa las configuraciones de memoria tras la ejecución de la sentencia.

3.4.2 Un ejemplo de interpretación abstracta

Antes de describir las nuevas funciones de *división*, *poda* y *materialización* necesarias para la ejecución simbólica de una sentencia, vamos a ejemplificar la ejecución simbólica de la sentencia $x \rightarrow nxt = NULL$ sobre el grafo de la figura 3.13, para una mejor comprensión de las distintas fases de la ejecución mostradas en la figura 3.14.

Para la sentencia en cuestión, tenemos que la única parte de la misma que hay que *enfocar* es $lhs = x \rightarrow nxt$, para hacer que en cada grafo que vaya a ser modificado por la semántica abstracta de la sentencia $[x \rightarrow nxt = NULL]$, el nodo apuntado por $x \rightarrow nxt$ represente sólo a las porciones de memoria realmente referenciadas por dicho selector. Por tanto a este grafo se le van a aplicar las dos operaciones que enfocan en cada grafo dicho selector, $DIVIDE$ y $PRUNE$.

Con la operación de división, se obtienen los grafos de la figura 3.15. El objeto de la división es que en cada grafo el nodo apuntado por x sólo tenga un enlace por nxt. Como podemos observar en el grafo original (fig. 3.13) el nodo n_1 apuntado por x, referencia a los nodos n_2 y n_3 por nxt. Esto es así, puesto que el grafo está representando tanto a listas de dos elementos (el primero apunta al último por nxt) como a listas de más de dos elementos (el

primero apunta a algún elemento intermedio en vez de al último). Como podemos observar en la figura 3.15, en los grafos obtenidos tras la división, tan solo existe un destino para nxt desde el nodo n_1 (n_2 en rsg'_1, y n_3 en rsg'_2).

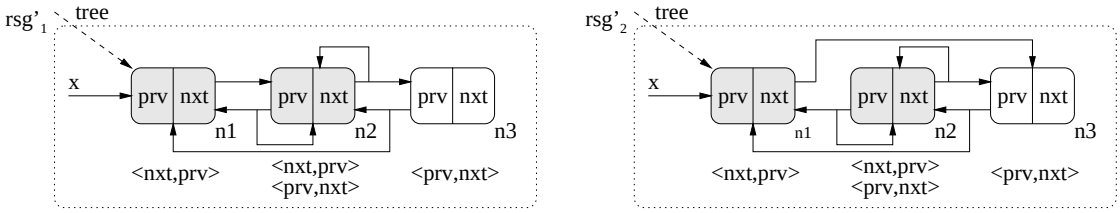


Figura 3.15: Grafos obtenidos tras la división.

Tras esta división, puede haber nodos y selectores que no pertenezcan a un determinado grafo, puesto que representan porciones de memoria y enlaces representados en otro. Para localizar dichos nodos o selectores, basta con mirar las propiedades de los nodos. Aquellos nodos y selectores que no satisfagan las propiedades de los nodos, es que no pertenecen al grafo actual, puesto que la división tan solo separa la información del grafo y no modifica las propiedades de los nodos. Así, por ejemplo, en el grafo rsg'_1 vemos que el selector $<n_3, prv, n_1>$, no satisface el *cyclelink* $<prv, nxt>$ del nodo n_3. Este *cyclelink* indica que cualquier nodo apuntado por n_3 y el selector prv, debe a su ver referenciar a n_3 por nxt. Antes de la división esto se cumplía, pero ahora no, puesto que dicho selector se ha pasado al grafo rsg'_2. Por tanto, el enlace $<n_3, prv, n_1>$ puede ser eliminado de rsg'_1, obteniendo de esta forma el grafo rsg''_1 de la figura 3.16. Como veremos a continuación, por un proceso un poco más complejo, el nodo n_2 del grafo rsg'_2 y todos sus enlaces son eliminados dando lugar al grafo rsg''_2 de la figura 3.16 (dicho nodo y sus enlaces, están representados en rsg''_1). Empezaremos advirtiendo que del nodo n_2 de rsg'_2 se borraría el selector $<n_2, prv, n_1>$ pues, como en el caso anterior, no satisface el *cyclelink* $<prv, nxt>$ de dicho nodo. Además, como tras dividir, se tiene la certeza que n_1 apunta a n_3 por el selector nxt, y como dicho nodo no es *shared* por dicho selector, se puede eliminar el enlace $<n_2, nxt, n_3>$, puesto que no puede coexistir con el anterior. Esto hace que el enlace $<n_3, prv, n_2>$ no satisfaga el *cyclelink* $<prv, nxt>$ de n_3, y por tanto es eliminado. Tras todo esto, el nodo n_2 puede ser eliminado pues es inaccesible desde cualquier variable puntero usada en el código. Tras la poda, se puede observar que el grafo rsg''_1 está representando las listas doblemente enlazadas de más de dos elementos, mientras que el grafo rsg''_2 representa las de exactamente dos elementos.

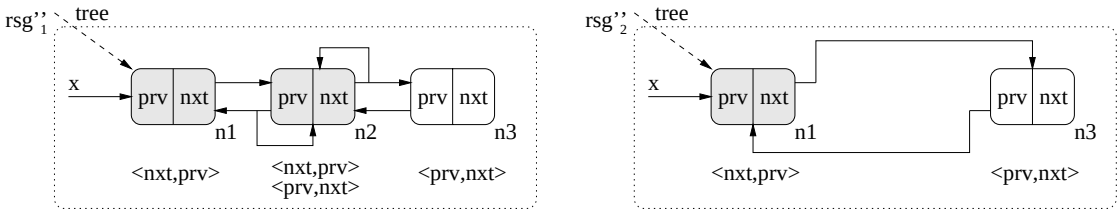


Figura 3.16: Grafos obtenidos tras la poda.

Una vez enfocados los grafos, hay que aplicar la semántica abstracta de la sentencia. Pero antes de hacerlo, hay que *enfocar* un nodo en el grafo rsg''_1. En concreto tenemos que *materializar* un nuevo nodo desde el nodo n_2 que represente exactamente aquellas porciones de memoria realmente apuntadas por $x \rightarrow nxt$. Básicamente, la materialización lo que hace es

crear un nodo exactamente igual a n_2 que es el realmente apuntado por n_1 con el selector nxt. Por tanto, el selector de n_1 por nxt sobre n_2 es borrado. Aparte, aplicando las restricciones impuestas por las propiedades (*cyclelinks*, *shared*, etc), se eliminan enlaces sobre n_2 y se crean los propios sobre n_4. Como resultado de materializar un nodo de n_2 en el grafo rsg''_1, obtenemos el grafo rsg'''_1 mostrado en la figura 3.17

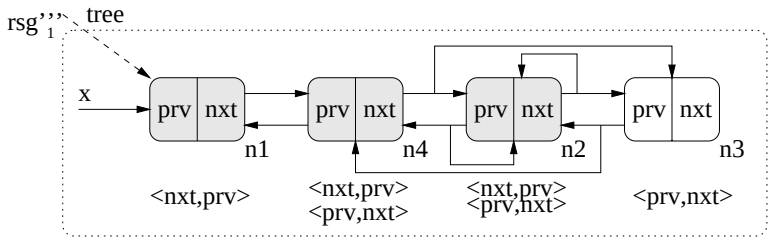


Figura 3.17: Grafo obtenido tras la materialización de un nuevo nodo.

Podemos ver que el nuevo nodo n_4 es una copia del nodo n_2, en el sentido de que tiene sus mismas *propiedades* y es el único apuntado por $x \rightarrow nxt$. Vemos como n_1 ya no referencia a n_2 por nxt puesto que las porciones de memoria realmente referenciadas, ahora están siendo representadas en n_4. Ahora n_2 representa al resto de porciones *no referenciadas* por $x \rightarrow nxt$. También se puede apreciar como ahora no existe un enlaces por prv entre n_2 y n_1. Esto es debido a que no satisface el *cyclelink* $<prv, nxt>$ de n_2. A excepción de esos enlaces, los restantes de n_4, como se puede observar en el grafo, son los mismos que poseía n_2. Merece especial mención, la no existencia de un enlace del tipo $<n_4, nxt, n_4>$, que sería copia del enlace $<n_2, nxt, n_2>$. Este no se introduce gracias a la información $SHSEL(n_4, nxt)$ =*false* y al hecho de la existencia segura del enlace $<n_1, nxt, n_4>$. Al existir este enlace, la información *shared por selector* indica que no puede existir otro enlace sobre n_4 por el selector nxt.

Llegados a este punto en el que hemos enfocado tanto grafos como nodos para aislar los enlaces que aparecen en la sentencia, es hora de modificar los grafos, aplicando las modificaciones necesarias para plasmar los cambios producidos por la sentencia $x \rightarrow nxt = NULL$. Ahora bastará con borrar cualquier enlace nxt que tengan los nodos apuntados por x en los grafos anteriormente obtenidos. En la figura 3.18 mostramos los dos grafos obtenidos a partir de los anteriores, eliminando dicho enlace.

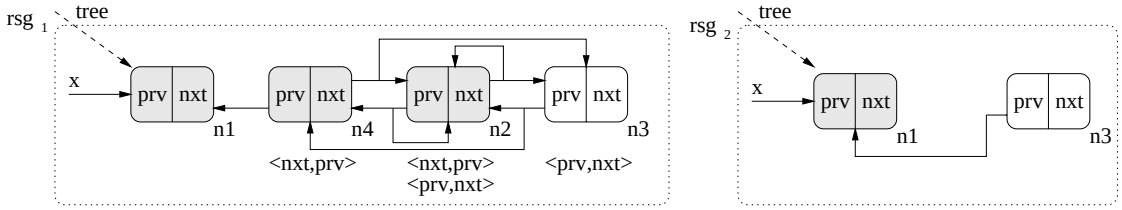


Figura 3.18: Grafos obtenidos tras la aplicación de la semántica abstracta de la sentencia $x \rightarrow nxt = NULL$.

Como podemos observar, en ambos grafos, el nodo n_1 ya no referencia a nadie por el selector nxt. Esta aplicación de la semántica abstracta además de modificar los grafos al igual que la división, poda y materialización de nodos, también modifica las propiedades de algunos nodos, cosa que no sucedía en dichas operaciones. De ahí la importancia de que el nodo modificado sólo representase aquellas porciones de memoria realmente involucradas en la sentencia. Así, por ejemplo, el nodo n_4 del grafo rsg_1 vería modificada su propiedad *reference*

patterns puesto que ya no es referenciado por ningún enlace por el selector nxt. Algo similar ocurre con el nodo n_3 de rsg_2.

Para terminar la ejecución simbólica de la sentencia, hay que comprimir los grafos uniendo los nodos compatibles y posteriormente habría que unir los grafos compatibles. La compresión de los grafos rsg_1 y rsg_2 da como resultado los mismos grafos, puesto que todos los nodos son incompatibles. El caso más difícil de ver, es el del nodo n_4 que se obtuvo a partir del nodo n_2 y por tanto compartían las mismas propiedades. Ahora no son compatibles, puesto que el nodo n_2 es referenciado por el selector nxt mientras que el nodo n_4 ya no lo es (*SELINset* distintos).

Una vez comprimidos los grafos, hay que ver si dichos grafos son compatibles y pueden ser unidos y representados en uno solo, en definitiva, hay que crear el $RSRSG_{out}$. Si observamos los grafos, vemos que si son compatibles, puesto que sus relaciones de alias son idénticas (la variable x no apunta donde otra variable en ninguno de los grafos) y los nodos apuntados por x son compatibles (tienen propiedades similares). Puesto que son compatibles, los grafos son unidos (función $MERGE\_RSG$) obteniendo el único grafo que representa la lista doblemente enlazada de dos o más elementos, cuyo primer elemento no tiene enlace por nxt (como expresa la ejecución de la sentencia $x \rightarrow nxt = NULL$). El grafo unión se puede ver en la figura 3.19.

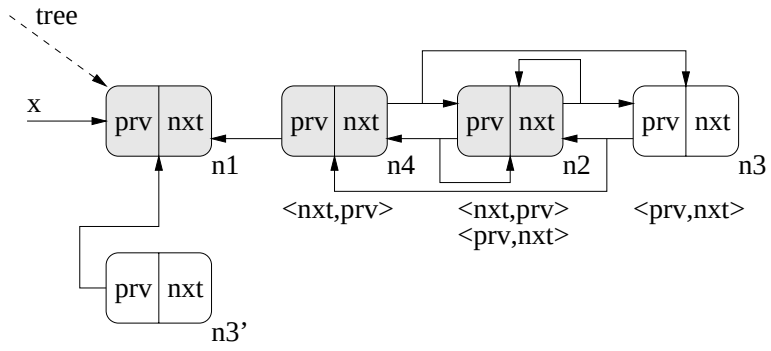


Figura 3.19: Grafo obtenido tras la unión de los grafos compatibles rsg_1 y rsg_2.

El nodo n_1 del nuevo grafo es la unión de los nodos n_1 compatibles de ambos grafos. El nodo n_3' se corresponde con el nodo n_3 de rsg_2. Este no se ha unido, en este caso, con el nodo n_3 de rsg_1 puesto que sus propiedades no son *compatibles*, ya que n_3 es referenciado por selectores nxt mientras que n_3' no (*SELINset*). Por lo demás, vemos como todos los nodos y enlaces de ambos grafos se mantienen en el nuevo, ya que este debe representar todas las configuraciones de memoria representadas en los grafos originales.

A continuación vamos a describir de una manera formal las operaciones utilizadas en la presentación de la interpretación abstracta de las sentencias, como son: la *división* de los grafos, la *poda* y por último la *materialización* de nodos.

3.4.3 División de grafos

Como hemos introducido en la sección anterior, la *división* de los grafos tiene lugar cuando se *enfocan* grafos debido a que en la sentencia aparece una referencia del tipo $pv \rightarrow sel$. Por tanto, esta operación de división tendrá lugar para las sentencias $x \rightarrow sel = NULL$, $x \rightarrow sel = y$ y $y = x \rightarrow sel$.

Ya hemos mencionado que el objetivo de esta división es dividir el grafo de entrada en

varios, de manera que en cada uno de ellos, el nodo directamente apuntado por la variable x apunte a un único nodo por el selector sel. Básicamente, con la división, pretendemos separar las características individuales de configuraciones de memoria representadas en un único grafo, relacionadas con el enlace en cuestión.

Podemos definir la operación $DIVIDE(rsg, x, sel) = \{rsg_1, ..., rsg_n\}$ que divide el grafo rsg en el conjunto de grafos $\{rsg_1, ..., rsg_n\}$, de manera que el nodo apuntado por x en cada uno de ellos, sólo referencie a otro nodo por el selector sel. La división es realizada de la siguiente manera. Sea $n \in N(rsg)$ de manera que n es apuntado por x ($<x, n> \in PL(rsg)$), entonces, $\forall < n, sel, n_i > \in NL(rsg)$, creamos un grafo rsg_i de la siguiente manera:

- $N(rsg_i) = N(rsg)$
- $PL(rsg_i) = PL(rsg)$
- $NL(rsg_i) = NL(rsg) \setminus \{< n, sel, n_j > \in NL(rsg), \forall n_j \neq n_i\}$

Podemos ver que el nuevo grafo rsg_i es exactamente igual al original, a excepción de que son eliminados todos los enlaces desde el nodo apuntado por x (nodo n) por el selector sel, distintos del elegido para la división. Esta operación deja mucha información redundante en cada grafo, perteneciente a otros grafos resultantes de la división. Por este motivo, los grafos obtenidos tras la *división* son *podados* para eliminar esta información redundante. La operación de *poda* se presenta en el siguiente apartado.

3.4.4 Poda de grafos

Como ya se ha comentado, hay situaciones, por ejemplo tras la división, en las que los grafos contienen nodos y enlaces que no pertenecen a las configuraciones de memoria representadas por el grafo y por ese motivo se pueden eliminar, obteniendo así una representación más exacta de las estructuras. La manera de determinar si los nodos o los enlaces pertenecen o no al grafo, consiste en comprobar que satisfacen las propiedades de los nodos. Así, por ejemplo, un nodo n de un grafo rsg, con la propiedad $SELIN(n, sel) = 1$ implica que debe existir otro nodo n_i apuntando a n por el selector sel ($<n_i, sel, n> \in NL(rsg)$). Si dicho nodo no existe en el grafo, eso implica que el nodo n no satisface sus propiedades en las configuraciones de memoria representadas por rsg y por tanto puede ser eliminado junto con sus selectores.

Para formalizar el proceso de *poda* de un grafo, debemos definir dos funciones booleanas que determinen si un nodo, $N\_PRUNE(n)$, o un enlace, NL_PRUNE ($< n_1, sel, n_2 >$) satisfacen las propiedades de los nodos o no:

- Un nodo n puede ser eliminado del grafo si sus enlaces no satisfacen la propiedad *reference patterns* del mismo. Es decir, si las funciones $SELIN/\ SELOUT$ indican que el nodo es referenciado por un selector sel o referencia a algún otro por sel, entonces deben existir enlaces sobre o desde el nodo por el selector sel. En otro caso, se está violando la propiedad *reference patterns* del nodo y puede ser eliminado del grafo actual.

$$N\_PRUNE(n) = \begin{cases} 1 \text{ si } \begin{cases} (\exists sel_i \mid SELIN(n, sel_i) = 1 \ \wedge \ \nexists n_2 \in \\ \qquad N(rsg), < n_2, sel_i, n > \in NL(rsg)) \ \vee \\ (\exists sel_i \mid SELOUT(n, sel_i) = 1 \ \wedge \ \nexists n_2 \in \\ \qquad N(rsg), < n, sel_i, n_2 > \in NL(rsg)) \end{cases} \\ 0 \text{ en caso contrario.} \end{cases}$$

- Por otro lado, un enlace puede ser eliminado si no satisface las condiciones impuestas por la propiedad *cyclelinks*. Así, si un nodo n tiene el *cyclelink* $<sel_1, sel_2>$ en su propiedad, indica que las porciones de memoria representadas por n que apuntan a alguna otra por sel_1, deben ser referenciadas por estas otras mediante el selector sel_2. Por este motivo, si el nodo n apunta a otro nodo n_2 por medio del selector sel_1 y éste no referencia a n_1 con el selector sel_2, entonces se puede eliminar el enlace $<n_1, sel, n_2>$ puesto que no satisface las propiedades de las porciones de memoria representadas en el nodo. Formalmente definimos:

$$NL\_PRUNE(< n_1, sel_i, n_2 >) = \begin{cases} 1 \text{ si } \exists \ < sel_i, sel_j > \in \\ \qquad CYCLELINKS(n_1) \wedge \\ \qquad \nexists < n_2, sel_j, n_1 > \in NL(rsg) \\ 0 \text{ en otro caso.} \end{cases}$$

Una vez determinados que nodos y enlaces de un grafo pueden ser eliminados puesto que no satisfacen las propiedades de los nodos, se pasa a la eliminación de los mismos por medio de la *poda*. Esta *poda* es un proceso iterativo de eliminación de nodos y enlaces. La eliminación de un nodo o enlace implica la modificación de los enlaces de entrada o salida de sus nodos adyacentes. Dichos nodos puede que ahora no satisfagan sus propiedades y por tanto deben ser eliminados también. Un ejemplo puede ser un nodo con $SELIN(n, sel_1) = 1$ y sobre el cual exista un enlace por sel_1 desde otro nodo n_2, no sería eliminado pues satisface sus propiedades. Sin embargo si n_2 es eliminado, se eliminan todos su enlaces y por tanto el nodo n_1 ya no es referenciado por sel_1 pasando a no satisfacer sus propiedades y por tanto debe de ser eliminado también.

El proceso completo puede ser expresado como $PRUNE(rsg) = rsg_n$, donde el grafo rsg_n es el resultado de podar el grafo rsg. El proceso iterativo comienza con $rsg_0 = rsg$. Entonces, $\forall i = 1..n$:

- $N(rsg_i) = N(rsg_{i-1}) \setminus \{n \in N(rsg_{i-1}) | N\_PRUNE(n) = 1\}$
- $PL(rsg_i) = PL(rsg_{i-1}) \setminus \{< pvar, n > \in PL(rsg_{i-1}) | N\_PRUNE(n) = 1\}$
- $NL(rsg_i) = NL(rsg_{i-1}) \setminus \{< n_1, sel, n_2 > \in NL(rsg_{i-1}) | (N\_PRUNE(n_1) = 1) \vee (N\_PRUNE(n_2) = 1) \vee (NL\_PRUNE(< n_1, sel, n_2 >) = 1)\}$

En cada paso, creamos un nuevo grafo rsg_i a partir del creado en el paso anterior rsg_{i-1}, eliminando nodos y enlaces que no satisfacen las propiedades, utilizando para ello las funciones N_PRUNE y NL_PRUNE. Al eliminar nodos y enlaces, la propiedad *SPATH* de algunos nodos se puede ver afectada, en el sentido de que puede que el nodo ya no sea referenciado por un camino de longitud uno desde una variable puntero (ver propiedad *SPATH* en sección 3.2.6). De este modo definimos:

$$SPATH(n) = SPATH(n) \setminus \{< pv, sel > \ | \ < pv, n_{pv} > \in PL(rsg), \\ < n_{pv}, sel, n > \in NL(rsg) \ \wedge \ ((N\_PRUNE(n_{pv}) = 1) \vee \\ (NL\_PRUNE(< n_{pv}, sel, n >) = 1))\}$$

Vemos como son eliminados de los *SPATH* de los nodos, aquellos que partían de un nodo, n_{pv}, que es eliminado, o utilizaban un enlace, $<n_{pv}, sel, n>$, que es también eliminado.

El proceso termina, cuando se alcanza un grafo rsg_n que cumple:

- $\forall n \in N(rsg_n), N\_PRUNE(n) = 0 \wedge$
- $\forall < n_1, sel, n_2 > \in NL(rsg_n), NL\_PRUNE(< n_1, sel, n_2 >) = 0$

Es decir, se termina el proceso cuando se obtiene un grafo en el cual todos sus nodos y enlaces satisfacen las propiedades del mismo.

La *poda* de grafos, como ya veremos, no sólo se utiliza tras la *división* vista anteriormente, sino que será utilizada cuando se modifica algún grafo para amoldarlo a una nueva situación, y se quiere eliminar nodos y enlaces que no pertenecen a esta nueva situación, ya que no satisfacen las propiedades de los nodos.

3.4.5 Materialización de un nuevo nodo

Al materializar un nuevo nodo, en el proceso de enfoque para una sentencia que hace referencia a un enlace del tipo $x \rightarrow sel$, desde otro existente, pretendemos separar en nodos distintos las porciones de memoria realmente referenciadas por el enlace de aquellas que no lo son. De este modo las posteriores modificaciones sobre las porciones apuntadas por el enlace $x \rightarrow sel$ se llevarán a cabo sólo sobre el nodo que realmente las representa, sin que afecte a otras posibles porciones compatibles con estas últimas.

Para ello se llevara a cabo una materialización de un nodo exactamente igual al referenciado por sel desde n_x, siendo n_x el nodo apuntado directamente por la variable x, el cual representará las porciones de memoria que son referenciadas por $x \rightarrow sel$, y el antiguo nodo representará las porciones compatibles con las anteriores pero que no son referenciadas por $x \rightarrow sel$.

Esta materialización puede llevar consigo la generación de dos grafos distintos a partir del original. Si el nodo referenciado por n_x mediante sel, n_1, es apuntado directamente por otra variable puntero pv, la materialización generaría un nodo n_1' copia de n_1. Por tanto este nuevo nodo sería también apuntado por dicha variable pv, obteniendo de esta forma un grafo en donde una misma variable apunta a dos nodos distintos. Esto no cumple las restricciones sobre los RSGs en las que en un rsg cada variable puntero puede apuntar como máximo a un solo nodo. Por este motivo, si el nodo apuntado por $x \rightarrow sel$ es a su vez apuntado directamente por una variable puntero pv, se va a proceder a la materialización de dos grafos. En principio será exactamente iguales, a excepción que uno de ellos, será el que realmente contenga el enlace $x \rightarrow sel$, mientras que el otro no lo contendrá. Una posterior *poda* de los mismos, eliminará enlaces y nodos que pertenecen al otro grafo. De este modo se habrá obtenido un grafo en el que el nodo n_1 (apuntado por $x \rightarrow sel$) representará a todas las porciones realmente apuntadas por dicho enlace, mientras que en el otro grafo, el nodo n_1 representará las porciones compatibles con las anteriores, no apuntadas por $x \rightarrow sel$.

1. Materialización de grafos nuevos

   Por tanto si el nodo referenciado por n_x y sel, n_{pv}, es apuntado directamente por una variable puntero pv, la materialización del nodo se llevará a cabo materializando un nuevo grafo a partir del original. Como resultado se obtienen dos grafos:

   - Uno con el selector sel desde el nodo n_x al nodo apuntado por pv.
   - Otro sin esa referencia.

En ambos grafos se utiliza la información de la existencia o no existencia segura de dicha referencia, para refinar las propiedades de los nodos y obtener así una representación más precisa. Es decir, una vez conocida la existencia segura del selector $<n_x, sel, n_{pv}>$ en uno de los grafos, se pueden eliminar aquellos enlaces que no puedan coexistir con éste, debido a las propiedades del nodo ($SHSEL(n_{pv}, sel)$ =*false*). Por este motivo, tras crear los dos grafos uno con el enlace y otro sin él, los dos nuevos grafos son *podados* con la función $PRUNE$ descrita en el apartado anterior.

Definimos pues $MATERIALIZE\_RSG(rsg, x, sel) = \{rsg_1, rsg_2 \mid rsg_1 = PRUNE(rsg'_1), rsg_2 = PRUNE(rsg'_2)\}$. El grafo rsg'_1 que conserva el enlace $<n_x, sel, n_{pv}>$, queda definido como:

- $N(rsg'_1) = N(rsg)$
- $PL(rsg'_1) = PL(rsg)$
- $NL(rsg'_1) = NL(rsg) \setminus (DL = \{< n_i, sel_j, n_{pv} > |$
  - $[(SHARED(n_{pv}) = 0) \wedge (sel_j \neq sel)] \vee$
  - $[(SHSEL(n_{pv}, sel) = 0) \wedge (n_i \neq n) \wedge (sel_j = sel)]\})$

siendo $n_x, n_{pv} \in N(rsg'_1)$ tal que $<x, n_x > \in PL(rsg'_1), < n_x, sel, n_{pv} > \in NL(rsg'_1)$, y DL el conjunto de enlaces eliminados por no cumplir las propiedades *shared* de n_{pv}.

Además, las siguientes propiedades de los nodos implicados, se ven modificadas de la siguiente manera:

- $SELINset(n_{pv}) = SELINset(n_{pv}) \cup \{sel\}$
- $PosSELINset(n_{pv}) = PosSELINset(n_{pv}) \setminus \{sel\}$
- $SPATH(n_{pv}) = SPATH(n_{pv}) \setminus \{< pv, sel_i > \mid < pv, n_1 > \in PL(rsg'_1) \wedge <n_1, sel_i, n_{pv} > \in DL\}$
- $SELOUTset(n_x) = SELOUTset(n_x) \cup \{sel\}$
- $PosSELOUTset(n_x) = PosSELOUTset(n_x) \setminus \{sel\}$

En rsg_1 representamos las estructuras que realmente poseen el enlace $x \rightarrow sel$, por tanto dejamos la referencia de n_x a n_{pv} por sel. Esto implica que ahora el nodo n_x realmente tenga un enlace de salida por sel y por eso es introducido dicho selector en su conjunto $SELOUTset$ y es eliminado, si existía, de $PosSELOUTset$. De igual forma, en este grafo, el nodo n_{pv} posee realmente un enlace de entrada por sel, de ahí que sea introducido es su propiedad $SELINset(n_{pv})$.

Por otro lado, la información *shared* de n_{pv} es usada para eliminar enlaces que no pueden aparecer junto con el enlace $x \rightarrow sel$ (conjunto DL). Como el nodo n_{pv} es ya referenciado por n_x y el selector sel, si n_{pv} no es $SHARED$ implica que no puede haber ningún otro enlace por un selector distinto a sel, luego estos enlaces no son copiados a rsg_1. Además, si $SHSEL(n_{pv}, sel)$ =*false*, no puede existir ningún otro enlace por el propio sel distinto del que viene de n_x. Además, del conjunto de $SPATH$ de n_{pv} son eliminados aquellos *simple paths* formados a partir de enlaces borrados (conjunto DL).

En cuanto a rsg'_2, el grafo donde no existe $<n_x, sel, n_{pv}>$:

- $N(rsg'_2) = N(rsg)$
- $PL(rsg'_2) = PL(rsg)$

- $NL(rsg_2') = NL(rsg) \setminus \{< n_x, sel, n_{pv} >\}$

siendo de nuevo $n_x, n_{pv} \in N(rsg)$ tal que $<x, n_x> \in PL(rsg), < n_x, sel, n_{pv} > \in NL(rsg)$.

Como vemos en rsg_2 mantenemos todo el grafo original, a excepción del enlace $x \rightarrow sel$ que se ha mantenido en el grafo rsg_1. Ya que se sabe de la no existencia del enlace $<n_x, sel, n_{pv}>$ en dicho grafo, se pueden modificar algunas propiedades de dichos nodos:

- $SELOUTset(n_x) = SELOUTset(n_x) \setminus \{sel \mid (sel \in PosSELOUTset(n_x)) \ \wedge \ (\nexists n_i, < n, sel, n_i > \in NL(rsg_2'))\}$
- $PosSELOUTset(n_x) = PosSELOUTset(n_x) \setminus \{sel \mid (sel \in PosSELOUTset(n_x)) \ \wedge \ (\nexists n_i, < n, sel, n_i > \in NL(rsg_2'))\}$
- $SELINset(n_{pv}) = SELINset(n_{pv}) \setminus \{sel \mid (sel \in PosSELINset(n_{pv})) \ \wedge \ (\nexists n_i, < n_i, sel, n_{pv} > \in NL(rsg_2'))\}$
- $PosSELINset(n_{pv}) = PosSELINset(n_{pv}) \setminus \{sel \mid (sel \in PosSELINset(n_{pv})) \ \wedge \ (\nexists n_i, < n_i, sel, n_{pv} > \in NL(rsg_2'))\}$
- $SPATH(n_{pv}) = SPATH(n_{pv}) \setminus \{< x, sel >\}$
- $SHARED(n_{pv}) = 0$ si $\nexists n_i, n_j \mid \ < n_i, sel_i, n_{pv} >, < n_j, sel_j, n_{pv} > \in NL(rsg_2') \wedge sel_i \neq sel_j$
- $SHSEL(n_{pv}, sel) = 0$ si $[\nexists n_i \in N(rsg_2') | < n_i, sel, n_{pv} > \in NL(rsg_2')] \vee [\exists_1 n_i \in N(rsg_2') | < pvar, n_i > \in PL(rsg_2') \wedge <n_i, sel, n_{pv} > \in NL(rsg_2')]$

Si el nodo n_x tiene el selector sel en su $PosSELOUT$ quiere decir que en dicho nodo puede haber porciones de memoria con y sin enlace de salida por el selector sel. Si en el grafo rsg_2, el nodo n_x ya no posee ningún enlace por sel, quiere decir que en este grafo tan solo se han quedado las porciones que no tenían enlace por sel, con lo que se puede quitar sel tanto de $SELOUTset$ como de $PosSELOUTset$ del nodo (las porciones que si tenía dicho enlace está representadas en el grafo rsg_1).

Lo mismo ocurre con el nodo n_{pv} y su selectores de entrada. Si dicho nodo posee a sel en su $PosSELINset$, quiere decir que representa porciones que pueden o no tener un enlace por el selector sel. Si en rsg_2 no existe ningún otro nodo que referencie a n_{pv} por sel, dicho nodo está representado sólo a aquellas porciones de memoria que no poseían dicho enlace de entrada (las que si lo poseen están representadas en rsg_1). Por este motivo, se puede eliminar sel de los conjuntos $PosSELINset$ y $SELINset$ del nodo.

En cuanto a los *simple paths* de n_{pv}, puesto que se ha eliminado el enlace desde el nodo apuntado por x mediante el selector sel, hay que eliminar $<x, sel>$ de su conjunto $SPATH$.

Por último, la información *shared* del nodo n_{pv} en este grafo, puede verse modificada, puesto que sobre este nodo se ha eliminado el enlace $<n_x, sel, n_{pv}>$. Por un lado, si n_{pv} ya no es referenciado desde dos nodos distintos por distintos selectores, no puede ser *shared* y por tanto $SHARED(n_{pv}) = 0$ (lo será el nodo correspondiente en rsg_1). Por otro lado, si al eliminar el selector $<n_x, sel, n_{pv}>$ ya no queda ningún otro sobre el nodo n_{pv} por el selector sel, dicho nodo no puede ser *shared* por el selector sel, $SHSEL(n_{pv}, sel) = 0$. Tampoco lo puede ser, si sólo existe un enlace por sel desde un nodo apuntado directamente por una variable puntero. Esto es así, ya que el nodo apuntado por una variable, está representado las porciones de memoria referenciadas por dicha variable

puntero. Por tanto, el nodo está representando porciones de memoria pertenecientes a distintas configuraciones de memoria, puesto que en una misma configuración una variable tan solo puede apuntar a una porción de memoria. Eso hace que el enlace desde dicho nodo, represente un único enlace en cada configuración de memoria, y por tanto las porciones de memoria representadas por n_{pv} no pueden ser apuntadas por más de un enlace sel.

Resumiendo, cuando tratamos de materializar un nodo apuntado por $x \rightarrow sel$ desde otro nodo que es apuntado por una variable puntero, creamos dos grafos rsg_1 y rsg_2, el los que el primero contiene el nodo materializado junto con el enlace, y el segundo representa las posibles estructuras que no contenían dicho enlace.

2. Materialización de nodos

   Como hemos visto, cuando $x \rightarrow sel$ referencia a un nodo n apuntado directamente por una variable puntero, se divide dicho grafo en dos, uno que posee el enlace y otro que no. Sin embargo, si el nodo referenciado n, no es apuntado por ninguna variable puntero, sigue siendo necesario una materialización de un nodo nuevo. Esta materialización va a separar en dos nodos las porciones de memoria representadas por n. El nuevo nodo materializado, n_m, va a representar aquellas porciones de memoria realmente referenciadas por $x \rightarrow sel$, mientras que el nodo n se va a modificar para que sólo represente las porciones restantes (no apuntadas por $x \rightarrow sel$ pero compatibles con estas).

   Al igual que con la materialización de los grafos, es posible un refinamiento de las propiedades de los nodos (uno va a tener definitivamente el enlace y el otro no) para así obtener una representación más exacta de las estructuras. De esta forma, puesto que cambian las propiedades de algunos nodos y el conjunto de enlaces se ve modificado, se aplica una *poda* del grafo modificado para así obtener esta representación más exacta.

   Definimos ahora la función $MATERIALIZE\_NODE(rsg, x, sel) = rsg_m$, con $rsg_m = PRUNE(rsg'_m)$, que genera un nuevo grafo rsg_m a partir del grafo original rsg, materializando un nodo desde el nodo destino de $x \rightarrow sel$. Sean $n_x, n \in N(rsg)$ tal que $<x, n_x> \in PL(rsg), < n_x, sel, n > \in NL(rsg)$, definimos entonces rsg'_m como:

   - $N(rsg'_m) = N(rsg'_m) \cup \{n_m\}$ donde
     - $TYPE(n_m) = TYPE(n)$
     - $STRUCT(n_m) = STRUCT(n)$
     - $SELINset(n_m) = SELINset(n) \cup \{sel\}$
     - $PosSELINset(n_m) = PosSELINset(n) \setminus \{sel\}$
     - $SELOUTset(n_m) = SELOUTset(n)$
     - $PosSELOUTset(n_m) = PosSELOUTset(n)$
     - $SHARED(n_m) = SHARED(n)$
     - $SHSEL(n_m, sel_i) = SHSEL(n, sel_i) \forall sel_i \in S$
     - $TOUCH(n_m) = TOUCH(n)$
     - $CYCLELINKS(n_m) = CYCLELINS(n)$
   - $PL(rsg'_m) = PL(rsg)$
   - $NL(rsg'_m) = NL(rsg) \ \cup \ \{< n_x, sel, n_m >\}$
     - $\cup \{n_m, sel_i, n_j > \mid < n, sel_i, n_j > \in NL(rsg'_m)\}$

- $\cup (AL = \{< n_i, sel_j, n_m > | < n_i, sel_j, n > \in NL(rsg'_m) \wedge$
  $[(SHARED(n_m) = 1 \ \wedge \ sel_j \neq sel) \vee$
  $(SHSEL(n_m, sel) = 1 \ \wedge \ sel_j = sel)]\})$
- $\cup \{< n_m, sel_i, n_m > | < n, sel_i, n > \in NL(rsg'_m) \wedge$
  $[(SHSEL(n_m, sel) = 1 \ \wedge \ sel_i = sel) \vee$
  $(SHARED(n_m) = 1 \ \wedge \ sel_i \neq sel]\}$
- $\setminus \{< n_x, sel, n >\}$

donde el conjunto AL, es el conjunto de enlaces que apuntan al nuevo nodo materializado n_m, desde otros nodos, y que, como veremos más adelante, se utiliza para construir el conjunto de *SPATH* del nuevo nodo.

A simple vista puede parecer algo compleja la construcción del nuevo grafo, rsg'_m, pero si nos fijamos paso a paso como se construye veremos que no lo es tanto.

Primero vemos que el conjunto de nodos de rsg'_m es igual al del grafo original, añadiendo uno nuevo, n_m que es el nodo materializado a partir de n (nodo apuntado por $x \rightarrow sel$). Por tanto, dicho nodo nuevo n_m es una copia de n (se copian todas sus propiedades) a excepción de sus *reference patterns*. La diferencia es que, puesto que n_m va a representar las porciones de memoria apuntadas por $x \rightarrow sel$, dicho nodo definitivamente tiene un enlace por el selector sel, por tanto introducimos sel en su conjunto *SELINset* y lo eliminamos, si existía, de *PosSELINset*.

En cuanto al conjunto de enlaces entre nodos $NL(rsg'_m)$, debe ser igual al de rsg, añadiendo todos los enlaces que posee el nodo n al nuevo nodo n_m que respeten las propiedades de este último. Así, se añade el enlace $<n_x, sel, n_m>$ puesto que ahora n_x sólo va a apuntar al nodo materializado n_m por sel. Además, el nodo n_m tendrá exactamente los mismos enlaces de salida que n. A la hora de copiar los enlaces de entrada de n en n_m tenemos que ver si son compatibles con el introducido anteriormente ($<n_x, sel, n_m>$). Así, si el nodo no es *SHARED* ni es *SHSEL* por ningún selector, no se copia ninguno de los selectores de entrada de n puesto que no serían compatibles con $<n_x, sel, n_m>$. Sin embargo, si el nodo es *shared*, algunos de los enlaces de entrada a n pueden ser introducidos sobre n_m. Por último, hay que eliminar el enlace $<n_x, sel, n>$, ya que ahora sólo existe sobre el nuevo nodo materializado n_m.

Como ejemplo, podemos ver de nuevo, la materialización de un nodo en el grafo que representa una lista doblemente enlazada, para la ejecución de la sentencia $x \rightarrow nxt = NULL$. En la figura 3.20 podemos ver el grafo antes (rsg_1) y después (rsg_2) de la materialización.

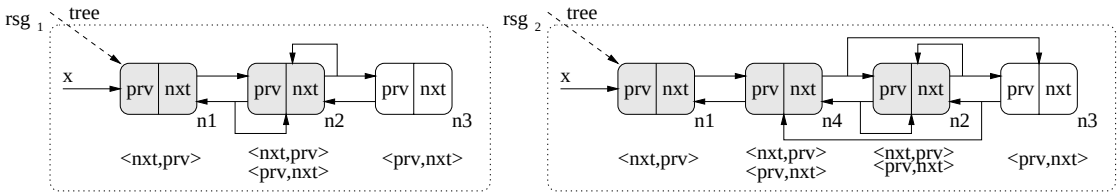


Figura 3.20: Materialización de un nuevo nodo.

Como se puede observar, el nodo apuntado por $x \rightarrow nxt$ en rsg_1 es n_2, y por tanto desde este nodo se va a materializar uno nuevo (n_4 de rsg_2) para representar las porciones de memoria representadas por n_2 que realmente son apuntadas por $x \rightarrow nxt$. Antes

de continuar, hay que decir que todos los nodos tienen la información *SHSEL* =*false* para todos los selectores, y que los nodos sombreados tienen *SHARED* =*true*. Como se a descrito al presentar la función $MATERIALIZE\_NODE$, el nuevo nodo n_4 es copia exacta del nodo original n_2. En cuanto a los enlaces, lo fundamental es que el nodo n_1 (n_x en la presentación de la función) ahora no referencia a n_2 por nxt, sino que sólo lo hace sobre n_4. Ahora, hay que copiar todos los enlaces de n_2 (tanto de salida como de entrada al nodo) sobre su copia, n_4, pero sólo aquellos que no entran en conflicto con el selector fijo $<n_1, nxt, n_4>$ y las propiedades del nodo n_4. Los enlaces de salida del nodo n_2 son copiados íntegramente al nodo n_4 ($<n_4, nxt, n_3>, <n_4, nxt, n_2>, <n_4, prv, n_1>, <n_4, prv, n_2>$). Como se puede observar, de los mencionados anteriormente, el enlace $<n_4, prv, n_2>$ no aparece en el grafo rsg_2, aunque por la materialización debería aparecer. Este enlace es eliminado en la *poda* del grafo puesto que no cumple en *cyclelink* de n_4, $<prv, nxt>$.

En cuanto a los enlaces de entrada a n_2, no todos son copiados sobre n_4, pues algunos no cumplen las restricciones impuestas por la propiedades. Los enlaces $<n_3, prv, n_2>$ y $<n_2, prv, n_2>$ son trasladados a n_4 ($<n_3, prv, n_4>$ y $<n_2, prv, n_4>$) puesto que no incumplen las propiedades del nodo. Sin embargo, hay un enlace sobre n_2, $<n_2, nxt, n_2>$ que no es trasladado a n_4, ya que como $SHSEL(n_4, nxt)$ =*false* y es segura la existencia de $<n_1, nxt, n_4>$, no puede existir ningún otro enlace por nxt sobre el nodo n_4. Aquí se ve claramente como la información almacenada en los nodos, más concretamente la información *shared* en este caso, hace que la transformación del grafo sea mucho más fiel a las estructuras que está representando. Si el nodo n_4 tuviera $SHSEL(n_4, nxt)$ =*true*, el enlace $<n_2, nxt, n_2>$, provocaría la aparición de dos enlaces en n_4, $<n_2, nxt, n_4>$ y $<n_4, nxt, n_4>$.

Aparte de la generación del nuevo grafo con el nodo materializado, rsg'_m, algunas de las propiedades de los nodos original n y materializado n_m, de dicho grafo, se pueden modificar, para amoldarse mejor a la nueva situación (n_m representa sólo a porciones que son referenciadas por $x \rightarrow sel$).

Las nuevas propiedades de n son:

- $SELINset(n) = SELINset(n) \setminus \{sel\}$ si
  $(sel \in PosSELINset(n)) \ \wedge \ (\nexists n_i, < n_i, sel, n > \in NL(rsg'_m))$
- $PosSELINset(n) = PosSELINset(n) \setminus \{sel\}$ si
  $(sel \in PosSELINset(n)) \ \wedge \ (\nexists n_i, < n_i, sel, n > \in NL(rsg'_m))$
- $SPATH(n) = SPATH(n) \setminus \{< x, sel >\}$
- $SHARED(n) = 0$ si $\nexists n_i, n_j \mid \ < n_i, sel_i, n >, < n_j, sel_j, n > \in NL(rsg'_m) \wedge sel_i \neq sel_j$
- $SHSEL(n, sel) = 0$ si $[\nexists n_i \in N(rsg'_m) | \ < n_i, sel, n > \in NL(rsg'_m)] \vee [\exists_1 n_i \in N(rsg'_m) | < pvar, n_i > \in PL(rsg'_m) \wedge < n_i, sel, n > \in NL(rsg'_m)]$

Como ocurría con la materialización de un nuevo grafo, el nodo n que ya no es referenciado por $x \rightarrow sel$, si no es apuntado por ningún otro selector sel entonces se elimina dicho selector de sus conjuntos *SELINset* y *PosSELINset*. De igual forma, si al eliminar dicho enlace por sel ya no es apuntado por distintos selectores, o no hay más de un nodo (apuntado por una variable puntero) referenciándolo por sel, se inicializa su información *shared*. De su conjunto *SPATH* es eliminado $<x, sel>$ ya que se ha eliminado el enlace que unía n_x y n por el selector sel.

En cuanto a las propiedades de n_m:

- $SPATH(n_m) = \{< x, sel >\} \cup \{< pv, sel_j > | < pv, n_i > \in PL(rsg'_m) \wedge < n_i, sel_j, n_m > \in AL\}$

El conjunto de *simple paths* del nuevo nodo, está formado por $<x, sel>$ puesto que ahora el nodo n_m es el único apuntado por sel desde n_x, y por aquellos *simple paths* correspondientes a los nuevos enlaces introducidos sobre el nodo n_m (conjunto AL).

De esta manera queda definida la función $MATERIALIZE\_NODE$, la cual, como hemos visto, genera un nuevo grafo copia del original, que tiene un nuevo nodo igual al nodo que era apuntado por $x \rightarrow sel$, y que representa sólo a aquellas porciones de memoria realmente referenciadas por dicho enlace. Las restantes quedarán representadas por el original.

Con la definición de estas dos funciones, hemos visto como separar o bien en dos grafos distintos o bien en dos nodos distintos, las porciones de memoria realmente referenciadas por $x \rightarrow sel$ de aquellas, compatibles con estas, que no lo son. La idea, recordamos, es que a la hora de modificarlas, según indique la semántica abstracta de la sentencia, sólo se cambien las propiedades y enlaces de las porciones de memoria correctas, siendo lo menos conservativo que se pueda. Esta semántica abstracta asociada a las distintas sentencias que operan con punteros se presenta en el apéndice B.

3.5 Resultados experimentales

Hemos implementado el método basado en RSRSGs en un *pseudocompilador* que analiza código C, y devuelve para cada sentencia del código, el RSRSG asociado a la misma, que describe todas las posibles configuraciones de memoria que pueden aparecer tras la ejecución de dicha sentencia.

Antes de pasar a comentar los códigos analizados y los resultados obtenidos para cada uno de ellos, vamos a presentar una modificación de la manera de analizar el código que hemos denominado *Análisis Progresivo*.

3.5.1 Análisis Progresivo

Como hemos comentado en la presentación de este método, la manera de mantener en nodos separados, porciones de memoria con características distintas, es asociar un conjunto de propiedades a las porciones de memoria y a los nodos, de manera que un nodo tan solo puede representar pociones de memoria de propiedades similares. Por tanto, el número de nodos que pueden aparecer en un grafo depende del número de propiedades y del rango de valores que puede tomar cada propiedad. Cuantas más propiedades se usen, más combinaciones de valores de las mismas pueden aparecer, lo que implica un mayor número de nodos. Igual ocurre si el rango de una propiedad aumenta, también aumenta el número distinto de combinaciones de propiedades aumentando el número distinto de nodos.

Por otro lado, las propiedades se han ido introduciendo para que el método sea capaz de aproximar con mayor exactitud las estructuras dinámicas que pueden aparecer en la ejecución de los códigos. Por tanto la exactitud en la descripción aumenta conforme aumentan las propiedades y su rango, pero también aumentan mucho los requerimientos de memoria y

tiempo para el análisis. Si hay más nodos por grafo, hay más posibilidades de grafos distintos, lo que aumenta el número de grafos a analizar, aumentando el tiempo de análisis.

Hemos comprobado que no siempre son necesarias todas las propiedades para alcanzar una descripción precisa de las estructuras en todos los códigos. Códigos simples pueden ser analizados utilizando un menor número de propiedades, mientras que para códigos más complejos se necesita la utilización de todas ellas. Por este motivo, hemos implementado el método para que lleve a cabo un *análisis progresivo* de los códigos, empezando con menos restricciones a la hora de sumarizar nodos y aumentándolas cuando sea necesario para obtener una aproximación más exacta de las configuraciones de memoria.

Hemos creado tres niveles de análisis, L_1, L_2 y L_3, cuya diferencia es la utilización de más propiedades o de propiedades más complejas a medida que se aumenta el nivel.

A continuación presentamos las características de los tres niveles:

- L_1: Este nivel implementa el método basado en los RSRSGs presentado en este capítulo, sólo que con las siguientes restricciones:

  - La propiedad *TOUCH* no es utilizada, es decir, ni se construye, ni es tenida en cuenta por la semántica abstracta de las sentencias. Donde radica la importancia de la no utilización de esta propiedad es en las funciones *C_NODES_RSG* (sección 3.2.10) y *C_NODES* (sección 3.3.1). Estas funciones determinan cuando dos nodos (de un mismo grafo o de grafos distintos respectivamente) tienen propiedades similares para poder ser sumarizados en uno solo. La variación en dichas funciones es la no comparación de la información *TOUCH* de los nodos, por tanto esta información no influirá a la hora de sumarizar los nodos.
  - Sólo se tienen en cuenta en la propiedad *SPATH*, los caminos de longitud cero, es decir, lo que denominábamos *SPATH*0. Con esto se consigue que a la hora de comparar nodos para ver si pueden ser o no sumarizados, en referencia a los caminos de acceso a las porciones de memoria, tan solo se tiene en cuenta si el nodo es o no apuntado por variables puntero. Este comportamiento se consigue utilizando como parámetro k el valor 0 en la función *C_SPATH* utilizada en *C_NODES_RSG* y *C_NODES*.

  Como podemos observar, en este nivel se ha reducido el número de propiedades a comparar (no se utiliza *TOUCH*) y se ha reducido el número de posibles valores que puede tomar otra propiedad (en *SPATH* sólo pueden aparecer caminos de longitud cero). Esto provocará un mayor número de sumarizaciones, reduciendo el tamaño y número de grafos durante el análisis, a costa de mezclar más configuraciones de memoria en un solo grafo.

- L_2: Este nivel se basa en el anterior pero usando dentro de la propiedad *SPATH* de los nodos, también los caminos de longitud uno. Ahora, cuando se comparen nodos se tendrá en cuenta no sólo si son apuntados por variables puntero, sino si están "cerca" de nodos apuntados por variables puntero. Para conseguir este comportamiento basta con dar el valor 1 al parámetro k de las función *C_SPATH*.

- L_3: Este nivel es el más complejo pues va a tener en cuenta todas las propiedades y todos sus posibles valores. Es como el nivel L_2, pero añadiendo la comparación de la propiedad *TOUCH* a las funciones *C_NODES_RSG* y *C_NODES*.

	Tiempo			Memoria (MB)		
Nivel	L_1	L_2	L_3	L_1	L_2	L_3
Código Ejemplo	0'09"	0'15"	0'16"	2.11	2.78	3.02
Matriz × Vector	0'03"	0'05"	0'07"	1.37	1.85	2.17
Matriz × Matriz	0'51"	1'36"	1'57"	8.13	11.45	12.68
Factorización LU	12'51"	-	-	99.46	-	-
Barnes-Hut	17'01"	1'47"	3'21"	44.46	12.44	21.31

Tabla 3.2: Requerimientos de tiempo y memoria para el análisis de los códigos.

El proceso de análisis empezaría por el nivel más bajo L_1 utilizando menos recursos (tiempo y memoria). Si los resultados obtenidos no proporcionan información suficiente para la optimización del código, se pasaría a analizar el código en el nivel L_2, que necesitará más espacio y memoria que el anterior. Si aún así los resultados obtenidos no son satisfactorios se pasaría al nivel L_3 de análisis, donde toda la potencia del método es usada para la determinación de la forma de las estructuras dinámicas de datos.

Con el compilador implementado se han analizado varios códigos C que crean, modifican y recorren estructuras de datos dinámicas. Los códigos analizados son: el código que genera la estructura ejemplo presentada en la sección 3.2.1, un código que multiplica una matriz dispersa por un vector, otro que multiplica dos matrices dispersas, la factorización LU de una matriz dispersa, y el núcleo del algoritmo de simulación Barnes-Hut.

En la tabla 3.2 se muestran el tiempo y memoria consumidos por el compilador para analizar en cada uno de los niveles de análisis, los cinco códigos que son presentados en el apéndice E.1. Los códigos han sido analizados en un Pentium III a 500 MHz con 128 MB de memoria. Los cuatro primeros códigos se analizaron correctamente en el nivel L_1, aunque en la tabla se presentan los requerimientos para los tres niveles del análisis. El código Barnes-Hut ha obtenido la representación más exacta de la estructura de datos en el nivel L_3, como se verá más adelante. Para la factorización LU tan solo se muestran los resultados del nivel $L1$ puesto que para los otros dos niveles, el compilador agotaba los 128 MB de memoria. Estos niveles no son necesarios para dicho código puesto que en L_1 se obtiene una representación correcta de la estructura de datos.

A continuación presentamos las peculiaridades de cada uno de los códigos.

3.5.2 Código ejemplo

El código analizado, que se presenta en el apéndice E.1.1, genera, recorre y modifica la estructura de datos presentada en la figura 3.21. Como se puede observar, la estructura creada consta de tres partes diferenciadas. Una lista cabecera doblemente enlazada por los selectores *nxt* y *prv*. Cada elemento de esta lisa apunta a un árbol binario distinto por medio del selector *tree*. Las hojas de estos árboles apuntan con el selector *list* a distintas listas, también enlazadas por los selectores *nxt* y *prv*.

Aparte de crear y recorrer dicha estructura, el código realiza la permutación de los árboles apuntados por dos elementos de la lista cabecera, como ejemplo de modificación de la estructura una vez creada. Dicha modificación no cambia la forma general de la estructura de datos.

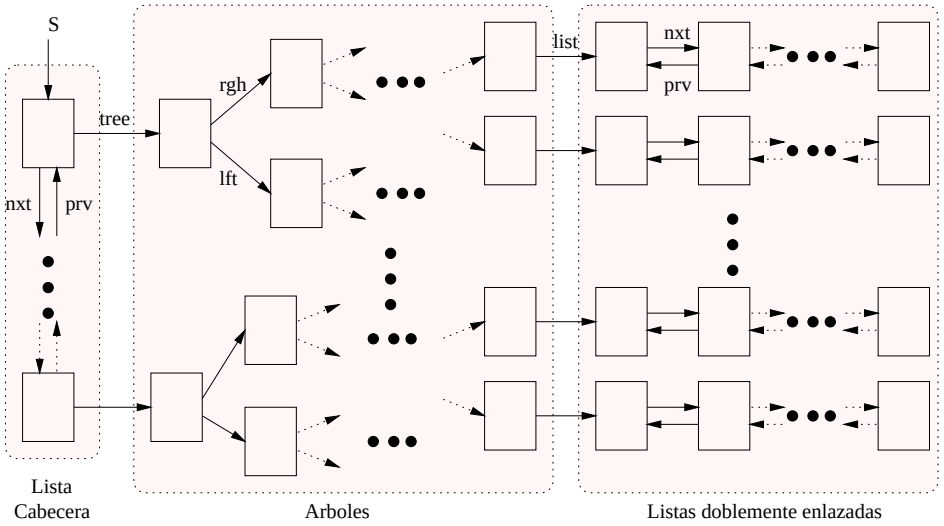


Figura 3.21: Ejemplo de estructura de datos dinámica compleja.

En la figura 3.22 se muestra una representación compacta del RSRSG obtenido para la última sentencia del código. Este grafo debe representar todas las posibles formas de las configuraciones de memoria tras la ejecución completa del código (creación, recorrido y modificación de la estructura). En definitiva, de este grafo se pueden deducir las características principales de la forma de la estructura de datos creada en el código y presentada en la figura 3.21. Se puede observar como los nodos n_1, n_2 y n_3 representan el primer elemento, los elementos centrales y el último elemento respectivamente de la lista cabecera. Los nodos n_4, n_5 y n_6 a la raíz, los elementos centrales y las hojas respectivamente de los árboles. Y por último los nodos n_7, n_8 y n_9 al primer elemento, los centrales y el último respectivamente de las listas apuntadas desde las hojas de los árboles.

Tenemos que recordar que los nodos sombreados representan nodos con su propiedad $SHARED = true$. Aparte, por cuestiones de claridad, en el grafo no aparece ninguna otra propiedad. Para su correcta interpretación, basta con añadir que la propiedad $SHSEL(n, sel)$ es *false* para todos los nodos y para todos los selectores.

Vamos a ver como es posible extraer las características sobre la forma de la estructura de datos a partir de dicho grafo:

- El puntero S apunta a una estructura acíclica (lista cabecera) si es recorrida utilizando tan solo un selector (nxt o prv) puesto que los nodos n_1, n_2 y n_3 no son *shared* por ninguno de dichos selectores. Esto implica que las porciones de memoria representados por ellos no son apuntadas por más de un selector nxt o prv y por tanto es imposible llegar dos veces a la misma porción en un mismo recorrido. El nodo n_2 es $SHARED$ puesto que representa porciones de memoria que son apuntadas a la vez por los selectores nxt (desde el elemento que le precede) y prv (desde el elemento siguiente).

- Los elementos de esta lista cabecera, apuntan a otro tipo de estructura también acíclica (árboles), puesto que los nodos no son $SHSEL$ por ningún selector. Es decir, siguiendo un camino cualquiera de selectores lft y rgh, nunca se va a visitar dos veces la misma porción de memoria. Por el mismo motivo, distintos árboles no comparten ningún elemento en común. Es decir, ya que a ninguna porción de memoria se puede acceder

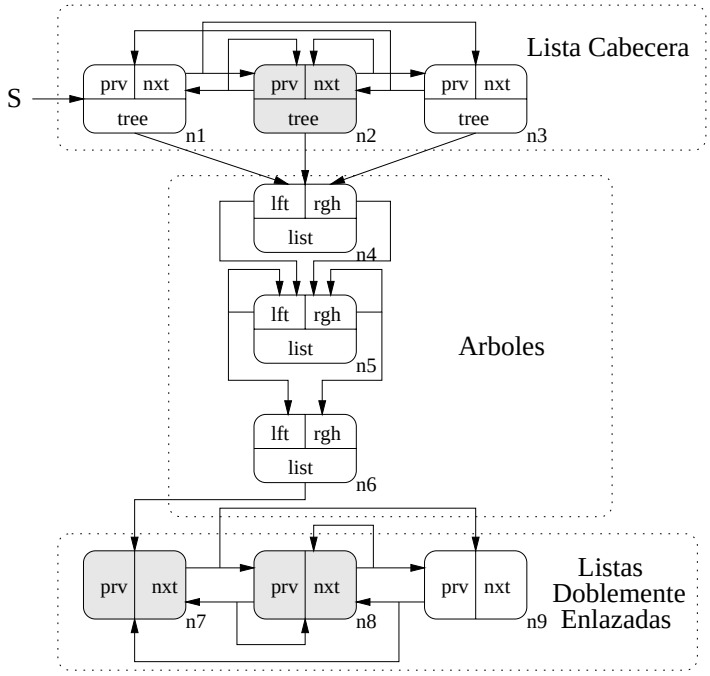


Figura 3.22: Representación simplificada del RSRSG obtenido para la estructura ejemplo.

desde dos distintas porciones, cada porción de memoria puede pertenecer tan solo a un árbol. Y por último, dos elementos distintos de la lista cabecera tienen que apuntar a dos árboles distintos, puesto que el nodo n_4 que representa las raíces de los árboles, no es *shared* por el selector *tree*.

- Por último, las estructuras apuntadas desde las hojas de los árboles (nodo n_6), tienen las mismas características que la lista cabecera, por lo que se puede deducir que son acíclicas si se recorren por un solo selector (*nxt* y *prv*). Además, aunque el nodo n_7, que representa a los primeros elementos de dichas listas, es *SHARED* (puesto que es apuntado por *tree* desde las hojas de los árboles y por *prv* desde el siguiente elemento de la lista), distintas hojas del mismo árbol o de distintos, no apuntan a la misma lista, puesto que $SHSEL(n_7, list) = false$. Al igual que pasaba con los árboles, puesto que ningún nodo tiene $SHSEL = true$ para ningún selector, distintas listas no pueden compartir elementos (un porción de memoria no puede ser accedida desde porciones pertenecientes a dos listas).

Vemos como se pueden extraer la características fundamentales de la forma de la estructura de datos. Esta información es muy útil, puesto que, por ejemplo en un recorrido de la lista cabecera en un bucle, utilizando el selector *nxt*, se puede saber que las porciones accedidas en cada paso del bucle son distintas de las accedidas en otro paso, puesto que la lista cabecera es acíclica si se utiliza sólo un selector (*nxt* en este caso) y las estructuras a las que se accede desde cada elemento de dicha lista no comparten elementos entre si. Este tipo de información puede ser tenida en cuenta a la hora de decidir si es posible la ejecución en paralelo de distintas iteraciones del bucle.

3.5.3 Multiplicación Matriz dispersa por Vector

Este código realiza la multiplicación de una matriz dispersa por un vector, $r = M \times v$. El código es presentado en el apéndice E.1.3. La matriz dispersa, M, se almacena en memoria usando la estructura *LLRS* (Linked List Row Storage), es decir, como una lista cabecera doblemente enlazada con punteros a otras listas doblemente enlazadas que contienen los elementos no nulos de las filas de la matriz. Los vectores dispersos, v y r, también son almacenados en listas doblemente enlazadas. La forma de las estructuras de datos se presenta en la figura 3.23.

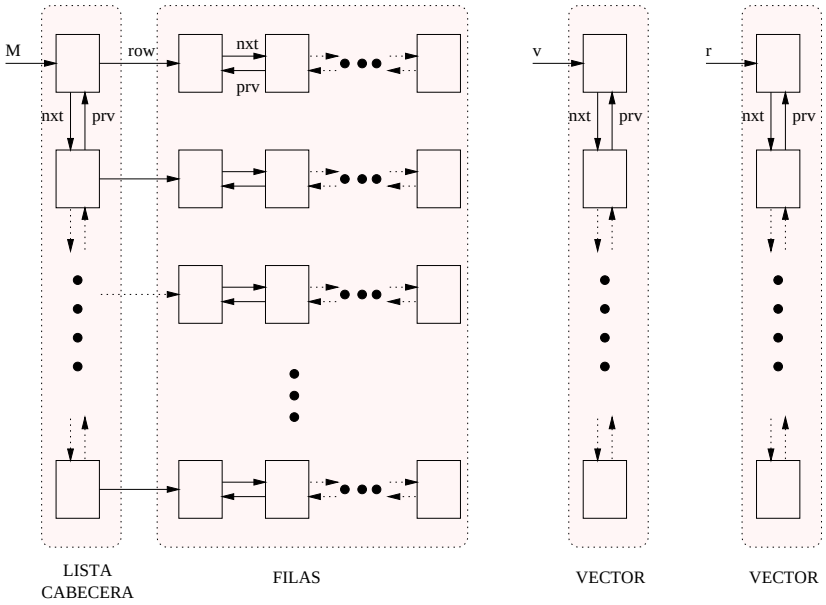


Figura 3.23: Estructuras del código matriz dispersa por vector.

En la figura 3.24 podemos ver una representación compacta del RSRSG obtenido para la última sentencia del código.

Podemos observar como en dicho RSRSG son distinguibles las tres estructuras apuntadas por M, v y r, puesto que son mantenidas en nodos separados. Esto es debido a la propiedad *STRUCTURE* que evita que porciones de las distintas estructuras sean representadas en el mismo nodo, puesto que pertenecen a componentes conexas distintas.

De nuevo, todos los nodos del grafo tienen la propiedad *SHSEL* =*false* para todos los selectores. De la misma forma en que se han deducido las características de la estructura de datos ejemplo a partir de su RSRSG, podemos concluir desde este grafo, que las listas doblemente enlazadas que aparecen en el código (las que representan la lista cabecera, las columnas y los vectores) son acíclicas si son recorridas siguiendo tan solo un selector (*nxt* o *prv*). Además, distintos elementos de la lista cabecera de la matriz M apuntarán a filas distintas, que no comparten elementos entre sí, puesto que el primer nodo de las listas que representan las filas de la matriz no es *shared* por el selector *row*.

El RSRSG obtenido para la sentencia justo antes de empezar la multiplicación, es igual al presentado en la figura 3.24 excepto que el subgrafo apuntado por r no aparece, puesto que dicho vector es creado durante la multiplicación.

De las propiedades deducidas del RSRSG, sería factible la paralelización de la multiplica-

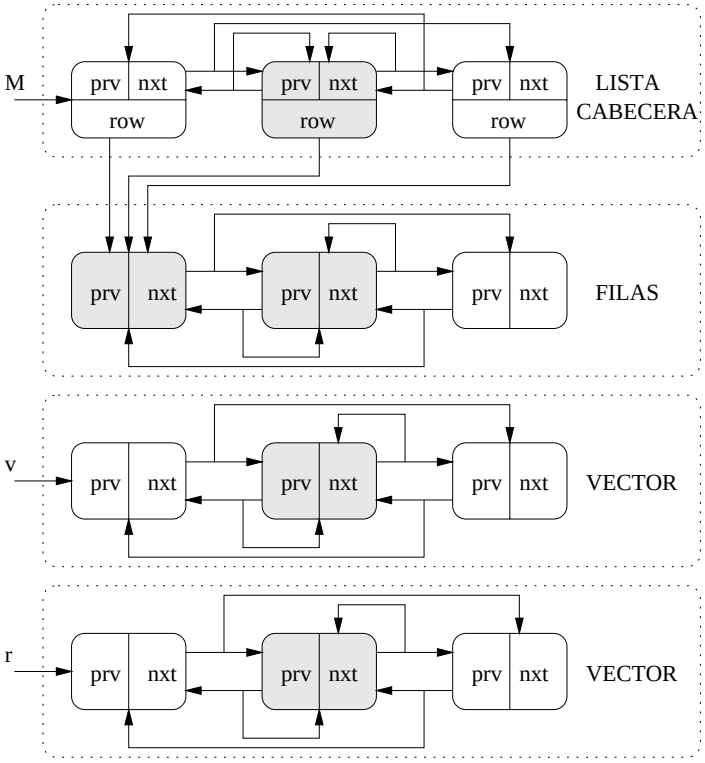


Figura 3.24: Representación simplificada del RSRSG obtenido para el código matriz dispersa por vector.

ción, puesto que se puede deducir que las porciones de memoria accedidas desde cada elemento de la lista cabecera son distintas de las accedidas desde otro elemento de dicha lista, es decir, se ha deducido que las filas de la matriz son independientes.

3.5.4 Multiplicación Matriz dispersa por Matriz dispersa

En este código se implementa la multiplicación de dos matrices dispersas, $C = A \times B$, representadas mediante listas doblemente enlazadas exactamente iguales a la matriz M del código anterior. En la figura 3.25 se presenta la representación de las estructuras al final del proceso de multiplicación, y en el apéndice E.1.5 se presenta el código.

Una representación simplificada del RSRSG obtenido para la última sentencia del código se presenta en la figura 3.26, donde tan solo aparece la matriz apuntada por la variable A. Las estructuras apuntadas por B y C son iguales a la apuntada por A.

Las estructuras apuntadas por las variables A, B y C son exactamente iguales y coinciden con la apuntada por M en el código anterior, por lo que se pueden deducir las mismas propiedades que las deducidas para esta. En definitiva podemos deducir que las estructuras apuntadas desde los elementos de las listas cabecera de las matrices, no comparten ningún elemento entre ellas, y que dos elementos distintos de estas lista cabecera no pueden apuntar a la misma fila/columna de la matriz. De nuevo del RSRSG se puede deducir una información muy valiosa sobre la forma de la estructura de datos utilizadas en el código.

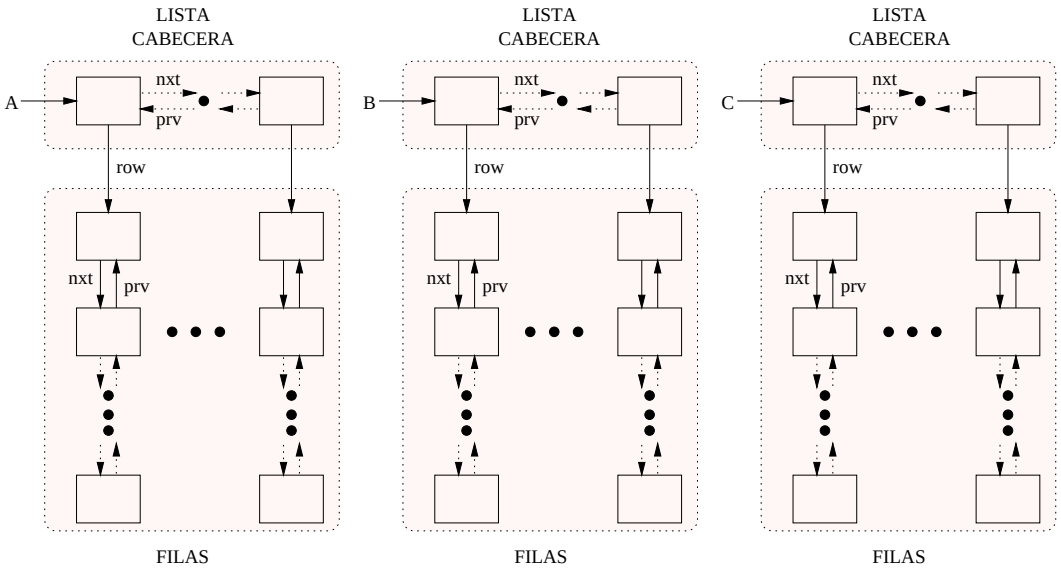


Figura 3.25: Estructuras del código matriz dispersa por matriz dispersa.

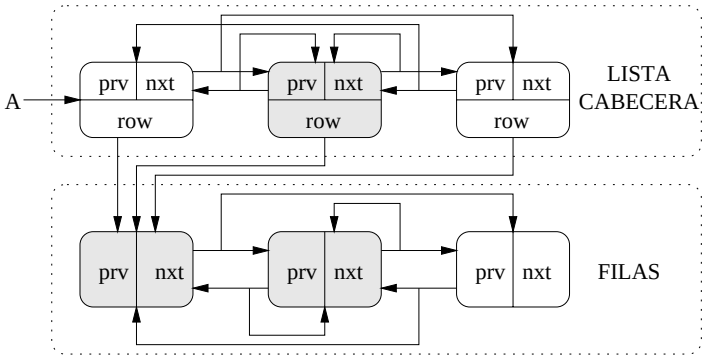


Figura 3.26: Representación simplificada del RSRSG obtenido para el código matriz dispersa por matriz dispersa.

3.5.5 Factorización LU de una matriz dispersa

El núcleo de muchas aplicaciones científicas consiste en la resolución de grandes sistemas lineales dispersos. Podemos encontrar ejemplos de este tipo de aplicaciones en problemas de optimización, programación lineal, simulación, análisis de circuitos, dinámica de fluidos y en general, solución numérica de ecuaciones diferenciales.

Por tanto, este código representa un buen caso de estudio y es un código computacional representativo de muchos otros problemas irregulares. Más precisamente, el problema representa a aquellos en los que la carga computacional crece con el tiempo de ejecución (*fill in*) y los coeficientes de la matriz cambian sus coordenadas debido a permutaciones de columnas (*pivoting*).

En concreto, el código analizado y que se presenta en el apéndice E.1.7, lleva a cavo la factorización LU de la matriz dispersa, usando un método general [1][24]. Estos métodos resuelven directamente el problema disperso y comparten la misma estructura del bucle que el código denso.

```
do k = 1, n
    Busca pivot=A_kj
    if (j ≠ k)
        intercambia A(1 : n, k) y A(1 : n, j)
    endif
    A(k + 1 : n, k) = A(k + 1 : n, k)/A(k, k)
    do j = k + 1, n
        do i = k + 1, n
            A(i, j) = A(i, j) − A(i, k)A(k, j)
        enddo
    enddo
enddo
```

Figura 3.27: Algoritmo LU (Aproximación general, versión "right-looking")

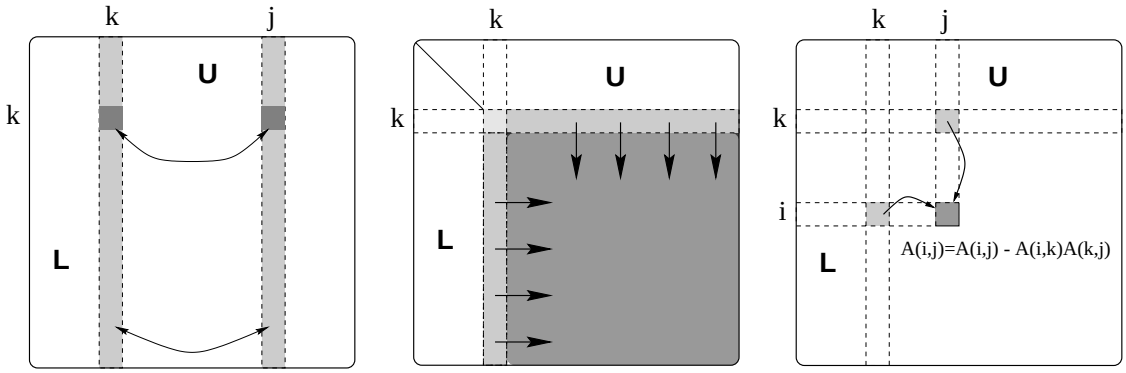


Figura 3.28: Operaciones de "pivoting" (a) y actualización (b), y "fill-in" (c) en "right-looking LU"

La figura 3.27 muestra el código para el algoritmo LU "right-looking", donde una matriz $n \times n$, A, es factorizada. El código incluye el intercambio de columnas lo que proporciona estabilidad numérica y conserva la dispersión en la matriz. En la figura 3.28 aparecen los patrones para las operaciones de intercambio y actualización, junto con la generación de nuevas entradas. Como se puede observar, sólo se necesita un acceso eficiente por columnas.

Normalmente, para ahorrar memoria y tiempo de computación, las entradas cero de las matrices dispersas no son almacenadas explícitamente. Se han desarrollado una amplia variedad de métodos para almacenar las entradas distintas de cero [5][23], pero aquí tan solo vamos a considerar la basada en listas enlazadas.

De nuevo la matriz factorizada, A, es almacenada usando una estructura *LLCS*: una lista cabecera doblemente enlazada con punteros a otras listas doblemente enlazadas que almacenan los elementos distintos de cero de las columnas de la matriz. En la figura 3.29(a) se presenta la estructura de datos usada en el código, y en la figura 3.29(b) una representación simplificada del RSRSG obtenido para la última sentencia del código.

El mismo grafo es obtenido justo antes de que comience el bucle que realiza la factorización de la matriz, lo que implica que la forma de la estructura apuntada por A no cambia durante la factorización, aunque durante dicha factorización se insertan elementos en las columnas y se permutan columnas entre elementos de la lista cabecera.

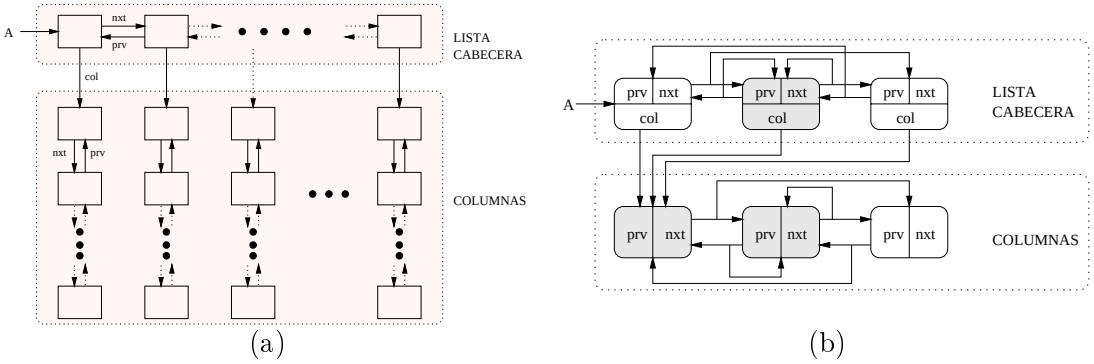


Figura 3.29: Representación simplificada del RSRSG obtenido para el código matriz dispersa por matriz dispersa.

Este código, aunque maneja el mismo tipo de estructura que el de los dos códigos anteriores, es mucho más complejo, puesto que consta de hasta tres bucles anidados y dentro de ellos existen muchas estructuras de control tipo IF en las que se deciden acciones como la permutación de columnas, la inserción de un nuevo elemento al final o en mitad de una columna, etc. Esta complejidad en el código, implica la existencia de muchos más caminos en el grafo de flujo de control que en los códigos anteriores. Puesto que nuestro análisis en tiempo de compilación no puede determinar que caminos de control se van a seguir en tiempo de ejecución, se tienen que analizar todos los posibles. Esto provoca que existan muchas más posibles configuraciones de memoria a analizar por cada sentencia, aumentando considerablemente el tamaño de los conjuntos RSRSGs asociados a cada sentencia. Además, este código utiliza muchas más variables puntero para llevar a cabo sus operaciones, lo que en el análisis implica un mayor número de nodos por grafo (puesto que hay más posibles valores para sus conjuntos *SPATH*), lo que a su vez implica un mayor número de grafos por RSRSG, debido a que hay un mayor número de nodos que tienen que ser compatibles para que dos grafos sean fundidos en uno solo.

La consecuencia inmediata de este aumento de complejidad en el análisis es que para el análisis de este código se necesita más tiempo y memoria que para el análisis de los anteriores. De ahí las diferencias mostradas en la tabla 3.2.

En cuanto a la información deducible del grafo, poco se puede añadir a lo comentado para los dos códigos anteriores. La variable A apunta a una lista doblemente enlazada acíclica si se utiliza sólo un selector (como efectivamente ocurre en el código, en el que sólo se utiliza nxt para avanzar en la lista cabecera). Cada elemento de esta lista cabecera apunta a otra lista acíclica distinta de cualquier otro elemento de la lista cabecera, lo que implica una independencia entre las columnas de la matriz. Un posterior análisis del código y de los grafos asociados a las sentencia, podría determinar la posibilidad de actualización en paralelo de las columnas de la matriz, e incluso la posibilidad de actualización de varias columnas en paralelo.

3.5.6 Simulación Barnes-Hut

El código analizado aquí es el núcleo del algoritmo presentado en [4] usado en astrofísica, y también forma parte del conjunto de códigos de aplicaciones para el diseño y evaluación de sistemas multiprocesador de memoria compartida *SPLASH* ([70][75]). La aplicación simula la

evolución de un sistema de cuerpos bajo la influencia de fuerzas de gravedad. Es una simulación gravitacional clásica "N-body", donde cada cuerpo se modela como una masa puntual. La simulación avanza a intervalos de tiempo, en cada uno de los cuales se calculan las fuerzas sobre cada cuerpo y se actualizan las posiciones y otros atributos de los cuerpos.

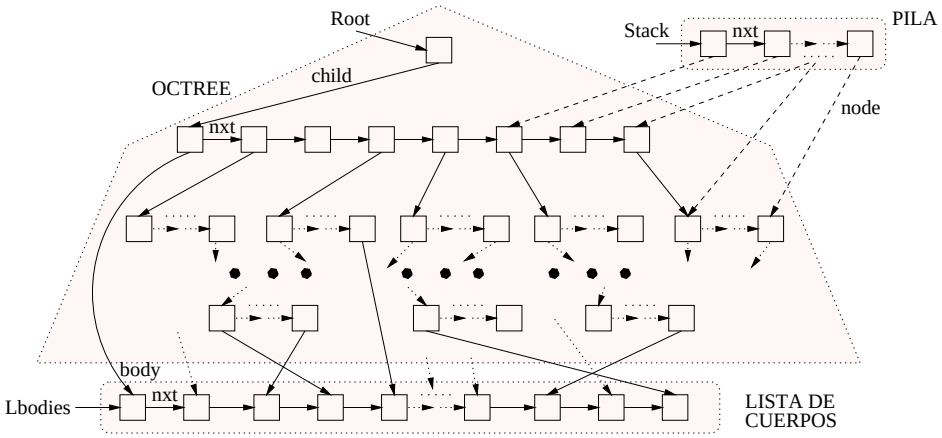


Figura 3.30: Estructuras del código Barnes-Hut.

La estructura de datos usada en este código se basa en un octree jerárquico, el cual representa el espacio en tres dimensiones. En la figura 3.30 presentamos una visión esquemática de la estructura usada en el código. Lo cuerpos se almacenan en un lista enlazada, apuntada por la variable *Lbodies*. El octree representan las subdivisiones del espacio en tres dimensiones, de forma que la raíz representa el espacio completo. Cada uno de sus ocho hijos representa una subdivisión de dicho espacio 3D, hasta llegar a las hojas, las cuales representan subdivisiones del espacio en las que sólo existe un cuerpo. Por este motivo, las hojas del árbol apuntan al cuerpo que encierran, almacenado en la lista *Lbodies*. Los ocho hijos de un nodo del árbol, son almacenados como una lista enlazada por *nxt*, de ocho elementos, de manera que el primer elemento de la lista es apuntado desde el padre por medio del selector *child*.

Los tres principales pasos del algoritmo son:

- i) Creación del octree y de los punteros desde las hojas del mismo a los cuerpos que contienen, almacenados en la lista *Lbodies*. Básicamente, se recorre la lista de cuerpos, y por cada cuerpo, se recorre el trozo del octree creado hasta el momento, buscando el nodo que representa el espacio 3D en el que está situado el cuerpo. Si no hay ningún cuerpo en dicho nodo, se introduce el actual, pero si existe alguno, el espacio representado por el nodo se tiene que subdividir creando sus correspondientes ocho hijos, e introduciendo el cuerpo actual y el que ya existía en su correspondiente lugar dentro de esa nueva subdivisión.

- ii) Para cada subdivisión del espacio, es decir, para cada nodo del octree, se calcula el centro de masas y la masa total de todos los cuerpos almacenados en dicha rama del octree. Esto supone un recorrido primero en profundidad del octree, ya que la masa y centro de masas de un nodo se calcula en base a los de sus ocho hijos (los de las hojas se toman directamente de los cuerpos a los que apuntan). En este tipo de recorrido cada nodo es visitado dos veces en un recorrido: primero cuando es accedido desde su padre por primera vez; segundo cuando se han calculado los valores para el subárbol apuntado por *child* desde el nodo y se actualizan sus valores.

- iii) Para cada cuerpo, recorrer el árbol para calcular las fuerzas sobre dicho cuerpo. Si el nodo del árbol representa una porción del espacio muy alejada del cuerpo, se utilizan el centro de masas y la masa calculada en el paso anterior, y no se avanza en dicha rama del octree. Si por el contrario, el espacio representado por el nodo está próximo al cuerpo, se toman sus ocho hijos y se repite el proceso. En este tipo de recorridos, los nodos del octree tan solo se visitan una vez para cada cuerpo.

Todos los recorridos del octree en el código son llevados a cabo mediante llamadas recursivas. Puesto que nuestro método no soporta, por el momento, análisis interprocedural, se ha llevado a cabo una transformación manual del código, de manera que la recursividad se ha implementado mediante bucles y la utilización de una estructura tipo *pila*. En cada paso de dichos bucles, la *pila* es usada para almacenar punteros a los nodos que tienen que ser visitados en sucesivas iteraciones del bucle. Esta transformación introduce más complejidad en la estructura de datos, puesto que aparece una nueva estructura enlazada con punteros hacia los elementos del octree. En la figura 3.30 aparece esta estructura *pila* y en el apéndice E.1.9 presentamos el código analizado.

La figura 3.31 muestra una simplificación del RSRSG obtenido para la cabecera del bucle que realiza el recorrido del paso iii) del algoritmo, analizado en el nivel L_3. En dicho grafo se pueden apreciar perfectamente los nodos pertenecientes al octree, a la lista de cuerpos y a la *pila* utilizada para emular la recursividad.

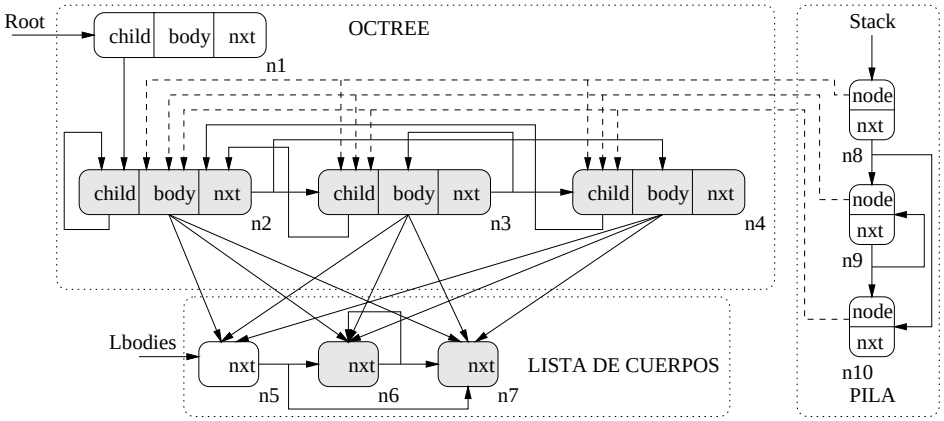


Figura 3.31: Representación simplificada del RSRSG obtenido para el código Barnes-Hut en el paso iii).

Si se eliminan los nodos que representan los elementos de la *pila* (nodos n_8, n_9 y n_{10}), se obtiene el RSRSG resultante para la sentencia correspondiente al final de cada paso de simulación (una vez que desaparece la *pila* usada para los recorridos), el cual es mostrado en la figura 3.32.

Podemos observar que básicamente se mantiene la estructura del octree y de la lista de cuerpos, con la diferencia que en este último, los nodos que representan los elementos del octree no son *shared*, mientras que en el grafo de la figura 3.31 si. Esto es debido, a que las porciones de memoria representadas por los nodos n_2, n_3 y n_4 pueden ser apuntadas por dos selectores distintos: los nodos iniciales de las listas de ocho hijos por el selector *child* desde el nodo padre y por el selector *node* desde algún elemento de la pila; y los restantes elementos de las listas de ocho hijos, por el selector *nxt* desde su hermano de la izquierda, y por *node* desde algún elemento de la pila. Al desaparecer la pila y estos enlaces sobre los nodos, el

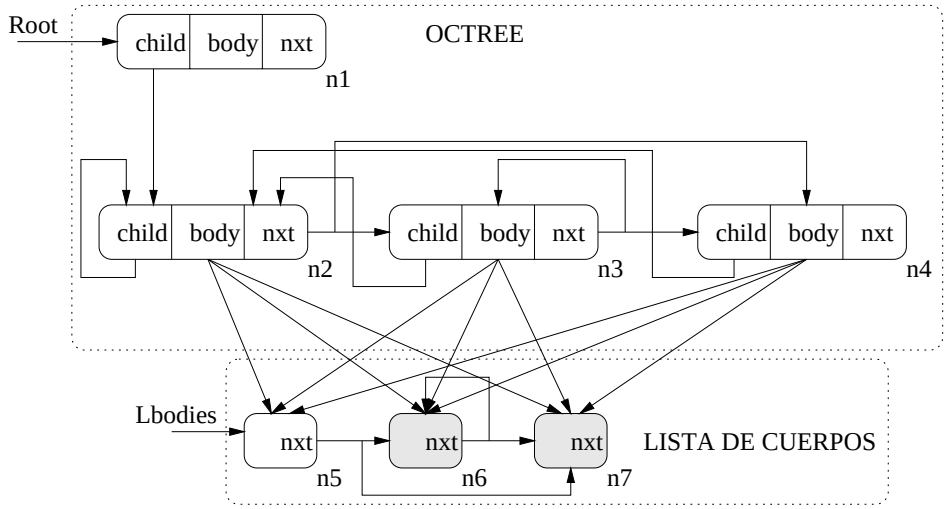


Figura 3.32: Representación simplificada del RSRSG obtenido para el código Barnes-Hut al final de cada paso de simulación.

atributo *SHARED* de los mismos pasa a *false*.

Tenemos que comentar, que este código ha sido analizado en los tres niveles del método, obteniéndose distintos resultados en cada caso.

- **Nivel** L_1: tras el análisis del código en este nivel, el grafo obtenido para la última sentencia es igual al mostrado en la figura 3.32, excepto que los nodos n_5, n_6 y n_7 son *shared* por el selector *body* ($SHSEL(n, body) = true$). Esto implicaría que un mismo cuerpo podría ser apuntado desde varias hojas del octree, cuando en realidad esto no ocurre en la estructura real. El problema aparece en la creación del octree, insertando los elementos de la lista *Lbodies*. Cuando un cuerpo cae en la misma subdivisión que otro, se subdivide de nuevo esa porción de espacio, insertando los ocho correspondientes hijos al nodo actual. Al crear la lista de estos nuevos ocho hijos, puesto que en L_1 no se están usando los *SPATH* de longitud uno, la lista de los nuevos hijos creados apuntada por una variable puntero (*new*), se mezclan con las listas de hijos creadas anteriormente y ya analizadas. A la hora de insertar los cuerpos, esta mezcla, provoca que exista la posibilidad de que dos nodos de dichas listas apunten al mismo cuerpo. Esta mezcla se ha producido por no mantener las "cercanías" de las variables puntero (*new*) aisladas unas de otras (*simple paths* de longitud uno).

  Por tanto, la información que se puede deducir de los RSRSG proporcionados por el análisis en este nivel, es demasiado conservativa y prevendría cualquier intento de paralelización del código.

- **Nivel** L_2: En este nivel, el grafo obtenido para la última sentencia del código si que es el presentado en la figura 3.32 donde ningún nodo es *shared* por ningún selector. Por tanto, en este nivel, el compilador es capaz de determinar de una forma muy precisa las características de las estructuras de datos creadas en el paso i) (ya que los pasos ii) y iii) tan solo recorren la estructura de datos, sin modificarla). Puesto que el atributo *SHSEL* es *false* para todos los nodos y selectores, la variable *Root* apunta a una estructura acíclica (dos recorridos distintos nunca visitan una misma porción de memoria, si no fuera así, algún nodo tendría el valor *true* en *SHSEL* para algún selector). Además los

elementos de esta estructura (el octree) pueden apuntar a distintos elementos de otra estructura acíclica apuntada por *Lbodies*. Se sabe que apuntan a elementos distintos de esta lista, puesto que los nodos que la representan tienen *SHSEL* a *false*, por tanto ninguna porción de memoria ahí representada puede ser apuntada por dos selectores *body* a la vez. Los nodos n_6 y n_7 son shared puesto que las porciones de memoria que representan son apuntadas desde otras dos por los selectores *nxt* (desde el cuerpo anterior) y *body* (desde algún elemento del octree).

Visto este RSRSG se podría pensar que el nivel L_2 es suficiente para el análisis de este código, pero esto no es así. El problema radica en la estructura *pila* que hemos tenido que insertar para eliminar la recursividad. Aunque la estructura de datos creada en el paso i) no se modifica en los recorridos llevados a cabo en los pasos ii) y iii), con la inserción de la *pila*, la forma de dicha estructura si que cambia temporalmente durante el análisis. En concreto, los nodos del octree son apuntados por los elementos de esta *pila* por el selector *node*. Esta complejidad adicional, provoca que en este nivel, el RSRSG para los dos bucles correspondientes a los recorridos ii) y iii) no sea el mostrado en la figura 3.31. El obtenido sería ese mismo grafo, pero con el atributo $SHSEL(n, node) = true$ para los nodos del octree.

¿Que se deduce de esa información? Pues que las porciones de memoria del octree pueden ser referenciadas desde más de un elemento de la *pila*, por lo que podrán aparecer más de una vez en el bucle que recorre la estructura. Esta información, de nuevo es conservativa, puesto que nunca se va a producir.

¿Por qué ocurre esta pérdida de certidumbre? El problema radica en que los nodos que representan porciones de memoria del octree visitadas en el recorrido son mezclados con los nodos que representan porciones de memoria no visitadas, lo que implica que un nodo ya visitado que está en la *pila* se interprete como que puede ser visitado otra vez y de nuevo insertado en la *pila*, de ahí que pueda ser apuntado por más de un selector *node* ($SHSEL(n, node) = true$).

Con la información que se deduce de los RSRSGs obtenidos en este nivel, vemos que es posible determinar con exactitud las características de la estructura de datos creada. Pero debido a la inserción de la *pila*, la posible paralelización de los recorridos sobre el octree se ve truncada, puesto que de los RSRSGs se deduce la posibilidad de visitar más de una vez un mismo nodo del octree en un recorrido.

- **Nivel** L_3: En este nivel se obtiene la máxima exactitud posible del método. En este caso se obtiene el RSRSG de la figura 3.32 para la última sentencia y el de la figura 3.31 para el bucle del recorrido iii). En ambos grafos, como hemos comentado, todos los nodos tienen $SHSEL = false$ para todos los selectores.

  La mejora con el nivel L_2 precisamente es el RSRSG del bucle del recorrido iii) en el que ahora los nodos del octree no son *shared* por *node*, por lo que se sabe que los elementos de la pila apuntan a distintos elementos del octree. Esta información permitiría deducir que el recorrido que hace ese bucle, sacando elementos de la pila, nunca va a pasar dos veces por la misma porción de memoria, con lo que se podría pensar en paralelizar dicho recorrido.

  Esta mejora ha sido posible gracias a que la utilización de la propiedad *TOUCH* en este nivel, mantiene por separado los nodos del octree visitados en un recorrido de los que aún no lo han sido. Por lo tanto al seleccionar nuevos nodos a introducir en la *pila*, tan solo se toman de aquellos que no han sido visitados.

Sin embargo el RSRSG obtenido en este nivel para el bucle del recorrido ii), sigue teniendo el atributo $SHSEL(n, node) = true$ para los nodos del octree. Esto se debe a que, aunque se utiliza la información $TOUCH$ para mantener porciones visitadas separadas de las no visitadas, en este recorrido, los nodos son visitados dos veces, como hemos comentado anteriormente (una vez para tomar sus hijos y la segunda para actualizar los valores de masa y centro de masas calculados a partir de la de sus hijos). Debido a que este análisis es en tiempo de compilación, es incapaz de deducir que la segunda vez que se visita un elemento del octree, no inserta sus hijos en la *pila*, puesto que ya los insertó la primera vez. Esto hace que se inserten de nuevo sus hijos en la pila, y de ahí que haya nodos *shared* por *node* desde la misma. Para poder evitar esta falta de exactitud, habría que poder extraer información muy compleja de las estructuras condicionales (IF) que determinan si los hijos de un elemento del octree son insertados o no en la *pila*.

En conclusión, en este nivel se deducen perfectamente las características de forma de la estructura de datos creada, y la información proporcionada permitiría una posible paralelización de los recorridos del octree del paso iii).

Para terminar, hay que comentar un comportamiento que merece una explicación y que podemos observar en la tabla 3.2. Como ya comentamos, el aumento de nivel de análisis trae consigo la utilización de más propiedades lo que provoca en principio un aumento de los requerimientos de memoria y tiempo durante el análisis. Vemos que esto se cumple para todos los códigos analizados excepto para el código Barnes-Hut. Como se puede observar, para este código, los niveles L_2 y L_3 consumen menos memoria y tardan menos que el nivel L_1. Esto es debido a que, como hemos visto, el RSRSG para la estructura creada en el paso i) para el nivel L_1 tiene nodos con $SHSEL = true$. Esto trae consigo una consecuencia inmediata, la interpretación de las sentencias sobre estos grafos tiene que ser más conservativa. Con esto nos referimos a que, como se ha descrito por ejemplo al presentar la *poda* de los grafos, cuando los nodos no son *shared* se pueden eliminar selectores si se cumplen una serie de propiedades. Lo mismo ocurre por ejemplo en la operación de *materialización*. También hay un aumento en el número de nodos, puesto que no se mezclan los que tiene $SHSEL = true$ con los que no lo tienen. Aparte, la existencia de más selectores, provoca la existencia de más nodos aún, puesto que hay más posibilidades para sus *reference patterns* (*SELINset*, *SELOUTset*, *PosSELINset*, *PosSELOUTset*). Este aumento en el número de nodos distintos y de selectores provoca a su vez un mayor número de grafos distintos en los RSRSGs, puesto que hay más nodos que tiene que ser compatibles para que dos grafos sean unidos. Y por último, la existencia de un mayor número de grafos y de nodos en los grafos, provoca que haya que efectuar un mayor número de pasadas hasta que se alcanza un punto fijo en el análisis. Estos factores son los que provocan que el nivel l_1 aún utilizando menos propiedades, al obtener una representación "peor" de la estructura de datos, necesite más memoria y tiempo de ejecución para el análisis del código.

4 Arrays de punteros en los RSRSGs

En este capítulo se presenta un extensión importante sobre el método basado en los RSRSG para permitir el análisis de un tipo de estructuras muy comunes en códigos C que manejan estructuras dinámicas de datos. En concreto, nos estamos refiriendo a soportar estructuras del tipo *array de punteros* a otras porciones de memoria. Hasta donde conocemos, ninguno de los trabajos relacionados con el análisis de forma soporta este tipo de estructuras.

Como hemos dicho, hemos creado una abstracción de este tipo de campos dentro de estructuras de datos más complejas, puesto que es muy común encontrarlos en códigos C. Así por ejemplo, una manera muy común de representar matrices dispersas es por medio de un array de tantas posiciones como filas o columnas tenga la matriz, en donde cada elemento del mismo es un puntero a una lista que almacena los elementos no nulos. Otro ejemplo común, son las estructuras tipo árbol, donde cada nodo tiene un número determinado de hijos, por ejemplo los *quadtrees* u *octrees* donde se sabe que cada elemento tiene cuatro y ocho hijos respectivamente. Es común encontrar dentro de cada nodo un campo tipo array con cuatro u ocho punteros a sus respectivos hijos.

En definitiva, vemos que por un sólo nombre de selector se hace referencia a un conjunto de enlaces, tantos como posiciones tenga el array. Por este motivo, este tipo de campos no están soportados por nuestro método, en el que un nombre de selector, representa a lo sumo, tan solo un enlace desde una porción de memoria. Por este motivo, hemos creado una abstracción de los arrays de punteros, como campo dentro de una estructura, denominada *multiselector*. Estos *multiselectores* van a poder representar, con un único nombre, varios enlaces desde una misma porción de memoria hacia otras.

4.1 Multiselectores

Como ya hemos comentado, con los multiselectores vamos a dar soporte a campos de tipo array de punteros dentro de las estructuras dinámicas de datos. Es decir, el método será capaz de analizar códigos con declaraciones de estructuras del siguiente tipo:

```
typedef NUEVA_ESTRUCTURA {
  ...
  struct NUEVA_ESTRUCTURA *sel1[256];
  struct OTRA_ESTRUCTURA **sel2;
```

...
}

En la figura 4.1 presentamos una representación gráfica de copo podría ser una estructura del tipo *NUEVA_ESTRUCTURA*, apuntada por una variable *x*.

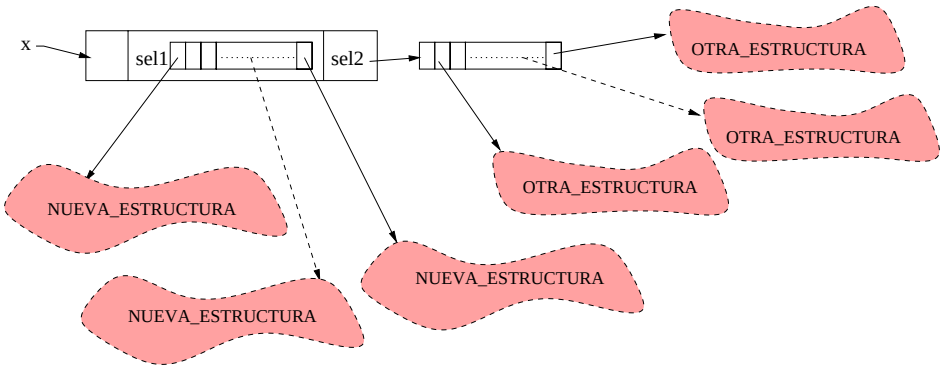


Figura 4.1: Ejemplo de estructura con arrays de punteros.

Las porciones de memoria del tipo *NUEVA_ESTRUCTURA*, contendrán un array de 256 punteros a otras porciones de memoria de este mismo tipo de estructura, llamado $sel1$, y además tendrá un puntero a un array de tamaño desconocido, con punteros a porciones de memoria del tipo *OTRA_ESTRUCTURA*, con el nombre $sel2$. En cualquiera de los dos casos, la utilización de los enlaces almacenados en dichos arrays se realiza de la misma forma, un nombre y un índice para determinar el enlace. Así por ejemplo, para una variable pv del tipo *NUEVA_ESTRUCTURA*, podemos encontrarnos ocurrencias del tipo $pv \rightarrow sel1[i]$ y $pv \rightarrow sel2[j]$ para hacer referencia a un enlace concreto de la porción de memoria apuntada por pv.

A la hora de intentar dar soporte a este tipo de estructuras dentro del nuestro método, hay que representar de alguna manera estos nuevos *selectores* dentro de los nodos. Como ya hemos comentado, no se puede utilizar la abstracción *selector*, puesto que un selector representa en cada caso un sólo enlace desde cada porción de memoria representada por un nodo, por tanto no puede representar a los arrays de punteros que implican más de un enlace desde una misma porción de memoria. Se podría pensar en mantener un nombre de selector distinto para cada una de las posiciones del array $(sel1_1, sel1_2, ..., sel1_{256})$, pero esto no es factible puesto que además de poder existir un número muy elevado de posiciones en un array, hay muchas ocasiones en las que no es posible determinar en tiempo de compilación el tamaño de dicho array (array $sel2$). Vemos que sucede algo similar a lo que ocurre con las porciones de memoria y los nodos que las representan en los grafos. No es posible tener un nodo por cada posible porción de memoria que pueda aparecer en las configuraciones de memoria, puesto que esto haría totalmente inviable el análisis. Para solucionar esto, los métodos basados en grafos, colapsan en un mismo nodo muchas porciones de memoria para mantener un número finito de nodos por grafo.

Nosotros hemos aplicado la misma idea a los enlaces de un array de punteros. Vamos a colapsar todos los enlaces de un array en un elemento que vamos a denominar *multiselector*, con un nombre determinado. De esta forma, todos los enlaces del array van a estar representados por un nombre, el del *multiselector*. Tenemos que decir que los *multiselectores* no sólo van a poder representar campos de porciones de memoria que son arrays de punteros, sino que, de forma general, podrán ser usados para representar a una variable tipo array de punteros

a estructuras de datos. Bastará con ver la variable array, como un puntero a una porción de memoria que contiene tan solo un campo tipo array de punteros. En una declaración de variable del tipo:

$ESTRUCTURA * pv[100];$

la reserva de espacio para los 100 punteros la llevará a cabo el compilador insertando una instrucción *malloc*(), lo que provocará que la semántica abstracta de esta sentencia cree un nodo que contenga sólo un campo de tipo array de punteros a estructuras tipo $ESTRUCTURA$.

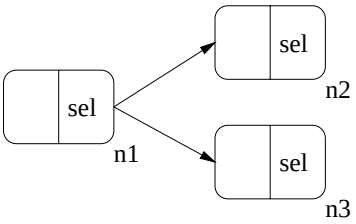


Figura 4.2: Enlaces entre nodos.

Por otro lado, al igual que en nuestro método se asocia información adicional a los nodos, referente a las porciones de memoria que representa para obtener una representación más exacta de las configuraciones de memoria, los *multiselectores* también van a tener información adicional sobre los enlaces que poseen, para poder manejar de una manera más "inteligente" estos enlaces, cuando sea posible.

Por ejemplo, tenemos tres nodos en un RSG, n_1, n_2 y n_3, enlazados de la forma que muestra la figura 4.2. Si el campo *sel* de los nodos es un *selector*, entonces podemos decir que cada porción de memoria representada por n_1 puede apuntar a una porción de memoria representada por n_2 o bien a una porción de memoria representada por n_3, utilizando el campo *sel*. Sin embargo, si *sel* fuera un *multiselector*, entonces cada porción de memoria representada por n_1 podría apuntar a la vez a varias porciones de memoria representadas por n_2 además de a varias porciones representadas por n_3, o a todas a la vez, siempre utilizando el mismo campo *sel*. Como se puede observar hay una gran diferencia entre los *selectores* y los *multiselectores*, por lo que su manejo será también muy distinto.

Así, hay que ver como se van a interpretar instrucciones que contengan en su *lhs* o *rhs* referencias a algún enlace de un array de punteros, del tipo $x \rightarrow sel[i] = y$ o $y = x \rightarrow sel[i]$. Como se ha descrito en el capítulo 3, el método de análisis es capaz de enfocar los enlaces representados por un *selector* simple, y modificarlos según indica la semántica abstracta de la sentencia. Si nos fijamos, las instrucciones que utilicen enlaces de un array de punteros, básicamente son iguales que las que utilizan un puntero simple, es decir cuando se tiene $y = x \rightarrow sel[i]$, la variable y va a apuntar a la porción de memoria referenciada desde x por exactamente un enlace, al igual que ocurre en $y = x \rightarrow sel$. Igual ocurre con $x \rightarrow sel[i] = y$ y $x \rightarrow sel = y$, donde en ambos casos se crea exactamente un enlace desde la porción apuntada por x y la apuntada por y. La diferencia entre esas sentencias radica en que las que utilizan un *selector* simple saben exactamente que enlace se crea, se utiliza o se elimina, mientras que las que utilizan un *multiselector*, en tiempo de compilación no saben con exactitud cual de todos los enlaces representados por le array, es creado, usado o eliminado. Por tanto, vemos que la manera de interpretar las sentencias con *multiselectores* será la misma que la de las sentencias normales, pero si conseguimos "enfocar" de entre todos los enlaces representados por el *multiselector*, aquellos que realmente puede que sean los referenciados en la sentencia.

Una vez enfocado el *multiselector*, puede ser tratado como un *selector* normal, aplicando la semántica abstracta de la sentencia correspondiente.

Sin ningún tipo de información adicional en los *multiselectores*, el enfoque del enlace utilizado en una sentencia tipo $x \rightarrow sel[i]$, consistiría en considerar que dicho enlace puede ser cualquiera de los enlaces existentes para el *multiselector sel* en el nodo apuntado por x. Para poder hacer un "enfoque" un poco más selectivo y exacto, hemos introducido algo de información adicional en los *multiselectores*. En referencia a esta información adicional hemos introducido dos conceptos en los *multiselectores*: las **instancias** de un multiselector y las **clases de multireferencias**.

Instancias de un multiselector

Por *instancia* de un multiselector vamos a referirnos a un subconjunto de posiciones del array con algo en común. Este subconjunto puede representar desde una posición del array hasta el conjunto completo de las mismas. Cada instancia de un multiselector se va a diferenciar de las demás por un identificador de instancia, que va a estar muy relacionado con las variables utilizadas en el código como índices de los arrays de punteros, que desde ahora denominaremos **ivars**.

Dentro de un multiselector puede haber dos tipos de instancias:

- *Instancia simple*: estas instancias representan tan solo una posición del array de punteros y los enlaces que salen de dicha instancia representan los enlaces que puede tener la posición del array representada.

  El identificador de estas instancias es un conjunto de *ivars*, de manera que la instancia representa la posición del array cuyo valor poseen todas las *ivars* del identificador de la instancia. Por ejemplo, si las variables i y j valen 3, la instancia i, j de un multiselector representaría los posibles enlaces que puede tener la posición 3 del array de punteros que representa el multiselector.

- *Instancia múltiple*: esta instancia representa varias posiciones del array y los enlaces que posee representa todos los enlaces que pueden tener las posiciones representadas por la instancia.

  El identificador de estas instancias también es un conjunto, pero ahora no puede aparecer ninguna *ivar*, puesto que si apareciera, estaría representando una posición concreta del array y no un subconjunto de posiciones. Por tanto, en principio, tan solo podría aparecer como identificador de estas instancias el conjunto vacío ($\emptyset$), que representaría al total de las posiciones del array no representadas por otras instancias simples del multiselector. Para que pueda existir más de una instancia múltiple que nos proporcionen información útil, se va a permitir la distinción de instancias múltiples en relación con las posiciones de los arrays recorridas en los bucles, como veremos más adelante.

Por tanto, por defecto, todo multiselector va a constar de una *instancia múltiple*, $\emptyset$, que va a representar a todas las posiciones del array, que tendrá todos los enlaces que puedan tener todas las posiciones de dicho array. Así, por ejemplo si nos encontramos en un sentencia una referencia del tipo $x \rightarrow sel[i]$, tendremos que ver de todos los enlaces del multiselector *sel* del nodo apuntado por x (n_x) cuales son a las que se puede referir dicha sentencia. Si el multiselector *sel* de n_x no posee una instancia simple que contenga en su identificador a

la *ivar* i, habrá que crear una instancia simple que contenga a i en su identificador y que represente todos los posibles enlaces que pueda tener la posición i del array en esa sentencia. Al proceso de creación de una instancia simple a partir de una múltiple para representar la posición que puede tomar una *ivar*, se le llama **instanciación**.

Del proceso de *instanciación* podemos decir que:

- crea una instancia simple que representa enlaces simples desde una posición del array de punteros, y que por lo tanto podrá ser tratado de forma similar a los enlaces representados por un *selector* simple.
- Al crear la instancia, se deben asociar todos aquellos enlaces del multiselector que pueda tener la posición i del array.

En relación a la asignación de los enlaces a la instancia simple creada en la instanciación, tenemos que decir, que puesto que en tiempo de compilación es posible que no pueda determinarse con exactitud que valor tomará i en una determinada sentencia, por defecto se tomarán todos los enlaces que tengan las demás instancias del multiselector. Esto hace referencia a que la posición i del array puede ser cualquiera de las representadas por las otras instancias. Este tratamiento que se le da a la instanciación es muy conservativo, por lo que nuestro método va a mantener información acerca de las instancias de manera que se pueda determinar relaciones entre las mismas. De esta manera, algunas veces se podrá determinar que una nueva instancia simple, creada en la instanciación de la posición i, no puede tener los enlaces que posee otra determinada instancia I_1, ya que la relación entre ambas nos indica que la posición representada por i no pertenece al conjunto de posiciones del array representadas por I_1.

Para determinar las relaciones entre las instancias, nos vamos a basar en las relaciones que se puedan extraer del código sobre las *ivars* usadas para indexar los arrays de punteros. Así en una fase de preproceso del código, para cada bloque del mismo, se determinará para cada dos *ivars* si en dicho bloque toman el mismo valor (eq), si no toman el mismo valor (neq) o si no se sabe si pueden tener o no el mismo valor (unk). Por ejemplo, si se va a crear una instancia nueva para la posición i en un multiselector que posee otra instancia con j en su identificador, podemos encontrar las siguientes situaciones:

- Si la relación entre i y j es eq, se sabe que en esa sentencia el valor de i es igual al de j y por tanto las posiciones que represente i serán las mismas que las representadas por j. Como ya existe una instancia que representa las posiciones que puede tomar j, bastará con añadir al identificador de esa instancia la *ivar* i, para indicar que ahora también representa a las de i (que son las mismas puesto que $i = j$).
- Si la relación entre i y j es neq, entonces no pueden tomar el mismo valor en dicha sentencia. Por tanto se creará una nueva instancia para i, y se copiarán los enlaces del multiselector, a excepción de los enlaces de la instancia que contiene a j en su identificador. Esto es así puesto que esos enlaces son para posiciones del array que en esta sentencia no puede tomar i (puesto que $i \neq j$). Como vemos, esta información hace que la elección de los enlaces para una nueva instancia sea selectiva, evitando la inclusión de enlaces superfluos.
- Por último si la relación entre ambas *ivars* es desconocida (unk), se tendrá que crear una nueva instancia que contenga a i y al crear sus enlaces habrá que tomar también

los que tenga la instancia que contiene a j. Aquí la información extraída del código no es suficiente y por tanto se tiene que tomar una postura conservativa.

Aún se puede extraer del código más información relacionada con las *ivars* que puede mejorar la instanciación. En concreto nos referimos a los cuerpos de los bucles, que recorren con una *ivar* las posiciones de un array de punteros. Veamos el siguiente código:

```
1:  for (i=0; i<n; i++)
    {
      ...
2:    x→sel[i] = ...
      ...
3:    for (j= i+1; j<n; j++)
        {
          ...
4:        ... = x→sel[j];
          ...
        }
      ...
    }
```

Una de las relaciones que se puede deducir es que en el cuerpo del bucle (3) las variables i y j son distintas. Sin embargo es deducible también que dentro del bucle (1) la variable i no toma dos veces el mismo valor (en un mismo recorrido) y lo mismo se puede decir de la variable j y el bucle (3). Esta información podría ser utilizada para, de alguna manera, evitar que cuando se crea una nueva instancia para i en la sentencia (2), se tomen los enlaces de instancias que representen posiciones por las que anteriormente ha pasado i, ya que no puede tomar dos veces el mismo valor en un mismo recorrido. La identificación de las regiones que recorre una variable de inducción en un array se puede hacer con bastante precisión, incluso cuando aparecen funciones de acceso no afines, como se muestra en [42, 33].

Para recoger este tipo de información, es necesario poder "acordarse" de las posiciones de los arrays que han visitado las *ivars* que son utilizadas como variable de inducción en algún bucle. Esta información está muy relacionada con la "permanencia" de las instancias en los multiselectores.

- *Permanencia de las instancias*: Como sabemos, cuando nos encontramos una sentencia con una referencia del tipo $x \rightarrow sel[i]$, se tiene que crear una instancia del multiselector *sel* en el nodo apuntado por x (n_x) con la *ivar* i en su identificador. Pero, ¿hasta cuando permanece dicha instancia en el multiselector *sel* del nodo n_x? Pues bien, como dicha instancia representa los enlaces de las posiciones que podía representar i cuando fue creada, se puede mantener dicha instancia mientras no se asigne un nuevo valor a la *ivar* i. Al cambiar de valor i, $x \rightarrow sel[i]$ hará referencia a selectores de otras posiciones del array y por tanto la instancia que contenía a i en su identificador ya no lo debe tener. Cuando se asigna un nuevo valor a una *ivar*, dicha variable es eliminada de los identificadores de las instancias. Esto provocará que varias instancias se fundan en una sola con todos los enlaces de ambas. Por ejemplo si el identificador de la instancia es $\{i\}$ y se elimina la *ivar* i, entonces nos quedaría la instancia $\emptyset$, de manera que si existe otra instancia $\emptyset$, se uniría con esta.

En cada paso del bucle (1) se da un valor nuevo a la *ivar* i, por lo que cada vez que empieza el análisis del bucle (1) se eliminaría i de las instancias. Pero como hemos visto sería muy útil "acordarse" de que posiciones ha visitado i, es decir mantener en una instancia distinta

las posiciones que i ha tomado anteriormente. Lo que hemos hecho en nuestro método, es que al empezar a analizar un bucle de este tipo en el que se sabe que la *ivar* i no va a tomar dos veces el mismo valor en la ejecución del bucle, en vez de sólo eliminar i de las instancias, se va a anotar la instancia con un valor que se identifique como *valor anterior* de i. En este caso, se eliminaría i de las instancias y se introduciría dicho valor, que denominaremos $va(i)$ (valor anterior de i). De este modo, al instanciar en la sentencia (2) con i, no se tomarán los enlaces de aquellas instancias que contengan $va(i)$ en su identificador, puesto que representan posiciones del array que no puede tomar el valor actual de i.

Por tanto, además de las relaciones entre las *ivars* como habíamos visto antes, también vamos a necesitar relaciones entre las propias *ivars* y los valores anteriores de las misas ($va(ivar)$). En este caso, los posibles valores de la relación tan solo pueden ser de desigualdad (neq) y de desconocimiento (unk), ya que no tiene mucho sentido informar de que una *ivar* i es igual a todos sus valores anteriores ($i = va(i)$).

- *Renombre de las instancias*: Cada vez que se asocia un nuevo valor a una *ivar*, todas las instancias de los multiselectores de los grafos que contienen a la *ivar* en su identificador, son renombradas, eliminando dicha *ivar* del identificador. Si además en dicha sentencia se da la relación $ivar \neq va(ivar)$, se introducirá $va(ivar)$ en el identificador de la instancia. Al finalizar el análisis de un bucle, ya no tiene sentido mantener en los identificadores $va(ivar)$ donde *ivar* es la variable de inducción del bucle, de manera que $va(ivar)$ es eliminada de los identificadores de las instancias.

  Tras el renombre, las instancias que pasen a tener el mismo identificador, serán fundidas uniendo sus enlaces en una sola instancia.

Todos estos conceptos relacionados con las instancias se verán en mayor profundidad a lo largo de este capítulo.

Clases de multireferencias

Como hemos visto, se han introducido las instancias dentro de los multiselectores para poder seleccionar de entre todos los posibles selectores del multiselector, los correspondientes a la posición que puede referenciar $sel[i]$, y usarlo como si de un *selector* normal se tratara. Se le ha añadido información extra, para que la selección de estos posibles enlaces sea un poco menos conservativa y use información disponible en el código del programa.

Aún así, el manejo de los multiselectores por parte del método es menos preciso que el de los selectores normales. Por ejemplo, cuando dos nodos se sumarizan en uno solo, se unen los distintos destinos de sus selectores, como por ejemplo sel_1. Cuando posteriormente, en alguna sentencia, se referencia el selector sel_1 de dicho nodo, el método "enfoca" dicho enlace (operaciones de división, poda y materialización vistas en el capítulo 3), de manera que se aislan los distintos enlaces representados por sel_1 que provienen de los distintos nodos sumarizados. Esta operación es posible ya que un selector normal representa selectores simples desde una porción de memoria determinada a otra. Como en realidad se trata de enlaces distintos, se pueden separar en grafos diferentes y así ser manejados con más exactitud (sin mezclarse con los demás).

Sin embargo, no es posible hacer lo mismo sobre los multiselectores, puesto que por definición, estos representan más de un enlace desde una porción de memoria a otras, pertenecientes

todos a la misma configuración de memoria. Por tanto, ahora dos enlaces que parten de un multiselector pueden representar tanto enlaces pertenecientes a la misma porción de memoria (varias posiciones del array) como enlaces pertenecientes a distintas porciones de memoria. Esto es así, ya que cuando dos nodos con multiselectores se unan, se unirán también los enlaces de los mismos, por tanto habrá enlaces que pertenezcan a la misma configuración y otros que pertenezcan a distintas configuraciones. El problema es que una vez mezclados los de una configuración y otra, luego es imposible distinguirlos.

En la figura 4.3 vemos el proceso de sumarización de los nodos n_1 y n_3 en el nodo n_5 (sus propiedades son similares). Los destinos del selector sel_1 de dicho nodo son los mismos que tenía dicho selector en los nodos n_1 y n_3. Cuando se referencie el selector sel_1 del nodo n_5 en alguna sentencia, se tomarán por separado los dos destinos puesto que al ser sel_1 un selector simple, en una determinada configuración de memoria tan solo puede apuntar a un sólo destino (n_2 o n_4). De ahí que se puedan separar en dos grafos distintos, uno con n_5 apuntando a n_2 por sel_1 y otro apuntando a n_4.

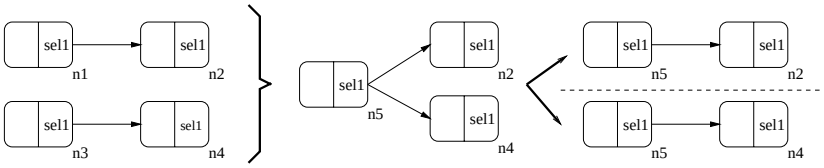


Figura 4.3: Sumarización y división con selectores simples.

En la figura 4.4 se presenta el caso de sumarización de nodos con un multiselector ms_1. Tras la unión, al igual que en el caso anterior, los destinos del multiselector ms_1 del nuevo nodo son los que tenían los nodos unidos (n_2, n_4 y n_5). Sin embargo, cuando una sentencia haga referencia a un enlace del multiselector ms_1 del nodo n_6, no se podrán separar los distintos enlaces de ms_1 puesto que es un multiselector, y todos los enlaces que representa pueden pertenecer a la misma porción de memoria. En este caso, tendríamos que suponer que las porciones de memoria representadas por n_6 pueden referenciar a la vez a las porciones de memoria representadas por los nodos n_2, n_4 y n_5 por el array de punteros ms_1 (distintas posiciones del array). Como se puede observar, se ha perdido la información de que en ningún caso una porción de memoria representada por n_6 puede apuntar simultáneamente a porciones de memoria representadas en n_2 y n_4 o n_2 y n_5, mediante punteros del array ms_1.

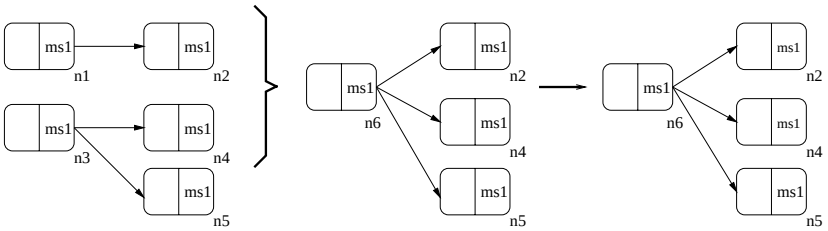


Figura 4.4: Sumarización con multiselectores sin clases de multireferencias.

Para que el manejo de los multiselectores sea un poco más preciso, hemos creado las *clases de multireferencias* (mc). Una clase de multireferencias va a agrupar bajo un identificador, un subconjunto de los enlaces de un multiselector, que pueden pertenecer todos a una misma porción de memoria. Así, por ejemplo la figura 4.5, podemos ver un nodo n_1 con un multiselector ms_1 con dos enlaces a los nodos n_2 y n_3, pertenecientes a la misma clase de

multireferencia mc_1. Hay otro nodo n_4, también con el multiselector ms_1 con dos enlaces a los nodos n_5 y n_6, pertenecientes a otra clase, mc_2. Si las propiedades de los nodos n_1 y n_4 fueran similares, dichos nodos se sumarizarían, creando un nuevo nodo n_7 que representaría todas las porciones de memoria de los dos nodos sumarizados. El multiselector ms_1 del nodo n_7 tomará todos los enlaces que tenía dicho multiselector en los nodos n_1 y n_4, por tanto apuntará a los nodos n_2, n_3, n_5 y n_6. La diferencia ahora es que cada enlace conserva su clase de multireferencia, de modo que el enlace a $n_2 \in mc_1$, el de $n_3 \in mc_1$, sin embargo el enlace a $n_5 \in mc_2$ al igual que el de n_6.

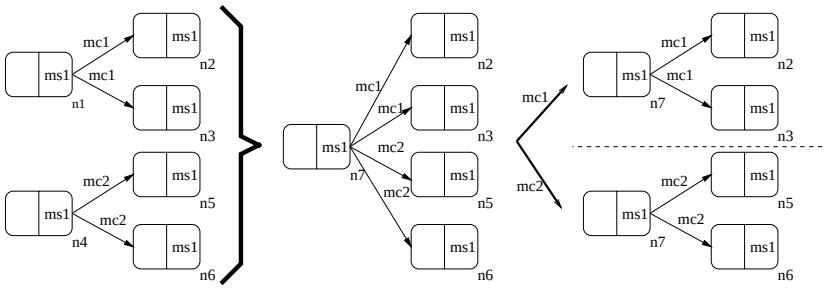


Figura 4.5: Ejemplo de sumarización usando clases de multireferencias.

Cuando en una sentencia se haga referencia un enlace del multiselector ms_1 del nodo n_7, ahora si es posible separar en grafos distintos los distintos conjuntos de selectores que pueden aparecer en una misma porción de memoria (operación similar al *divide* de los selectores simples). Esta división, consiste en separar en distintos grafos las distintas clases de multireferencias que existan en el multiselector ms_1 del nodo. En la figura 4.5 se puede observar como quedaría la unión de los nodos usando las clases de multireferencias, y como gracias a ellas, se pueden separar de nuevo los distintos destinos del multiselector ms_1.

- *División por clases de multireferencias*: consiste en separar en grafos distintos los enlaces pertenecientes a distintas clases de multireferencias de un multiselector perteneciente a un nodo determinado. Para crear cada grafo, bastará con eliminar los enlaces pertenecientes a todas las clases de multireferencias que no sean la correspondiente al grafo que se está creando. Una posterior *poda* del grafo acabará por "enfocar" el grafo, eliminando nodos y selectores pertenecientes a otra clase de multireferencia.

Vemos como las clases de multireferencia mantienen, dentro de un mismo nodo, información de enlaces pertenecientes a muy diversos nodos compatibles. Se podría pensar que el número distinto de clases de multireferencias no tiene límite pudiendo crecer indefinidamente, provocando que cada grafo fuera distinto de todos los anteriores y haciendo así imposible alcanzar un punto fijo. En realidad esto no puede ocurrir, ya que el número de clases de multireferencias distintas está limitado por el número de enlaces distintos que puede tener un multiselector. Este número de enlaces es finito, ya que los destinos de estos enlaces son los nodos del grafo, que como se presentó en el capítulo 3, el número distinto de estos que pueden aparecer en un grafo está limitado.

La idea es la siguiente, si se tienen dos clases de multireferencias, mc_1 y mc_2, que tras alguna modificación del grafo, tienen los mismos nodos destino para sus enlaces, entonces dichas clases ya no son distinguibles (representan el mismo subconjunto de enlaces). Si esto ocurre, se puede eliminar una de ellas, quedando su información representada por la otra.

Los conceptos y operaciones relacionados con las clases de multireferencias introducidos aquí, serán presentados formalmente a lo largo de este capítulo.

4.2 Multiselectores en los RSGs

En esta sección vamos a introducir los multiselectores en los *Grafos de Forma de Referencias* (RSG), viendo los cambios que esto supone con respecto al paradigma presentado en el capítulo 3.

Hay que empezar modificando la definición de configuración de memoria (definición 3.2.1), añadiendo la posibilidad de la existencia, dentro de las porciones de memoria, de arrays de punteros a otras porciones.

Definición 4.2.1 Vamos a representar las configuraciones de memoria con la tupla $M = (L, P, S, MS, PS, LS, LMS)$, donde L, P, S, PS y LS se toman de la definición 3.2.1, a la que se han añadido MS y LMS, de modo que:

- **MS** es el conjunto de arrays de punteros declarados dentro de las estructuras del código, que denominaremos como *multiselectores*.
- **LMS** es el conjunto de enlaces entre porciones de memoria por medio de punteros pertenecientes a arrays de punteros (multiselectores), de la forma $<l_1, ms, ind, l_2>$, donde la porción de memoria $l_1 \in L$ referencia a la porción de memoria $l_2 \in L$ por medio del puntero que hay en la posición $ind \in \mathbb{N}$ del multiselector $ms \in MS$.

Con $MS(m)$ y $LMS(m)$ nos referiremos a dichos conjuntos pertenecientes a la configuración de memoria m. □

En la figura 4.6 podemos ver una representación gráfica de una configuración de memoria, M, que representa una estructura tipo árbol, donde cada elemento puede tener hasta cuatro hijos. Los enlaces a dichos hijos se almacenan en un array de punteros *child* de cuatro posiciones. Los conjuntos que definen M son: $L(M) = \{l_1, l_2, l_3, l_4, l_5, l_6\}$, $P(M) = \{tree\}$, $S(M) = \emptyset$, $MS(M) = \{child\}$, $PS(M) = \{< tree, l_1 >\}$, $LS(M) = \emptyset$ y $LMS(M) = \{< l_1, child, 1, l_2 >, < l_1, child, 3, l_3 >, < l_2, child, 4, l_4 >, < l_3, child, 1, l_5 >, < l_3, child, 4, l_6 >\}$.

Como ya se vio en el capitulo anterior, el RSRSG asociado a cada sentencia es una aproximación de todas las configuraciones de memoria que pueden aparecer tras la ejecución de dicha sentencia. Por tanto, vamos a ver como se ve afectada al definición de los RSGs (definición 3.2.2) tras la inclusión de los multiselectores.

Definición 4.2.2 Tras la inclusión de los multiselectores en los RSGs, un RSG estará representado por la tupla $RSG = (N, P, IV, S, MS, PL, NL, MSL)$, donde de nuevo, N, P, S, PL y NL se toman de la definición 3.2.2, a la cual se le han añadido IV, MS y MSL, de modo que:

- **IV** es el conjunto de variables utilizadas como índice de los multiselectores en el código.
- **MS** es el conjunto de multiselectores declarados en el código (arrays de punteros).
- **MSL** es el conjunto de enlaces por medio de un multiselector de la forma $<n_s, ms, ins, mc, n_d>$, donde:

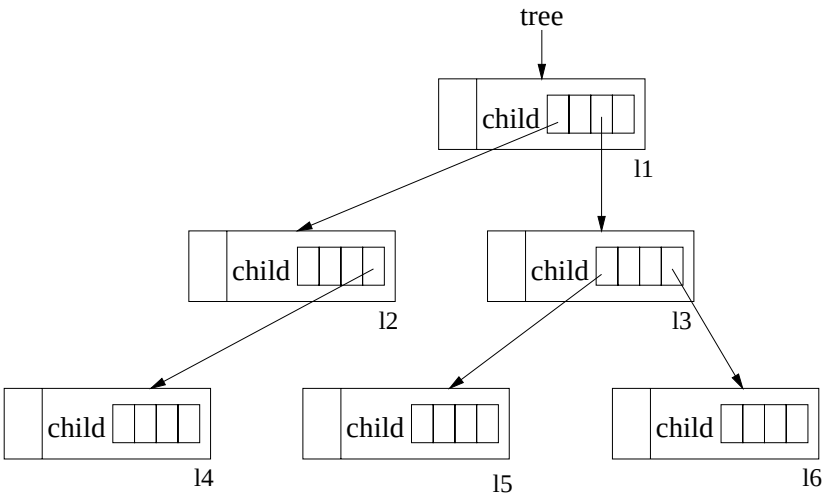


Figura 4.6: Ejemplo de configuración de memoria con multiselectores.

- $n_s \in N(rsg)$ es el nodo origen del enlace,
- $ms \in MS(rsg)$ es el identificador del multiselector al que pertenece el enlace,
- $ins =< ivs, vivs >$ es el identificador de la *instancia* del multiselector a la que pertenece el enlace (ivs y $vivs$ se discuten a continuación),
- $mc \in \mathbb{N}$ es el identificador de la clase de multireferencia a la que pertenece el enlace y
- $n_d \in (N(rsg) \cup NULL)$ es el nodo destino del enlace, que puede ser $NULL$ si las posiciones representadas por la instancia no referencian a ningún nodo.

El identificador de instancia ins está formado por dos conjuntos $<ivs, vivs>$ de la forma:

- $ivs = \{iv \mid iv \in IV(rsg)\}$, conjunto de *ivars* que en la sentencia a la que pertenecen las configuraciones de memoria representadas por rsg, están representadas por la instancia ins. Para referirnos a este conjunto para una instancia ins determinada, usaremos la notación $ivs(ins)$.
- $vivs = \{iv \mid iv \in IV(rsg)\}$, conjunto de *ivars* que anteriormente a la sentencia actual a la que pertenecen las configuraciones de memoria representadas por el rsg, han estado representadas por la instancia ins (las *ivars*, iv, que aparecen en este conjunto también son representadas como $va(iv)$, como hemos visto en la sección 4.1 al describir las instancias de los multiselectores). Para referirnos a este conjunto de *ivars* para una instancia determinada ins, usaremos la notación $vivs(ins)$.

De nuevo usaremos $IV(rsg)$, $MS(rsg)$ y $MSL(rsg)$ para hacer referencia a cada uno de esos conjuntos del grafo rsg. Aunque podremos encontrarnos IV y MS sin hacer referencia al grafo rsg ya que son conjuntos que no dependen del grafo actual, sino que dependen del código y por tanto son comunes para todos los grafos. □

Como ya hemos visto, las instancias se pueden dividir en dos tipos, según el conjunto de posiciones del multiselector que representan:

- *Instancia simple* es aquella que representa exactamente una posición del array y por tanto puede ser manejada como un selector simple. La instancia ins es simple si $ivs(ins) \neq \emptyset$.

Es decir, si en el identificador de la instancia aparece una *ivar*, la instancia está representando únicamente las posiciones concretas que dicha *ivar* pueda seleccionar del array de punteros.

- *Instancia múltiple* representa más de una posición del array de punteros. Una instancia ins es múltiple si $ivs(ins) = \emptyset$.

  Si no aparece ninguna *ivar* en el identificador de la instancia, la instancia puede representar a la vez más de una posición del array.

En la figura 4.7 mostramos un posible RSG, rsg, que representa la configuración de memoria mostrada en la figura 4.6. Tenemos que indicar, que en el RSG de la figura 4.7, los nodos circulares representan instancias del multiselector que los apunta (en este caso *child*). Dentro de dichas instancias, aparece el conjunto de identificadores de dichas instancias. Cuando aparece *va(i)* quiere indicar que la *ivar* i pertenece al conjunto *vivs* de dicha instancia. Los conjuntos que definen rsg son los siguientes: $N(rsg) = \{n_1, n_2, n_3\}$, $P(rsg) = \{tree\}$, $IV(rsg) = \{i\}$, $S(rsg) = \emptyset$, $MS(rsg) = \{child\}$, $NL(rsg) = \emptyset$ y $MSL(rsg) = \{< n_1, child, < \emptyset, \{i\} >, mc_1, n_2 >, < n_1, child, < \emptyset, \emptyset >, mc_1, n_2 >, < n_1, child, < \emptyset, \emptyset >, mc_1, NULL >, < n_2, child, < \{i\}, \emptyset >, mc_3, n_3 >, < n_2, child, < \emptyset, \emptyset >, mc_2, n_3 >, < n_2, child, < \emptyset, \emptyset >, mc_2, NULL >, < n_2, child, < \emptyset, \emptyset >, mc_3, NULL >, < n_3, child, < \emptyset, \emptyset >, mc_4, NULL >\}$

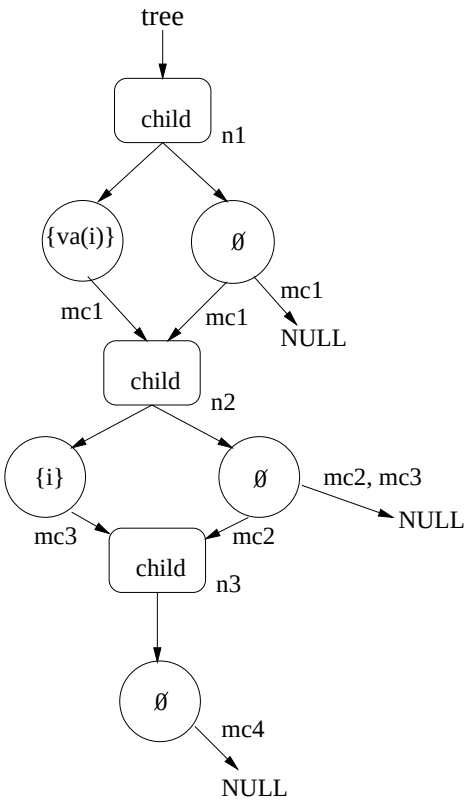


Figura 4.7: Ejemplo de RSG con multiselectores.

Como podemos apreciar, el multiselector *child* del nodo n_1 tiene dos instancias ($< \emptyset, \{i\} >$ representada en el grafo como $\{va(i)\}$ y $< \emptyset, \emptyset >$ representada como $\emptyset$). Ambas instancias son múltiples, ya que no existe ninguna *ivar* en sus conjuntos *ivs*. Esto quiere decir que ambas instancias representa varias de las posiciones del array de punteros de las porciones de memoria representadas por el nodo n_1. La única instancia *simple* que aparece en el grafo, es

la instancia $< i, \emptyset >$ (representada con $\{i\}$ en la grafo de la figura) del multiselector *child* del nodo n_2. En este caso, dicha instancia representa a aquellas posiciones del array cuyo valor puede tomar i. Como ya hemos comentado anteriormente, esta instancia *simple* puede ser tratada como un selector simple puesto que en cada configuración de memoria representada por el grafo, representa un único enlace.

Una vez presentados de una manera formal los multiselectores dentro de los RSRSGs, vamos a ver las propiedades de los nodos que se ven afectadas por dicha inserción.

4.2.1 Los multiselectores y las propiedades de los nodos

En esta sección vamos a presentar tan solo las modificaciones sobre aquellas propiedades que se ven afectadas por el soporte de los multiselectores en los RSRSGs. Las propiedades afectadas, son las que tienen una relación directa con los enlaces de la estructura de datos. Así, las dos únicas que no se ven afectadas son la propiedad *type* ya que depende del tipo de la variable puntero que genera el nodo y *touch* que depende de las variables puntero que "visitan" un nodo. Las restantes propiedades se ven modificadas de la forma que se presenta a continuación.

Propiedad STRUCTURE

Recordamos que esta propiedad mantiene un valor común para todos aquellos nodos pertenecientes a una misma componente conexa de las estructuras representadas por los grafos. De esta forma, se mantienen en nodos distintos, las porciones de memoria pertenecientes a estructuras que no comparten ningún elemento.

A la hora de determinar si dos porciones de memoria pertenecen a la misma *estructura*, nos basábamos en la existencia de una tercera porción desde la cual existen caminos (sucesión de porciones y enlaces) hacia las dos porciones anteriores. Por lo tanto, la modificación necesaria para el soporte de los multiselectores, es que estos caminos pueden estar formados por sucesiones de porciones y enlaces que pueden ser o bien a través de selectores o bien a través de multiselectores.

Formalmente,

$$\begin{gathered} STRUCTURE(n_a) = STRUCTURE(n_b) = val \text{ si } \forall l_a, F_n(l_a) = n_a, \forall l_b, F_n(l_b) = n_b, \\ \exists l_n, l_1, ..., l_i, l'_1, ..., l'_i \in L \mid \\ (REF(l_n, l_1) = 1 \ \wedge \ REF(l_1, l_2) = 1 \ \wedge \ ... \ \wedge \ REF(l_i, l_a) = 1) \ \wedge \\ (REF(l_n, l'_1) = 1 \ \wedge \ REF(l'_1, l'_2) = 1 \ \wedge \ ... \ \wedge \ REF(l'_i, l_b) = 1) \end{gathered}$$

siendo $F_n(l) = n$ la función que devuelve el nodo n que representa la porción de memoria l y $REF(l_1, l_2)$ una función booleana que devuelve *true* si existe un enlace desde la porción l_1 a la porción l_2, ya sea por un selector o por un multiselector:

$$REF(l_1, l_2) = \begin{cases} 1 \text{ si } \exists < l_1, sel_i, l_2 > \in LS \ \vee \ \exists < l_1, ms_i, ind_j, l_2 > \in LMS \\ 0 \text{ en caso contrario.} \end{cases}$$

Propiedad SIMPLE PATHS

Como ya vimos, los *simple paths* son caminos de acceso a los nodos, utilizando como máximo un *selector* para llegar a dicho nodo a partir de una variable puntero. Con la inserción de los multiselectores, nos encontramos que ahora en los *simple paths* pueden aparecer multiselectores además de selectores simples.

Por ejemplo, si la variable x apunta al nodo n_x, éste a su vez apunta a n_1 por medio de un puntero de un array de punteros o multiselector ms_1, tenemos un camino de acceso de longitud uno al nodo n_1 de la forma $<x, ms_1>$.

Formalmente, definimos de nuevo la propiedad *SPATH* para un nodo $n \in N(rsg)$ como:

$$SPATH(n) = \{sp_1, ..., sp_n\} \text{ donde } sp_i =< pv, sel^* > \text{ con } pv \in P, donde$$

$$sel^* = \begin{cases} \emptyset \text{ si } < pv, n > \in PL(rsg) \\ sel \in S \text{ si } \exists n_i \in N(rsg), < pv, n_i > \in PL(rsg) \wedge \\ \quad < n_i, sel, n > \in NL(rsg) \wedge (\nexists pv_i \in P, < pv_i, n > \in PL(rsg)) \\ ms \in MS \text{ si } \exists n_i \in N(rsg), < pv, n_i > \in PL(rsg) \wedge \\ \quad < n_i, ms, ins, mc, n > \in MSL(rsg) \wedge (\nexists pv_i \in P, < pv_i, n > \in PL(rsg)) \end{cases}$$

donde el cambio respecto a la definición del capítulo anterior es la posibilidad de que el enlace provenga de un multiselector, en vez de que sólo pudiera ser de un selector.

Propiedad REFERENCE PATTERNS

Esta propiedad hace referencia a los *selectores* que apuntan o pueden apuntar a un nodo, y por los que apunta o puede apuntar dicho nodo a otros. Esta información se mantenía por medio de cuatro conjuntos de selectores, *SELINset*, *SELOUTset*, *PosSELINset* y *PosSELOUTset*. Ahora, en dichos conjuntos además de aparecer selectores podrán aparecer multiselectores, si un determinado nodo es apuntado o puede ser apuntado por un determinado multiselector, o si apunta o puede apuntar a otros nodos por dicho multiselector.

Formalmente, para un nodo $n \in N(rsg)$ los conjuntos queda definidos como:

$$\begin{aligned} \cdot\ & SELINset(n) = \{sel_i \in S \mid \forall l, F_n(l) = n,\ \exists l_i \in L(m), < l_i, sel_i, l > \in LS(m)\} \cup \\ & \{ms_i \in MS \mid \forall l, F_n(l) = n,\ \exists l_i \in L(m), < l_i, ms_i, ind, l > \in LMS(m)\} \\ \cdot\ & SELOUTset(n) = \{sel_i \in S \mid \forall l, F_n(l) = n,\ \exists l_i \in L(m), < l, sel_i, l_i > \in LS(m)\} \cup \\ & \{ms_i \in MS \mid \forall l, F_n(l) = n,\ \nexists < l, ms_i, ind, NULL > \in LMS(m)\} \\ \cdot\ & PosSELINset(n) = \{sel_i \in S \mid \exists l, F_n(l) = n,\ \exists l_i \in L(m), < l_i, sel_i, l > \in LS(m)\} \cup \\ & \{ms_i \in MS \mid \exists l, F_n(l) = n,\ \exists l_i \in L(m), < l_i, ms_i, ind, l > \in LMS(m)\} \\ \cdot\ & PosSELOUTset(l) = \{sel_i \in S \mid \exists l, F_n(l) = n,\ \exists l_i \in L(m), < l, sel_i, l_i > \in LS(m)\} \cup \\ & \{ms_i \in MS \mid (\exists l_1, F_n(l_1) = n,\ \exists < l_1, ms_i, ind_1, NULL > \in LMS(m) \wedge \\ & (\exists l_2, F_n(l_2) = n,\ \exists l_3 \in L(m), < l_2, ms_i, ind_2, l_3 > \in LMS(m))\} \end{aligned}$$

Vemos que las diferencias con las definiciones de dichos conjuntos en el capítulo anterior, son que ahora se permite la existencia de multiselectores en dichos conjuntos. Así, un multiselector ms aparecerá en los conjuntos *SELINset* o *PosSELINset*, si el nodo al que pertenecen, definitivamente es referenciado por un enlace perteneciente a un multiselector ms

o es posible que lo sea (algunas porciones de memoria representadas por el nodo lo son y otras no). Para los enlaces de "salida", almacenados en *SELOUTset* y *PosSELOUTset*, la cosa cambia. Ahora, para que un multiselector *ms* pertenezca a *SELOUTset* de un nodo, se debe cumplir que todas las posiciones del array representado por *ms* tengan un enlace hacia otra porción de memoria, no puede haber ninguna con el valor $NULL$. Por el contrario para que pertenezca a *PosSELOUTset*, debe existir alguna posición que apunte a una porción de memoria y debe existir alguna otra que sea $NULL$, ya sean de la misma porción de memoria o de varias representadas por el nodo. El multiselector *ms* no pertenece a ninguno de los dos conjuntos, si todas las posiciones del array son $NULL$, es decir, no existe ningún enlace por el multiselector *ms*.

Información SHARED

Esta información comprende dos propiedades que indican si alguna de las porciones de memoria representadas por un nodo puede o no ser referenciada más de una vez por distintos selectores (*SHARED*) o por un mismo selector (*SHSEL*). Ahora, dentro de esta información tendrán cabida los multiselectores, si alguna de las porciones puede ser apuntada por varios enlaces y alguno de ellos pertenece a una posición de algún multiselector.

Cada una de las nuevas propiedades queda como sigue:

- *SHARED*(n) con $n \in N(rsg)$, es una función booleana que devolverá *true*, si alguna de las porciones de memoria l_1 representadas en n, puede ser referenciada por otras porciones por medio de selectores o multiselectores distintos.

$$SHARED(n) = \begin{cases} 0 \text{ si } \nexists l, l_1, l_2 \in L \mid (F_n(l) = n) \wedge \\ \quad (((< l_1, sel_1, l >, < l_2, sel_2, l > \in LS) \wedge (sel_1 \neq sel_2)) \vee \\ \quad ((< l_1, ms_i, ind_i, l >, < l_2, ms_j, ind_j, l > \in LMS) \wedge \\ \quad (ms_i \neq ms_j)) \vee \\ \quad (< l_1, sel_i, l > \in LS, < l_2, ms_j, ind_j, l > \in LMS)) \\ 1 \text{ en caso contrario (puede haber una porción de} \\ \quad \text{memoria representada por } n \text{ que sea referenciada} \\ \quad \text{por diferentes selectores y/o multiselectores).} \end{cases}$$

- *SHSEL*(n, sel) con $n \in N(rsg)$ y $sel \in S \cup MS$, es una función booleana que devuelve *true* si alguna de las porciones de memoria representadas por el nodo n puede ser referenciada más de una vez por el selector o multiselector sel, desde otras porciones de memoria.

$$SHSEL(n, sel) = \begin{cases} 0 \text{ si } \nexists l, l_1, l_2 \in L \mid (F_n(l) = n) \ \wedge ((sel \in S) \wedge \\ \quad (< l_1, sel, l >, < l_2, sel, l > \in LS \ \wedge \ l_1 \neq l_2)) \ \vee ((sel \in MS) \\ \quad \wedge (< l_1, sel, ind_1, l >, < l_2, sel, ind_2, l > \in LMS)) \\ 1 \text{ en caso contrario (puede haber una porción de} \\ \quad \text{memoria en } n \text{ referenciada por el selector o} \\ \quad \text{multiselector } sel, \text{ varias veces).} \end{cases}$$

Propiedad CYCLELINKS

Como recordamos, esta propiedad mantenía parejas de selectores que formaban ciclos entre dos porciones de memoria, siguiendo consecutivamente los selectores de la pareja. Con la inserción de los multiselectores, ahora es posible que algún componente de un *cyclelink* sea un multiselector. Como veremos, la inserción de un multiselector en un *cyclelink* está más restringida que la de un selector normal.

$$\begin{aligned} CYCLELINKS(n) = \{ &< sel_i, sel_j > \mid sel_i, sel_j \in S \wedge (\forall l_i,\ F_n(l_i) = n, \\ &\text{si } \exists l_j \in L, < l_i, sel_i, l_j > \in LS \text{ entonces } \exists < l_j, sel_j, l_i > \in LS)\} \cup \\ &\{< ms_i, sel_j > \mid ms_i \in MS, sel_j \in S \wedge (\forall l_i,\ F_n(l_i) = n, \\ &\text{si } \exists l_j \in L, < l_i, ms_i, ind_j, l_j > \in MLS \text{ entonces } \exists < l_j, sel_j, l_i > \in LS)\} \end{aligned}$$

Como podemos observar, los multiselectores sólo pueden aparecer como el primer elemento de la pareja de un *cyclelink*. Es decir, se recoge información del tipo: si en un nodo n, $<ms_i, sel_j> \in CYCLELINKS(n)$, entonces siempre que referencie a otro nodo n_j por ms_i, se sabe que n_j debe referenciar a n por sel_j. Un ejemplo de este tipo de relaciones las podemos encontrar en un árbol donde los hijos de un nodo se almacenan en un array de punteros, y cada nodo hijo tiene un selector al padre. Sin embargo, el caso contrario, en el que el multiselector ms_i aparezca como segundo elemento del par no está contemplado, ya que haría referencia a casos de estructuras extremadamente raros y sería una información muy costosa de mantener. Un cyclelink de la forma $<sel_j, ms_i>$ en un nodo n indicaría que los nodos alcanzados por sel_j desde n, referenciarían de nuevo a n por medio de todos los enlaces que representa el multiselector ms_i, es decir, absolutamente todas las posiciones del array de punteros representado por ms_i deberían apuntar a n.

Una vez vistas las propiedades afectadas por la inclusión de los multiselectores, pasamos a ver como se mantiene de tamaño finito un RSG que contiene multiselectores.

4.2.2 Compresión de un RSG con multiselectores

En la sección 3.2.10 del capítulo 3, vimos que una vez definidas todas las propiedades, dependiendo del valor que tomaran para cada porción de memoria, la función F_n las trasladaba a nodos de un grafo, de forma que porciones con propiedades similares se mapeaban todas sobre el mismo nodo.

También se vio que para que el tamaño de los grafos no crezca y se mantenga finito, cuando son modificados por la semántica abstracta de las sentencias, son *comprimidos*, ya que puede haber nodos que antes de las modificaciones no eran compatibles y después de las mismas si. En este caso, los nodos compatibles son sumarizados en uno solo que recoge la información y selectores de los dos originales.

En la sección 3.2.10 se presentaba la función $COMPRESS\_RSG(rsg)$ que es la encargada de comprimir el grafo rsg buscando nodos compatibles. Para determinar si dos nodos de un mismo grafo son compatibles, se definió la función C_NODES_RSG, que comparaba las propiedades de ambos nodos para ver si son similares. La mayoría de las propiedades tienen que ser iguales para que los nodos sean compatibles. Esto no varía con la inserción de los multiselectores: si aparece algún multiselector en alguna de esas propiedades para un nodo, también deberá aparecer para la misma propiedad en el otro nodo a comparar.

Sin embargo, para comparar los *reference patterns* de los nodos, C_NODES_RSG hace

uso de otra función booleana, $C\_REFPATH(n_i, n_j)$ que devuelve *true* si la propiedad *reference patterns* es compatible en los nodos n_i y n_j. Esta función se ve afectada levemente, quedando como sigue:

$$C\_REFPAT(n_i, n_j) =$$
$$\begin{cases} 1 \text{ si } \forall sel_i \in S \cup MS, (SELIN(n_i, sel_i) = SELIN(n_j, sel_i) \ \vee \ SELIN(n_i, sel_i) = 2 \\ \quad \vee \ SELIN(n_j, sel_i) = 2) \ \wedge \ (SELOUT(n_i, sel_i) = SELOUT(n_j, sel_i) \\ \quad \vee \ SELOUT(n_i, sel_i) = 2 \ \vee \ SELOUT(n_j, sel_i) = 2) \\ 0 \text{ en otro caso.} \end{cases}$$

en donde la única diferencia es que ahora sel_i puede ser un selector o un multiselector.

Una vez definida la compatibilidad entre los nodos de un RSG donde existen multiselectores, ya se pueden comprimir los grafos con la función $COMPRESS\_RSG(rsg) = rsg_c$, que genera el nuevo grafo rsg_c a partir de rsg uniendo aquellos nodos compatibles (C_NODES_RSG).

Para la unión de dos nodos, la función $COMPRESS\_RSG(rsg) = rsg_c$ hace uso de otra función, $MERGE\_NODES(n_1, n_2) = n$, que devuelve las propiedades del nodo n que representa a los nodos compatibles n_1 y n_2 tras la sumarización. Todo sigue igual en esta función, a excepción de la generación de las propiedades *SHSEL* y *CYCLELINKS*, que quedan de la siguiente manera:

- $SHSEL(n, sel_i) = SHSEL(n_1, sel_i) = SHSEL(n_2, sel_i) \ \forall sel_i \in S \cup MS$
- $$\begin{aligned} &CYCLELINKS(n) = \{< sel_i, sel_j > \ | \\ &< sel_i, sel_j > \in (CYCLELINKS(n_1) \cap CYCLELINKS(n_2)) \ \vee \\ &< sel_i, sel_j > \in CYCLELINKS(n_1) \ \wedge \ \nexists n_k \in N(rsg), \\ &\qquad (< n_2, sel_i, n_k > \in NL(rsg) \ \vee \ < n_2, sel_i, ins_j, mc_k, n_k > \in MSL(rsg)) \ \vee \\ &< sel_i, sel_j > \in CYCLELINKS(n_2) \ \wedge \ \nexists n_k \in N(rsg), \\ &\qquad (< n_1, sel_i, n_k > \in NL(rsg) \ \vee \ < n_1, sel_i, ins_j, mc_k, n_k > \in MSL(rsg))\} \end{aligned}$$

En la propiedad *SHSEL* ahora sel_i puede ser un selector o un multiselector, y en *CYCLELINKS*, ahora puede aparecer un multiselector como primer elemento del par.

La función de compresión, $COMPRESS\_RSG(rsg) = rsg_c$, se ve completada con la definición del conjunto de multiselectores del nuevo grafo, rsg_c, quedando de la siguiente manera:

$$\begin{aligned} MSL(rsg_c) = \{&< MAP\_RSG(n_i), ms, ins, mc', MAP\_RSG(n_j) > \ | \\ &< n_i, ms, ins, mc, n_j > \in MSL(rsg)\} \end{aligned}$$

como se puede observar, se cambian los nodos origen y destino de los multiselectores usando la función $MAP\_RSG(n)$ que devuelve el nodo de rsg_c que representa al nodo n de rsg. Además, la clase de multireferencia a la que pertenece cada enlace, se cambia, asignado a cada mc de los multiselectores antiguos una nueva mc', de forma que las nuevas clases sean distintas para los multiselectores que provienen de cada nodo. De esta forma, los enlaces pertenecientes a las distintas clases de un nodo no se mezclarán con los enlaces pertenecientes a las del otro nodo. Vemos pues como se van generando nuevas clases de multireferencias que nos permiten distinguir que conjuntos de enlaces pueden aparecer todos juntos en una misma porción de memoria representada por el nuevo nodo.

Como se comentó en la sección 4.1, si se dan nuevos identificadores de clases diferentes cada vez, el número de estos crecería indefinidamente. Como vimos, en realidad el número de clases de multireferencias distintas está limitado, ya que en definitiva una mc es un conjunto de enlaces de un multiselector, y el número de enlaces en un RSG está limitado. Esto implicaba, que dos clases distintas mc_1 y mc_2 serán la misma, si comprenden el mismo conjunto de enlaces del multiselector. En este caso se puede eliminar una de las dos clases. Por tanto, tras la compresión del grafo, se aplica un proceso de eliminación de clases de multireferencias repetidas, para mantener finito en número de clases de cada multiselector.

Definimos $MERGE\_MC(rsg)$, como la función que modifica las clases de multireferencias de los multiselectores del grafo rsg, eliminando clases repetidas. La modificación del conjunto de multiselectores de rsg es la siguiente:

$$MSL(rsg) = \{< n, ms_i, ins_j, MAP\_MC(mc_k), n_d > \mid \\ < n, ms_i, ins_j, mc_k, n_d > \in MSL(rsg)\}$$

donde

$$MAP\_MC(mc_k) = \begin{cases} mc_m \text{ si } (\forall < n, ms_i, ins_j, mc_k, n_d > \exists < n, ms_i, ins_j, mc_m, n_d >) \\ \qquad \wedge (\forall < n, ms_i, ins_j, mc_m, n_d > \ \exists \ < n, ms_i, ins_j, mc_k, n_d >) \\ \qquad \wedge (mc_m < mc_k) \\ mc_k \text{ en caso contrario.} \end{cases}$$

Con $MAP\_MC(mc_k)$ renombramos aquellas clases de multireferencias que aparecen exactamente en los mismos enlaces que otra, y el el identificador de la otra clase es menor.

Por tanto la nueva función de compresión quedaría como $COMPRESS\_RSG(rsg) = MERGE\_MC(rsg_c)$. De este modo queda completamente definida la función de compresión de grafos con la presencia de multiselectores en los mismos.

4.3 Los multiselectores y la semántica abstracta

La semántica abstracta de las sentencias, como ya hemos visto, especifica para cada sentencia los cambios producidos en los grafos de un RSRSG tras ser analizados por dicha sentencia. Antes de comentar los cambios producidos sobre la semántica abstracta y sobre la interpretación de las sentencias sobre los grafos, debidos a la inclusión de los multiselectores, vamos a ver en que afectan estos multiselectores a la hora de mantener "reducido" el RSRSG asociado a cada sentencia.

4.3.1 Compresión de los RSRSGs

El método mantiene un conjunto reducido de grafos (RSRSG) para cada sentencia del programa, representando todas las posibles configuraciones de memoria que se pueden dar tras la ejecución de dicha sentencia. Como vimos en la sección 3.3, los grafos de un RSRSG que cumplan una serie de condiciones, se dice que son compatibles, y es factible su unión en un único grafo que va a representar todas las configuraciones de memoria representadas anteriormente por los grafos originales.

Para unir los grafos compatibles de un RSRSG y así reducir su número, se utiliza la función $COMPRESS\_\ RSRSG(rsrsg)$, que comprueba los grafos que son compatibles con la función $COMPATIBLE(rsg_1, rsg_2)$ y los une con la función $MERGE\_RSG(rsg_1, rsg_2)$.

La compatibilidad entre dos grafos, vimos que dependía de sus relaciones de alias y de la compatibilidad de ciertos nodos en ambos grafos. La inclusión de los multiselectores no afecta para nada a las funciones utilizadas para determinar la compatibilidad entre dos grafos, por tanto la función $COMPATIBLE(rsg_1, rsg_2)$ no se ve afectada.

Sin embargo, la función $MERGE\_RSG(rsg_1, rsg_2) = rsg$, genera un nuevo grafo uniendo los grafos rsg_1 y rsg_2. Los conjuntos $N(rsg)$, $PL(rsg)$ y $NL(rsg)$ del nuevo grafo se generan de la misma manera, la única modificación que supone la inclusión de los multiselectores en esta función, es especificar como se genera el conjunto de multiselectores del nuevo grafo, $MSL(rsg)$. Este conjunto queda definido de la siguiente manera:

$$\begin{gathered} MSL(rsg) = \{< MAP(n_i), ms_i, ins_j, mc'_k, MAP(n_k) > \mid \\ \forall < n_i, ms_i, ins_j, mc_k, n_k > \in MSL(rsg_1)\} \cup \\ \{< MAP(n_i), ms_i, ins_j, mc'_k, MAP(n_k) > \mid \forall < n_i, ms_i, ins_j, mc_k, n_k > \in MSL(rsg_2)\} \end{gathered}$$

donde la función $MAP(n)$ devuelve el nodo del nuevo grafo que representa al nodo n perteneciente a alguno de los grafos a unir. Al igual que con $COMPRESS\_RSG$, a la hora de copiar los multiselectores de rsg_1 y rsg_2 en el nuevo grafo, las clases de multireferencias, mc_k, se cambian por otras nuevas, mc'_k, de forma que no coincidan las de un grafo con las del otro. Por este motivo, para mantener reducido el número de clases de multireferencias, hay que aplicar la función $MERGE\_MC$ al nuevo grafo, de forma que:

$$MERGE\_RSG(rsg_1, rsg_2) = MERGE\_MC(rsg)$$

De esta forma la compresión de los RSRSGs asociados a cada sentencia queda definida para grafos con multiselectores. Pasamos a continuación a describir como quedará la interpretación abstracta de las sentencias durante el análisis.

4.3.2 Interpretación abstracta de las sentencias

En la sección 3.4.1 del capítulo anterior, se presentó la manera en que las sentencias eran interpretadas sobre los grafos para generar los cambios en los mismos producidos por la ejecución de dicha sentencia.

Vimos que antes de poder modificar los grafos por la acción de una sentencia, el enlace referenciado en la misma era "enfocado" para que su manejo fuera lo más preciso posible. Este proceso de enfoque consiste en la aplicación de una serie de operaciones sobre los grafos: *división* y *poda* de los grafos, y *materialización* de grafos y nodos cuando era necesario. El sentido de todas estas operaciones está ampliamente descrito en la sección 3.4.1. En la figura 3.14 de dicha sección, se presenta de forma esquemática las distintas fases para la obtención del $RSRSG_{out}$ tras la interpretación abstracta de una sentencia a partir de un $RSRSG_{in}$ de entrada.

Este proceso se va a ver modificado cuando la sentencia haga referencia a un enlace de un multiselector. La diferencia fundamental radica en localizar el enlace adecuado para llevar a cabo las transformaciones, ya que ahora el multiselector representa a un conjunto de enlaces para una proción de memoria determinada. En el caso de selectores simples, $x \rightarrow sel$ representa un único enlace para cada porción de memoria representada por el nodo apuntado por x. Por tanto si existen varios selectores en el nodo apuntado por x se sabe que en realidad

pertenecen a porciones de memoria distintas y se pueden aislar en grafos distintos mediante la operación $DIVIDE$, para aplicar la semántica abstracta a cada enlace concreto.

Sin embargo para multiselectores, $x \rightarrow sel[i]$, el multiselector sel, está representando múltiples enlaces pertenecientes a la misma porción de memoria apuntada por x. Por tanto antes de "enfocar" el enlace, como se hace con los selectores simples, hay que "enfocar" los enlaces de la posición concreta del array referenciada en la sentencia (i). Es decir, aislar de todos los enlaces del multiselector sel del nodo apuntado por x, aquellos enlaces que realmente puedan ser referenciados por la sentencia $x \rightarrow sel[i]$. De esta forma, las posteriores modificaciones se llevarán a cabo sobre los enlaces que realmente puedan ser referenciados en la sentencia y no sobre todos los del multiselector.

Por tanto, se necesita aplicar una fase de "pre-enfoque" para obtener una instancia del multiselector que represente los posibles enlaces referenciados en la sentencia, de modo que puedan ser tratados de igual forma que lo son los selectores normales.

A grandes rasgos, para las sentencias que utilizan multiselectores, antes de introducir los grafos en la fase de *División* y *Poda* (figura 3.14) las cuales se deben aplicar sobre selectores simples, hay que "pre-enfocar" los multiselectores, transformando los grafos para que a la fase de "enfoque" lleguen grafos en los que $x \rightarrow sel[i]$ represente enlaces simples (*instancia simple*).

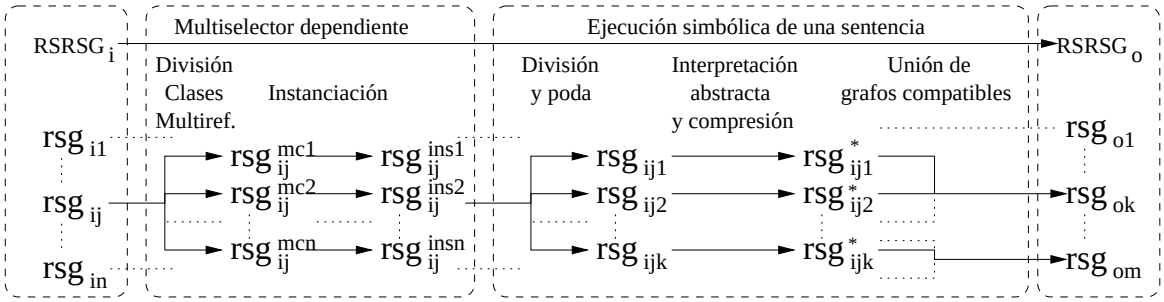


Figura 4.8: Descripción esquemática de la ejecución simbólica de una sentencia con multiselectores.

En la figura 4.8 se muestra un esquema similar al de la figura 3.14, con la nueva descripción esquemática de la ejecución simbólica de sentencias en las que se ve involucrado algún multiselector. Como se puede apreciar, la diferencia fundamental con la anterior es la inclusión de una fase de "pre-enfoque" sólo para las instrucciones que usan multiselectores, antes de la ejecución simbólica de la sentencia ya presentada en el capítulo anterior. Vemos que esta nueva fase, aplicada a las sentencias con referencias del tipo $x \rightarrow sel[i]$, consta de dos operaciones: la *división por clases de multireferencias* y la *instanciación*.

1. **División por clases de multireferencias**: Como vimos al principio de este capítulo, una clase de multireferencia es una agrupación de enlaces de un selector, que pueden pertenecer todos a la vez a una porción de memoria. La *división por clases de multireferencias* consistirá en separar en distintos grafos las clases de multireferencias del multiselector sel del nodo apuntado por x. De este modo, en cada grafo, dicho multiselector sólo tendrá enlaces que realmente pueden aparecer juntos en alguna porción de memoria.

2. **Instanciación**: Una vez que en el multiselector en cuestión tan solo aparecen los enlaces que pueden aparecer juntos, hay que crear una instancia simple del multiselector que

represente a todas las posibles posiciones individuales del array de punteros, que puede direccionar i. De este modo, los enlaces de esta instancia podrán ser considerados exactamente igual que los enlaces de un selector normal. Por este motivo, una vez instanciada, se puede pasar a la fase de *división* y *poda* normales, para enfocar un enlace en concreto (ver figura 4.8).

A continuación pasamos a ver más detalladamente cada una de estas dos operaciones de la fase de "pre-enfoque".

División por clases de multireferencias

Antes de describir formalmente la división por clases de multireferencias, vamos a comentar algunos aspectos importantes relacionados con dichas clases de multireferencias.

Como se introdujo en la sección 4.1, todo multiselector $<n_s, sel, ins, mc, n_d>$ pertenece a una clase de multireferencia mc. Estas clases se construyen para indicar que no todos los enlaces que aparecen en un multiselector de un nodo pueden pertenecer a una misma porción de memoria representada por dicho nodo. tan solo pueden aparecer juntos aquellos enlaces que pertenezcan a la misma clase de multireferencia mc. Las clases de multireferencias se crean cuando se sumarizan varios nodos con multiselectores, para mantener información en el nodo sumario de las configuraciones de enlaces que existían en los nodos sumarizados. En la sección 4.2.2 hemos presentado la nueva función $COMPRESS\_RSG$ que es la encargada de sumarizar nodos. Hemos presentado la manera en que las clases de multireferencias son renombradas para que en la unión no se pierdan las configuraciones de los nodos unidos. También hemos descrito el proceso para mantener finito el número de clases de multireferencias de un multiselector.

Pasamos a presentar ahora, la división de un grafo para separar las diferentes clases de multireferencias y así aislar las distintas configuraciones de enlaces que realmente se pueden dar para un determinado multiselector.

Dada la referencia $x \rightarrow sel[i]$ que aparece en una sentencia, definimos MC_DIVIDE (x, sel, rsg) con $x \in P$ y $sel \in MS$ como $\{rsg_1, ..., rsg_n\}$. La división crea una serie de grafos de la siguiente manera. Sea $n \in N(rsg) \mid \ < x, n > \in PL(rsg)$, entonces $\forall mc_i, <$ $n, sel, ins_j, mc_i, n_d > \in MSL(rsg)$ creamos un nuevo grafo rsg'_i de modo que:

- $N(rsg'_i) = N(rsg)$
- $PL(rsg'_i) = PL(rsg)$
- $NL(rsg'_i) = NL(rsg)$
- $MSL(rsg'_i) = MSL(rsg) \setminus \{< n, sel, ins_k, mc_m, n_e > \in MSL(rsg), \forall mc_m \neq mc_i\}$

Vemos que los grafos rsg'_i son copias de rsg, eliminando aquellos multiselectores desde el nodo n por sel que pertenecen a una clase distinta a mc_i (clase asociada al grafo actual rsg'_i). Por tanto en el nuevo grafo tan solo quedarán los enlaces pertenecientes a esta clase de multireferencia. Aparecerán tantos grafos como clases distintas existan para el multiselector sel en el nodo n.

Para obtener el definitivo rsg_i, el grafo es *podado* para eliminar nodos y enlaces pertenecientes a configuraciones de memoria distintas a la que pertenece mc_i y que por tanto

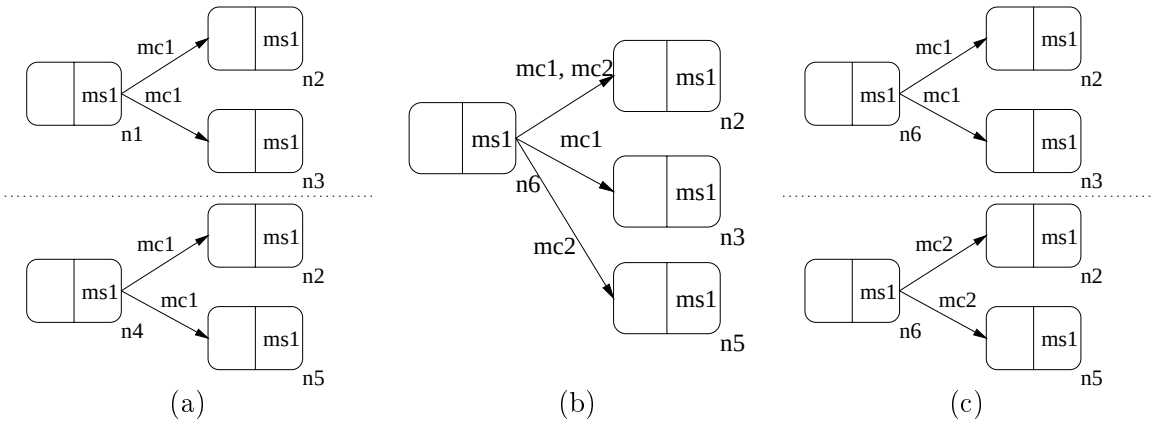


Figura 4.9: Ejemplo de sumarización y división por clases de multireferencias.

serán representados en otro de los grafos resultantes de la división. Esto aumenta la precisión con que las configuraciones de memoria son representadas en el grafo, haciendo su posterior manipulación más exacta. De este modo $rsg_i = PRUNE(rsg'_i)$.

En el ejemplo de la figura 4.9 podemos observar como las operaciones de sumarización y de división afectan a las clases de multireferencias. En 4.9(a) vemos dos porciones de memoria pertenecientes a dos grafos. Al unirlos, los nodos n_1 y n_4 que son compatibles, se sumarizan en un nuevo nodo n_6 (4.9(b)). Como se puede apreciar, el nuevo nodo n_6 conserva los enlaces por el multiselector ms_1 que poseían tanto n_1 como n_4, aunque cada uno con distinta clase de multireferencia (mc_1 para los enlaces que tenía n_1 y mc_2 para los de n_4). Vemos que el enlace por ms_1 desde el nodo n_6 al nodo n_2 pertenece a dos clases de multireferencias. Esto es debido a que dicho enlace aparece tanto en el nodo n_1 del primer grafo, como en el nodo n_4 del segundo. En la figura 4.9(c) vemos el resultado de la operación MC_DIVIDE. Como hemos dicho, se crea un grafo para cada clase de multireferencia (mc_1 y mc_2), eliminado los enlaces que no pertenecen a la clase por la que se divide. Como se puede observar, se obtienen los mismos conjuntos de enlaces para el multiselector que existían en los grafos originales.

Instanciación

Al igual que con la división por clases de multireferencias, antes de pasar a describir el proceso de instanciación, vamos a presentar una serie de elementos relacionados con las instancias que nos van a ayudar a comprender su funcionamiento.

Como hemos visto anteriormente, el identificador de una instancia $ins =< ivs, vivs >$, está compuesto por dos conjuntos de variables índice (IV). El conjunto ivs es el conjunto de $ivars$, las cuales en la sentencia a la que pertenecen las configuraciones de memoria representadas por el grafo al que pertenece ins, pueden tener como valor las posiciones del array que representa la instancia ins. Mientras que el conjunto $vivs$ está formado por aquellas que tomaron dicho valor en algún momento de la ejecución antes de llegar a la sentencia actual, también representadas como *va(ivar)*.

Cuando instanciamos un multiselector pretendemos aislar los enlaces que pueden tener unas posiciones del array determinadas y que así puedan ser manejados como si de selectores normales se tratara. Como ya vimos al principio del capítulo, para la selección más precisa de los enlaces que pueden aparecer en una nueva instancia, es fundamental conocer las relaciones existentes entre las variables índice.

Relaciones entre variables índice. En una fase de preproceso del código, se crea una tabla de relaciones entre variables índice denominada $IVRT$. El valor de $IVRT(iv_1, iv_2, st)$, donde $iv_1, iv_2 \in IV(rsg)$, o son el valor anterior de alguna variable índice ($iv_1 = va(iv_1')$) y st es la sentencia actual, es la relación existente entre ambas $ivars$ o valores anteriores de las mismas. Los valores que puede tomar $IVRT(iv_1, iv_2, st)$ son:

- **eq** si el valor de iv_1 es igual al valor de iv_2 en la sentencia st, con $iv_1, iv_2 \in IV(rsg)$. Como ya dijimos, la relación de igualdad sólo tiene sentido entre valores actuales de variables índice.

- **neq** si el valor de iv_1 es distinto del valor de iv_2 en la sentencia st. Ahora tanto iv_1 como iv_2 pueden pertenecer a $IV(rsg)$ o ser $va(iv)$, pudiendo indicar que una variable índice en dicha sentencia nunca tomará un valor que previamente haya tenido en esa misma sentencia.

- **unk** si no se sabe la relación existente ente iv_1 y iv_2 en la sentencia st.

La construcción de $IVRT$ se realiza por bloques de código delimitados por asignaciones a variables índice. Dentro de cada bloque, las relaciones entre las $ivars$ no cambian, por lo que tan solo es necesario recalcularlas en cada asignación. No hemos desarrollado un análisis complejo para extraer estas relaciones, sino que hemos aplicado un par de reglas simples que nos dan la suficiente información para el análisis de muchos códigos. Análisis más complejos podrían ser adaptados para determinar las relaciones entre variables utilizadas como índices en los arrays de punteros. Las reglas que hemos utilizado son las siguientes:

- $st : i = j$;

  A partir de esta sentencia, la variable i va a tener las mismas relaciones que la variable j, además de añadirse la relación $i = j$ (eq). La tabla de relaciones quedaría de la siguiente manera:

  · $IVRT(i, j, st) = eq$
  · $IVRT(i, iv, st) = IVRT(j, iv, st),\ \forall\ iv \in IV$

- $st : i = value$;

  Cuando a una $ivar$ se le asocia cualquier valor no relacionado con otra $ivar$, las relaciones de i con las otras $ivars$ dejan de ser conocidas. Por eso se ponen todas con el valor unk

  · $IVRT(i, iv, st) = unk,\ \forall\ iv \in IV$

- $st_1 : for(i = value_1; i < value_2; i++)\{...\}; st_2 : ...$

  En los bucles de este tipo, donde la variable de inducción del bucle es la propia variable índice de los arrays de punteros, se sabe que dentro del cuerpo de dicho bucle, la variable i no va a tomar dos veces el mismo valor. Esto implica que la relación entre i y $va(i)$ sea neq, indicando que los valores anteriores en el bucle de i ($va(i)$) son distintos del valor actual de i.

  Al final del bucle (st_2) esta relación debe de ser eliminada, puesto que fuera del mismo no tiene sentido $va(i)$.

  · $IVRT(i, iv, st_1) = unk,\ \forall\ iv \in IV$
  · $IVRT(i, va(i), st_1) = neq$
  · $IVRT(i, va(i), st_2) = unk$

- $st_1 : for(j = i + value_1; j < value_2; j++)\{...\}; st_2...$

  En este tipo de bucles, además de las mismas relaciones comentadas para el caso anterior, se sabe que la variable j no tomará nunca el mismo valor que la variable i en el cuerpo del bucle (con $value_1$ positivo). Por tanto entre i y j se establecerá la relación neq.

  $\cdot\ IVRT(j, iv, st_1) = unk,\ \forall\ iv \in IV$
  $\cdot\ IVRT(j, va(j), st_1) = neq$
  $\cdot\ IVRT(j, i, st_1) = neq$
  $\cdot\ IVRT(j, va(j), st_2) = unk$

Las relaciones en cualquier sentencia en la que no se asignan valores a ninguna $ivar$, se toman de la sentencia inmediatamente anterior. Cuando una sentencia tenga más de una predecesora (ej. primera sentencia después de un if), se mantienen sólo aquellas relaciones que coinciden en todas las predecesoras, pasando a unk aquellas que no coincidan.

Operación de Instanciación. Cuando nos encontramos con una sentencia st que hace referencia a $x \rightarrow sel[i]$, tras la división por clases de multireferencias, debemos crear una instancia simple del multiselector sel en el nodo n apuntado por x, que represente los enlaces que realmente puede referenciar $x \rightarrow sel[i]$.

Vamos a definir la operación $INSTANCE(x, sel, i, st, rsg) = rsg'$ con $x \in P$, $sel \in MS$, $i \in IV$ y st la sentencia con $x \rightarrow sel[i]$. Sea $n \in N(rsg)$ el nodo tal que $<x, n> \in PL(rsg)$, entonces definimos el conjunto de multiselectores del nuevo grafo rsg' de la siguiente manera:

- Si ya existe una instancia en el multiselector sel del nodo n que contenga a i, no hace falta crear una instancia nueva, puesto que los enlaces relacionados con la posición i del array, ya están recogidos en esta otra instancia.

  si $\exists < n, sel, ins, mc, n_d > \in MSL(rsg), i \in ivs(ins)$:

  $$MSL(rsg') = MSL(rsg)$$

- Si no existe ninguna instancia de sel en n que contenga a i, pero existe alguna que contiene una $ivar$ j cuya relación con i es de igualdad (eq), tampoco se va a crear una instancia nueva. La instancia que contiene a j representa los enlaces de las posiciones que puede tomar j y como $i = j$, esta instancia también representa a las que puede tomar i. En este caso, lo único que se hace es modificar el identificador de la instancia que contiene a j, insertando i.

  si $\exists j \in IV, IVRT(j, i, st) = eq\ \wedge\ \exists < n, sel, ins, mc, n_d > \in MSL(rsg), j \in ivs(ins)$

  $$\begin{array}{c} MSL(rsg') = MSL(rsg)\ \setminus\ \{< n, sel, ins, mc_j, n_d >, \forall mc_j, n_d\} \cup \\ \{< n, sel, ins', mc_j, n_d >, \forall mc_j, n_d \mid ivs(ins') = ivs(ins) \cup \{i\}, \\ vivs(ins') = vivs(ins)\} \end{array}$$

  Los multiselectores de n por sel pertenecientes a la instancia ins que contiene a j, son sustituidos por otros exactamente iguales, excepto cambiando ins por ins', la cual tiene a i en su ivs.

- Si no existe ninguna instancia con i ni con una $ivar$ igual a i (relación eq), se crea una nueva instancia para representar a las posiciones que puede tomar i, por tanto la nueva instancia contendrá a i en su identificador (ivs). Esta nueva instancia, debe tener

todos los enlaces de aquellas instancias que ya existen y que representen posiciones del array que i puede tomar en esta sentencia. Para determinar que instancias son las "compatibles" con i, bastará con mirar sus identificadores. Si la instancia no contiene ninguna $ivar$ o $va(ivar)$ con relación neq con i, entonces será compatible. Es decir, si la instancia contiene a j o a $va(i)$ y la relación entre i y j, o i y $va(i)$ es neq, los enlaces de esta instancia no serán pasados a la nueva. Esto es así puesto que la instancia está representando posiciones del array que no puede tomar en esa sentencia la variable i (relación neq). Por lo tanto, los enlaces que contiene, no pueden ser enlaces de las posiciones representadas por i.

$$MSL(rsg') = MSL(rsg) \cup \{< n, sel, ins_i, mc_j, n_k > \mid ivs(ins_i) = \{i\},$$
$$vivs(ins_i) = \emptyset, \forall < n, sel, ins_l, mc_j, n_k > \in MSL(rsg),$$
$$\nexists iv \in ivs(ins_l) \cup vivs(ins_l) \ \wedge \ IVRT(i, iv, st) = neq\}$$

Con esto queda definida la operación de instanciación de un multiselector.

Ya se han definido la dos operaciones que componen la fase de "pre-enfoque" de las sentencias que utilizan multiselectores. A continuación presentamos un ejemplo de esta fase sobre el grafo presentado en la figura 4.10. El grafo de esta figura, representa un array de punteros, col, apuntado por la variable a, en una sentencia dentro de un bucle donde se crean listas enlazadas y se insertan en las posiciones de dicho array. La variable del bucle, i, también es usada para indexar el array. Tenemos que indicar de nuevo, que las instancias de un multiselector las vamos a representar con nodos circulares, y en su interior pondremos el identificador de la instancia (conjunto de $ivars$ y de $va(ivars)$). Como se puede observar, el multiselector col posee dos instancias, $\{va(i)\}$ que representa todas aquellas posiciones del array por las que ya a pasado i, y $\emptyset$ representando todas aquellas por las que aún no ha pasado.

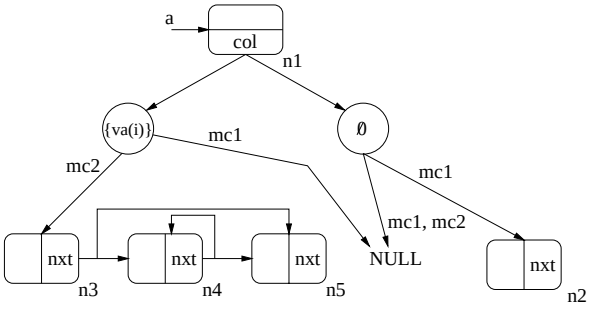


Figura 4.10: Grafo ejemplo para "pre-enfocar" el multiselector $a \rightarrow col[i]$.

Como se puede observar, en los enlaces que parten desde estas instancias, aparece un conjunto de identificadores, que no son más que las clases de multireferencias a las que pertenece cada enlace. Vemos que existen dos clases, la mc_1 con enlaces al nodo n_2 y a $NULL$, y la clase mc_2 con enlaces al nodo n_3 y a $NULL$. Estas clases nos indican que ninguna de las configuraciones de memoria representadas por el grafo tiene posiciones del array que apunten a listas y otras a elementos sueltos. Precisamente, el primer paso del "pre-enfoque" consiste en la *división por clases de multireferencias*, que separa en distintos grafos los enlaces de distintas clases. La figura 4.11 presenta los grafos obtenidos tras esta división. El primer grafo corresponde a la clase mc_1 mientras que el segundo corresponde a mc_2. Como se puede observar, ahora ninguna de las porciones de memoria representadas por n_1, contiene enlaces en el array a elementos sueltos y a listas a la vez. Para obtener estos grafos, en cada caso se han

eliminado los selectores pertenecientes a la otra clase, y se a *podado* el grafo (despareciendo el resto de nodos y enlaces).

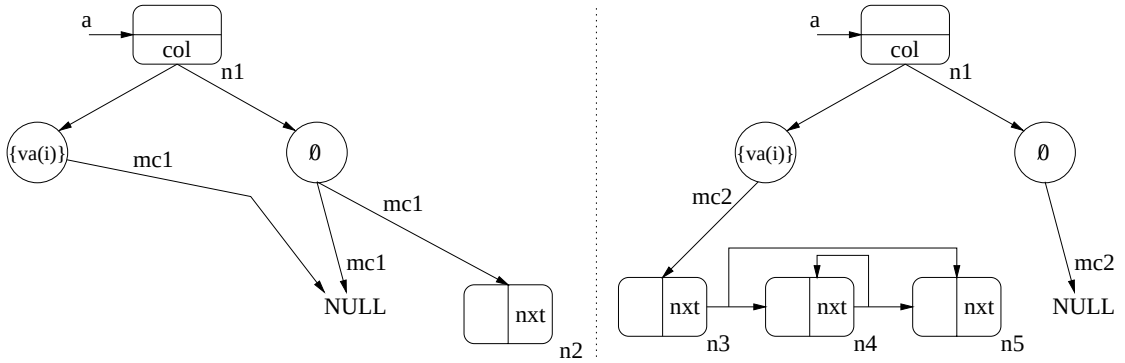


Figura 4.11: Grafos resultantes de la división por clases de multireferencias para $a \rightarrow col[i]$.

Una vez divididos, en cada grafo hay que instanciar las posiciones que pueda tomar i en la sentencia a la que pertenecen dichos grafos. Como en ninguno de los dos existe una instancia que contenga a i, hay que crear instancias nuevas en ambos grafos. En la figura 4.12 podemos ver ambos grafos tras la instanciación, suponiendo la relación $IVRT(i, va(i), st) = neq$ para la sentencia a la que pertenecen los grafos. Se crea la instancia $\{i\}$, y se copian todos los enlaces de $\emptyset$ a la nueva instancia. Los de $\{va(i)\}$ del primer grafo no se copian puesto que $i \neq va(i)$. Los enlaces desde la instancia $\{i\}$ al nodo n_2 y a $NULL$ pueden ser tratados como selectores simples, puesto que ya sólo representan enlaces desde una única posición del array, como un selector normal. A estos enlaces es a los que luego la fase de "enfoque" aplicará las operaciones de *división*, *materialización*, etc.

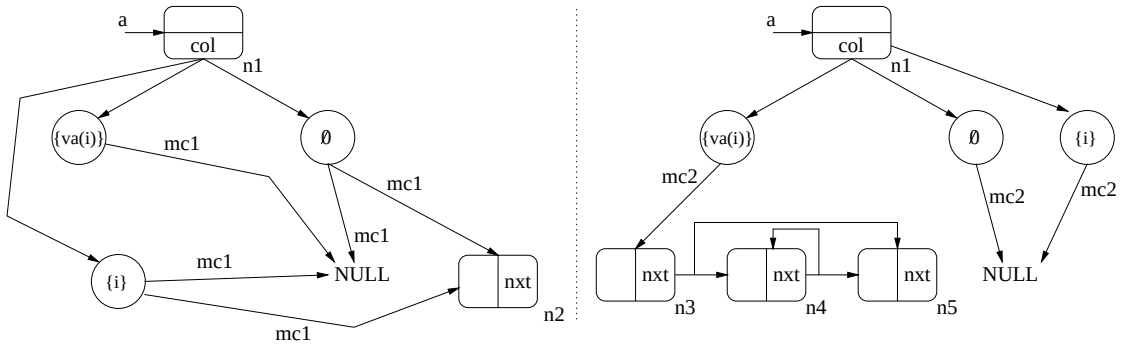


Figura 4.12: Grafos tras la instanciación para $a \rightarrow col[i]$.

En el segundo grafo de la figura 4.12, puesto que $IVRT(i, va(i), st) = neq$, se ha creado una nueva instancia $\{i\}$, pero los enlaces para esta nueva instancia, de nuevo, tan solo se han tomado de la instancia $\emptyset$, ya que i no puede tomar el mismo valor que ha tomado anteriormente en el bucle ($i \neq va(i)$). Esto hace que la posición i del array, según el grafo instanciado, sólo pueda ser $NULL$, y no apuntar a una lista. Esto se corresponde con lo que ocurre en el bucle, en el que como hemos indicado se creaban listas que luego se insertaban en las posiciones del array, recorridas por la variable i del bucle.

Ya hemos presentado las dos operaciones de "pre-enfoque" de los enlaces de un multiselector y una serie de conceptos relacionados con las mismas. Antes de pasar a describir los

cambios que provocan los multiselectores en las operaciones de "enfocado" (división, poda, materialización) y en la semántica abstracta de las sentencias, vamos a tratar un tema muy relacionado con las instancias: *permanencia de las instancias en los multiselectores.*

Permanencia de las instancias en los multiselectores.

Cuando se crea una instancia que contiene la *ivar* i, dicha instancia va a representar los enlaces que pueden tener las posiciones del array representadas por i. Por tanto, esta instancia dejará de tener sentido en cuanto la variable i cambie de valor, puesto que ya no representará las mismas posiciones del array. Algo similar ocurre con las $va(ivar)$, es decir, con instancias que representan posiciones del array que ha tomado una determinada *ivar* anteriormente en una sentencia determinada perteneciente a un bucle. Al terminar el cuerpo del bucle, ya no tiene sentido mantener la información de las posiciones por las que ha pasado i, ya que fuera del bucle no se sabe cual será el nuevo valor que se asigne a i en relación a los que ha tomado en el bucle.

Por este motivo hemos creado una nueva *pseudoinstrucción* $DeleteIndex(iv)$ que se encarga de eliminar de las instancias la *ivar* iv, y se utilizará cuando ya no tenga sentido mantener la instancia que contiene a iv.

Por tanto, la *pseudoinstrucción* $DeleteIndex(i)$ se introducirá en el código, justo después de cualquier instrucción que cambie el valor de la *ivar* i $(i = ...)$. Esto incluye también la cabecera de los bucles $for(i = value_1; i < value_2; i++)$ en la que cada vez que se pasa por ella se asocia un nuevo valor a la *ivar* i. La *pseudoinstrucción* $DeleteIndex(va(i))$ se insertará justo después del bloque de código de un bucle con variable índice i.

La interpretación abstracta de la *pseudoinstrucción* $DeleteIndex(iv)$ sobre un grafo rsg, se define como la aplicación de la función $DELINDEX(iv, rsg)$. Por tanto:

$$ST_{[DeleteIndex(iv)]}(rsg) = DELINDEX(iv, rsg)$$

Definimos la función $DELINDEX(iv, rsg) = rsg'$ con $iv \in IV$ como la función que genera un nuevo grafo rsg' a partir de rsg. Se eliminan de las instancias, las ocurrencias de la *ivar* iv y en el caso de que se sepa que en el bloque de código al que pertenece la sentencia, iv no repite su valor $(i \neq va(i))$, se inserta $va(i)$ en su lugar. Por tanto, para crear rsg' se copian directamente todos los conjuntos de rsg, excepto MSL que se ve modificado.

- $N(rsg') = N(rsg)$
- $PL(rsg') = PL(rsg)$
- $NL(rsg') = NL(rsg)$
- La creación de $MSL(rsg)$ va a depender de la *ivar* iv:
  - Si $iv \in IV \ \wedge \ IVRT(iv, va(iv), st) \neq neq$

$$MSL(rsg') = MSL(rsg) \setminus \{< n, ms, ins, mc, n_d >\} \cup$$
$$\{< n, ms, ins', mc, n_d > \ | \ ivs(ins') = ivs(ins) \setminus \{iv\}, vivs(ins') = vivs(ins)\} \forall < n, ms, ins, mc, n_d > \in MSL(rsg), iv \in ivs(ins)$$

    Es decir, si iv es una *ivar*, cuya relación con su $va(iv)$ no es neq, lo que implica que no se sabe la relación del nuevo valor de iv con los anteriores, entonces se elimina iv de ivs de las instancias de los multiselectores en las que aparezca iv.

– Si $iv \in IV \ \wedge \ IVRT(iv, va(iv), st) = neq$

$$MSL(rsg') = MSL(rsg) \setminus \{< n, ms, ins, mc, n_d >\} \cup$$
$$\{< n, ms, ins', mc, n_d > \ | \ ivs(ins') = ivs(ins) \setminus \{iv\}, vivs(ins') =$$
$$vivs(ins) \cup \{iv\}\} \forall < n, ms, ins, mc, n_d > \in MSL(rsg), iv \in ivs(ins)$$

Por el contrario si iv es una *ivar* y su relación con $va(iv)$ es neq, se sabe que el valor que se asigne a iv será distinto a cualquier valor asignado anteriormente a iv en esta sentencia. Esto permite, que además de eliminar iv de todas las instancias de los multiselectores, se pueda añadir en su lugar $va(iv)$, para que estas nuevas instancias representen enlaces de posiciones por las que ha pasado iv y no va ha volver a pasar.

– Si $iv = va(iv'), iv' \in IV$

$$MSL(rsg') = MSL(rsg) \setminus \{< n, ms, ins, mc, n_d >\} \cup$$
$$\{< n, ms, ins', mc, n_d > \ | \ ivs(ins') = ivs(ins), vivs(ins') =$$
$$vivs(ins) \setminus \{iv'\}\} \forall < n, ms, ins, mc, n_d > \in MSL(rsg), iv' \in vivs(ins)$$

En el caso de que iv sea el valor anterior de alguna *ivar* iv', como hemos comentado, únicamente se elimina dicha iv' de las instancias (en concreto del conjunto $vivs$).

Por ejemplo, si en el primer grafo de la figura 4.12, suponemos que se aplicara la semántica abstracta de $DeleteIndex(i)$ y $IVRT(i, va(i), st) = neq$, la instancia $\{i\}$ cambiaría. En principio se eliminaría i de dicha instancia quedando $\emptyset$. Si no se conociera la relación entre i y $va(i)$, ya habría terminado la ejecución de la sentencia, y los enlaces de $\{i\}$ pasarían a la instancia $\emptyset$ (no habría cambios en los enlaces de $\emptyset$ puesto que tiene los mismos que tiene $\{i\}$). Pero como se sabe que $IVRT(i, va(i), st) = neq$, además de eliminar i de la instancia, se introduce $va(i)$. Por tanto, ahora los enlaces de la instancia $\{i\}$ pasarían a la instancia, que ya existe, $\{va(i)\}$. Ahora sí que cambian los enlaces de esta instancia, puesto que antes no apuntaba a n_2 y ahora si.

Para terminar, en la figura 4.13(a) podemos ver un trozo de código con dos bucles anidados y en la figura 4.13(b) como quedaría este mismo código una vez insertadas las *pseudoinstrucciones DeleteIndex*, junto con las relaciones que aparecerían en $IVRT$ (sólo se muestran las conocidas (eq, neq), las que no aparece son unk).

4.3.3 Operaciones del "enfoque" con multiselectores

En esta sección vamos a ver como afectan los multiselectores a las operaciones descritas en el capítulo anterior, para enfocar los enlaces referenciados en una determinada sentencia. En concreto nos estamos refiriendo a las operaciones de *división*, *poda* y las distintas *materializaciones*.

División de grafos con multiselectores

Esta operación, para una instrucción que contiene $x \rightarrow sel$, divide el grafo de entrada en tantos grafos como enlaces tenga el nodo apuntado por x mediante el selector sel, de forma que en cada grafo aparezca tan solo un enlace para dicho selector. Este comportamiento no se ve afectado por la inclusión de los multiselectores, la diferencia es que se tiene que ampliar

```
...
10 i = j;
...
20 for (i=0; i<100; i++) {
...
30   for (j=i+1; j<100; j++) {
...
40   }
...
50 }
...
```

(a)

```
...                                          IVRT
10 i = j;                                    (i , j , eq)
   DeleteIndex(i);
...
20 for (i=0; i<100; i++) {                   (i , va(i ), neq)
     DeleteIndex(i);
...
30   for (j=i+1; j<100; j++) {               (i , va(i ), neq),
       DeleteIndex(j);                       (j , va(j ), neq),
...                                          (i , j , neq)
40   }
     DeleteIndex(va(j));                     (i , va(i ), neq)
...
50 }
   DeleteIndex(va(i));
...
```

(b)

Figura 4.13: (a) Ejemplo de código con dos bucles anidados. (b) Mismo código con *pseudoinstrucciones DeleteIndex* y tabla de relaciones $IVRT$.

el alcance de esta función, para que también divida los grafos para sentencias que contengan $x \rightarrow sel[i]$.

En este caso, el grafo de entrada ya habrá sufrido un proceso de "pre-enfoque" de manera que en dicho grafo tan solo habrá una clase de multireferencia (operación de división por clases de multireferencias) y en el nodo apuntado por x, en el multiselector sel existirá una instancia que contenga a i (instanciación), que representa los enlaces que pueden tener las posiciones del array representadas por i. Son estos enlaces por los que hay que dividir de nuevo el grafo, como si fueran los enlaces de un selector normal. Es decir, la instancia que contiene a i puede representar varias posiciones del array, pero en una determinada porción de memoria tan solo representa una, la i (de ahí que ahora pueda se considerada como un selector normal).

Por tanto la división para un multiselector consistirá en la misma operación que se hacia con los selectores, crear tantos grafos como enlaces tenga la instancia del multiselector que contenga a i, de forma que en cada uno de ellos tan solo exista un enlace para esta instancia.

La operación divide quedaría de la siguiente forma, $DIVIDE(rsg, x, sel, i) = \{rsg_1, ..., rsg_n\}$. Sea $n \in N(rsg)$, tal que $<x, n> \in PL(rsg)$, entonces si $sel \in S$, $\forall\ < n, sel, n_i > \in NL(rsg)$, o si por el contrario $sel \in MS$, $\forall\ < n, sel, ins, mc, n_i >, i \in ivs(ins)$, se crea un nuevo grafo rsg_i de la siguiente manera:

- $N(rsg_i) = N(rsg)$
- $PL(rsg_i) = PL(rsg)$
- $NL(rsg_i) = NL(rsg) \setminus \{< n, sel, n_j > \in NL(rsg), sel \in S, \forall n_j \neq n_i\}$
- $MSL(rsg_i) = MSL(rsg_i) \setminus \{n, sel, ins, mc, n_j > \in\ MSL(rsg), sel \in MS, i \in ivs(ins), \forall n_j \neq n_i\}$

Poda de grafos con multiselectores

Como ya vimos en el capítulo anterior, la poda de los grafos es llevada a cabo para eliminar nodos y enlaces que se sabe con certeza que no representan porciones de memoria ni enlaces,

pertenecientes a las configuraciones de memoria representadas por el grafo a podar. Esta operación es usada tras la división para eliminar aquellas porciones que pertenecen a otro de los grafos obtenidos de la división, pero también es usada en otras operaciones, siempre para eliminar información no perteneciente al grafo actual.

Vimos que para determinar si un nodo pertenece a las configuraciones de memoria de un grafo, se chequeaba su propiedad *reference patterns* mediante la función *N_ PRUNE*. Para hacer lo mismo con los enlaces de un nodo, se chequeaban sus *cyclelinks* con la función *NL_ PRUNE*. La función *N_ PRUNE* será modificada para que tenga en cuenta los enlaces mediante multiselectores. La función *NL_ PRUNE* se quedará igual para los selectores simples, y se creará una nueva función, *MSL_ PRUNE* para chequear *cyclelinks* para los enlaces pertenecientes a multiselectores.

En cuanto a *N_ PRUNE*, si las funciones $SELIN/\ SELOUT$ indican que el nodo es referenciado por un *multiselector sel* o referencia a algún otro nodo por el *multiselector sel*, entonces debe existir algún enlace por dicho multiselector sobre o desde el nodo por el multiselector *sel*.

$$N\_PRUNE(n) = \begin{cases} 1 \text{ si } \begin{cases} (\exists sel_i \in S \mid SELIN(n, sel_i) = 1 \ \wedge \ \nexists n_2 \in \\ \qquad N(rsg), < n_2, sel_i, n > \in NL(rsg)) \ \vee \\ (\exists sel_i \in S \mid SELOUT(n, sel_i) = 1 \ \wedge \ \nexists n_2 \in \\ \qquad N(rsg), < n, sel_i, n_2 > \in NL(rsg)) \ \vee \\ (\exists sel_i \in MS \mid SELIN(n, sel_i) = 1 \ \wedge \ \nexists n_2 \in \\ \qquad N(rsg), < n_2, sel_i, ins_j, mc_k, n > \in MSL(rsg)) \ \vee \\ (\exists sel_i \in MS \mid SELOUT(n, sel_i) = 1 \ \wedge \\ \qquad \exists < n, sel_i, ins_j, mc_k, NULL > \in MSL(rsg)) \end{cases} \\ 0 \text{ en caso contrario.} \end{cases}$$

Las modificaciones en esta función han sido la introducción de la posibilidad de que sel_i sea un multiselector. Para el caso de que el nodo tenga $SELIN(n, sel_i) = 1$, tiene que ser referenciado desde otro nodo por sel_i. Sin embargo, si $SELOUT(n, sel_i) = 1$, indica que todos las posiciones del array representado por sel_i en el nodo n tienen que apuntar a algún otro nodo. Por tanto, si existe alguna que sea $NULL$, ya no está cumpliendo la propiedad, aunque haya otras posiciones (instancias de sel_i) que si apunten a algún nodo.

La nueva función *MSL_ PRUNE*, es muy similar a *NL_ PRUNE* pero para los multiselectores. Si un *cyclelink* $<ms_1, sel_2>$ existe en un nodo n, todos los nodos destino de los enlaces por ms_1, deben apuntar de nuevo a n por sel_2. Si esto no ocurre es porque el enlace por el multiselector no pertenece a las configuraciones de memoria representadas ahora por el grafo. La diferencia con los selectores simples, es que ahora además de eliminar dicho enlace del multiselector, se eliminan también todos aquellos que pertenezcan a su misma clase de multireferencia. Como ya vimos, las clases de multireferencias agrupaban aquellos enlaces que pertenecían todos a una misma porción de memoria. Si uno de ellos no puede pertenecer a las porciones representadas por un nodo, ninguno de los demás puede hacerlo tampoco. Aquí se ve claramente como las clases de multireferencias ayudan a que el "enfoque" sea mucho más preciso, eliminando enlaces que realmente no pueden existir en las configuraciones representadas por el grafo actual.

Aplicando todo esto, definimos la función como:

$$MSL\_PRUNE(< n_1, sel_i, ins_j, mc_k, n_2 >) =$$
$$\begin{cases} 1 \text{ si } \begin{cases} \cdot\ \exists < sel_i, sel_j > \in CYCLELINKS(n_1)\ \wedge\ \nexists < n_2, sel_j, n_1 > \in NL(rsg)\ \vee \\ \cdot\ MSL\_PRUNE(< n_1, sel_i, ins, mc, n_d >) = 1\ \wedge\ mc = mc_k \end{cases} \\ 0 \text{ en caso contrario.} \end{cases}$$

Una vez creadas las funciones que nos informan cuando un nodo, un selector o un multiselector debe ser eliminado del grafo, se puede definir la operación *PRUNE* para grafos con multiselectores. Como ya se vio, *PRUNE* es un proceso iterativo de *poda* mientras existan nodos o enlaces a eliminar, terminando cuando todos los nodos y enlaces del grafo satisfacen las propiedades de los nodos.

El proceso completo, de nuevo, puede ser expresado como $PRUNE(rsg) = rsg_n$, donde el grafo rsg_n es el resultado de podar el grafo rsg. El proceso iterativo comienza con $rsg_0 = rsg$. Entonces, $\forall i = 1..n$:

- $N(rsg_i) = N(rsg_{i-1}) \setminus \{n \in N(rsg_{i-1}) | N\_PRUNE(n) = 1\}$
- $PL(rsg_i) = PL(rsg_{i-1}) \setminus \{< pvar, n > \in PL(rsg_{i-1}) | N\_PRUNE(n) = 1\}$
- $NL(rsg_i) = NL(rsg_{i-1}) \setminus \{< n_1, sel, n_2 > \in NL(rsg_{i-1}) | (N\_PRUNE(n_1) = 1) \vee (N\_PRUNE(n_2) = 1) \vee (NL\_PRUNE(< n_1, sel, n_2 >) = 1)\}$
- $MSL(rsg_i) = MSL(rsg_{i-1}) \setminus \{< n_1, sel, ins, mc, n_2 > \in MSL(rsg_{i-1}) | (N\_PRUNE(n_1) = 1)\ \vee\ (N\_PRUNE(n_2) = 1) \vee (MSL\_PRUNE(< n_1, sel, ins, mc, n_2 >) = 1)\}$

La única modificación introducida en *PRUNE* es que ahora se eliminan también enlaces por multiselectores, si en nodo origen o destino han sido eliminados (*N_PRUNE*), o si es el propio enlace del multiselector el que no cumple las propiedades del nodo al que pertenece (*MSL_PRUNE*).

Como vimos, al eliminar nodos y enlaces, la propiedad *SPATH* de algunos nodos cambia, de modo que definimos:

$$SPATH(n) = SPATH(n)\ \setminus\ \{< pv, sel >\ |\ < pv, n_{pv} > \in PL(rsg)\ \wedge\ N\_PRUNE(n_{pv}) = 1\}\ \setminus\ \{< pv, sel >\ |\ sel \in S, < pv, n_{pv} > \in PL(rsg)\ \wedge\ NL\_PRUNE(< n_{pv}, sel, n >) = 1\}\ \setminus\ \{< pv, sel >\ |\ sel \in MS,\ < pv, n_{pv} > \in PL(rsg)\ \wedge\ MSL\_PRUNE(< n_{pv}, sel, ins, mc, n >) = 1\ \wedge\ \nexists < n_{pv}, sel, ins_i, mc_j, n > \in MSL(rsg)\}$$

Los *simple paths* son modificados igual que se hacía para los selectores normales, pero ahora, la eliminación de un enlace por un multiselector, puede modificar el conjunto *SPATH* del nodo destino.

Sin embargo hay un diferencia con los selectores simples. Si se elimina un enlace desde un nodo n_{pv} apuntado por la variable pv por el multiselector sel hacia un nodo n, directamente no se elimina el *simple path* $<pv, n>$ de dicho nodo. Antes hay que comprobar que no existe ningún otro enlace desde el nodo n_{pv} por el multiselector sel hacia el nodo n. Esto no podía ocurrir con un selector simple, pero si con un multiselector puesto que representa varios enlaces pertenecientes a una misma porción de memoria (distintas posiciones del array de punteros).

El proceso iterativo de *poda* termina, cuando se alcanza un grafo rsg_n que cumple:

- $\forall n \in N(rsg_n), N\_PRUNE(n) = 0 \wedge$
- $\forall < n_1, sel, n_2 > \in NL(rsg_n), NL\_PRUNE(< n_1, sel, n_2 >) = 0$
- $\forall < n_1, sel, ins, mc, n_2 > \in MSL(rsg_n), MSL\_PRUNE(< n_1, sel, ins, mc, n_2 >) = 0$

en donde todos sus nodos y enlaces (ya sea por selectores o multiselectores) cumplen las propiedades de los nodos.

El proceso de "enfoque" de un enlace continúa con la materialización de un nuevo nodo que represente las porciones de memoria realmente referenciadas por el enlace.

Materialización de un nuevo nodo con multiselectores

Este es el último paso del proceso de enfoque de un determinado enlace, y tiene por objetivo, aislar aquellas porciones de memoria que realmente son referenciadas por el enlace, de aquellas que no lo son. Como ya sabemos, un único nodo representa varias porciones de memoria de propiedades similares, con lo que se consigue representar mediante un grafo finito configuraciones de memoria de tamaño indeterminado. Cuando se van a analizar los efectos de una sentencia que implica una referencia del tipo $x \rightarrow sel$, se toma el nodo destino de dicha referencia, y se materializa uno nuevo de forma que este represente a las porciones de memoria que realmente son referenciadas por $x \rightarrow sel$ y el antiguo representará a las porciones compatibles pero que no son referenciadas por dicho enlace.

Esta materialización consiste básicamente en copiar el nodo, y a la hora de determinar los enlaces sobre el mismo, se tiene en cuenta que ahora es seguro que tiene un enlace desde $x \rightarrow sel$ y por tanto se pueden eliminar algunos enlaces que tenía el nodo antiguo si no son compatibles con este. Tras esta materialización, la semántica abstracta se aplica únicamente sobre porciones y enlaces realmente relacionados con $x \rightarrow sel$.

Vimos también que a veces no se puede materializar un simple nodo, sino que es necesario crear un grafo nuevo. Esto ocurre cuando el nodo referenciado por $x \rightarrow sel$ es apuntado directamente por una variable puntero. Si se "copiara" dicho nodo en otro, existirían dos nodos en un mismo grafo apuntados por la misma variable puntero, cuando una de las restricciones de los RSGs es que en cada uno una variable puede apuntar como máximo a un nodo. Esto provoca que se copie el grafo, de forma que uno de ellos representa las configuraciones de memoria en las que realmente existe el selector $x \rightarrow sel$ y el otro las restantes.

La cuestión ahora es determinar en que forma se ven afectadas estas operaciones en presencia de enlaces a través de multiselectores. Ahora cuando se materializa un nodo, los enlaces que apuntan al mismo, son copias de algunos de los que tenía el nodo antiguo, y pueden provenir de multiselectores. Por otro lado, podría ser necesario materializar un nodo para aislar las porciones apuntadas por un multiselector, en sentencias que incluyan $x \rightarrow sel[i]$. Este último caso, aunque pueda parecer bastante distinto al de un selector simple, no lo es, ya que como hemos visto, antes de llegar a la materialización hay un proceso de "pre-enfoque" que modifica los grafos para que los selectores representados por $x \rightarrow sel[i]$ puedan ser tratados como enlaces simples a través de selectores normales. Veremos pues que prácticamente la materialización en este caso es igual a la presentada en el capítulo anterior.

Sin embargo, antes de pasar a describir formalmente estas operaciones incluyendo a los multiselectores, vamos a prestar atención a algo que en principio no parece tener mayor

importancia. Nos referimos a como los enlaces que apuntan al nodo del que se materializa son copiados al nodo materializado, cuando se trata de enlaces por un multiselector.

Supongamos un nodo n_1 del cual se va a materializar un nuevo nodo n_m. Los enlaces por selectores normales que apuntan a n_1 son copiados sobre n_m si son compatibles con el enlace por el que se materializa (en base a la información *shared* del nodo n_m se puede determinar si un selector determinado puede coexistir con el que ha provocado la materialización). Esta operación bastante simple con selectores, es mucho más compleja si se trata de copiar los enlaces sobre n_1 a n_m pero por medio de multiselectores, debido a las clases de multireferencias de los multiselectores. Para entender mejor el problema y su solución, vamos a presentar un pequeño ejemplo.

En la figura 4.14 se presenta la sumarización de dos nodos n_a y n_b procedentes de dos grafos distintos, para dar lugar a un nuevo nodo n_c. Dichos nodos poseen un multiselector *sel*, con el que n_a apunta a los nodos n_1 y n_3, mientras que n_b lo hace sobre los nodos n_1 y n_2. Al unirse los nodos, como hemos visto en la sección 4.2.2, las clases de multireferencias son creadas para mantener las configuraciones de enlaces que pertenecen a cada nodo sumarizado. Así vemos que la clase mc_1 representa los enlaces provenientes del nodo n_1 y la clase mc_2 a los del nodo n_2, pudiendo de este modo determinar en el nuevo nodo n_c qué combinaciones de enlaces a través del multiselector *sel* son posibles y cuales no.

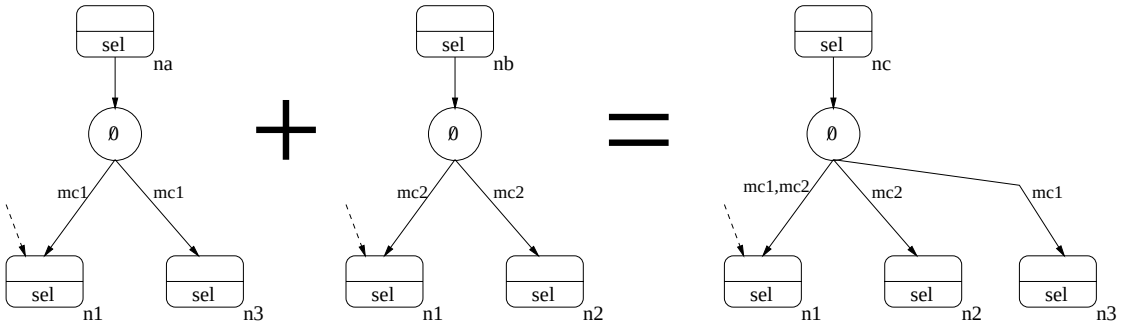


Figura 4.14: Sumarización de dos nodos con multiselectores.

Supongamos que el nodo n_1 es apuntado por otro nodo por algún selector, y es necesario materializar un nuevo nodo, n_4 para representar aquellas porciones de memoria de n_1 realmente referenciadas por dicho selector. En principio, como con los selectores normales, se crearía un nodo n_4 exactamente igual a n_1, y que es apuntado desde los mismos nodos que n_1 (se supone que el nodo es *shared* y por eso estos enlaces pueden aparecer junto a enlace por el que se materializa). Como se puede apreciar en la figura 4.15, al nodo n_4 llegaría un enlace desde *sel* del nodo n_c perteneciente a las clases de multireferencias mc_1 y mc_2 (como el enlace sobre el nodo del que se materializa, n_1). Es aquí donde no se están teniendo en cuenta todas las posibilidades de representación, y en concreto se está perdiendo aquella que proviene de los nodos n_a y n_b de la figura 4.14. En concreto, si extraemos la información proporcionada por las clases de multireferencias, deducimos que los enlaces del nodo n_c a los nodos n_1 y n_4 siempre aparecen juntos (pertenecen al mismo conjunto de clases). Sin embargo, dichos enlaces puede provenir de grafos distintos, como es el caso de la figura 4.14 y por tanto nunca deberían aparecer en una misma configuración de enlaces.

El problema es que a la hora de materializar un nuevo nodo (n_4) de uno existente (n_1), no se sabe de las clases a las que pertenecen los multiselectores que apuntan a n_1, cuales deberían pasar a n_4 y cuales no. Esta imposibilidad de determinar con exactitud esta in-

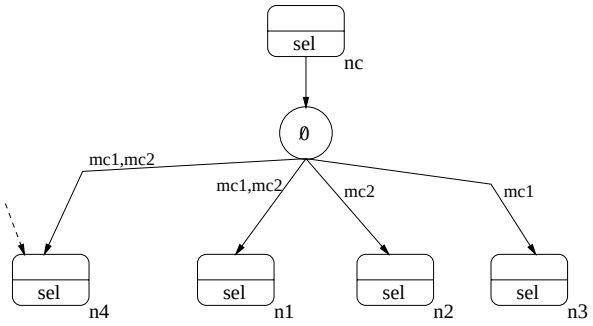


Figura 4.15: Materialización de un nuevo nodo apuntado por un multiselector.

formación, nos hace ser conservativos, y dar cabida a todas las posibilidades de distribución de los multiselectores sobre los dos nodos. Así vamos a contemplar la posibilidad de que los enlaces pertenezcan a las mismas clases (figura 4.15), pero además, vamos a considerar todas las combinaciones de repartición exclusiva de las clases para los dos enlaces. Así en la figura 4.16 aparecen distintas reparticiones de las clases de multireferencias a los enlaces por el multiselector *sel* desde el nodo n_c a los nodos n_1 y n_4. En el primer caso, se contempla la posibilidad de que el enlace a n_4 pertenezca a mc_1 mientras que el de n_1 pertenezca a mc_2. En el segundo, se contempla lo contrario, que el de n_4 pertenezca a mc_2 mientras que el de n_1 a mc_1. En el tercero, consideramos que el enlace a n_4 pertenece a las clases mc_1 y mc_2, mientras que el de n_1 no pertenece a ninguna de ellas, y por último, el caso contrario en el que el de n_4 no pertenece a ninguna de ellas mientras que el de n_1 pertenece a las dos. En los cinco grafos presentados en las figuras 4.15 y 4.16, se contemplan todas las posibilidades de combinaciones de enlaces de las que podría provenir el enlace original de n_c a n_1 perteneciente a mc_1 y mc_2.

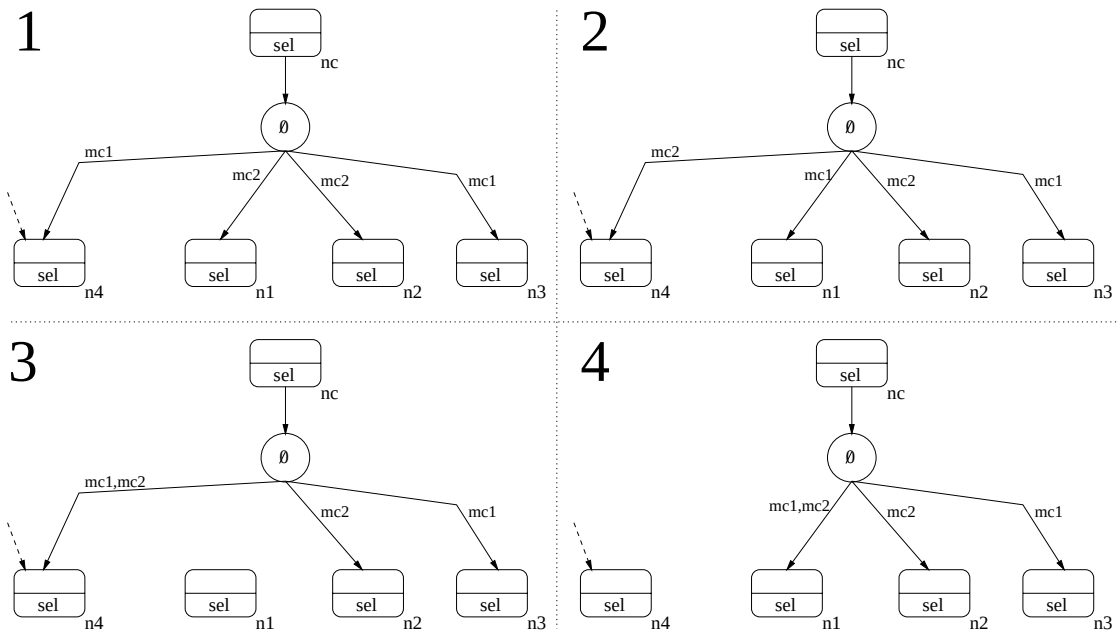


Figura 4.16: Posibles distribuciones de la clases de multireferencias.

A la hora de materializar el nodo apuntado por multiselector, deberán aparecer todas esas

posibilidades, por lo que se crean nuevas clases de multireferencias para cubrirlas. Así en la figura 4.17(a), presentamos como quedaría la materialización tras la creación de las nuevas clases para la distribución de los enlaces sobre el nuevo y el antiguo nodo. En este grafo, las clases mc_1 y mc_2 se corresponden con las mismas del grafo de la figura 4.15. Las clases mc_1 y mc_2 de los cuatro grafos de 4.16 son: las del grafo uno, mc_3 y mc_4; las del grafo dos, mc_5 y mc_6; las del grafo tres mc_7 y mc_8; y las del grafo cuatro, mc_9 y mc_{10}. Como vemos, el número de clases de multireferencias puede crecer mucho, pero hay que tener en cuenta que no todas las clases se mantienen, ya que dos clases que aparezcan exactamente en los mismos enlaces, representan la misma información y una de ellas puede ser eliminada. Así, las clases siguientes son equivalentes pues aparecen en los mismos enlaces: mc_3 y mc_7, mc_6 y mc_8, mc_4 y mc_{10}, y por último, mc_5 y mc_9. De esta forma, eliminando las clases mc_7, mc_8, mc_9 y mc_{10} se consigue el resultado de la materialización mostrado en la figura 4.17(b). En esta figura ya si quedan plasmadas todas las posibles distribuciones de los enlaces del multiselector, que aunque son muchas más de las reales, la falta de información adicional nos hace tener que tomarlas todas en consideración.

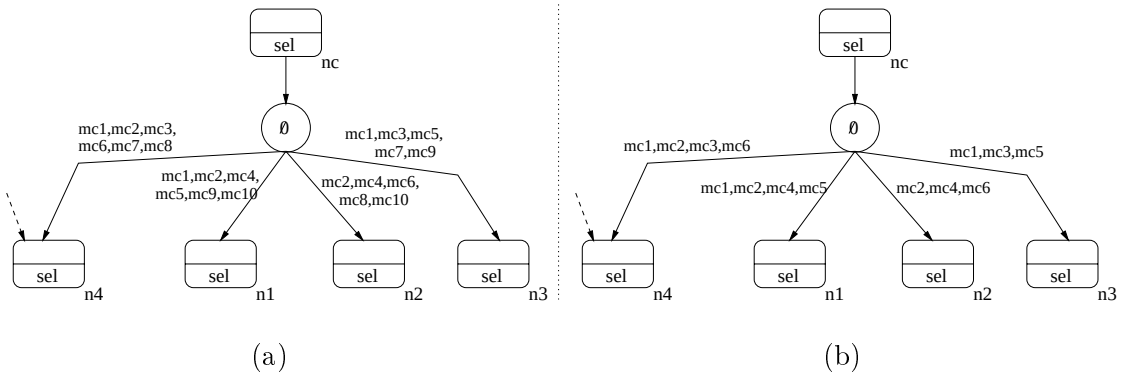


Figura 4.17: Materialización completa, (a) sin simplificar clases de multireferencias, (b) simplificadas.

Este proceso de creación de nuevas clases de multireferencias lo vamos a denominar *materialización de clases de multireferencias* y va a tener lugar cuando el nodo materializado sea apuntado por otros nodos mediante multiselectores.

A continuación pasamos a describir las operaciones de materialización de grafos y nodos, tal y como las presentamos en la sección 3.4.1, pero ahora dando soporte a los multiselectores. Se van a definir tan solo una función $MATERIALIZE\_RSG$ y $MATERIALIZE\_NODE$ independientemente de que materialicen para un selector ($x \rightarrow sel$) o par aun multiselector ($x \rightarrow sel[i]$), puesto que las diferencias son mínimas. Dentro de cada una de ellas, se tendrá en cuenta si *sel* es un selector o un multiselector.

Materialización de nuevos grafos. Esta materialización sucede cuando el nodo, n_{pv}, referenciado por *sel* (ya se selector o multiselector) desde el nodo apuntado por x (n_x) es a su vez apuntado por una variable puntero pv. Se obtienen dos grafos, uno con dicho selector y otro sin él.

La nueva función para materializar grafos queda de la siguiente manera:

$$MATERIALIZE\_RSG(rsg, x, sel, i) = \{rsg_1, rsg_2 \mid rsg_1 = PRUNE(rsg'_1), \\ rsg_2 = PRUNE(rsg'_2)\}$$

La construcción de los grafos rsg'_1 y rsg'_2 se presenta a continuación.

El grafo rsg'_1 queda definido como:

- $N(rsg'_1) = N(rsg)$
- $PL(rsg'_1) = PL(rsg)$
- $NL(rsg'_1) = NL(rsg) \setminus (DL_1 = \{< n_i, sel_j, n_{pv} > \mid [(SHARED(n_{pv}) = 0) \wedge (sel_j \neq sel)] \vee [(SHSEL(n_{pv}, sel) = 0) \wedge (n_i \neq n) \wedge (sel_j = sel)]\})$
- $MSL(rsg'_1) = MSL(rsg) \setminus (DL_2 = \{< n_i, sel_j, ins, mc, n_{pv} > \mid [(SHARED(n_{pv}) = 0) \wedge (sel_j \neq sel)] \vee [(SHSEL(n_{pv}, sel) = 0) \wedge (n_i \neq n) \wedge (sel_j = sel)]\}) \setminus (DL_3 = \{< n_i, sel_j, ins, mc, n_2 > \in MSL(rsg) \mid mc = mc_d \wedge < n_i, sel_j, ins_k, mc_d, n_{pv} > \in DL_2\})$

Como se puede observar, si el nodo n_{pv} no es *shared* se eliminan otros enlaces no compatibles con el enlace por el que se materializa. Se eliminan tanto enlaces por selectores simples (DL_1), como enlaces por multiselectores (DL_2) que no cumplen alguna de los atributos de la información *shared*. Pero con los multiselectores, la eliminación de enlaces va más allá. Si un enlace por un multiselector sobre n_{pv} es borrado, $<n_i, sel_j, ins_k, mc_d, n_{pv}>$, porque no puede pertenecer a las configuraciones de memoria representadas ahora por el grafo, ningún enlace por este mismo multiselector que pertenezca a la misma clase de multireferencias, mc_d, puede aparecer, puesto que aparecería siempre junto al enlace borrado. De esta forma se borran otra serie de multiselectores (DL_3) sobre el nodo n_{pv}. Esto hace que la materialización sea más refinada en cuanto a los enlaces por multiselector.

Como consecuencia de la eliminación de esos selectores y multiselectores, las propiedades de algunos nodos se ven afectadas.

El nodo n_{pv}, definitivamente es apuntado por sel:

- $SELINset(n_{pv}) = SELINset(n_{pv}) \cup \{sel\}$
- $PosSELINset(n_{pv}) = PosSELINset(n_{pv}) \setminus \{sel\}$

Por otro lado el nodo n_x, definitivamente apunta por sel a n_{pv}:

- $SELOUTset(n_x) = SELOUTset(n_x) \cup \{sel\}$
- $PosSELOUTset(n_x) = PosSELOUTset(n_x) \setminus \{sel\}$

Además la eliminación de los enlaces por selectores y multiselectores, provoca cambios en los *SPATH* de muchos nodos:

$$SPATH(n) = SPATH(n) \setminus \{< pv, sel > \mid < pv, n_1 > \in PL(rsg'_1) \wedge ((< n_1, sel, n > \in DL_1) \vee (< n_1, sel, ins, mc, n > \in DL_2 \cup DL_3 \wedge \nexists < n_1, sel, ins_j, mc_k, n > \in MSL(rsg'_1)))\}$$

El grafo rsg'_2 que representa las configuraciones en las que el enlace no existe, queda definido como:

- $N(rsg'_2) = N(rsg)$
- $PL(rsg'_2) = PL(rsg)$

- $NL(rsg_2') = NL(rsg) \setminus \{< n_x, sel, n_{pv} > \mid sel \in S\}$
- $MSL(rsg_2') = MSL(rsg) \setminus \{< n_x, sel, ins, mc, n_{pv} > \mid sel \in MS, i \in ivs(ins)\} \cup \{< n_x, sel, ins, mc, NULL > \mid sel \in MS, i \in ivs(ins)\}$

Si *sel* es un selector, se borra el enlace por dicho selector desde n_x a n_{pv} (representado en rsg_1). Si por el contrario, *sel* es un multiselector, se sustituye el enlace desde n_x por *sel* con la instancia que contiene la *ivar i* al nodo n_{pv} por uno a $NULL$ (de nuevo el enlace borrado está representado en rsg_1). Al tener la seguridad de que este enlace (por selector o multiselector) no existe en rsg_2', se pueden actualizar en concordancia las propiedades de algunos nodos.

El nodo n_x en este grafo puede que no tenga enlaces de salida por *sel*. Se comprueba, y si es cierto, se actualiza los *reference patterns* de dicho nodo. Si *sel* es un multiselector, hay que comprobar que no existen ningún otro enlace por dicho multiselector, para modificar la información referente a selectores de salida:

- $SELOUTset(n_x) = SELOUTset(n_x) \setminus \{sel \mid (sel \in PosSELOUTset(n_x)) \wedge [(\nexists n_i, < n, sel, n_i > \in NL(rsg_2') \wedge sel \in S) \vee (\nexists n_i, < n, sel, ins, mc, n_i > \in MSL(rsg_2') \wedge sel \in MS \wedge n_i \neq NULL)]\}$
- $PosSELOUTset(n_x) = PosSELOUTset(n_x) \setminus \{sel \mid (sel \in PosSELOUTset(n_x)) \wedge [(\nexists n_i, < n, sel, n_i > \in NL(rsg_2') \wedge sel \in S) \vee (\nexists n_i, < n, sel, ins, mc, n_i > \in MSL(rsg_2') \wedge sel \in MS \wedge n_i \neq NULL)]\}$

Por otro lado el nodo n_{pv}, ahora no va a tener el enlace por el selector o multiselector *sel* desde n_x. Esto puede provocar cambios en sus selectores de entrada si ya no es referenciado por ningún otro *sel*. Por ese mismo motivo también puede cambiar su información *shared* si ya no es referenciado por más de un selector o multiselector.

- $SELINset(n_{pv}) = SELINset(n_{pv}) \setminus \{sel \mid (sel \in PosSELINset(n_{pv})) \wedge [(\nexists n_i, < n_i, sel, n_{pv} > \in NL(rsg_2') \wedge sel \in S) \vee (\nexists n_i, < n_i, sel, ins, mc, n_{pv} > \in MSL(rsg_2') \wedge sel \in MS)]\}$
- $PosSELINset(n_{pv}) = PosSELINset(n_{pv}) \setminus \{sel \mid (sel \in PosSELINset(n_{pv})) \wedge [(\nexists n_i, < n_i, sel, n_{pv} > \in NL(rsg_2') \wedge sel \in S) \vee (\nexists n_i, < n_i, sel, ins, mc, n_{pv} > \in MSL(rsg_2') \wedge sel \in MS)]\}$
- $SPATH(n_{pv}) = SPATH(n_{pv}) \setminus \{< x, sel > \mid sel \in S\} \setminus \{< x, sel > \mid sel \in MS \wedge \nexists < n_x, sel, ins, mc, n_{pv} > \in MSL(rsg_2')\}$
- $SHARED(n_{pv}) = 0$ si $\nexists n_i, n_j \in N(rsg_2') \mid (< n_i, sel_i, n_{pv} >, < n_j, sel_j, n_{pv} > \in NL(rsg_2') \wedge sel_i \neq sel_j \in S) \vee (< n_i, sel_i, ins_i, mc_i, n_{pv} >, < n_j, sel_j, ins_j, mc_j, n_{pv} > \in MSL(rsg_2') \wedge sel_i \neq sel_j \in MS) \vee (< n_i, sel_i, n_{pv} > \in NL(rsg_2') \wedge < n_j, sel_j, ins_j, mc_j, n_{pv} > \in MSL(rsg_2'))$
- $SHSEL(n_{pv}, sel) = 0$ si $[sel \in S \wedge (\nexists n_i \in N(rsg_2') \mid < n_i, sel, n_{pv} > \in NL(rsg_2') \vee \exists_1 n_i \in N(rsg_2') \mid < pvar, n_i > \in PL(rsg_2') \wedge < n_i, sel, n_{pv} > \in NL(rsg_2'))] \vee [sel \in MS \wedge (\nexists n_i \in N(rsg_2') \mid < n_i, sel, ins, mc, n_{pv} > \in MSL(rsg_2') \vee \exists_1 n_i \in N(rsg_2') \mid < pvar, n_i > \in PL(rsg_2') \wedge \exists_1 < n_i, sel, ins, mc, n_{pv} > \in MSL(rsg_2') \wedge \exists iv \in IV(rsg), iv \in ivs(ins))]$

Al borrar en enlace por el multiselector *sel* de n_x a n_{pv} se elimina $<x, sel>$ de los *SPATH* de n_{pv} si no existe algún otro enlace apuntándole desde otra instancia de *sel* en n_x (no se elimina directamente como se hace un un selector normal puesto que el multiselector realmente representa varios enlaces desde la misma porción de memoria). En cuanto a $SHARED(n_{pv})$ será *false*, si tras la eliminación del enlace desde n_x no hay dos enlaces por distintos selectores o multiselectores. En cuanto a $SHSEL(n_{pv}, sel)$, si *sel* es un multiselector, se pondrá a *false* su valor, si ya no es apuntado por ningún otro multiselector *sel* o tan solo es apuntado por uno, desde un nodo apuntado por una variable puntero (esto nos asegura que representa tan solo una porción de memoria en cada configuración de memoria representada) y además perteneciente a una instancia simple (aparece una *ivar* en el identificador de la instancia). Esta última restricción es porque si la instancia es múltiple, el enlace sobre n_{pv} puede representar varias referencias desde una misma porción de memoria y por tanto pertenecientes todas a la misma configuración de memoria. Si hay una *ivar* en la instancia, ésta representa posiciones simples del array en cada configuración de memoria, por lo que el enlace puede representar como mucho una referencia en cada configuración de memoria representada.

De este modo quedan definidos los dos grafos resultantes de la *materialización* ya sea de un selector normal o de un multiselector. Pasamos a continuación a describir como queda la materialización de un nuevo nodo cuando se utilizan multiselectores en los grafos.

Materialización de un nodo nuevo. Esta materialización de un nuevo nodo se lleva a cabo cuando el enlace referenciado en la sentencia, ya sea por un selector normal, $x \rightarrow sel$, o por un multiselector, $x \rightarrow sel[i]$, apunta a un nodo, n, que no es apuntado por una variable puntero. El nuevo nodo n_m representará las porciones de memoria de n realmente referenciadas por el enlace de la sentencia desde n_x (nodo apuntado por x).

Es ahora, cuando en un mismo grafo van a estar representados los dos nodos, el sumario, n, y el nuevo materializado, n_m, y al copiar los multiselectores que pueda haber sobre n, hay que hacer una *materialización de las clases de multireferencias* a las que pertenecen, como se ha expuesto al principio de esta sección. Esta materialización de clases, puede provocar que el grafo deba ser divido. Como se ha comentado, existe una fase de "pre-enfoque" que lo primero que hace, cuando la sentencia referencia un multiselector, es dividir por clases de multireferencias, de manera que en cada grafo haya tan solo una clase para dicho multiselector. Pero, como hemos visto, si el nodo a materializar es referenciado por multiselectores, se materializan nuevas clases. Se puede dar el caso, de que el nodo del que se va ha materializar debido a un enlace por un multiselector, sea referenciado por otro enlace de dicho multiselector. Esto va a provocar que tras la materialización, el enlace por el que se ha materializado, pertenezca a más de una clase de multireferencia, con lo cual, debemos aplicar la operación MC_DIVIDE, para separar de nuevo en distintos grafos las distintas clases del multiselector al que pertenece el enlace. De esta forma, sólo tenemos combinaciones de enlaces que realmente es posible que aparezcan todos en el array de una porción de memoria.

Por lo demás, la función $MATERIALIZE\_NODE$ se comporta de manera muy similar a como se expuso en la sección 3.4.1, con la salvedad de que ahora los nodos pueden referenciar y ser referenciados por medio de multiselectores.

De este modo definimos la nueva operación de materialización de nodos como:

$$MATERIALIZE\_NODE(rsg, x, sel, i) =$$
$$\begin{cases} \{rsg_m \mid rsg_m = PRUNE(rsg''_m) \ \wedge \ sel \in S\} \ \vee \\ \{rsg_{m_i} \mid \forall \, rsg_{m_i} \in MC\_DIVIDE(x, sel, rsg''_m) \ \wedge \ sel \in MS\} \end{cases}$$

La operación MC_DIVIDE sólo se llevará a cabo si sel es un multiselector, con lo que la operación $MATERIALIZE\_NODE$ devolverá más de un grafo. En cualquier caso, el grafo o grafos obtenidos, se crean a partir de rsg''_m, que se construirá, como veremos más adelante, a partir del grafo rsg'_m, construido de la siguiente manera. Sean $n_x, n \in N(rsg)$ tal que $<x, n_x> \in PL(rsg), <n_x, sel, n> \in NL(rsg)$ si $sel \in S$, o $<n_x, sel, ins, mc, n> \in MSL(rsg)$ si $sel \in MS$, definimos entonces rsg'_m como:

- $N(rsg'_m) = N(rsg'_m) \cup \{n_m\}$ donde
  - $TYPE(n_m) = TYPE(n)$
  - $STRUCT(n_m) = STRUCT(n)$
  - $SELINset(n_m) = SELINset(n) \cup \{sel\}$
  - $PosSELINset(n_m) = PosSELINset(n) \setminus \{sel\}$
  - $SELOUTset(n_m) = SELOUTset(n)$
  - $PosSELOUTset(n_m) = PosSELOUTset(n)$
  - $SHARED(n_m) = SHARED(n)$
  - $SHSEL(n_m, sel_i) = SHSEL(n, sel_i) \forall sel_i \in S$
  - $TOUCH(n_m) = TOUCH(n)$
  - $CYCLELINKS(n_m) = CYCLELINS(n)$
- $PL(rsg'_m) = PL(rsg)$
- $NL(rsg'_m) = NL(rsg) \setminus \{<n_x, sel, n> \mid sel \in S\} \cup \{<n_x, sel, n_m> \mid sel \in S\}$
  - $\cup \{n_m, sel_i, n_j> \mid <n, sel_i, n_j> \in NL(rsg'_m)\}$
  - $\cup (AL_1 = \{<n_i, sel_j, n_m> \mid <n_i, sel_j, n> \in NL(rsg'_m) \wedge$
    $[(SHARED(n_m) = 1 \wedge sel_j \neq sel) \vee$
    $(SHSEL(n_m, sel) = 1 \wedge sel_j = sel)]\})$
  - $\cup \{<n_m, sel_i, n_m> \mid <n, sel_i, n> \in NL(rsg'_m) \wedge$
    $[(SHSEL(n_m, sel) = 1 \wedge sel_i = sel) \vee$
    $(SHARED(n_m) = 1 \wedge sel_i \neq sel]\}$
- $MSL(rsg'_m) = MSL(rsg) \setminus \{<n_x, sel, ins, mc, n> \mid sel \in MS \wedge i \in ivs(ins)\} \cup \{<n_x, sel, ins, mc, n_m> \mid sel \in MS \wedge i \in ivs(ins)\}$
  - $\cup \{n_m, sel_i, ins_i, mc_i, n_j> \mid <n, sel_i, ins_i, mc_i, n_j> \in MSL(rsg'_m)\}$
  - $\cup (AL_2 = \{<n_i, sel_j, ins_j, mc_j, n_m> \mid <n_i, sel_j, ins_j, mc_j, n> \in MSL(rsg'_m) \wedge$
    $[(SHARED(n_m) = 1 \wedge sel_j \neq sel) \vee$
    $(SHSEL(n_m, sel) = 1 \wedge sel_j = sel)]\})$
  - $\cup \{<n_m, sel_i, ins_i, mc_i, n_m> \mid <n, sel_i, ins_i, mc_i, n> \in MSL(rsg'_m) \wedge$
    $[(SHSEL(n_m, sel) = 1 \wedge sel_i = sel) \vee$
    $(SHARED(n_m) = 1 \wedge sel_i \neq sel]\}$

Vemos que la creación de rsg'_m a cambiado poco, se crea el nuevo nodo n_m copiando las propiedades de n, y los enlaces que puedan coexistir con el enlace por el que se materializa. De esta forma, n_m tiene los mismos enlaces por selectores y multiselectores que n. Es apuntado

por aquellos selectores y multiselectores, los cuales, dependiendo de los atributos *SHARED* y *SHSEL* de n_m, pueden coexistir con el nuevo selector o multiselector desde n_x por sel (que aparece en la sentencia). Los conjuntos AL_1 y AL_2 contienen enlaces por selectores y multiselectores, respectivamente, que han sido insertados sobre n_m y que serán utilizados para calcular la propiedad *SPATH* del mismo. Aparte de *SPATH*, otras propiedades de varios nodos se ven afectadas debido a la distribución de enlaces llevada a cabo al separar en nodos distintos las porciones de memoria representadas en n. Las propiedades de n se verán modificadas para reflejar el hecho de que ahora n ya no es apuntado desde n_x por sel, y las de n_m precisamente por lo contrario, porque sí que es apuntado por n_x y sel.

Las nuevas propiedades de n son:

- $SELINset(n) = SELINset(n) \setminus \{sel \mid (sel \in PosSELINset(n)) \wedge$
  $[(\nexists n_i, < n_i, sel, n > \in NL(rsg'_m) \ \wedge \ sel \in S) \vee$
  $(\nexists n_i, < n_i, sel, ins, mc, n > \in MSL(rsg'_m) \ \wedge \ sel \in MS)]\}$

- $PosSELINset(n) = PosSELINset(n) \setminus \{sel \mid (sel \in PosSELINset(n)) \wedge$
  $[(\nexists n_i, < n_i, sel, n > \in NL(rsg'_m) \ \wedge \ sel \in S) \vee$
  $(\nexists n_i, < n_i, sel, ins, mc, n > \in MSL(rsg'_m) \ \wedge \ sel \in MS)]\}$

- $SPATH(n) = SPATH(n) \setminus \{< x, sel > \ \mid sel \in S\} \setminus \{< x, sel > \ \mid$
  $sel \in MS \ \wedge \nexists < n_x, sel, ins, mc, n > \in MSL(rsg'_m)\}$

- $SHARED(n) = 0$ si $\nexists n_i, n_j \in N(rsg'_m) \mid$
  $(< n_i, sel_i, n >, < n_j, sel_j, n > \in NL(rsg'_m) \wedge sel_i \neq sel_j \in S) \vee$
  $(< n_i, sel_i, ins_i, mc_i, n >, < n_j, sel_j, ins_j, mc_j, n > \in MSL(rsg'_m) \wedge sel_i \neq sel_j \in MS) \vee (< n_i, sel_i, n > \in NL(rsg'_m) \wedge < n_j, sel_j, ins_j, mc_j, n > \in MSL(rsg'_m))$

- $SHSEL(n, sel) = 0$ si $[sel \in S \ \wedge \ (\nexists n_i \in N(rsg'_m) | < n_i, sel, n > \in NL(rsg'_m) \vee$
  $\exists_1 n_i \in N(rsg'_m) | < pvar, n_i > \in PL(rsg'_m) \wedge < n_i, sel, n > \in NL(rsg'_m))] \vee$
  $[sel \in MS \ \wedge \ (\nexists n_i \in N(rsg'_m) | < n_i, sel, ins, mc, n > \in MSL(rsg'_m) \vee \exists_1 n_i \in N(rsg'_m) | < pvar, n_i > \in PL(rsg'_m) \wedge \exists_1 < n_i, sel, ins, mc, n > \in MSL(rsg'_m) \ \wedge \ \exists iv \in IV(rsg), iv \in ivs(ins))]$

Las propiedades cambian exactamente igual a como lo hacen para el nodo n_{pv} en la materialización de un grafo nuevo, expuesta en el apartado anterior.

En cuanto a las propiedades de n_m, el cual representa ahora porciones de memoria apuntadas todas desde n_x por sel:

- $SPATH(n_m) = \{< x, sel >\} \ \cup \ \{< pv, sel_j > \mid \ < pv, n_i > \in PL(rsg'_m) \wedge$
  $< n_i, sel_j, n_m > \in AL_1\} \ \cup \ \{< pv, sel_j > \mid \ < pv, n_i > \in PL(rsg'_m) \wedge$
  $< n_i, sel_j, ins, mc, n_m > \in AL_2\}$

Los *SPATH* del nodo materializado, n_m, se crean viendo los enlaces por selectores (AL_1) o multiselectores (AL_2) que se han insertado sobre n_m desde nodos apuntados directamente por variables puntero.

Por último, como hemos comentado al principio de esta sección, si el nuevo nodo materializado, n_m, es referenciado por algún enlace de un multiselector, hay que llevar a cabo la *materialización de nuevas clases de multireferencias* en dicho multiselector, para cubrir todas las posibles configuraciones de enlaces en las que pueden aparecer dicho enlace y el que apuntaba desde el multiselector al nodo del cual se ha materializado, n.

Por cada enlace desde un multiselector insertado sobre n_m (conjunto AL_2), $<n_a, ms_b, ins_i, mc_j, n_m>$, hay que crear nuevas clases de multireferencias en el multiselector ms_b, que representen a todas las existentes en el mismo, pero repartiéndolas exclusivamente entre el enlace al nodo materializado, n_m, y al nodo del que se materializa, n. Una función $MATERIALIZE\_MC$ es la encargada de crear estas nuevas clases.

Antes de definir esa función, vamos a presentar otra que nos devuelve el conjunto de *clases de multireferencias* al que pertenece un determinado enlace de un multiselector. Esta nueva función se denomina MCS y queda definida de la siguiente manera:

$$MCS(< n_s, ms, ins, mc, n_d >, rsg) = \{mc_i \mid \forall < n_s, ms, ins', mc_i, n_d > \in MSL(rsg)\}$$

Esta función MCS se usará para determinar las clases a las que pertenecen los enlaces por multiselector insertados sobre n_m (AL_2) y así determinar las nuevas clases a repartir. La creación de las nuevas clases consiste en lo siguiente: se va a repetir el siguiente proceso para cada una de las clases mc_i a las que pertenece el enlace en cuestión (MCS).

- Elegido un mc_i, crear clases nuevas, mc_i', para todas las clases que tiene el enlace (MCS).
- Insertar la nueva clase correspondiente a mc_i (mc_i') en los enlaces hacia el nodo materializado n_m.
- Insertar en los enlaces hacia el nodo del cual se ha materializado, n, la nuevas clases correspondientes a las clases que aparecen en dichos enlaces, excepto mc_i', que se le ha asociado al enlace hacia n_m.
- Para todos los demás enlaces del multiselector (con destino distinto a n y n_m), insertar las nuevas clases correspondientes a las clases a las que pertenece cada uno de estos selectores. Por ejemplo, si existe un enlace a otro nodo n_o, perteneciente a las clases mc_1 y mc_2, se insertarán en dicho enlace las nuevas clases correspondientes a estas dos (mc_1' y mc_2').

De esta forma se consigue la pertenencia exclusiva de los enlaces hacia n_m y n a las clases que pertenecía el enlace al nodo n antes de materializar.

Formalmente definimos la operación $MATERIALIZE\_MC(rsg, AL)$, donde se van a materializar nuevas clases para los enlaces de los multiselectores del conjunto de multiselectores AL de la siguiente forma:

$$\begin{array}{l} \forall < n_s, ms_i, ins_j, mc_k, n_m > \in AL \\ \left\{ \begin{array}{l} \forall mc_i \in MCS(< n_s, ms_i, ins_j, mc_k, n_m >) \\ \left\{ \begin{array}{l} \cdot\ MSL(rsg) = MSL(rsg) \cup \{< n_s, ms_i, ins', mc_i', n_m > \ \mid \\ \qquad \forall < n_s, ms_i, ins', mc, n_m > \in MSL(rsg)\} \cup \\ \cdot\ \{< n_s, ms_i, ins, mc_j', n > \ \mid \forall < n_s, ms_i, ins, mc_j, n > \in MSL(rsg) \wedge \\ \qquad mc_j \neq mc_i\} \cup \\ \cdot\ \{< n_s, ms_i, ins, mc_j', n_d > \ \mid \forall < n_s, ms_i, ins, mc_j, n_d > \in MSL(rsg)\} \end{array} \right. \end{array} \right. \end{array}$$

Como vemos, para cada nuevo enlace insertado (AL) y para cada clase de multireferencia que aparece en dicho enlace, se insertan las nuevas clases en los enlaces a n_m, n y demás nodos, según se ha descrito anteriormente.

Para obtener el definitivo rsg''_m resultante de la materialización de un nuevo nodo (función $MATERIALIZE\_NODE$) hay que aplicar esta operación de materialización de clases de multireferencias al grafo rsg'_m, para los enlaces por multireferencias creados sobre n_m, AL_2. Hay que recordar, que para que el número de clases de multireferencias no crezca indefinidamente, hay que buscar clases equivalentes (aparecen exactamente en los mismos enlaces) y eliminar una de ellas. Para esta tarea usaremos la función MAP_MC definida al final de la sección 4.2.2.

Sea $rsg_t = MATERIALIZE\_MC(rsg'_m, AL_2)$, definimos entonces rsg''_m como:

- $N(rsg''_m) = N(rsg_t)$
- $PL(rsg''_m) = PL(rsg_t)$
- $NL(rsg''_m) = NL(rsg_t)$
- $MSL(rsg''_m) = \{< n_s, ms, ins, MAP\_MC(mc), n_d > \mid \forall < n_s, ms, ins, mc, n_d > \in MSL(rsg_t)\}$

Una vez definida por completo la operación de materialización de un nuevo nodo, quedan descritas todas las operaciones que componen las fases de "pre-enfoque" y "enfoque" de los enlaces referenciados en las sentencias, para grafos con multiselectores. La semántica abstracata de las sentencias que manejan selectores y multiselectores se presenta en el apéndice C.

4.4 Resultados experimentales

Todas las modificaciones presentadas en este capítulo sobre los RSRSGs y la semántica de las sentencias, para el soporte de los arrays de punteros, han sido implementadas en el *pseudo-compilador* que genera los RSRSGs asociados a cada sentencia de un código. En esta sección vamos a presentar los resultados obtenidos tras el análisis de los cuatro últimos códigos presentados en la sección 3.5, donde las estructuras dinámicas de datos usadas han sido modificadas utilizando arrays de punteros en las mismas. Los códigos analizados se presentan en el apéndice E.2. Estas nuevas estructuras realmente son muy usadas por su facilidad de creación y uso.

Como ya presentamos en la sección 3.5, el compilador puede llevar a cabo el análisis en tres niveles distintos, donde la complejidad va aumentando, añadiendo más propiedades al análisis cuando es necesario. El análisis comienza en el nivel más bajo que necesita menos recursos (tiempo y memoria) y se va aumentando de nivel si de los grafos obtenidos no se puede deducir una información adecuada sobre la estructura de datos.

Recordamos que los tres niveles son:

- L_1: En este nivel no se utiliza la propiedad $TOUCH$ y los únicos *simple paths* que son tomados en cuenta son los de longitud cero ($SPATH0$).
- L_2: Es igual al nivel anterior pero utilizando también los *simple paths* de longitud uno ($SPATH1$).
- L_3: Este es el nivel más completo, en el además se utiliza la información proporcionada por $TOUCH$.

	Tiempo			Memoria (MB)		
Nivel	L_1	L_2	L_3	L_1	L_2	L_3
Matriz × Vector	0'03"	0'04"	0'05"	0.92	1.03	1.2
Matriz × Matriz	0'12"	0'14"	0'16"	1.19	1.31	1.49
Factorización LU	2'50"	3'03"	-	3.96	4.18	-
Barnes-Hut	61'24"	69'55"	0'54"	40.14	42.86	3.06

Tabla 4.1: Requerimientos de tiempo y memoria para el análisis de los códigos con multiselectores.

Los códigos analizados son el producto matriz dispersa por vector, el producto matriz dispersa por matriz dispersa, la factorización LU de una matriz dispersa, y el núcleo de la simulación N-Body Barnes-Hut. Más adelante presentamos las estructuras usadas en cada código asi como los resultados obtenidos tras su análisis. En la tabla 4.1 se muestran los requerimientos de tiempo y memoria que ha necesitado el compilador para el análisis de los cuatro códigos. Para los tres primeros códigos se ha obtenido una descripción exacta de la estructura de datos en el nivel L_1 del compilador, aunque en la tabla también se muestran los requerimientos para los demás niveles. Sin embargo, de nuevo, para el código Barnes-Hut, la representación más exacta ha sido obtenida en el nivel L_3, en principio debido a la estructura "pila" adicional que nos vemos obligados a introducir para simular la recursividad, como presentamos en la sección 4.4.4. Los tiempos mostrados en la tabla corresponden al análisis en un procesador Pentium III a 500 MHz con 128 MB de memoria principal.

A continuación presentamos los aspectos principales relacionados con cada uno de los códigos analizados.

4.4.1 Multiplicación matriz dispersa por vector

Este código implementa la multiplicación de una matriz dispersa por un vector, $r = M \times v$. La diferencia con el presentado en el capítulo anterior es que la matriz M se almacena como un puntero a una estructura que contiene un array row con punteros a listas doblemente enlazadas que almacenan los elementos no nulos de las filas de la matriz. Los vectores v y r se almacenan también en listas doblemente enlazadas. El código analizado puede ser consultado en el apéndice E.2.2. Un esquema de las estructuras de datos usadas se presenta en la figura 4.18(a). El vector r es construido durante el proceso de multiplicación.

Después del proceso de análisis, el RSRSG obtenido para cada sentencia, representa de una forma precisa las características fundamentales de la forma de las estructuras de datos. En la figura 4.18(b) presentamos una simplificación del RSRSG obtenido para la última sentencia del código. Tenemos que recordar que en dichas representaciones compactas, los nodos circulares representan instancias de los multiselectores (row en este caso), y en su interior aparecen los conjuntos ivs y $vivs$ que definen cada instancia. En los ejemplos mostrados aparecerá siempre $\{\emptyset\}$ puesto que se ha llevado a cabo la pseudoinstrucción $DeleteIndex$ de todas las variables índice del código. Además, los nodos que aparezcan sombreados representan nodos con el atributo $SHARED$ igual a $true$.

En el RSRSG podemos observar claramente las tres estructuras involucradas en el código (M, v y r). Podemos ver como los vectores están representados por tres nodos con selectores nxt y prv. El nodo central representa todos los elementos centrales de los vectores y

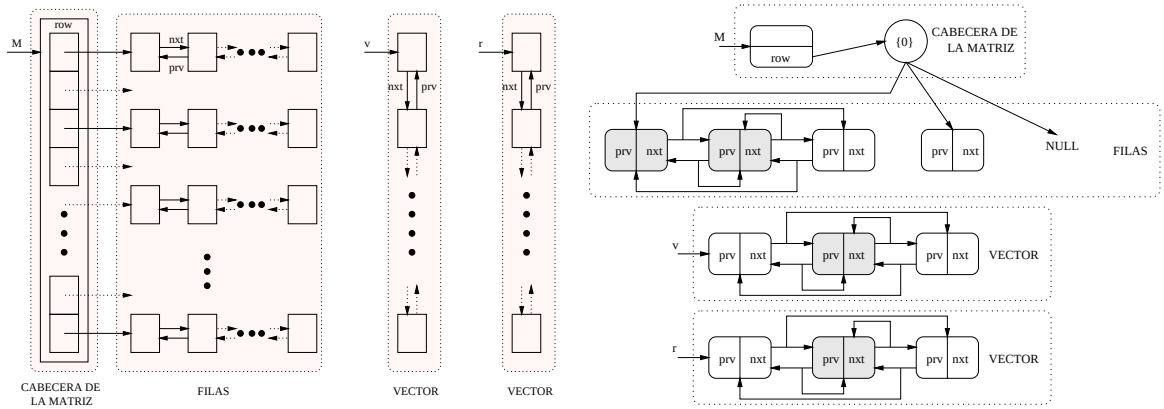


Figura 4.18: Estructuras del código matriz dispersa por vector (a) y representación compacta del RSRSG obtenido.

los de los extremos, el primer y último elemento respectivamente. Se puede apreciar que el nodo central es *SHARED* ya que representa elementos que son referenciados a la vez por *nxt* (desde el elemento anterior) y por *prv* (desde el elemento posterior). Se puede ver también, como el multiselector *row*, tiene una sola instancia ($\{\emptyset\}$) la cual puede ser un puntero *NULL*, apuntar a un solo elemento (lista de un solo elemento) o a una lista de ellos (fila con más de un elemento). Como se puede observar el primer elemento de las filas es *SHARED* puesto que es referenciado por *prv* desde el segundo elemento y además por *row* desde la cabecera de la matriz. Aún así, la información *SHSEL* de todos los nodos y para todos los selectores/multiselectores es *false*. Esto implica lo siguiente:

- Las listas doblemente enlazadas que aparecen en el código (filas de la matriz y vectores) son acíclicas cuando se recorren siguiendo tan solo un tipo de selector (*nxt* or *prv*), puesto que no hay ningún elemento que sea referenciado por el mismo selector desde dos elementos distintos.

- Los distintos enlaces del multiselector *row* apuntan a listas distintas ya que el primer nodo de dichas listas no es *SHSEL* por el multiselector *row*, por tanto ningún elemento es apuntado por más de un enlace *row*.

- Además las listas apuntadas por row no comparten elementos entre sí, puesto que si lo hicieran, habría alguno con más de un selector (*nxt*, *prv*) referenciándolo, y esto no es así ya que *SHSEL* es *false* para todos los nodos y selectores.

Por tanto, de un posterior análisis del RSRSG obtenido para cada sentencia, se pueden deducir las principales características de la estructura de datos usada en el código, reportando la independencia de las estructuras que representan las filas de la matriz y que son recorridas en los sucesivos pasos del bucle principal. Esta información podría ser muy útil para una posible paralelización automática del código, analizando los accesos a las estructuras y la forma de las mismas proporcionada por los RSRSGs.

4.4.2 Multiplicación matriz dispersa por matriz dispersa

Este código implementa la multiplicación de dos matrices dispersas, $C = A \times B$. De nuevo, el cambio en las estructuras de datos es igual que en el código anterior, la cabecera de la matriz ahora no es una lista doblemente enlazada, sino un array *row* de punteros a listas doblemente enlazadas representando las filas de la matriz. Por tanto los punteros A, B y C apuntarían a una estructura similar a la que apunta M en la figura 4.18(a). El código analizado se presenta en el apéndice E.2.4.

La figura 4.19 muestra una simplificación del RSRSG obtenido para la última sentencia del código, donde se pueden apreciar las tres matrices. La estructura apuntada por C es construida en el bucle que realiza la multiplicación de las otras dos matrices.

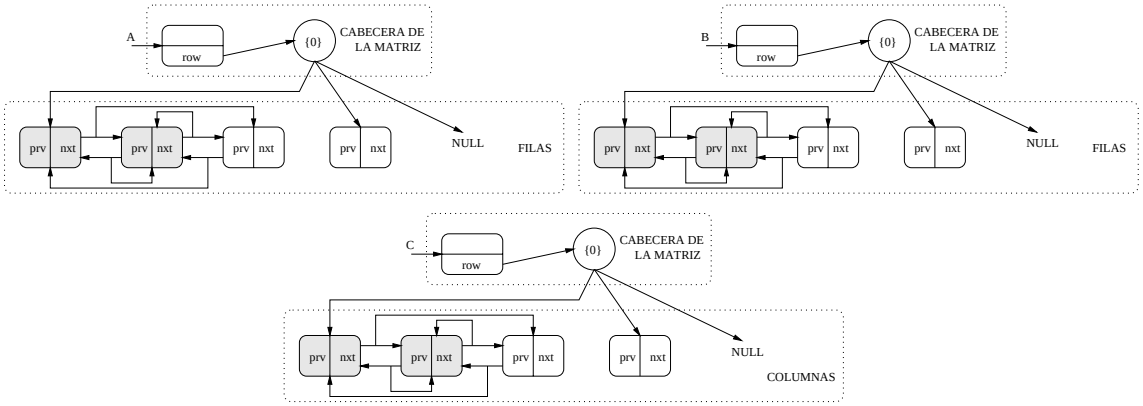


Figura 4.19: Representación compacta del RSRSG obtenido para la última sentencia del código matriz dispersa por matriz dispersa.

De dicho RSRSG podemos extraer las características principales de las estructuras de datos, exactamente de la misma forma que hemos expuesto en el apartado anterior con la matriz del producto matriz vector. Así se puede deducir de los grafos, que hay tres estructuras disjuntas, accesibles por los punteros A, B y C. Las tres son similares, constan de un multiselector con punteros $NULL$, a un elemento o a una estructura acíclica si se recorre utilizando tan solo uno de los selectores que posee, nxt o prv. Además, distintas posiciones del array apuntan a distintas estructuras (no son *SHSEL* por *row*) que no comparten elementos, por tanto son independientes. Lo que implica que recorridos sobre las estructuras apuntadas por dos posiciones distintas del array *row* (por ejemplo en distintos paso del bucle), no visitarán nunca un mismo elemento.

De nuevo, de los RSRSG se va a poder recuperar la información necesaria para determinar la forma de las estructuras de datos utilizadas, para poder así determinar si varios accesos desde variables puntero a posiciones de estas estructuras pueden o no acceder a una misma porción de memoria.

4.4.3 Factorización LU de una matriz dispersa

Este código es el mismo que el analizado en la sección 3.5.5, únicamente cambiando el tipo de estructura que almacena la matriz, como se puede observar en el código presentado en el apéndice E.2.6. Al igual que en los dos códigos anteriores, la lista doblemente enlazada que hace de cabecera de la matriz, ha sido sustituida por un array de punteros *col* a las listas que

representan las columnas de la matriz. La estructura es similar a la apuntada por M en la figura 4.18(a) del código matriz dispersa por vector.

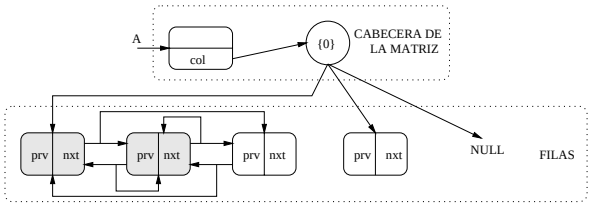


Figura 4.20: Representación compacta del RSRSG obtenido para la última sentencia del código de factorización LU.

En la figura 4.20 se muestra la representación compacta del RSRSG obtenido justo después de la inicialización de la matriz A y después de la factorización LU, lo que implica que la forma fundamental de la estructura de datos no cambia tras la ejecución de la factorización. De nuevo, las mismas apreciaciones hechas en los dos códigos anteriores sobre las matrices, pueden ser aplicadas a la matriz A que es factorizada. La información fundamental que se puede deducir, como hemos comentado para los códigos anteriores, es la independencia de las estructuras que representan las filas de la matriz, además de su carácter acíclico cuando son recorridas utilizando tan solo un tipo de selector, *nxt* o *prv*. Un análisis posterior se podría basar en esta información extraída de los RSRSGs asociados a cada sentencia, y determinar que las columnas de la matriz pueden ser actualizadas en paralelo puesto que los distintos pasos de los bucles recorren estructuras que no comparten elementos entre si.

De nuevo hay que tener en cuenta lo siguiente: aunque la estructura manejada en este código es igual a las de los códigos anteriores, la complejidad de éste es mucho mayor a la hora de ser analizado. Las razones son que existen muchos bucles anidados dentro de los cuales hay estructuras condicionales, IF, lo que provoca que el número de posibles caminos en el flujo de control se dispare, lo que implica un mayor número de posibles configuraciones de memoria a analizar. A esto se le une, que el código hace uso de muchas variables puntero para el manejo de las estructuras, lo que provoca que el número de nodos en los RSGs aumente y por tanto el número de RSGs en cada RSRSG también aumenta. Aún así, si miramos la tabla 4.1, vemos que se han reducido mucho los requerimientos en tiempo y memoria para el código que utiliza multiselectores, con respecto al que utilizaba listas doblemente enlazadas. El motivo es que en el antiguo, la lista cabecera era también recorrida por variables puntero, lo que provocaba grafos con un mayor número de nodos y por tanto un mayor número de los mismos por cada RSRSG. Ahora, estos recorridos se hacen mediante un bucle que accede a distintos índices (i, j) del multiselector *col*.

Pero de nuevo, el análisis en el nivel L_3 ha agotado los 128 MB de memoria principal. Esto es debido al uso de la información $TOUCH$. Puesto que en este código existen muchos bucles anidados que hacen recorridos dependientes unos de otros, de las estructuras dinámicas (las columnas de la matriz) con diferentes punteros, hay muchas variables puntero que pueden aparecer en el $TOUCH$ de los nodos. Esto provoca que haya muchos más nodos, puesto que no tienen la propiedad $TOUCH$ igual. Como hemos comentado antes, esto dispara el tamaño de los grafos y lo que es peor, dispara el número de posibles grafos en cada $RSRSG$. Esto unido a que todos estos grafos tienen que ser analizados teniendo en cuenta todos los posibles caminos del flujo de control, hace que los requerimientos en espacio y tiempo se disparen también.

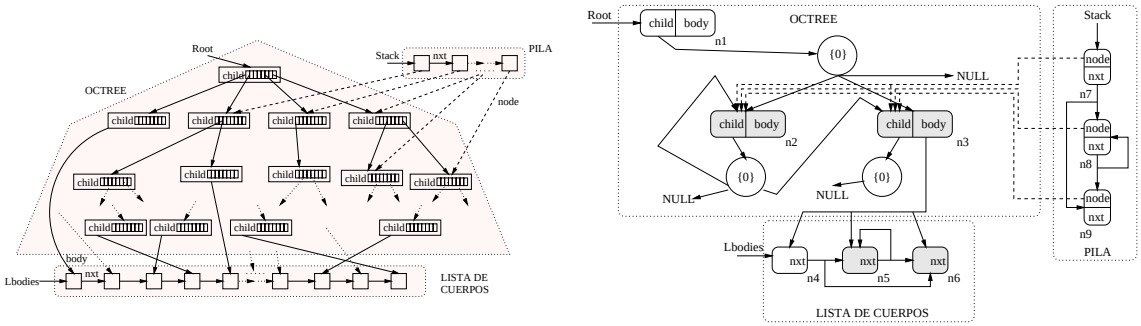
Pero tenemos que recordar, que ya en el nivel L_1 de análisis, se han obtenido unos RSRSGs de los que se pueden deducir las características fundamentales de la estructura de datos, por lo que ni el análisis en nivel L_2 ni el L_3 tendrían que ser llevados a cabo.

4.4.4 Simulación Barnes-Hut

Los detalles de este código han sido presentados en la sección 3.5.6, por lo que aquí nos centraremos en presentar los cambios en la estructura de datos y los resultados obtenidos tras el análisis del código modificado que es presentado en el apéndice E.2.8.

Como ya comentamos la estructura principal de este código es un octree que representa todo el espacio, y una lista donde se almacenan los cuerpos que existen. La diferencia de la estructura utilizada aquí con la presentada en la sección 3.5.6 es la forma de almacenar el octree. Cada nodo del árbol va a tener un campo *child* que va a ser un array de ocho punteros a sus posibles ocho hijos. Aparte, las hojas del árbol apuntarán a los cuerpos almacenados en la lista *Lbodies* mediante un selector llamado *body*. Este tipo de estructura mejora mucho su manejo por parte del código, puesto que por ejemplo, a la hora de crear un hijo nuevo basta con reservar memoria para un nodo más y dentro de él ya están los punteros a los hijos, mientras que en el modelo anterior, había que crear la lista de los ocho hijos. Además, para acceder a un determinado hijo basta con acceder al correspondiente $child[i]$, y no hay que recorrer con un puntero la lista de los hijos hasta encontrar el *i-ésimo*.

En la figura 4.21(a) presentamos esquemáticamente la forma de las estructuras de datos utilizadas en el código. Vemos como *Root* apunta a la raíz del octree, y como las hojas de dicho árbol apuntan a elementos de la lista apuntada por *Lbodies*. Vemos como además aparece una estructura adicional apuntada por *Stack*. Esta estructura es una pila que se introduce para simular mediante bucles, la recursividad que existe en el código para llevar a cabo los distintos recorridos. Los elementos de esta pila apuntan a los nodos del octree que hay que visitar en las siguientes iteraciones del bucle.



(a) (b)

Figura 4.21: Estructura de datos para el código Barnes-Hut (a), y representación compacta del RSRSG obtenido en uno de los recorridos del octree.

Por lo demás, el código es igual, con sus tres fases: i) creación del octree, ii) recorrido para determinar el centro de masas y la masa total de cada nodo y iii) recorridos del octree, para cada cuerpo de *Lbodies*, calculando la fuerza total sobre dicho cuerpo.

En la figura 4.21(b) presentamos una representación simplificada del RSRSG obtenido para la primera sentencia del bucle que recorre el octree en el paso iii). El RSRSG obtenido

para la última sentencia del código es el mismo, pero eliminando completamente los nodos y enlaces de la pila (estructura apuntada por *Stack*), y con la propiedad $SHARED = false$ para los nodos n_2 y n_3, puesto que ya no son apuntados desde la pila. Tenemos que indicar que los RSRSGs obtenidos en los tres niveles del análisis para esta última sentencia son iguales, es decir, en los tres niveles el análisis es capaz de determinar las características de la estructura de datos que se crea.

Si nos centramos en las características que se pueden deducir de este RSRSG, indicando que todos los nodos tienen el atributo *SHSEL* igual a *false* para todos los selectores, (dejando por ahora de lado la pila y los enlaces al octree) vemos que:

- la variable *Root* apunta a una estructura formada por elementos que en su interior tiene un multiselector *child* (por claridad, tan solo se han dejado las instancias $\emptyset$). Como se aprecia, por el multiselector *child* puede apuntar a dos tipos de nodos. El nodo n_2, que representa porciones de memoria que apuntan por *child* a otras, y no apuntan a nada por *body*, es decir los elementos centrales del árbol. Por otro lado, el nodo n_3 representa porciones de memoria que no apuntan a ninguna otra por *child* pero si apuntan a elementos de la lista *Lbodies* por *body*, o sea, representa las hojas del árbol.

- Siguiendo cualquier recorrido por enlaces de *child*, nunca se visitará dos veces el mismo elemento del octree, puesto que los nodos n_2 y n_3 no son *shared* por ningún selector, por tanto una misma porción de memoria por ellos representada no es apuntada por dos enlaces *child* (ya sean de distintas porciones de memoria o de una misa por diferentes posiciones del array).

- Las porciones representadas por n_3 (las hojas) apuntan por *body* a diferentes cuerpos de la lista *Lbodies*, puesto que los nodos que la representan (n_4, n_5 y n_6) no son *shared* por el selector *body* y por tanto ninguna de las porciones allí representadas puede ser apuntada por dos selectores *body* a la vez.

Como vemos, del RSRSG se pueden deducir las principales características de la estructura de datos. Como hemos dicho, se obtiene el mismo RSRSG para la última sentencia del código en los tres niveles del análisis. Sin embargo, el RSRSG presentado en la figura 4.21(b) para el bucle del paso iii) del algoritmo, tan solo se obtiene en el nivel L_3. Para los otros dos niveles, los nodos n_2 y n_3 aparecen con el atributo *SHSEL* para el selector *node* igual a *true*. El motivo es el mismo expuesto en la sección 3.5.6, y es porque en el recorrido no se mantienen por separado los nodos visitado y por tanto introducidos en la pila, de aquellos que aún no lo han sido. Esto provoca que un elemento pueda ser introducido varias veces en la pila, ya que no se tiene en cuenta que ningún elemento es visitados dos veces en ese recorrido. En el nivel L_3, como vimos, el uso de la información *TOUCH* sobre las variables que han visitado cada nodo en los recorridos, evita que se mezclen y por eso se obtiene una representación más exacta.

Tenemos que decir, que de nuevo para el bucle que realiza el recorrido del paso ii), los tres niveles devuelven $SHSEL = true$ para los nodos n_2 y n_3 y el selector *node*. Esto es debido, como ya dijimos, a que en este recorrido, realmente los elementos son visitados dos veces, una para seleccionar a sus hijos, y la segunda, para calcular los valores para su centro de masas y masa total, una vez calculada la de us hijos. De la información proporcionada en las sentencias condicionales del bucle, el análisis no puede deducir que en la segunda vez que se visita un nodo no se vuelven a introducir sus hijos en la pila. Por tanto, como ese es un camino válido en el grafo de flujo de control debe de ser analizado.

Sin embargo, para el bucle iii), del RSRSG obtenido en L_3 se puede extraer que los elementos del octree tan solo aparecen una sola vez en la pila, y por tanto no se va a visitar dos veces el mismo elemento del octree en el bucle. Esta información si que puede ser tenida en cuenta en un posterior análisis para una posible paralelización del recorrido del árbol.

Para terminar, de nuevo vemos que para este código se produce una situación peculiar, y es que el nivel L_3 consume menos recursos que los dos anteriores, aunque utiliza una representación más compleja (como vemos el aumento de complejidad entre L_2 y L_3 se ve reflejado en un aumento de los requerimientos en tiempo y memoria). La explicación es la misma que ya dimos en la sección 3.5.6, y es que, en los niveles L_1 y L_2, todos los recorridos de los bucles de paso iii) se realizan con grafos donde los nodos del octree son *shared* por el selector *body*. Esto provoca la aparición de más nodos (más diversidad en la información *shared*) y enlaces, puesto que a la hora de enfocar (la operación de *poda* sobre todo) no se pueden eliminar del grafo tantos elementos, ya que la información *shared* a *true* hace que el método se comporte más conservativamente. Cuando se utiliza $TOUCH$ en L_3 y desaparece el valor *true* de *SHSEL*, los grafos obtenidos son mucho más simples puesto que pueden ser *podados* más eficazmente.

Conclusiones y principales aportaciones

El objetivo de esta tesis ha sido el análisis automático de la forma que toman las estructuras dinámicas de datos en códigos que utilizan punteros, como un paso hacia la resolución del problema de la paralelización automática de este tipo de códigos. La motivación de esta elección ha sido el gran auge que en los últimos tiempos están teniendo lenguajes de programación como C, C++, Fortran90, Java, etc., como herramientas para la resolución de problemas irregulares y simbólicos. Su potencia radica en la facilidad con la que se pueden crear estructuras dinámicas muy complejas que aumentan el rendimiento de los códigos además de facilitar la tarea de desarrollo. El problema es el desconocimiento del compilador sobre la forma de estas estructuras dinámicas que son creadas y modificadas en tiempo de ejecución. Sin este conocimiento no se puede llevar a cabo un análisis de dependencias preciso y por tanto es muy difícil la tarea de determinar que porciones de código que acceden a estas estructuras son independientes y pueden ser ejecutados en paralelo. Nuestro objetivo ha sido diseñar las técnicas de compilación que permitan obtener de forma automática información sobre la forma que pueden tomar las estructuras dinámicas en cada punto del programa para que en base a ella se lleve a cabo un análisis de dependencias mucho más preciso.

Las principales aportaciones y líneas de investigación que pretendemos seguir en el futuro se discuten a continuación.

Principales aportaciones

A continuación se recogen los principales aspectos desarrollados en este trabajo, cuyo objetivo ha sido la obtención de método para descubrir las características de las estructuras dinámicas de datos usadas en códigos reales escritos en C.

- Se han estudiado los métodos existentes para tratar de descubrir la forma de las estructuras dinámicas de datos en programas que utilizan punteros. Dentro de los mismos hay distintos modos de enfocar el problema. Están los basados en anotaciones del programador, de utilidad para la paralelización semi-automática, pero no para la automática. Otros utilizan una descripción de las relaciones existentes entre las variables puntero usadas en el programa, que producen unos resultados bastante conservadores. Y por último, los basados en grafos, que describen las propiedades de las estructuras mediante grafos. Entre estos nos encontramos métodos muy diferentes con menor y mayor éxito según el tipo de estructuras analizadas. En ningún caso se obtienen resultados satisfactorios para estructuras complejas de códigos reales.

- De entre los métodos basados en grafos, el que mejores resultados obtiene es el de Sagiv et al [67] con sus *Static Shape Graph* (SSG), aunque no puede reconocer estructuras doblemente enlazadas. Hemos propuesto e implementado una mejora sobre su método, permitiendo que en los SSGs existan más de un nodo sumario y que contengan más información por nodo. Esto permite que el método sea capaz de reconocer estructuras más complejas como la utilizada por un algoritmo de descomposición LU de una matriz dispersa. Precisamente se ha analizado con la implementación del método un código sintético que crea y modifica la misma estructura utilizada en dicho algoritmo, obteniéndose unos resultados satisfactorios.

- Aunque la mejora sobre los SSGs a permitido el reconocimiento de estructuras más complejas, algunas de sus peculiaridades hacen que el método falle a la hora de analizar núcleos de algoritmos reales. Para solucionar esto, hemos propuesto un nuevo método basado en grafos, denominado *Conjunto Reducido de Grafos de Forma de Referencias* (RSRSG). Este método utiliza un conjunto de grafos para describir las posibles configuraciones de memoria que pueden aparecer tras la ejecución de una sentencia. Esto evita la mezcla de informaciones de configuraciones de memoria muy distintas que se producía con los SSGs. Además, los *Grafos de Forma de Referencias* (RSGs), han cambiado la manera de representar a las porciones de memoria, centrándose más en *cómo son referenciadas* y manteniendo por separado aquellas porciones que son singulares en relación a sus enlaces. Esto se ha conseguido asociando a cada nodo un conjunto nuevo de propiedades, reutilizando algunas de las propuestas en los SSGs. Otra mejora fundamental para aumentar la precisión con la que las sentencias modifican los grafos, ha sido el *enfoque* de las referencias que realmente son utilizadas por dichas sentencias. De este modo se hace menos conservativa la interpretación de la sentencia sobre los grafos.

  Todas estas mejoras han llevado consigo un aumento de los requerimientos en tiempo y memoria necesarios para llevar a cabo el análisis. Hemos detectado que no todos los códigos necesitan ser analizados al máximo de la complejidad del método, sino que reduciento las capacidades del análisis también se pueden obtener buenos resultados. Por ello hemos desarrollado tres niveles de análisis que se diferencian en la complejidad menor o mayor del método. El nivel más bajo utilizan menos propiedades por lo que necesita menos recursos durante el análisis. Si los resultados obtenidos en un nivel no son satisfactorios describiendo la estructura de datos, el análisis pasa al siguiente nivel. Se termina en el tercer nivel donde se aplica toda la potencia del método y por tanto se obtienen los resultados más precisos.

  Con la implementación del método RSRSG, en un *pseudocompilador* que lee código C y devuelve el RSRSG asociado a cada sentencia, se han analizado una serie de códigos reales basados en estructuras de datos dinámicas como son la multiplicación de una matriz dispersa por un vector, la multiplicación de dos matrices dispersas, la factorización LU de una matriz dispersa y el núcleo del algoritmo de simulación N-Body Barnes-Hut. Los RSRSGs obtenidos para cada código describen con gran precisión las estructuras dinámicas utilizadas.

- También hemos creado una abstracción nueva, los *multiselectores*, dentro de los RSRSGs para el soporte de los arrays de punteros como un elemento nuevo dentro de las porciones de memoria. Ningún método que nosotros conozcamos soporta los arrays de punteros como selectores de las porciones de memoria, y sin embargo muchos códigos irregulares los utilizan como una manera fácil y óptima para crear sus estructuras de datos. Se

ha modificado la manera de *enfocar* los enlaces de las sentencias para dar soporte a las peculiaridades de los *multiselectores* sin que tenga que ser demasiado conservativa la interpretación de las sentencias que los utilizan.

Esta nueva abstracción se ha implementado en el *pseudocompilador* y se han analizado los mismos códigos anteriores, pero utilizando los arrays de punteros dentro de las estructuras dinámicas que utilizan. De los RSRSGs obtenidos se han podido deducir las características principales de dichas estructuras de datos.

- Hasta donde nosotros conocemos, ninguno de los métodos propuestos anteriormente para detectar la forma de las estructuras de datos dinámicas de códigos que utilizan punteros, es capaz de obtener resultados satisfactorios para el tipo de estructuras y códigos para los que nuestro método se comporta de forma correcta.

Líneas de investigación futura

En este trabajo se ha presentado un método basado en grafos para la detección automática de estructuras dinámicas de datos, dando un primer paso en su implantación en el análisis de códigos reales. En este sentido, es muy común en este tipo de códigos la creación de las estructuras dinámicas por medio de llamadas recursivas a procedimientos. Por lo tanto, una de las ideas de trabajo futuro es la ampliación del método para el soporte de análisis interprocedural de forma que evitemos el paso de *inlining* de las llamadas a procedimientos y se soporte la recursividad.

Como hemos expuesto en esta memoria, aunque el método puede tarbajar con estructuras dinámicas muy complejas, hay códigos que por su complejidad hacen que el método se comporte conservativamente. Este comportamiento es debido al desconocimiento en tiempo de ejecución de los caminos del grafo de flujo de control que realmente pueden aparecer en tiempo de ejecución. Una línea de trabajo futuro será intentar llevar a cabo análisis del código mucho más complejos para acercar los caminos de flujo analizados a los reales, en la línea de las *pseudoinstrucciones* $FORCE$ del método.

En cuanto a la paralelización automática sería necesario trabajar en la obtención de un método de análisis de dependencias eficaz que utilice la información que se puede extraer de los RSRSGs asociados a cada sentencia del código. De esta información y de un estudio de como son recorridas las estructuras en el código, el análisis deberá determinar las secciones de código que pueden ser ejecutados en paralelo. Como último paso podríamos divisar el problema de cómo paralelizar recorridos sobre estructuras dinámicas de datos.

Tampoco descartamos la idea del refinamiento del método, añadiendo nuevas propiedades para el manejo de estructuras de datos mucho más complejas.

Apéndice A: Semántica Abstracta modificada de los SSGs

En este apéndice vamos a presentar las modificaciones llevadas a cabo sobre la semántica abstracta de las sentencias, para que los SSGs puedan contener cada una de los nuevos atributos que hemos asociado a los nodos, presentadas en el capítulo 2.

A.1 Atributo TYPE

Para que el método pueda manejar la información $type^{\#}$ ha sido necesario modificar de forma leve la semántica abstracta ([67]) de algunas sentencias. A continuación presentamos las variaciones en la semántica abstracta de aquellas sentencias que se ven afectadas.

1. Sentencia [x:= new]

   Esta sentencia reserva una nueva porción de memoria, que será representada por un nuevo nodo $n_{\{x\}}$ (puesto que x es la única variable que lo referencia). La única operación nueva que hay que realizar para manejar $type^{\#}$, es poner como *tipo* del nuevo nodo el mismo que el de la variable x. De este modo:

   $$type^{\#\prime}(n_{\{x\}}) = type\_var(x)$$

   el nuevo valor del atributo $type^{\#}$ ($type^{\#\prime}(n_{\{x\}})$) es tomado del *tipo* de la variable x.

2. Sentencia [x:= y]

   La ejecución de esta sentencia, hace que los nodos n_Z en los que $y \in n_Z$ sean sustituidos por nodos $n_{Z'}$ donde $Z' = Z \cup \{x\}$. Por tanto los nuevos nodos toman el valor $type^{\#}$ que tenían los nodos apuntados por y.

   $$type^{\#\prime}(n_Z) = type^{\#}(n_{Z-\{x\}})$$

   El nuevo *tipo* de cada nodo del grafo $type^{\#\prime}(n_Z)$ es el mismo que tenía antes de que x fuera añadida a Z.

3. Sentencia [x:= y.sel]

   En esta sentencia se produce la *materialización* de un nuevo nodo desde el nodo apuntado por el selector *sel* desde el nodo referenciado por la variable y, de manera que será el único nodo sobre el que incida dicho selector. Al igual que en el caso anterior, el nuevo nodo *materializado*, ahora apuntado por la variable x, tendrá igual *tipo* que el nodo desde el que fue *materializado*

$$type^{\#\prime}(n_Z) = type^{\#}(n_{Z-\{x\}})$$

4. La operación de *sumarización* también se ve afectada por esta nueva información. La *sumarización* une la información de aquellos nodos que tras alguna modificación por parte de la semántica abstracta de alguna sentencia, han pasado a tener el mismo "nombre" (mismo conjunto de variables referenciándolos). Ahora, con la información $type^{\#}$, para que dos nodos n_X y n_Y sean sumarizados, no sólo tienen que tener el mismo nombre, $X = Y$, sino que tienen que representar porciones de memoria del mismo *tipo*, $type^{\#}(n_X) = type^{\#}(n_Y)$.

5. Las operaciones de *unión* y *comparación* de grafos, usadas en la *interpretación abstracta* del programa, hacen comparaciones de nodos (en la *unión* para buscar que nodos de los distintos grafos deben ser unidos y en la *comparación* para buscar nodos equivalentes en ambos grafos). Ahora además de comprobar que el conjunto de variables que referencian los nodos es el mismo, también es necesario que tengan igual $type^{\#}$. En realidad esta restricción sólo afecta a los nodos sumario $n_{\emptyset}$, puesto que los demás nodos que son apuntados por al menos una variable puntero, serán del mismo *tipo* que dicha variable.

A.2 Atributo STRUCTURE

A continuación pasamos a describir las modificaciones sobre la semántica abstracta de algunas sentencias, necesarias para el manejo de la información $structure^{\#}$.

1. Sentencia [x:= y]

   La ejecución de esta sentencia no provoca cambios en la conectividad del grafo, puesto que lo único que hace es hacer que x apunte a los nodos referenciados por y. Por este motivo, los nuevos nodos, ahora apuntados también por x pertenecerán a la misma estructura a la que pertenecían antes de que x les apuntara.

$$structure^{\#\prime}(n_Z) = structure^{\#}(n_{Z-\{x\}})$$

2. Sentencia [x.sel:= nil]

   Esta sentencia elimina el enlace por el selector *sel* que hay desde la porción de memoria apuntada por x, y por tanto puede partir en dos la componente conexa a la que pertenece.

   $\forall n_X, x \in X, < n_X, sel, n_Z > \in Es^{\#}$:

   - if $C[Es^{\#\prime}](n_X) \cap C[Es^{\#\prime}](n_Z) = \phi$ then
     $\forall n \in C[Es^{\#\prime}](n_X), structure^{\#\prime}(n) = new\_structure_1$,
     $\forall m \in C[Es^{\#\prime}](n_Z), structure^{\#\prime}(m) = new\_structure_2$

- if $C[Es^{\#\prime}](n_X) \cap C[Es^{\#\prime}](n_Z) \neq \phi$, el atributo $structure^{\#}$ no cambia.

Si tras la *interpretación abstracta* de la sentencia, las componentes conexas de los nodos n_X ($C[Es^{\#\prime}](n_X)$) y n_Z ($C[Es^{\#\prime}](n_Z)$) no tienen ningún nodo en común, es que se ha roto la componente conexa en dos, y lo que se hace es asociar a cada nodo de esas nuevas componentes conexas un nuevo valor para el atributo *structure*, distinto en ambos casos.

3. Sentencia [x.sel:= y]

   Antes de presentar los cambios en la semántica abstracta asociada a esta sentencia, hay que decir que dicha sentencia no rompe ningún enlace, puesto que antes de que cualquier código sea analizado, es transformado de manera que antes de cualquier asignación a x o $x.sel$ va precedida de las sentencias $x := nil$ o $x.sel := nil$ respectivamente. Con esto se pretende pasar todas las modificaciones debidas a la eliminación de enlaces a dichas sentencias, eliminándolas de todas las demás.

   Es por ese motivo que esta sentencia no rompe ningún enlace, centrándonos sólo en el enlace que crea entre los nodos apuntados por x e y por medio del selector sel.

   $\forall n_X, n_Y, [x, n_X], [y, n_Y] \in Ev^{\#\prime}, < n_X, sel, n_Y > \in Es^{\#\prime}, compatible^{\#}(n_X, n_Y)$:

   - $\forall n \in C[Es^{\#\prime}](n_X), \forall m \in C[Es^{\#\prime}](n_Y)$
     $structure^{\#}(n) = structure^{\#}(m) = new\_structure$

   Tras la ejecución abstracta de esta sentencia, se crea un enlace entre los nodos apuntados por las variables x e y, por lo que, tanto dichos nodos como todos aquellos conectados con ambos pasan a formar parte de la misma componente conexa y es por eso que el atributo $structure^{\#}$ de todos ellos toma el mismo valor.

4. Sentencia [x:= y.sel]

   En esta sentencia tampoco se ve afectada la conectividad del grafo, puesto que lo único que implica es la utilización de un enlace por el selector sel para seleccionar el nuevo elemento al que apunte x. En definitiva se esta "recorriendo" la estructura sin modificarla.

   Como se comentó anteriormente, esta sentencia provoca la materialización de un nodo que es el que representa aquellas porciones de memoria realmente referenciadas por sel desde las porciones de memoria referenciadas por y. El atributo $structure^{\#}$ del nuevo nodo, es por tanto, el mismo que el del nodo del cual se materializa.

   $$structure^{\#\prime}(n_Z) = structure^{\#}(n_{Z-\{x\}})$$

5. En cuanto a la *sumarización*, al igual que con el atributo $type^{\#}$, de nuevo sólo aquellos nodos no referenciados por ninguna variable puntero y que su $structure^{\#}$ sea la misma, serán sumarizados. Esto provoca que porciones de memoria, incluso del mismo *tipo*, pertenecientes a distintas estructuras que no comparten ningún elemento, sean representadas en diferentes nodos sumario, evitando así mezclar características de estructuras distintas.

6. Las operaciones de *unión* y *comparación* de grafos se ven afectadas de igual forma que lo fueron con la inclusión de la información sobre el *tipo*. La nueva condición

que se impone ahora para que dos nodos sean considerados como representantes de las mismas porciones de memoria es además de que sus "nombres" coincidan, que su atributo $structure^{\#}$ también sea igual.

A.3 Atributo IS_SEL

Aparte de la introducción de esta nueva información en cada nodo, hemos tenido que modificar la semántica abstracta de las sentencias para mantener y utilizar de forma adecuada este nuevo atributo. Dichas modificaciones han sido las siguientes:

1. Sentencia [x:= nil]

   Esta sentencia hace que las porciones de memoria referenciadas por x ya no lo sean y por tanto, el nodo que las representa pierda dicha variable de su nombre. Esto puede provocar la sumarización de dicho nodo con algún otro. La información $is\_sel^{\#}$ para cada selector del nuevo nodo, debe ser conservativa con respecto a la que poseían los nodos sumarizados. Si en alguno de ellos era *true* para algún selector, debe seguir siendolo en el nodo sumarizado. Por tanto:

   $$is\_sel^{\#\prime}(n_Z, sel) = is\_sel^{\#}(n_Z, sel) \vee is\_sel^{\#}(n_{Z\cup\{x\}}, sel)\ \forall sel$$

   Como se ve la información $is\_sel^{\#\prime}(n_Z, sel)$ será *false*, sólo si lo era en los dos nodos que se han sumarizado.

2. Sentencia [x:= new]

   Cuando se crea un nuevo nodo ($n_{\{x\}}$) para representar las nuevas porciones de memoria apuntadas por x, la información $is\_sel^{\#}$ es inicializada a *false* para todos los selectores puesto que son nuevas porciones de memoria no referenciadas desde ninguna otra.

   $$is\_sel^{\#\prime}(n_{\{x\}}, sel) = false\ \forall sel$$

3. Sentencia [x:= y]

   Esta sentencia no provoca ningún cambio en la información $is\_sel^{\#}$ ya que las conexiones de las estructuras de datos no varían.

   $$is\_sel^{\#\prime}(n_Z, sel) = is\_sel^{\#}(n_{Z-\{x\}}, sel)\ \forall sel$$

   La información $is\_sel^{\#}$ permanece igual que cuando x no formaba parte del nombre del nodo.

4. Sentencia [x.sel:= nil]

   La ejecución de esta sentencia puede romper enlaces por el selector sel desde las porciones de memoria referenciadas por x. Por tanto es posible que nodos que representan las porciones de memoria que eran referenciadas por $x.sel$ con $is\_sel^{\#}(n_Z, sel) = true$, ahora pase a ser *false* si ya no hay más de un enlace por el selector sel sobre dichas porciones de memoria.

   Para poder manejar adecuadamente el atributo $is\_sel^{\#}$, definimos la siguiente función:

$$iss\_sel^{\#}[Es^{\#}](n, sel) = \begin{cases} true \text{ si } \exists n_{Z1}, n_{Z2}, compatible^{\#}(n_{Z1}, n_{Z2}, n) \wedge \\ \quad < n_{Z1}, sel, n >, < n_{Z2}, sel, n > \in Es^{\#} \wedge \\ \quad n_{Z1} \neq n_{Z2} \\ false \text{ en caso contrario.} \end{cases}$$

Esta función chequea si existen dos nodos n_{Z1} y n_{Z2} "compatibles" con el nodo n de manera que ambos lo referencien por medio del selector sel. La función de "compatibilidad" ($compatible^{\#}$) como ya comentamos, indica si varios nodos representan porciones de memoria que pueden pertenecer a un mismo estado de la memoria, o si por el contrario representan porciones de memoria que nunca pueden aparecer en un mismo estado de la memoria del programa.

Hemos complementado la semántica abstracta de esta sentencia para que calcule el nuevo valor del atributo $is\_sel^{\#\prime}$ de la siguiente manera:

$$is\_sel^{\#\prime}(n, sel) = \begin{cases} is\_sel^{\#}(n, sel) \wedge iis\_sel^{\#}[Es^{\#\prime}](n, sel) \text{ si} \\ \quad \exists n_X, [x, n_X] \in Ev^{\#} \wedge < n_X, sel, n > \in Es^{\#} \\ is\_sel^{\#}(n, sel) \text{ en otro caso.} \end{cases}$$

Lo que se hace es que la información $is\_sel^{\#\prime}(n, sel)$, si el nodo n era referenciado por $x.sel$ (nodo n_X por el selector sel), va a depender de la función $iis\_sel^{\#}[Es^{\#\prime}](n, sel)$. Si dicha función no encuentra dos nodos compatibles con n de forma que ambos lo referencien por sel, $is\_sel^{\#\prime}(n, sel)$ pasa a ser *false* para informar de esta nueva situación.

5. Sentencia [x.sel = y]

   Ahora se crea un nuevo enlace entre la porción de memoria apuntada por x y la apuntada por y, por medio del selector sel, por lo que la información $is\_sel^{\#}$ del nodo que representa las porciones de memoria apuntadas por y puede pasar a ser *true* debido a la incorporación del nuevo enlace. Tomará el valor *true* si existen dos nodos compatibles con el que representa la porción de memoria referenciada por y, que tras la ejecución de la sentencia, tienen un enlace sobre dicho nodo por el selector sel. De nuevo, para chequear esta condición se utiliza la función $iis\_sel^{\#}[Es^{\#\prime}](n, sel)$ descrita en el apartado anterior.

$$is\_sel^{\#\prime}(n, sel) = \begin{cases} is\_sel^{\#}(n, sel) \vee iis\_sel^{\#}[Es^{\#\prime}](n, sel) \text{ si } [y, n] \in Ev^{\#} \\ is\_sel^{\#}(n, sel) \text{ en otro caso.} \end{cases}$$

El atributo $is\_sel^{\#}(n, sel)$ pasa a ser *true* sólo en caso de que tras la ejecución de la sentencia, haya al menos dos nodos compatibles con n y que posean enlaces por sel sobre él (en caso de que $iis\_sel^{\#}[Es^{\#\prime}](n, sel) =$*true*).

Esta sentencia es la única que puede hacer que la información *shared* por selector de un nodo pase a ser *true*, puesto que crea un nuevo enlace entre las dos porciones de memoria apuntadas por las variables x e y. Esto se traduce en el SSG en la creación de un nuevo enlace por el selector sel entre el nodo que representa la porción de memoria apuntada por x (n_x) y el que representa la apuntada por y (n_y). Si el nodo n_y ya era referenciado por sel desde otro nodo n_z antes de la interpretación abstracta de la sentencia, es posible que ahora, algunas de las porciones de memoria representadas

por n_y sean referenciadas desde más de un sitio por el selector sel. El único caso en el que podemos estar seguros de que no existe esta posibilidad y por tanto no dar el valor *true* a $is\_sel^{\#}(n_y, sel)$ es que las porciones de memoria representadas por el nodo n_x no puedan pertenecer a un mismo estado de la memoria que las representadas por n_z, lo que implicaría que los enlaces de n_x a n_y y de n_z a n_y realmente pertenecen a configuraciones de memoria distintas. Esta información la podemos obtener de los "nombres" de los nodos por medio de la función $compatible^{\#}$ presentada anteriormente, y que es usada por la función $iis\_sel^{\#}[Es^{\#\prime}]$.

6. Sentencia [x:= y.sel]

   Esta es una sentencia muy importante a la hora de ver la utilidad de la información mantenida por el atributo $is\_sel^{\#}$ de cada nodo. Como hemos comentado ya, es en esta sentencia donde se produce la materialización de un nuevo nodo y por tanto vamos a necesitar toda la información disponible en el nodo del que es materializado, para regenerar con la mayor exactitud posible, los enlaces sobre los nodos que representan las porciones de memoria apuntadas por x e y.

   Antes de ver como el nuevo atributo afecta a la generación de los enlaces del nodo materializado, debemos decir que al ser ésta una sentencia en la que no se ve modificado ningún enlace, la información $is\_sel^{\#}$ de todos los nodos permanece inalterada. tan solo hay que tener en cuenta el valor de dicho atributo para el nuevo nodo materializado, que por otro lado será exactamente igual al valor de $is\_sel^{\#}$ del nodo del cual se materializa.

   Por tanto:

$$is\_sel^{\#\prime}(n_Z, seli) = is\_sel^{\#}(n_{Z-\{x\}}, seli) \quad \forall seli$$

   Se deja el atributo $is\_sel^{\#}$ igual para todos los nodos, y se copia sobre el nuevo nodo materializado, ahora referenciado por x, la misma información que tenia el nodo del que ha sido materializado ($n_{Z-\{x\}}$, antes de ser apuntado por la variable x).

   Como hemos comentado anteriormente, esta sentencia provoca la materialización de un nuevo nodo $n_{Z\cup\{x\}}$ a partir del nodo n_Z referenciado por el selector sel desde el nodo apuntado por la variable y (n_Y). Ésto se hace para separar en distintos nodos las porciones de memoria representadas por n_Z que realmente son apuntadas por $y.sel$ de aquellas que no lo son. Las primeras estarán ahora representadas por el nuevo nodo materializado $n_{Z\cup\{x\}}$ el cual a su vez es apuntado por la pvar x. A la hora de crear los enlaces desde y hacia el nuevo nodo, se toman todos aquellos que posee el nodo del cual se ha materializado n_Z. Para poder eliminar enlaces que realmente no pueden aparecer en el nuevo nodo, se utiliza la información almacenada en cada nodo.

   En definitiva el método original utiliza varias funciones booleanas para determinar si varios enlaces "son compatibles" en un nodo, es decir, si pueden existir a la vez en dicho nodo teniendo en cuenta la información almacenada en el mismo. En el método original la información de que se dispone es tan solo de $is^{\#}$ y por tanto sólo se puede utilizar este atributo a la hora de determinar si varios enlaces pueden aparecer en un determinado nodo. Nosotros hemos modificado dichas funciones para que tengan en cuenta además la nueva información proporcionada por $is\_sel^{\#}$ que va a permitir una reconstrucción más fiel de los enlaces del nodo materializado.

   En concreto se tienen que modificar dos funciones:

- Función $compat\_in^{\#}([y, n_Y], < n_Y, sel, n_Z >, < n_W, sel', n_Z >)$, determina si el nodo n_Z puede ser referenciado desde n_Y por sel y desde otro nodo n_W por sel'. Como hemos dicho el nuevo nodo materializado se extrae de n_Z, y se materializa para ser el destino univoco de $y.sel$, por tanto es seguro que va a haber un enlace por sel entre n_Y y $n_{Z\cup\{x\}}$. Esta función es usada para determinar si el enlace que hay entre el nodo n_W y n_Z por sel' debe ser trasladado sobre el nuevo nodo. Para determinar esto, la función se define como:

  $$compatible^{\#}(n_Y, n_Z, n_W) \wedge [y, n_Y] \in Ev^{\#} \wedge$$
  $$< n_Y, sel, n_Z >, < n_W, sel', n_Z > \in Es^{\#} \wedge n_Z \neq n_W \wedge$$
  $$((n_Y = n_W \wedge sel = sel') \vee is^{\#}(n_Z))$$

  donde se ve que para que sean compatibles dichos enlaces y se traslade al nuevo nodo materializado, dicho nodo tiene que ser "shared". Esto es así, puesto que si el nodo es "shared" puede ser referenciado desde más de una porción de memoria, por tanto aunque es referenciado desde n_Y por sel también puede serlo desde n_W por sel'.

  Esto implica que en cuanto el nodo sea "shared", cuando se materialize otro nodo desde él, tiene que ser apuntado desde todos los nodos que apuntaban al original, evitando la eliminación de enlaces que no se corresponden con los estados de memoria representados por el grafo.

  Nosotros hemos cambiado la definición de dicha función para que tenga en cuenta la nueva información proporcionada por el atributo $is\_sel^{\#}$, permitiendo la eliminación de más enlaces sobre el nuevo nodo materializado. La nueva definición de la función queda de la siguiente manera:

  $$compatible^{\#}(n_Y, n_Z, n_W) \wedge [y, n_Y] \in Ev^{\#} \wedge$$
  $$< n_Y, sel, n_Z >, < n_W, sel', n_Z > \in Es^{\#} \wedge n_Z \neq n_W \wedge$$
  $$((n_Y = n_W \wedge sel = sel') \vee (is^{\#}(n_Z) \wedge (sel \neq sel')) \vee$$
  $$(is\_sel^{\#}(n_Z, sel) \wedge (sel = sel')))$$

  Como podemos observar ahora, el enlace desde n_W por sel' sobre el nodo materializado puede coexistir con el enlace desde n_Y por sel si: (i) el nodo es "shared" y $sel \neq sel'$ o, (ii) si el nodo es "shared" por el selector sel y $sel = sel'$. La información disponible ahora hace que la selección de los enlaces sobre el nodo materializado sea más selectiva. Así aunque el nodo sea "shared", si no es "shared" por el selector sel, no habrá ningún nodo distinto a n_Y que apunte al nodo nuevo por sel.

- Función $compat\_self^{\#}([y, n_Y], < n_Y, sel, n_Z >, < n_Z, sel', n_Z >)$. Es similar a la anterior, pero ahora determina si un selector desde el nodo n_Z por sel' apuntándose a él mismo, puede ser trasladado a un selector del nuevo nodo $n_{Z\cup\{x\}}$ a él mismo por sel', sabiendo que n_Y definitivamente apunta al nuevo nodo.

  De nuevo, para determinar esta compatibilidad, el método original sólo hace uso de la información $is^{\#}$. La función queda definida como:

  $$compatible^{\#}(n_Y, n_Z) \wedge [y, n_Y] \in Ev^{\#} \wedge$$
  $$< n_Y, sel, n_Z >, < n_Z, sel', n_Z > \in Es^{\#} \wedge$$
  $$((n_Y = n_Z \wedge sel = sel') \vee is^{\#}(n_Z))$$

si el nodo es "shared", todo enlace de n_Z a n_Z se va a trasladar al nuevo nodo materializado, independientemente del selector que sea utilizado.

De nuevo, hemos modificado dicha función para que tome en consideración la nueva información $is\_sel^{\#}$, quedando la función como sigue:

$$\begin{array}{l} compatible^{\#}(n_Y, n_Z) \wedge [y, n_Y] \in Ev^{\#} \wedge \\ < n_Y, sel, n_Z >, < n_Z, sel', n_Z > \in Es^{\#} \wedge \\ ((n_Y = n_Z \wedge sel = sel') \vee (is^{\#}(n_Z)) \wedge (sel \neq sel')) \vee \\ (is\_sel^{\#}(n_Z, sel) \wedge (sel = sel'))) \end{array}$$

Ahora dependiendo del si el selector es igual o no a sel (selector desde n_Y al nuevo nodo) y de la información de si el nodo es "shared" o "shared por algún selector", los "auto-selectores" de n_Z son o no trasladados al nodo materializado. En el método original si el nodo era "shared" todos los "auto-selectores" de n_Z eran trasladados al nuevo nodo $n_{Z \cup \{x\}}$. Sin embargo, ahora, aunque sea "shared", si no es "shared" por el selector sel, no se pasarán aquellos "auto-selectores" que utilizan sel, puesto que ya hay un nodo (n_Y) que referencia a $n_{Z \cup \{x\}}$ por sel.

A.4 Atributo CYCLELINKS

A continuación, como con las propiedades anteriores, vamos a presentar las modificaciones que hemos tenido que llevar a cabo sobre la semántica abstracta de cada sentencia para la creación, mantenimiento y utilización de esta nueva propiedad.

1. Sentencia [x:= nil]

   Como ya hemos comentado en propiedades anteriores para esta sentencia, cuando la variable x deja de apuntar a un nodo, el "nombre" de dicho nodo cambia y por tanto si toma el mismo "nombre" que otro nodo existente en el grafo, ambos nodos son sumarizados. Por tanto, en lo que respecta a la propiedad $cyclelinks^{\#}$ del nuevo nodo, debe ser conservativa y mantener sólo aquellas parejas de selectores que son "compatibles" en ambos nodos.

   Pero ¿que es lo que consideramos como "compatibilidad" de *cycle link* entre dos nodos?. Vayamos por partes: si una pareja de selectores $<sel_1, sel_2>$ aparece en ambos nodos, n_1 y n_2, dicha pareja debe estar en el conjunto de *cycle links* de nodo sumarizado, puesto que la propiedad se cumple en todas las porciones de memoria representadas por ambos nodos. Sin embargo hay casos en los que un par $<sel_1, sel_2>$ perteneciente sólo a $cyclelinks^{\#}(n_1)$ puede permanecer en los *cycle links* del nuevo nodo sumarizado. Esto es posible si el otro nodo a sumarizar, n_2 no tiene ningún enlace por el selector sel_1, representando porciones de memoria que no apuntan a ningún sitio por sel_1. En este caso, para el nuevo nodo sumarizado, se sigue manteniendo la propiedad de que siguiendo el enlace por sel_1 desde cualquiera de las porciones de memoria que representa se llega a otra porción de memoria que referencia a la primera por sel_2. Esto ya se cumplía para las representadas por n_1 (puesto que $<sel_1, sel_2>$ está en $cyclelinks^{\#}(n_1)$) y no es erróneo para las representadas por n_2 puesto que estas no tienen enlaces por sel_1. En definitiva, si tomo un enlace por sel_1 desde el nuevo nodo, este enlace corresponde a las porciones de memoria representadas por n_1 y por tanto cumple la propiedad cíclica con sel_2.

Formalmente definimos el conjunto de *cycle links* del nuevo nodo como:

$$
\begin{aligned}
&cyclelinks^{\#\prime}(n_Z) = \{< sel1, sel2 > | \\
&< sel1, sel2 > \in (cyclelinks^{\#}(n_Z) \ \cap \ cyclelinks^{\#}(n_{Z\cup\{x\}})) \vee \\
&< sel1, sel2 > \in cyclelinks^{\#}(n_Z) \wedge \nexists n, < n_{Z\cup\{x\}}, sel1, n > \in Es^{\#} \vee \\
&< sel1, sel2 > \in cyclelinks^{\#}(n_{Z\cup\{x\}}) \wedge \nexists n, < n_Z, sel1, n > \in Es^{\#}\}
\end{aligned}
$$

como se ve, el conjunto de *cycle links* de nodo sumario n_Z se compone por todos los *cycle links* que tenían dicho nodo y el nodo que era apuntado por x ($n_{Z\cup\{x\}}$), o aquellos pertenecientes a alguno de ellos de manera que el otro no tuviera ningún enlace por el primer selector del par.

2. Sentencia [x:= new]

   Esta sentencia debe inicializar el conjunto de *cycle links* del nuevo nodo creado, que será apuntado únicamente por la variable x. Dicho conjunto debe de estar vacío, puesto que dicho nodo no posee ningún enlace.

$$cyclelinks^{\#\prime}(n_{\{x\}}) = \emptyset$$

3. Sentencia [x:= y]

   La interpretación de esta sentencia provoca el cambio de "nombre" de los nodos apuntados por la variable y, añadiendo a dicho "nombre" la variable x. Por tanto lo único que hay que hacer es que el nuevo nodo (n_Z con $x, y \in Z$) conserve el conjunto de *cycle links* que poseía el original ($n_{Z-\{x\}}$).

$$cyclelinks^{\#\prime}(n_Z) = cyclelinks^{\#}(n_{Z-\{x\}})$$

4. Sentencia [x.sel:= nil]

   Puesto que esta sentencia elimina el enlace por el selector *sel* de la porción de memoria apuntada por x, el conjunto de *cycle links* del nodo que representa dicha porción de memoria puede verse modificado. Si el nodo n_X con $x \in X$, el cual esta representando todas las porciones de memoria referenciadas por la variable x, posee el *cycle link* $<sel, sel_i>$, debe ser eliminado puesto que ahora las porciones de memoria representadas por el nodo ya no poseen enlaces por el selector *sel*.

   Además de eliminar dichos *cycle links* de n_X, los conjuntos de *cycle links* de los nodos apuntados desde n_X por *sel*, también se ven modificados. En concreto, si un nodo n es referenciado por *sel* desde n_X, y n a su vez referencia a n_X por sel_j, si el *cycle link* $<sel_j, sel>$ pertenece a $cyclelinks^{\#}(n)$ hay que eliminarlo. Es necesaria su eliminación puesto que la porción de memoria referenciada por $x.sel$ que está representada en n, ya no cumple el *cycle link* $<sel_j, sel>$, puesto que es posible que apunte a la porción de memoria referenciada por x por sel_j y sin embargo ésta no la vuelve a referenciar por *sel* (enlace borrado por la sentencia). Como ya hay una porción de memoria que puede que no cumpla la propiedad impuesta por dicho *cycle link*, es eliminado del nodo que la representa.

   Los conjuntos de *cycle links* que se ven afectados son los siguientes:

$$cyclelinks^{\#\prime}(n) = \begin{cases} cyclelinks^{\#}(n) \setminus < sel, seli > \text{ if } [x,n] \in Ev^{\#} \wedge \\ \qquad < sel, seli > \in cyclelinks^{\#}(n) \\ cyclelinks^{\#}(n) \setminus < selj, sel > \text{ if } [x,n_X] \in Ev^{\#} \wedge \\ \qquad < n_X, sel, n > \in Es^{\#} \wedge < n, selj, n_X > \in Es^{\#} \wedge \\ \qquad < selj, sel > \in cyclelinks^{\#}(n) \\ cyclelinks^{\#}(n) \text{ en otro caso.} \end{cases}$$

como se puede apreciar tan solo se eliminan los *cycle links* del nodo apuntado por x (n_X) cuyo primer selector es sel, y de aquellos nodos apuntados por n_X por el selector sel y que a su vez vuelven a referenciarlo por el selector sel_j, el *cycle link* $<sel_j, sel>$ (si es que existe en el nodo).

5. Sentencia [x.sel:= y]

   La creación de un enlace entre la porción de memoria apuntada por x y la apuntada por y por el selector sel puede provocar la aparición de nuevos elementos en los conjuntos de *cycle links* de los nodos que representan ambas porciones de memoria (n_X, n_Y). En concreto en el nodo n_x se introducirá el *cycle link* $<sel, sel_j>$, si el nodo n_Y apunta sólo a n_X por sel_j. Por el mismo motivo se puede introducir el *cycle link* $<sel_j, sel>$ en el nodo n_Y si aún no existía.

   Los *cycle links* de algunos nodos se ven afectados de la siguiente manera:

$$cyclelinks^{\#\prime}(n) = \begin{cases} cyclelinks^{\#}(n) \cup < sel, seli > \text{ if } [x,n],[y,n_Y] \in Ev^{\#} \\ \qquad \wedge\ compatible^{\#}(n,n_Y) \wedge < n_Y, seli, n > \in Es^{\#} \\ \qquad \wedge\ \nexists n_Z, compatible^{\#}(n,n_Z), n \neq n_Z, \\ \qquad < n_Y, seli, n_Z > \in Es^{\#} \\ cyclelinks^{\#}(n) \cup < seli, sel > \text{ if } [x,n_X],[y,n] \in Ev^{\#} \\ \qquad \wedge\ compatible^{\#}(n,n_X) \wedge < n, seli, n_X > \in Es^{\#} \\ \qquad \wedge\ \nexists n_Z, compatible^{\#}(n_X,n_Z), n_X \neq n_Z, \\ \qquad < n, seli, n_Z > \in Es^{\#} \\ cyclelinks^{\#}(n) \text{ en otro caso.} \end{cases}$$

   Antes de introducir los nuevos *cycle links* se comprueba que no existe ningún otro nodo n_Z "compatible" con los otros, de manera que el nodo apuntado por y referencie a n_Z y a n_X por el mismo selector (sel_i). Si existe, entonces no se introduce el *cycle link* puesto que es posible que la relación "cíclica" entre las porciones de memoria apuntadas por x e y no se cumpla.

6. Sentencia [x:= y.sel]

   De nuevo en esta sentencia es cuando se va a mostrar el potencial de la nueva información proporcionada por el atributo *cycle links*. Sabemos que esta sentencia va a provocar la materialización de un nuevo nodo desde el nodo apuntado por y para representar las porciones de memoria que realmente son referenciadas por $y.sel$.

   Por un lado, el nuevo nodo conserva el mismo conjunto de *cycle links* que el nodo del cual se materializa, por tanto:

$$cyclelinks^{\#\prime}(n_Z) = cyclelinks^{\#}(n_{Z-\{x\}})$$

Presentamos ahora las modificaciones realizadas a la semántica abstracta de la sentencia $x := y.sel$, de manera que la información proporcionada por los *cycle links* sea usada para eliminar enlaces del nuevo nodo materializado.

Para ello, una vez materializado el nuevo nodo, ahora apuntado por x al que se le han añadido sus enlaces correspondientes, el nuevo conjunto de selectores $Es^{\#\prime}$ se ve modificado, eliminado aquellos que no satisfacen los *cycle links* de los nodos.

Definimos el nuevo conjunto de selectores $Es^{\#\prime\prime}$ como:

$$Es^{\#\prime\prime} = Es^{\#\prime} \setminus \{< n_1, sel_1, n_2 > | < sel_1, sel_2 > \in cliclelinks^{\#\prime}(n_1), \\ < n_1, sel_1, n_2 > \in Es^{\#\prime}, < n_2, sel_2, n_1 > \notin Es^{\#\prime}\}$$

Como se puede observar, se elimina un enlace $<n_1, sel_1, n_2>$ si el nodo n_1 posee el *cycle link* $<sel_1, sel_2>$ y sin embargo el nodo n_2 no referencia a n_1 por sel_2 como indica el *cycle link*.

Apéndice B: Semántica Abstracta de los RSRSGs

En este apéndice vamos a presentar las semántica abstracta de los RSRSGs, de cada una de las seis sentencias simples que manejan punteros. Como ya se comentó en el capítulo 2, cualquier sentencia más compleja con punteros, puede ser descompuesta en una secuencia de estas seis sentencias simples, utilizando variables puntero temporales.

Por tanto, aquí vamos a definir $ST_{[st]}(rsg)$, que dependiendo de la sentencia $[st]$, va a transformar el grafo rsg de entrada, en un conjunto de grafos que representan las configuraciones de memoria anteriormente representadas por rsg, pero tras sufrir las modificaciones debidas a la ejecución de la sentencia.

Las seis sentencias básicas con punteros son: $x = NULL$, $x = malloc()$, $x = y$, $x \rightarrow sel = NULL$, $x \rightarrow sel = y$ y $y = x \rightarrow sel$. A continuación pasamos a describir la semántica abstracta de cada una de ellas.

B.1 Sentencia $[x = NULL]$

El efecto de esta sentencia es que las porciones directamente apuntadas por la variable x, van a dejar de serlo. Por tanto, hay que eliminar cualquier destino de la variable puntero x en el grafo. Esta acción provoca cambios en los *simple paths* de algunos nodos. Puesto que x es una variable puntero, aparecerá en los *simple paths* de aquellos nodos "cercanos" al nodo apuntado por x, y por tanto su desaparición provoca la eliminación de dichos *simple paths*.

Definimos la semántica abstracta de la sentencia $[x = NULL]$ como:

$$ST_{[x=NULL]}(rsg) = rsg'$$

donde el grafo rsg' queda definido de la siguiente manera:

- $N(rsg') = N(rsg)$
- $PL(rsg') = PL(rsg) \setminus \{< x, n_i >, \forall < x, n_i > \in PL(rsg)\}$
- $NL(rsg') = NL(rsg)$

Vemos, como el grafo rsg' es igual a rsg, excepto que se eliminan todas las referencias de la variable x a cualquier nodo. Como hemos comentado, esta operación provoca cambios en la propiedad $SPATH$ de algunos nodos. Los cambios son los siguientes, $\forall n \in N(rsg')$:

$$SPATH(n) = SPATH(n) \setminus \{< x, sel^* > \mid (sel^* = sel_i) \vee (sel^* = \emptyset)\}$$

B.2 Sentencia $[x = malloc()]$

En este caso, se inicializa una nueva porción de memoria referenciada directamente por la variable x. Esto, en los grafos, va a provocar la aparición de un nuevo nodo, que representará a dicha porción, y que será apuntado directamente por x.

Antes de modificar el grafo por dicha sentencia, se aplica la semántica abstracta de la sentencia $x = NULL$ ($ST_{[x=NULL]}$), para no tener en cuenta en la semántica abstracta de $[x = malloc()]$ las cuestiones relacionadas con la eliminación de las referencias de x. De este modo, esta sentencia tan solo se debe preocupar de la creación del nuevo nodo.

Definimos la semántica abstracta de la sentencia $[x = malloc()]$ como:

$$ST_{[x=NULL]}(rsg) = rsg'$$

donde el nuevo grafo rsg' queda definido como:

- $N(rsg') = N(rsg_N) \cup \{n_{new}\}$
- $PL(rsg') = PL(rsg_N) \cup \{< x, n_{new} >\}$
- $NL(rsg') = NL(rsg_N)$

siendo $rsg_N = ST_{[x=NULL]}(rsg)$. Como vemos, se añade un nuevo nodo n_{new} que es referenciado directamente por x. Las propiedades del nodo son las siguientes:

$$\begin{cases} TYPE(n_{new}) = \text{Type}(x) \\ STRUCTURE(n_{new}) = \text{ Nueva estructura} \\ SPATH(n_{new}) = \{< x, \emptyset >\} \\ SHARED(n_{new}) = 0 \\ SHSEL(n_{new}, sel_i) = 0 \ \forall sel_i \in S \\ SELINset(n_{new}) = \emptyset \\ PosSELINset(n_{new}) = \emptyset \\ SELOUTset(n_{new}) = \emptyset \\ PosSELOUTset(n_{new}) = \emptyset \\ CYCLELINKS(n_{new}) = \emptyset \\ TOUCH(n_{new}) = \emptyset \end{cases}$$

Lo más destacable del nuevo nodo es que pertenece a una nueva *estructura* y además posee un *simple path* de longitud cero desde x.

B.3 Sentencia $[x = y]$

Esta sentencia hace que las porciones de memoria apuntadas directamente por la variable y lo sean además por la variable x. Como todas las porciones apuntadas por y en un grafo está

representadas en un único nodo, el apuntado por y, bastará con añadir la correspondiente referencia desde x a dicho nodo.

De nuevo la propiedad *SPATH* de dicho nodo se verá modificada para reflejar el hecho de que ahora también es apuntado por x, y también de los nodos "cercanos" a él. De nuevo, antes de asignar el nuevo destino a la variable x, hay que eliminar el que tuviera, por lo tanto, al igual que con la sentencia $x = malloc()$, en esta también se aplica la semántica abstracta de $x = NULL$ antes de proceder a las modificaciones.

La semántica abstracta de la sentencia $[x = y]$ queda definida como:

$$ST_{[x=y]}(rsg) = rsg'$$

definiendo el nuevo grafo rsg' a partir del grafo $rsg_N = ST_{[x=NULL]}(rsg)$ de la siguiente manera:

- $N(rsg') = N(rsg_N)$
- $PL(rsg') = PL(rsg_N) \cup \{< x, n_y > \mid < y, n_y > \in PL(rsg_N)\}$
- $NL(rsg') = NL(rsg_N)$

Como podemos observar, todo queda igual, excepto que se añade la referencia desde x al nodo apuntado por y (n_y). La propiedad *simple path* de los nodos se modifica de la siguiente manera, $\forall n \in N(rsg')$:

$$SPATH(n) = SPATH(n) \cup \{< x, sel^* > \mid < y, sel^* > \in SPATH(n)\}$$

Como se ve, el conjunto de *simple paths* de un nodo se ve completado con uno empezando en x, si posee alguno que empiece en y (ya que ahora x apunta al mismo nodo que y).

B.4 Sentencia $[x \rightarrow sel = NULL]$

La acción de esta sentencia es eliminar el enlace que exista en la estructura de datos, desde la porción de memoria apuntada por x por el selector sel. En un grafo, deberemos eliminar los enlaces por el selector sel del nodo que representa las porciones de memoria apuntadas por x. Esta eliminación de enlaces, trae consigo el cambio en muchas de las propiedades de los nodos a ambos extremos del enlace. Hay que estar pues seguros de que se cambian las propiedades de nodos que realmente representan a las porciones de memoria involucradas en la sentencia.

El nodo que representa las porciones de memoria apuntadas por x no supone ningún problema, ya que en un rsg tan solo puede existir un nodo apuntado por una determinada variable puntero, y dicho nodo, n_x, representa todas las porciones apuntadas por dicha variable. El problema está en los nodos apuntados por n_x con el selector sel. Este nodo puede representar porciones de memoria apuntadas por $x \rightarrow sel$ como otras compatibles con éstas pero no referenciadas por este enlace. Para no modificar conservativamente las propiedades y enlaces de los nodos, aplicamos el proceso de *enfoque* para separar en distintos nodos o grafos las porciones involucradas en la sentencia de las que no.

Una vez que se tengan por separado dichas porciones, se podrá eliminar el enlace desde n_x, actualizando las propiedades de los nodos afectados, como *reference patterns*, información *shared*, etc., que se ven modificadas puesto que ha desaparecido un enlace del grafo.

Por tanto las modificaciones de la semántica abstracta de esta sentencia no se aplican directamente al grafo de entrada, sino que se aplican a los grafos obtenidos tras el *enfoque*, donde se utilizarán las funciones descritas anteriormente para *dividir* los grafos, *podarlos* y *materializar* grafos o nodos según sea necesario. Definimos el conjunto de grafos *enfocados* $F(rsg, x, sel)$ de un grafo de entrada rsg para el enlace $x \rightarrow sel$, como:

$F(rsg, x, sel) = \{rsg_i |$

- $[rsg_i = MATERIALIZE\_NODE(rsg_j, x, sel), \forall rsg_j \in PR(rsg, x, sel) \wedge$
  $< x, n_x > \in PL(rsg_j) \wedge < n_x, sel, n > \in NL(rsg_j) \wedge (\nexists pv \in P | < pv, n > \in PL(rsg_j))] \vee$
- $[\forall rsg_i \in MATERIALIZE\_RSG(rsg_j, x, sel), \forall rsg_j \in PR(rsg, x, sel) \wedge$
  $< x, n_x > \in PL(rsg_j) \wedge < n_x, sel, n > \in NL(rsg_j) \wedge (\exists pv \in P | < pv, n > \in PL(rsg_j))]\}$

Siendo PR el conjunto de grafos *podados*, resultantes de la operación de *división* de rsg:

$$PR(rsg, x, sel) = \{rsg_j \mid rsg_j = PRUNE(rsg'_j),\ \forall\ rsg'_j \in DIVIDE(rsg, x, sel)\}$$

Vemos, como el proceso de *enfoque* genera una serie de grafos (conjunto F), según el proceso descrito la sección 3.4.1: primero el grafo es *dividido* ($DIVIDE$) para separar en distintos grafos los distintos enlaces que pueda haber por $x \rightarrow sel$; luego estos grafos son *podados* ($PRUNE$) para eliminar nodos y selectores que representan porciones de memoria y enlaces de configuraciones de memoria representados en otros grafos, centrándose así el grafo en unas determinadas estructuras; por último, se lleva a cabo la *materialización* que tiene por objetivo separar en distintos nodos o grafos las porciones realmente referenciadas por $x \rightarrow sel$ de aquellas que no lo son. De manera que si n_x es el nodo apuntado por x en cada grafo, tras todas estas transformaciones, en cada grafo, el nodo n apuntado por n_x por el selector sel representa única y exclusivamente a aquellas porciones de memoria realmente referenciadas por $x \rightarrow sel$.

Una vez definido el conjunto de grafos *enfocados*, ya se puede definir la semántica abstracta de la sentencia:

$$ST_{[x \rightarrow sel = NULL]}(rsg) = \{rsg'_i \mid \forall rsg_i \in F(rsg, x, sel)\}$$

que da como resultado un conjunto de grafos transformando los grafos obtenidos tras *enfocar* rsg ($F(rsg, x, sel)$) de la siguiente manera:

- $N(rsg'_i) = N(rsg_i)$
- $PL(rsg'_i) = PL(rsg_i)$
- $NL(rsg'_i) = NL(rsg_i) \setminus \{< n_x, sel, n > \in NL(rsg_i) \mid\ < x, n_x > \in PL(rsg_i)\}$

Como la sentencia elimina el selector $x \rightarrow sel$, la modificación en el grafo consiste en dejar los conjuntos $N(rsg_i)$ y $PL(rsg_i)$ sin variar, y de $NL(rsg_i)$ eliminamos el único enlace que exista desde n_x por el selector sel (siendo n_x el nodo que representa las porciones de memoria referenciadas por x).

La desaparición de este enlace provoca el cambio en las propiedades de los nodos involucrados en dicho enlace. Sean los nodos n_x y n tal que $< x, n_x > \in PL(rsg_i), < n_x, sel, n > \in NL(rsg_i)$, las nuevas propiedades de n son:

- $SPATH(n) = SPATH(n) \setminus \{< x, sel >\}$
- $SELINset(n) = SELINset(n) \setminus \{sel \mid SHSEL(n, sel) = 0 \vee (\nexists n_1 \in N(rsg_i') \wedge < n_1, sel, n > \in NL(rsg_i'))\}$
- $PosSELINset(n) = PosSELINset(n) \setminus \{sel \mid SHSEL(n, sel) = 0 \vee (\nexists n_1 \in N(rsg_i') \wedge < n_1, sel, n > \in NL(rsg_i'))\} \cup \{sel \mid SHSEL(n, sel) = 1 \wedge (\exists n_1 \in N(rsg_i') \wedge < n_1, sel, n > \in NL(rsg_i'))\}$
- $SHARED(n) = 0$ si $\nexists n_i, n_j \mid < n_i, sel_i, n >, < n_j, sel_j, n > \in NL(rsg_i') \wedge sel_i \neq sel_j$
- $SHSEL(n, sel) = 0$ si $[\nexists n_i \in N(rsg_i') \mid < n_i, sel, n > \in NL(rsg_i')] \vee [\exists_1 n_i \in N(rsg_i') \mid < pvar, n_i > \in PL(rsg_i') \wedge < n_i, sel, n > \in NL(rsg_i')]$
- $CYCLELINKS(n) = CYCLELINKS(n) \setminus \{< sel_i, sel > \mid < n, sel_i, n_x > \in NL(rsg_i')\}$

Las propiedades que se ven afectadas en el nodo n (antes referenciado por n_x con sel) hacen referencia al hecho de que al desaparecer el enlace desde n_x por sel, el nodo puede que ya no sea apuntado por este tipo de selector (*reference patterns*) y por tanto si era *shared* es posible que deje de serlo. En concreto:

- Se elimina $<x, sel>$ del conjunto de sus *simple paths* puesto que ya no es referenciado desde n_x por sel (selector borrado por la sentencia).
- Se puede eliminar sel de sus selectores de entrada (*SELINset*) si el nodo no es *shared* por el selector sel ya que se ha borrado el único enlace por sel que tenía. Si el nodo era *shared* por sel, también se puede eliminar, pero sólo si ya no existe ningún otro nodo apuntando a n por sel (el último era el borrado).
- Con el conjunto de posibles selectores de entrada (*PosSELINset*) pueden ocurrir dos cosas totalmente distintas. Como en el caso del *SELINset*, si el nodo no es *shared* por sel o ya no hay más nodos apuntándole por sel, se puede quitar sel de *PosSELINset* (mismas razones que para *SELINset*). Sin embargo, si el nodo es *shared* por el selector sel y existe algún nodo apuntando a n por sel, se ha de introducir sel como posible selector de entrada. Esto es debido a que si el nodo es *shared* por sel, las porciones pueden ser referenciadas desde otras porciones distintas a las representadas por n_x. Aunque se quite el enlace que viene de n_x, puede que haya porciones a las que les haya desaparecido todos sus enlaces por sel, pero puede que otras tengan aún más de uno, desde otras porciones (ya que son *shared*). Por lo tanto, es aquí donde se introduce la incertidumbre en los *reference patterns*, puesto las porciones representadas por el nodo pueden ser o no apuntadas por el selector sel.
- En cuanto a la información *SHARED*, el nodo deja de serlo si tras eliminar el enlace por sel, ya no es apuntado por distintos nodos con distintos selectores.
- Algo similar ocurre con $SHSEL(n, sel)$. Ya no será *shared* por el selector sel si no existe ningún enlace sobre n por sel, o si existe tan solo uno desde un nodo apuntado por una variable puntero. Como ya dijimos, este nodo no puede representar varias porciones de memoria de una misma configuración, puesto que una variable puntero sólo puede apuntar a un sitio a la vez. Por tanto, este selector está representando un único enlace en cada configuración, por lo que el nodo no puede ser *shared* por sel.

- Por último, de los *cyclelinks* del nodo se elimina $<sel_i, sel>$ si n apunta a n_x por sel_i, ya que ahora n_x no apunta a n por sel como implicaría el *cyclelink*.

En cuanto a las propiedades del nodo n_x, las afectadas son:

- $SELOUTset(n_x) = SELOUTset(n_x) \setminus \{sel\}$
- $CYCLELINKS(n_x) = CYCLELINKS(n_x) \setminus \{< sel, sel_i > \forall sel_i \in S\}$

Para n_x, tan solo se elimina sel de sus selectores de salida ($SELOUTset$) puesto que se ha eliminado el único enlace que tenía por dicho selector, y se quitan todos los *cyclelinks* de la forma $<sel, sel_i>$, puesto que ya no hay ningún enlace por el selector sel.

Con la propiedad $STRUCTURE$ hay que tener un especial cuidado. Tras la eliminación del enlace $<n_x, sel, n>$, puede que se rompa una componente conexa en varias. Por este motivo, hay que ver si todavía existe algún nodo desde el cual se pueda llegar a ambos nodos, con lo cual seguirían formando parte de la misma componente y no habría que cambiar $STRUCTURE$ de ninguno de los nodos. Si este nodo no puede ser encontrado, entonces es que se han creado dos nuevas componentes no conexas entre sí, y por tanto el atributo $STRUCTURE$ de todos los nodos pertenecientes a las mismas, debe ser cambiado.

Para poder determinar lo anteriormente expuesto, vamos a definir el conjunto de nodos *conectados* con uno dado, $C(n, rsg)$, que no es más que todos los nodos del grafo rsg que cumplen que se puede encontrar un nodo en rsg desde el cual se puede llegar tanto a n como a dichos nodos.

$$\begin{gathered} C(n, rsg) = \{n_i \in N(rsg) \mid \exists n_j \in N(rsg), (\exists n_1, n_2, .., n_n \in N(rsg), \\ < n_j, sel_1, n_1 >, < n_1, sel_2, n_2 >, ..., < n_n, sel_{n+1}, n_i > \in NL(rsg)) \wedge \\ (\exists n'_1, n'_2, .., n'_n \in N(rsg), < n_j, sel'_1, n'_1 >, < n'_1, sel'_2, n'_2 >, ..., \\ < n'_n, sel'_{n+1}, n > \in NL(rsg))\} \end{gathered}$$

Una vez definido el conjunto C, es fácil determinar si se ha partido una componente conexa y en ese caso asignar un atributo nuevo $STRUCTURE$ a los nodos de las componentes resultantes:

Si $C(n_x, rsg'_i) \cap C(n, rsg'_i) = \emptyset$ entonces $\forall n_i \in C(n_x, rsg'_i), STRUCTURE(n_i) = S_1 \wedge \forall n_j \in C(n, rsg'_i), STRUCTURE(n_j) = S_2$, donde S_1 y S_2 son los identificadores de dos estructuras nuevas.

De esta forma queda descrita completamente la semántica abstracta de la sentencia $x \rightarrow sel = NULL$.

B.5 Sentencia $[x \rightarrow sel = y]$

Este es el caso contrario al anterior, en el que se crea un enlace por el selector sel entre las porciones de memoria referenciadas por x e y. Esto va a provocar la inserción de un nuevo enlace en los grafos a analizar entre los nodos n_x y n_y, que representan respectivamente a las porciones de memoria apuntadas por x e y.

Pero antes de introducir el nuevo enlace, esta sentencia también provoca la eliminación de un posible enlace por sel desde la porción apuntada por x. Para llevar esto a cabo, basta con aplicar la semántica abstracta de la sentencia $x \rightarrow sel = NULL$ al grafo a analizar, y

posteriormente, aplicar la semántica abstracta de la sentencia actual a los grafos resultantes para generar el nuevo enlace.

Definimos la semántica abstracta de esta sentencia como

$$ST_{[x \to sel=y]}(rsg) = \{rsg'_i \mid \forall rsg_i \in ST_{[x \to sel=NULL]}(rsg)\}$$

que da como resultado un conjunto nuevo de grafos, rsg'_i, transformando los obtenidos tras aplicar la semántica abstracta de la sentencia $x \to sel = NULL$, rsg_i. Los nuevos grafos, rsg'_i, se obtienen de la siguiente manera:

- $N(rsg'_i) = N(rsg_i)$
- $PL(rsg'_i) = PL(rsg_i)$
- $NL(rsg'_i) = NL(rsg_i) \cup \{< n_x, sel, n_y > \mid < x, n_x > \in PL(rsg_i) \wedge < y, n_y > \in PL(rsg_i)\}$

Vemos que la transformación consiste en insertar precisamente el enlace que indica la sentencia, entre las porciones de memoria apuntadas por x (representadas por n_x) y las apuntadas por y (representadas por n_y). Como consecuencia de la inserción de este nuevo enlace, las propiedades de ambos nodos deben ser modificadas para reflejar la presencia de este nuevo enlace. Si n_x y n_y son los dos nodos de rsg'_i de modo que $<x, n_x > \in PL(rsg'_i)$ y $<y, n_y > \in PL(rsg'_i)$, las propiedades modificadas de n_x son:

- $SELOUTset(n_x) = SELOUTset(n_x) \cup \{sel\}$
- $PosSELOUTset(n_x) = PosSELOUTset(n_x) \setminus \{sel\}$
- $CYCLELINKS(n_x) = CYCLELINKS(n_x) \cup \{< sel, sel_i > \mid (\exists < n_y, sel_i, n_x > \in NL(rsg'_i)) \wedge (SELOUT(n_y, sel_i) = 1) \wedge (\nexists n_2 \in N(rsg'_i), < n_y, sel_i, n_2 > \in NL(rsg'_i))\}$

Al añadir el nuevo enlace $x \to sel = y$, todas las porciones de memoria referenciadas por x van a tener un enlace de salida por sel, y como n_x representa sólo a todas las porciones de memoria apuntadas por sel, entonces se introduce sel en $SELOUT(n_x)$. Además, el *cyclelink* $<sel, sel_i>$ es introducido en n_x, si el nodo n_y definitivamente tiene un selector de salida por sel_i y sólo apunta a n_x.

Las propiedades de n_y quedan de la siguiente manera:

- $SELINset(n_y) = SELINset(n_y) \cup \{sel\}$
- $PosSELINset(n_y) = PosSELINset(n_y) \setminus \{sel\}$
- $SPATH(n_y) = SPATH(n_y) \cup \{< x, sel >\}$
- $SHARED(n_y) = 1$ si $\exists n_i \in N(rsg'_i) \mid < n_i, sel_i, n_y > \in NL(rsg'_i) \wedge sel_i \neq sel$
- $SHSEL(n_y, sel) = 1$ si $\exists n_i \in N(rsg'_i), n_i \neq n_x \mid < n_i, sel, n_y > \in NL(rsg'_i)$
- $CYCLELINKS(n_y) = CYCLELINKS(n_y) \cup \{< sel_i, sel > \mid (\exists < n_y, sel_i, n_x > \in NL(rsg'_i)) \wedge (SELOUT(n_y, sel_i) = 1) \wedge (\nexists n_2 \in N(rsg'_i), <n_y, sel_i, n_2 > \in NL(rsg'_i))\}$

En este caso, las porciones representadas por n_y, que son todas aquellas apuntadas por y, tienen todas un nuevo enlace por el selector sel desde las porciones representadas por n_x. Como es seguro que todas son apuntadas por sel, dicho selector es introducido en $SELINset(n_y)$ y eliminado, si existía, de $PosSELINset(n_y)$. Como, dicho enlace, proviene de porciones de memoria apuntadas por x, existe un nuevo *simple paths* para el nodo n_y que es $<x, sel>$.

En cuanto a la información *shared*, el nodo n_y será $SHARED$ si hay algún otro nodo que tiene un enlace sobre n_y por un selector distinto de sel. Lo mismo ocurre con $SHSEL(n_y, sel)$, pero en este caso el enlace que existe debe de ser con el propio selector sel. Para terminar, se añaden los mismos *cyclelinks* y bajo las mismas condiciones que para el nodo n_x.

De nuevo, el atributo $STRUCTURE$ se ve afectado para más nodos aparte de los dos involucrados en la sentencia. El motivo es que la inclusión del nuevo enlace, provoca que si las componentes conexas de n_x y n_y no eran la misma, tras la ejecución de la sentencia, si que lo serán. Bastará con poner el mismo valor del atributo $STRUCTURE$ de n_x a todos los nodos conectados con él tras la inserción del enlace (entre ellos estará n_y y todos los que pertenecían a su componente conexa), de la siguiente manera:

$$\forall n_i \in C(n_x, rsg'_i), STRUCTURE(n_i) = STRUCTURE(n_x)$$

Quedan así definidas todas las transformaciones necesarias para generar el conjunto de grafos que representas las configuraciones de memoria tras la aplicación de la sentencia $x \rightarrow sel = y$ a un rsg dado.

B.6 Sentencia $[y = x \rightarrow sel]$

Esta última sentencia, hace que la variable y apunte a las porciones de memoria referenciadas por $x \rightarrow sel$. Para que en los grafos, la variable y apunte a las porciones de memoria que realmente son referenciadas por $x \rightarrow sel$, debemos enfocar la parte derecha de la sentencia, como ya se ha visto en la semántica abstracta de $x \rightarrow sel = NULL$. Es decir, vamos a generar un conjunto de grafos de manera que en cada uno de ellos, el nodo apuntado por $x \rightarrow sel$ realmente sólo represente a las porciones de memoria que son referenciadas por dicho enlace, y no a otras compatibles con estas. El proceso consiste en la *división*, *poda* y la *materialización*, anteriormente comentados.

Para llevar a cabo esto, utilizaremos el conjunto de grafos enfocados a partir de rsg para el selector $x \rightarrow sel$, $F(rsg, x, sel)$ definido en la semántica abstracta de $x \rightarrow sel = NULL$.

Esta sentencia, además de crear el nuevo enlace para y, destruye el que pudiera tener antes, por lo tanto, para no tener en cuenta los aspectos de destrucción del enlace de y, aplicamos la semántica abstracta de la sentencia $x = NULL$ para la variable y y el grafo a analizar.

De este modo definimos:

$$ST_{[y=x \rightarrow sel]}(rsg) = \{rsg'_i \mid \forall rsg_i \in F(ST_{[y=NULL]}(rsg), x, sel)\}$$

que es el conjunto de grafos tras aplicar la semántica abstracta de la sentencia $y = x \rightarrow sel$ sobre un grafo de entrada rsg. Como se aprecia, lo primero que hacemos es aplicar la semántica de $y = NULL$ al grafo ($ST_{[y=NULL]}$), y posteriormente se crea el conjunto de grafos enfocados del anterior teniendo en cuenta el selector $x \rightarrow sel$ ($F(rsg, x, sel)$). tan solo queda por definir

como es transformado cada uno estos grafos, rsg_i, para que reflejen la ejecución de la sentencia en un nuevo grafo rsg'_i. Estos nuevos grafos son creados de la siguiente manera:

- $N(rsg'_i) = N(rsg_i)$
- $PL(rsg'_i) = PL(rsg_i) \cup \{< y, n > \mid < x, n_x > \in PL(rsg_i) \wedge < n_x, sel, n > \in NL(rsg_i)\}$
- $NL(rsg'_i) = NL(rsg_i)$

Como se puede observar, el conjunto de nodos y de enlaces entre nodos del grafo nuevo son los mismos que los del grafo original. Lo único que cambia es el conjunto de enlaces desde las variables puntero a los nodos, puesto que ahora, la variable y apunta al nodo n, que es referenciado por n_x y el selector sel (siendo n_x el nodo referenciado por x). No hay ningún problema al hacer esto, puesto que dicho nodo n sólo representa a las porciones de memoria que realmente son apuntadas por $x \rightarrow sel$, ya que para eso se ha *enfocado* el grafo original.

En esta sentencia, la única propiedad que se ve afectada es el conjunto de *simple paths* de algunos nodos, puesto que se ha introducido un punto de acceso nuevo desde la variable y al nodo n. En concreto hay que modificar el conjunto de *simple paths* de n, añadiendo $<y, \emptyset>$ puesto que es directamente apuntado por y, y el de los nodos directamente apuntados por n, añadiendo $<y, sel_i>$, con sel_i dependiendo del selector del enlace que hay desde n. Formalmente, $\forall n \in N(rsg'_i)$:

$$SPATH(n) = SPATH(n) \cup \{< y, \emptyset > \mid < y, n > \in PL(rsg'_i)\} \cup$$
$$\{< y, sel_i > \mid < y, n_y > \in PL(rsg'_i) \wedge < n_y, sel_i, n > \in NL(rsg'_i)\}$$

De esta manera queda definida completamente la semántica abstracta de la última sentencia simple con punteros. A continuación presentamos la semántica abstracta de las *pseudoinstrucciones* introducidas en el método para refinar la representación de las configuraciones de memoria por los grafos, bajo determinadas condiciones.

B.7 Semántica abstracta de las *pseudoinstrucciones*

Como se ha presentado en el capítulo 3, nuestro método basado en los RSRSG aproxima todas las configuraciones de memoria que pueden aparecer tras la ejecución de una sentencia mediante un conjunto reducido de grafos. Debido a que una misma sentencia puede alcanzarse siguiendo muy diversos caminos en el grafo de flujo de control, después de una sentencia podemos tener muchas configuraciones de memoria que tienen que ser aproximadas por un conjunto de grafos, RSGs. Conforme un mismo RSG va aproximando más y más configuraciones de memoria y va siendo transformado, se puede ir degradando la representación de las configuraciones de memoria por parte del grafo. Por este motivo hemos elegido un conjunto de propiedades asociadas a cada nodo, de manera que mantengan la mayor información posible sobre las estructuras representadas y para que su manipulación sea lo más precisa posible.

En el sentido de aumentar la exactitud del método, evitando en la medida de lo posible la información incierta, hemos comprobado que se puede aprovechar de una manera bastante útil, información proporcionada por las distintas sentencias del cuerpo del código. En concreto, las sentencias condicionales (sentencias if, while, case, etc..) cuya condición hace referencia a variables puntero, nos pueden proporcionar una información muy útil para refinar los grafos,

de manera que se ciñan mejor a las configuraciones de memoria que en tiempo de ejecución pueden aparecer en una sentencia. Para entender mejor esto, vamos a ver un trozo de código en el que en un bucle se recorre y se modifica una estructura de datos dinámica.

```
...
/* x apunta a una estructura dinamica */
while (x != NULL)
  {
    Instruccion A;
    ...
    y = x->next;
    if (y != NULL)
      {
        Instruccion B;
        ...
      }
    else
      {
        Instruccion C;
        ...
      }
    x = x→nxt;
  }
Instruccion D;
...
```

En primer lugar, cuando se analice este código, el RSRSG de entrada para la instrucción A es el de la instrucción *while*. El RSRSG de esta sentencia, será la unión de los RSRSGs de las sentencias inmediatamente anterior al *while* y de la última del cuerpo del mismo. Como vemos, la condición para entrar en el bucle es que $x! = NULL$. Esto implica, que en la instrucción A, en todas las configuraciones de memoria que pueden aparecer, la variable x apunta a alguna porción. Si se analiza el código sin tener esta información en cuenta, debido a los posibles caminos del flujo de control, a la instrucción A llegarán grafos que cumplan esta condición, como otros que no la cumplan (proveniente de la última instrucción del cuerpo del bucle). Todos estos grafos, serán analizados de nuevo, hasta alcanzar un punto fijo. Vemos, como hay grafos analizados, que representan configuraciones de memoria que en tiempo de ejecución nunca pueden aparecer en la instrucción A.

Si tenemos en cuenta la información que nos proporciona la condición del bucle, se podrían eliminar del RSRSG justo a la entrada del bucle (antes de la instrucción A) aquellos grafos que no representan configuraciones de memoria en las que $x! = NULL$. De este modo, se evitan analizar grafos que representan configuraciones de memoria que no se pueden dar en dicha sentencia, reduciendo el número de grafos y aumentando la exactitud del análisis (reduce el número de posibilidades de representación). Además, si el bucle no tiene ninguna sentencia de terminación distinta de la condición del lazo ($BREAK, EXIT, ...$), también se sabe que en la instrucción D, que es la inmediatamente posterior al bucle, la variable x tiene que ser igual a $NULL$, puesto que si no fuera así, la ejecución hubiera continuado con el cuerpo del bucle.

Algo similar ocurre en las instrucciones B y C. Si se tiene en cuenta la condición de la sentencia IF, en la instrucción B la variable y no puede ser igual a $NULL$ y en la instrucción C, debe ser igual a $NULL$.

En definitiva, lo que pretendemos es recortar el número de posibles caminos del grafo de flujo de control, acercándolos a los caminos reales en tiempo de ejecución, y para ello tenemos

que interpretar estas sentencias, que realmente no modifican las estructuras dinámicas de datos.

Para llevar esto a cabo, se han creado una serie de *pseudoinstrucciones* que se insertarán en aquellos lugares donde se conoce la certeza de una condición sobre un puntero. Estas *pseudoinstrucciones* tienen una *semántica abstracta* asociada, que es la encargada de eliminar de los grafos aquellas configuraciones de memoria que en tiempo de ejecución no pueden pasar por ese punto del programa. Las *pseudoinstrucciones* creadas son las siguientes:

- $FORCE(x == NULL)$: la variable x debe ser $NULL$ en ese punto del programa.
- $FORCE(x! = NULL)$: la variable x no debe apuntar a alguna porción de memoria en ese punto del programa.
- $FORCE(x == y)$: La variable x debe apuntar a la misma porción de memoria que la variable y, en ese punto del programa.
- $FORCE(x! = y)$: La variable x no puede apuntar a la misma porción de memoria que la variable y, en ese punto del código.
- $FORCE(x \rightarrow sel == NULL)$: La porción de memoria apuntada por la variable x no puede apuntar a ninguna otra porción por el selector sel, en dicho punto del programa.

Como hemos comentado dichas *pseudoinstrucciones* serán insertadas en aquellos puntos del código donde se tenga la certeza de que la condición sobre la variable puntero se cumple $(x == NULL, x! = NULL, x == y, x! = y, x \rightarrow sel == NULL)$. Además de utilizar las sentencias condicionales para obtener esta información, como hemos visto anteriormente, también podemos sacar información de otras sentencias con punteros, bajo la suposición de la *corrección del código*. Así, si se accede a alguno de los campos de una porción de memoria apuntada por una variable x (para lectura o escritura, sea un campo puntero o un campo de datos), si el código es correcto, la variable x no puede ser $NULL$, ya que si lo fuera, se produciría un error en tiempo de ejecución.

Por ejemplo, si nos encontramos las sentencias $x \rightarrow sel = ...$, $x \rightarrow data = ...$, $... = x \rightarrow sel$ o $... = x \rightarrow data$ (sel representa un campo puntero de la porción apuntada por x y $data$ un campo de datos), si el código es correcto, x debe ser distinto de $NULL$ puesto que si no, la ejecución de cualquiera de las sentencias produciría un error en tiempo de ejecución. Como se puede observar, da igual que esté en la parte lhs o rhs de la sentencia, ya que en ambas partes se estaría accediendo a un campo de una porción de memoria, y dicha porción debe de ser accedida mediante x. Por tanto, justo antes de este tipo de sentencias se insertará la *pseudoinstrucción* $FORCE(x! = NULL)$, con lo que se evitará que dichas sentencias analicen configuraciones de memoria que realmente no pueden aparecer (bajo la suposición de corrección del código).

Pasamos a continuación a describir la semántica abstracta de cada una de las *pseudoinstrucciones*, donde veremos en que forma reducen el número de configuraciones de memoria a analizar.

B.7.1 Pseudoinstrucción $FORCE(x == NULL)$

Esta condición impone la restricción que en las configuraciones de memoria que pueden pasar por ella, la variable puntero x no puede apuntar a ninguna porción de memoria. La cuestión

ahora es como se modifican los grafos que llegan a dicha *pseudoinstrucción*.

Si tenemos en cuenta cuando dos RSGs pueden ser unidos para que sus configuraciones de memoria sean representadas por un solo grafo, vemos que una de las condiciones que se les imponen es que sus relaciones de *alias* coincidan (sección 3.3.1). Esta restricción va es muy útil a la hora de interpretar estas *pseudoinstrucciones*, ya que todas las configuraciones de memoria que representa un determinado RSG, tienen el mismo conjunto de relaciones de *alias*. Esto implica que si la variable x apunta a un nodo junto con otras variables en un determinado RSG, en todas las configuraciones de memoria representadas por dicho grafo, x apuntará a la misma porción de memoria que las demás variables. Si x no apunta a ningún nodo, entonces es que x no apunta a ninguna porción de memoria en todas las configuraciones representadas por el grafo. No es posible que x apunte a un porción en una configuración y no apunte a ninguna otra en otra configuración, ambas representadas en un mismo RSG, ya que las relaciones de *alias* en ambas configuraciones son distintas y por tanto no serían nunca fundidas en un solo grafo.

Teniendo todo esto en cuenta, la *pseudoinstrucción* $FORCE(x == NULL)$ lo que hará es no dejar pasar aquellos grafos en los que x apunta a algún nodo, puesto que todas las configuraciones de memoria tienen $x! = NULL$. Los grafos en los que x no apunte a ningún nodo, se dejarán pasar a la siguiente sentencia sin modificar, puesto que cumplen la restricción.

Formalmente, definimos su semántica abstracta de la siguiente manera:

$$ST_{[FORCE(x==NULL)]}(rsg) = \begin{cases} \emptyset \text{ si } \exists n \in N(rsg) \mid < x, n > \in PL(rsg) \\ rsg \text{ en caso contrario.} \end{cases}$$

B.7.2 Pseudoinstrucción $FORCE(x! = NULL)$

Lo expuesto para la anterior *pseudoinstrucción* se aplica a esta también. La única diferencia es que ahora las configuraciones de memoria que pueden continuar son aquellas en las que la variable x apunta a alguna porción de memoria. Por tanto, la semántica abstracta de esta *pseudoinstrucción* dejará pasar sin modificar aquellos RSGs en los que la variable x apunta a algún nodo y no dejará pasar aquellos en los que no aparezca x. De esta forma, tan solo se analizarán los grafos que representan configuraciones de memoria en las que $x! = NULL$.

Su semántica abstracta queda de la siguiente manera:

$$ST_{[FORCE(x!=NULL)]}(rsg) = \begin{cases} \emptyset \text{ si } \nexists n \in N(rsg) \mid < x, n > \in PL(rsg) \\ rsg \text{ en caso contrario.} \end{cases}$$

B.7.3 Pseudoinstrucción $FORCE(x == y)$

Ahora las configuraciones que deben seguir adelante tienen que tener a las variables x e y apuntando a la misma porción de memoria. De nuevo, esta condición es sobre las relaciones de *alias* que se deben dar en las configuraciones de memoria, y por tanto se puede trasladar directamente a los grafos, puesto que las configuraciones de memoria que representa un grafo tienen todas las mismas relaciones de *alias*.

Por tanto, los grafos en los que x no apunte al mismo nodo que y representan configuraciones de memoria donde la condición no se cumple, y por tanto no tienen que ser analizados (representan configuraciones de memoria que en tiempo de ejecución no pueden aparecer en

dicho punto del programa). Sólo se dejarán pasar los grafos en los que x e y apunten al mismo nodo, o ambas no apunten a ninguno.

Su semántica abstracta es:

$$ST_{[FORCE(x==y)]}(rsg) = \begin{cases} \emptyset \text{ si } \exists n \in N(rsg) \mid \\ \quad (< x, n > \in PL(rsg) \wedge \ < y, n > \notin PL(rsg)) \vee \\ \quad (< x, n > \notin PL(rsg) \wedge \ < y, n > \in PL(rsg)) \\ rsg \text{ en caso contrario.} \end{cases}$$

B.7.4 Pseudoinstrucción $FORCE(x! = y)$

Esta *pseudoinstrucción* es totalmente similar a la anterior, sólo que para configuraciones de memoria en las que las variables x e y apuntan distintas porciones de memoria. Por tanto, ahora sólo pasarán los grafos en los que las variables apunten a distintos nodos.

Formalmente tenemos que:

$$ST_{[FORCE(x!=y)]}(rsg) = \begin{cases} \emptyset \text{ si } \begin{cases} (\exists n \in N(rsg) \mid < x, n >, < y, n > \in PL(rsg)) \vee \\ ((\nexists n_1 \in N(rsg) \mid \ < x, n_1 > \in PL(rsg)) \wedge \\ \quad (\nexists n_2 \in N(rsg) \mid \ < y, n_2 > \in PL(rsg))) \end{cases} \\ rsg \text{ en caso contrario.} \end{cases}$$

B.7.5 Pseudoinstrucción $FORCE(x \rightarrow sel == NULL)$

Este tipo de condición es más compleja de implementar, puesto que ahora, el puntero que tiene que ser $NULL$ no es una variable, sino un campo de la porción de memoria apuntada por x. Por este motivo, ya no podemos usar, como hemos hecho con las anteriores *pseudoinstrucciones*, las relaciones de *alias* para dejar o no pasar el grafo.

En este caso, en un mismo grafo puede haber configuraciones de memoria en las que $x \rightarrow sel == NULL$ y otras en las que esto no sea cierto. Por tanto, ahora la semántica abstracta no consistirá en dejar o no pasar un grafo, sino en modificar el grafo de manera que se eliminen aquellos nodos y enlaces que se sabe con certeza que pertenecen a una configuración de memoria en la que $x \rightarrow sel$ es distinto de $NULL$. La manera de hacer esto es igual a la presentada en la sección 3.4.1 cuando se presentaron las operaciones de *división* y *poda* de los grafos.

La semántica abstracta de esta *pseudoinstrucción* consistirá pues en eliminar cualquier enlace por el selector *sel* del nodo directamente apuntado por x. A continuación se hará una *poda* del grafo, de manera que se eliminarán todos los nodos y enlaces que no satisfacen las propiedades de los mismos. Si no las satisfacen, es porque el enlace $x \rightarrow sel$ realmente formaba parte de las configuraciones de memoria que representaba. Si tras la eliminación, hay nodos y enlaces que siguen satisfaciendo sus propiedades, es porque dicho enlace no pertenecía a las configuraciones de memoria que representan o bien no se sabe con certeza si pertenece o no y por tanto no son eliminados.

Para esta *pseudoinstrucción*, la semántica abstracta quedaría de la siguiente manera:

$$ST_{[FORCE(x \rightarrow sel == NULL)]}(rsg) = PRUNE(rsg')$$

donde rsg' se construye a partir de rsg de la siguiente manera:

- $N(rsg') = N(rsg)$
- $PL(rsg') = PL(rsg)$
- $NL(rsg') = NL(rsg) \setminus \{< n_x, sel, n > \mid \forall < n_x, sel, n > \in NL(rsg), < x, n_x > \in PL(rsg)\}$

Como se ha comentado anteriormente, la semántica abstracta de esta *pseudoinstrucción* elimina el enlace por *sel* que hay desde el nodo apuntado por x y *poda* el grafo resultante para eliminar nodos y enlaces pertenecientes a configuraciones de memoria que no cumplen la condición.

B.8 Información TOUCH y la semántica abstracta

La semántica abstracta de todas las sentencias presentadas hasta el momento, se encarga de modificar los grafos dependiendo de las modificaciones que provoca la sentencia en las configuraciones de memoria. Estas modificaciones suponen la creación y/o borrado de nodos y enlaces, además de la modificación, en concordancia con estas creaciones y/o borrados, de las propiedades de los nodos, entre ellas la propiedad *TOUCH*.

Pero la cuestión es que, debido a las peculiaridades de la información mantenida por *TOUCH* (presentada en la sección 3.2.8), los grafos tienen que ser modificados por sentencias que no son ninguna de las presentadas hasta el momento. Como ya se ha descrito cuando fue presentada la propiedad *TOUCH*, la información que mantiene hace referencia a las variables puntero de *inducción* que han visitado una porción de memoria, o desde las que se ha visitado dicha porción. Vimos que esta información tan solo tiene sentido dentro de los bucles que recorren las estructuras dinámicas, para mantener las porciones ya visitadas en el recorrido de las que aún no lo han sido.

Básicamente, las modificaciones que hay que introducir para el manejo de esta propiedad, son: i) asociar a las sentencias que *visitan* una porción de memoria ($x = y$ y $x = y \rightarrow sel$) una variable puntero de inducción que será introducida en el conjunto *TOUCH* del nodo visitado (podrá ser x o no, como se verá más adelante); ii) asociar a cada cabecera de bucle, un conjunto de variables puntero de inducción, que son las que recorren estructuras dinámicas dentro del cuerpo del bucle. Estas variables serán luego utilizadas para eliminarlas de la información *TOUCH* de cada nodo, cuando el grafo salga del cuerpo del bucle. Es decir, los recorridos que se acaban con el bucle, ya no tienen que ser mantenidos en *TOUCH*, reduciendo el número de nodos y por tanto el tamaño de los grafos.

Para conseguir todo esto, en una fase de preprocesado del código, antes del análisis de forma, se lleva a cabo un proceso de cálculo de los *path expressions* ($PE(pv) =< p, e, f, r >$) de las variables puntero utilizadas en el código, siguiendo el método de Hwang [49]. En base a estos *path expressions* se determinan los *punteros de inducción* asociados a cada bucle (es un *puntero de inducción* si $PE(pv).f \neq \emptyset$).

Vamos a denominar con B_i a las sentencias cabecera de los bucles del código, con $SUB(B_i)$ al conjunto de sentencias cabecera de bucles que están dentro del cuerpo del bucle B_i, y con $IPVARS(B_i)$ al conjunto de variables puntero de inducción detectadas para el bucle B_i. En base a estos conjuntos definimos otros conjuntos de punteros de inducción asociados a

las cabeceras de los bucles y a las sentencias que *visitan* porciones de memoria, que serán utilizados para mantener la información $TOUCH$ de los nodos.

- Asociamos a la cabecera de cada bucle B_i el conjunto de *punteros de inducción* denominado $IP\_TOUCH(B_i)$ tal que:

$$IP\_TOUCH(B_i) = IPVARS(B_i) \cup PVDOWN(B_i) \setminus PVUP(B_i)$$

donde

  - $PVDOWN(B_i) = \{pv \in P \mid (pv \in IPVARS(B_j)), B_j \in SUB(B_i)) \wedge$
    $(\exists\ ipv \in IPVARS(B_i) \mid PE(pv).e = ipv) \wedge$
    $(\nexists\ ipv_2 \in IPVARS(B_k), B_i, B_j \in SUB(B_k) \mid PE(ipv).e = ipv_2)\}$
  - $PVUP(B_i) = \{pv \in P \mid (pv \in IPVARS(B_i))$
    $\wedge\ (PE(pv).e \in IPVARS(B_j))\ \wedge\ (B_i \in SUB(B_j))\}$

Vemos que el conjunto de punteros inducción asociados a un bucle B_i para la información $TOUCH$, está formado por los punteros inducción detectados en el bucle B_i, cuyo punto de entrada en su *path expression* a la estructura de datos no depende de otro puntero inducción de un bucle que contiene a B_i. Además, se introducen también punteros de inducción de bucles en el cuerpo de B_i cuyos puntos de entrada son punteros de inducción del bucle B_i.

¿Que conseguimos con esto? Pues lo que se consigue es asociar al bucle B_i aquellos punteros de inducción que recorren estructuras dinámicas dentro del cuerpo de dicho bucle, pero que no dependen de recorridos de otros punteros de inducción en bucles que contienen a B_i.

En la figura B.1 volvemos a presentar el código ejemplo que aparece en la presentación de la propiedad $TOUCH$ (figura 3.9).

```
1:    x = M;
2:    while (condicion1)
        {
3:        z = x→col;
4:        while (condicion2)
            {
              /* procesar elemento apuntado por z */
5:            k = z→nxt;
6:            z = k;
7:            k = NULL;
            }
8:        y = x→nxt;
9:        x = y;
10:       y = NULL;
        }
```

Figura B.1: Código ejemplo.

En este caso el conjunto IP_TOUCH de la sentencia (2) estaría formado por $\{x, z\}$. La variable x es un puntero de inducción ($PE(x) =< x, M, nxt, \emptyset >$) del bucle (2) y la variable z lo es del bucle (4) ($PE(z) =< z, x, nxt, col >$). En principio habría que asociar al bucle (2) la variable x puesto que recorre una estructura en su cuerpo, y la variable z

al bucle (4) puesto que es este bucle donde z recorre una estructura. El problema es que, de ese modo, cuando se acaba de analizar el bucle (4) los nodos perderían la información referente a las "visitas" de z, cuando no debería ser así puesto que ese recorrido está ligado al recorrido de x, ya que z recorre un trozo de estructura empezando donde apunta x. Por tanto, como x aparece como punto de entrada en el *path expression* de z, esta última variable se va a asociar al bucle (2) (y no al (4)) que es donde aparece x como puntero de inducción.

- Asociamos a cada sentencia st del tipo $x = y$ y $x = y \rightarrow sel$ (sentencias que "visitan" porciones de memoria) una variable de inducción para que sea tenida en cuenta para la información $TOUCH$ de los nodos.

$$IP\_TOUCH(st) = \begin{cases} x \text{ si } x \in IPVARS(B_i) \ \wedge \ st \in Body(B_i) \\ z \text{ si } PE(x).e = z \ \wedge \ z \in IPVARS(B_i) \ \wedge \\ \qquad st \in Body(B_i) \\ \emptyset \text{ en otro caso.} \end{cases}$$

  La variable asociada a cada sentencia st será la introducida en la información $TOUCH$ del nodo al que apunte x tras el análisis de la sentencia, para indicar que las porciones representadas por dicho nodo han sido visitadas en un recorrido dependiente de dicha variable puntero. Como se puede observar, si la variable es un puntero de inducción del bucle al que pertenece la sentencia ($IPVARS(B_i)$) se introduce como IP_TOUCH de la sentencia. Si no lo es, pero el punto de entrada de su *path expression* si lo es, entonces se introduce dicho punto de entrada (z) en IP_TOUCH, para hacer constar que aunque la variable que visita la porción de memoria es x, lo hace en un recorrido que depende de z.

  Volviendo al código de la figura B.1, aplicando las reglas anteriores, las variables puntero de inducción asociadas a las sentencias que "visitan" porciones de memoria son: $IP\_TOUCH(3) = x$, $IP\_TOUCH(5) = z$, $IP\_TOUCH(6) = z$, $IP\_TOUCH(8) = x$ y $IP\_TOUCH(9) = x$. Como podemos observar, estas variables que son las que aparecerán en el conjunto $TOUCH$ de los nodos, tan solo están formadas por los punteros inducción detectados para los bucles (2) y (4), que son las variables desde las que empieza cualquier visita a las porciones de memoria en los recorridos realizados en dichos bucles.

Con los conjuntos IP_TOUCH asociados a los bucles y a las sentencias que visitan porciones de memoria, ya estamos en condiciones de mantener la información de que recorridos (variables puntero de inducción) han visitado las porciones representadas por los nodos. Por tanto, hay que definir los cambios en la semántica abstracta de las sentencias para que introduzcan y eliminen la información de $TOUCH$. Como vemos hay dos acciones con la información $TOUCH$, introducir una variable cuando visita las porciones de memoria representadas por el nodo, o eliminar una variable, cuando se termina el cuerpo del bucle al cual está asociada la variable puntero de inducción.

- **Introducción de una variable en** $TOUCH(n)$. Esto ocurre cuando las porciones de memoria representadas por el nodo n son "visitadas" en un recorrido. Como hemos comentado, tan solo hay dos sentencias que realicen esta "visita", $x = y$ y $x = y \rightarrow sel$. La modificación de la semántica abstracta de dichas sentencias es muy simple.

Sea $ST_{[x=y]_T}(rsg)$ la semántica abstracta de la instrucción $x = y$ modificada para que genere la información $TOUCH$. Queda definida de la siguiente manera:

$$ST_{[x=y]_T}(rsg) = rsg'_i$$

donde rsg'_i se crea a partir de $rsg_i = ST_{[x=y]}(rsg)$ que es el grafo obtenido tras aplicar la semántica abstracta de la sentencia $x = y$ al grafo rsg. El grafo rsg'_i es exactamente igual a rsg_i aplicando al nodo $n_x \in N(rsg'_i), < x, n_x > \in PL(rsg'_i)$, el siguiente cambio:

$$TOUCH(n_x) = TOUCH(n_x) \cup \{IP\_TOUCH(st)\}$$

Como vemos, se añade a $TOUCH$ del nodo "visitado" por x, la variable puntero de inducción asociada a la sentencia actual ($IP\_TOUCH(st)$).

Exactamente de la misma forma se define la nueva semántica abstracta de la sentencia $x = y \rightarrow sel$:

$$ST_{[x=y\rightarrow sel]_T}(rsg) = rsg'_i$$

donde de nuevo rsg'_i se crea a partir de $rsg_i = ST_{[x=y\rightarrow sel]}(rsg)$, cambiando el conjunto $TOUCH$ del nodo $n_x \in N(rsg'_i), < x, n_x > \in PL(rsg'_i)$ de la siguiente manera:

$$TOUCH(n_x) = TOUCH(n_x) \cup \{IP\_TOUCH(st)\}$$

Una vez visto como se introduce la información en los conjuntos $TOUCH$ de las sentencias, vamos a ver cuando es eliminada dicha información.

- **Borrado de una variable de** $TOUCH(n)$. Como hemos comentado, una variable de $TOUCH$ deja de ser útil cuando el bucle asociado con el recorrido de dicha variable ha terminado (con lo cual el propio recorrido a terminado y ya no es necesario mantener por separado porciones visitadas y no visitadas). Por tanto, cuando un bucle B_i termina, es decir, sus grafos pasan a la sentencia inmediatamente posterior, hay que eliminar de los conjuntos $TOUCH$ de los nodos del grafo, las variables asociadas con recorridos del bucle B_i, que acaba de terminar.

  Para llevar a cabo esta "limpieza" de la información $TOUCH$ al final de los bucles, hemos creado una *pseudoinstrucción* $ENDLOOP(B_i)$ de finalización de bucle, que es insertada justo detrás del cuerpo de los bucles. La semántica abstracta de esta *pseudoinstrucción* modificará los grafos que devuelve el bucle, eliminando de la propiedad $TOUCH$ de los nodos, aquellos punteros de inducción asociados al bucle B_i.

  La semántica abstracta queda definida de la siguiente manera:

$$ST_{[ENDLOOP(B_i)]}(rsg) = rsg'$$

  siendo rsg' una copia del grafo rsg, modificada de la siguiente manera:

$$\forall n \in N(rsg'), TOUCH(n) = TOUCH(n) \setminus IP\_TOUCH(B_i)$$

  donde se elimina del atributo $TOUCH$ de cada nodo del grafo, aquellas variables que aparezcan en el conjunto IP_TOUCH asociado al bucle que termina. La posterior

compresión de grafo, reducirá el numero de nodos uniendo aquellos que al eliminar algunas variables de *TOUCH*, les queda un conjunto de variables igual.

En la figura B.2 volvemos a presentar uno de los RSGs que se obtendrían para la sentencia (3) con información *TOUCH* en cada nodo, ya mostrado en la figura 3.12.

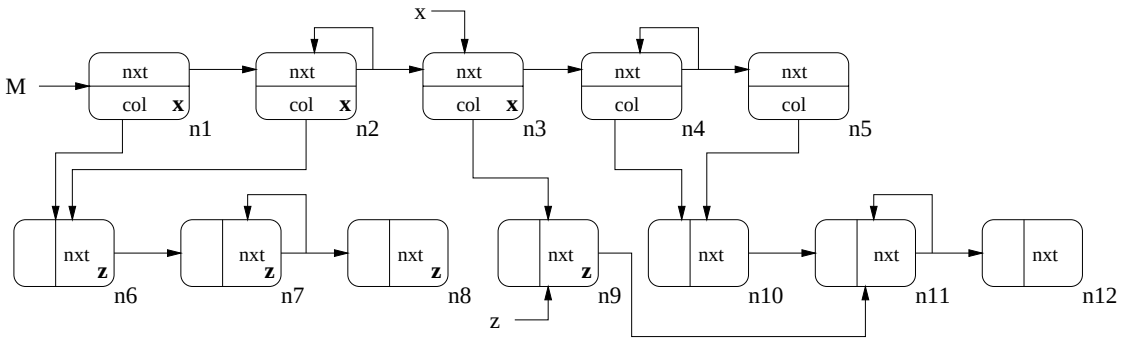


Figura B.2: RSG en la sentencia 3 con marcas de "visitas".

Podemos observar que aunque esta sentencia, (3), pertenece sólo al bucle externo, (2), en la información *TOUCH* se mantienen las variables x y z, aunque esta segunda es el puntero de inducción del bucle (4). Como hemos expuesto, esto es así puesto que el recorrido que realiza la variable z depende del que realiza la variable x (puesto que parte desde esta). Si se asociara z a su correspondiente bucle, (4), al terminar éste, se eliminaría z de los conjuntos *TOUCH* de los nodos. Esto implicaría que los nodos n_6, n_7 y n_8 serían indistinguibles de los nodos n_{10}, n_{11} y n_{12}, y por tanto serían sumarizados. Como vemos, el análisis volvería a tomar nodos que representan porciones de memoria ya visitadas y les volvería a aplicar las modificaciones oportunas. Al mantener z en el bucle (2), dicha variable no es eliminada de *TOUCH* hasta el final de dicho bucle, por lo que en (3) aparecen tanto x y z en el *TOUCH* de los nodos. Se puede observar, como z no pasará por ningún nodo ya visitado en el bucle, en pasos anteriores.

Apéndice C: Semántica Abstracta de los RSRSGs con multiselectores

En este apéndice vamos a describir los cambios producidos sobre la semántica abstracta de las sentencias, presentada en el apéndice B, asi como la semántica abstracta de las nuevas instrucciones que utilizan explícitamente alguna posición de un array de punteros. Por tanto vamos a tener dos partes bien diferenciadas, los cambios de la semántica de las sentencias simples con selectores normales ($x = NULL$, $x = malloc()$, $x = y$, $x \rightarrow sel = NULL$, $x \rightarrow sel = y$ y $x = y \rightarrow sel$), y por otro, las semántica abstracta de las nuevas sentencias con multiselectores ($x \rightarrow sel[i] = NULL$, $x \rightarrow sel[i] = y$ y $x = y \rightarrow sel[i]$) que debe ser completamente definida.

A parte de estas nuevas instrucciones con multiselectores, hemos introducido una nueva *pseudoinstrucción*, $FORCE(x \rightarrow sel[i] == NULL)$ totalmente equivalente a $FORCE(x \rightarrow sel == NULL)$ (ver apéndice B.7), pero para un multiselector en vez de para un selector.

C.1 Modificación de la semántica abstracta de las sentencias simples

En este apartado vamos a presentar las modificaciones sobre la semántica abstracta de las sentencias presentadas en el apéndice B.

Las semántica abstracta de las sentencias $x = NULL$, $x = malloc()$ y $x = y$ no se ve afectada por la inclusión de los multiselectores. Tan solo hay que decir, que el conjunto de multiselectores de los grafos devueltos por la semántica de cada sentencia, rsg' es exactamente igual al del grafo de entrada rsg para la sentencia $x = NULL$ ($MSL(rsg') = MSL(rsg)$), o al del grafo rsg_N para las sentencias $x = malloc()$ y $x = y$ ($MSL(rsg') = MSL(rsg_N)$).

C.1.1 Sentencia $[x \rightarrow sel = NULL]$

Indirectamente ya se han definido algunos cambios en la semántica abstracta de esta sentencia, ya que en ella se produce un proceso de "enfoque" del enlace $x \rightarrow sel$ con la operaciones

división, *poda* y las diversas *materializaciones*, y dichas operaciones ya han sido adaptadas para el soporte de grafos con multiselectores. Para este enfoque se necesita el conjunto de grafos *podados*, $PR(rsg, x, sel, i)$ donde ahora se necesita también la *ivar* i. Este conjunto se construye utilizando las nuevas operaciones $PRUNE(rsg)$ y $DIVIDE(rsg, x, sel, i)$. Por último, el nuevo conjunto de grafos *enfocados*, $F(rsg, x, sel, i)$, se construye utilizando las nuevas operaciones $MATERIALIZE\_NODE(rsg, x, sel, i)$ y $MATERIALIZE\_RSG(rsg, x, sel, i)$ sobre los grafos de $PR(rsg, x, sel, i)$.

Aparte del enfoque, la semántica abstracta de esta sentencia, elimina el enlace desde el nodo apuntado por x, n_x por el selector sel, en cada grafo obtenido tras el enfoque, rsg_i, generando uno nuevo, rsg'_i.

Puesto que esta sentencia no modifica ningún enlace por multiselectores, el conjunto de dichos multiselectores del nuevo grafo se obtiene directamente del grafo original:

- $MSL(rsg'_i) = MSL(rsg_i)$

Al eliminar el enlace $x \rightarrow sel$, cambian algunas propiedades del nodo $n \in RSG(rsg'_i)$, tal que $<x, n_x> \in PL(rsg'_i), < n_x, sel, n> \in NL(rsg'_i)$. Las definiciones de nuevas propiedades presentadas para esta sentencia en el capítulo anterior, que se ven modificadas por la inclusión de los multiselectores, son las relacionadas con la información *shared*. Antes, para comprobar si seguía siendo *shared* el nodo n, al cual se le ha eliminado un enlace de entrada, se miraban tan solo los enlaces por selectores simples. Ahora hay que mirar también los procedentes de multiselectores.

- $SHARED(n) = 0$ si $\nexists n_i, n_j \in N(rsg'_i)$ |
  $(< n_i, sel_i, n >, < n_j, sel_j, n > \in NL(rsg'_i) \wedge sel_i \neq sel_j \in S) \vee$
  $(< n_i, sel_i, ins_i, mc_i, n >, < n_j, sel_j, ins_j, mc_j, n > \in MSL(rsg'_i) \wedge sel_i \neq sel_j \in MS) \vee$
  $(< n_i, sel_i, n > \in NL(rsg'_i) \wedge\ < n_j, sel_j, ins_j, mc_j, n > \in MSL(rsg'_i))$

Por último, vimos que la propiedad *STRUCTURE* podía cambiar para muchos nodos puesto que al eliminar el enlace, una componente conexa podía romperse. Para llevar a cabo esta modificación, se utiliza el conjunto de nodos conectados a uno dado, $C(n, rsg)$. La definición de este conjunto debe de ser modificada, puesto que ahora dos nodos pueden conectarse además de por selectores, por multiselectores. La definición de este conjunto queda de la siguiente manera:

$$C(n, rsg) = \{n_i \in N(rsg) \mid \exists n_j \in N(rsg), (\exists n_1, n_2, .., n_n \in N(rsg), REF(n_j, n_1, rsg) = 1 \wedge REF(n_1, n_2, rsg) = 1 \wedge ... \wedge REF(n_n, n_i, rsg) = 1) \wedge (\exists n'_1, n'_2, .., n'_n \in N(rsg), REF(n_j, n'_1, rsg) = 1 \wedge REF(n'_1, n'_2, rsg) = 1 \wedge ... \wedge REF(n'_n, n, rsg) = 1)\}$$

y siendo $REF(n_1, n_2, rsg)$ una función booleana que devuelve *true* si los nodos n_1 y n_2 están enlazados por medio de un selector o multiselector en el grafo rsg:

$$REF(n_1, n_2, rsg) = \begin{cases} 1 \text{ si } \exists < n_1, sel_i, n_2 > \in NL(rsg) \vee \\ \qquad \exists < n_1, ms_i, ins_j, mc_k, n_2 > \in MSL(rsg) \\ 0 \text{ en caso contrario.} \end{cases}$$

C.1.2 Sentencia $[x \rightarrow sel = y]$

La semántica abstracta de esta sentencia, crea una serie de grafos nuevos rsg'_i, a partir de los grafos rsg_i obtenidos tras aplicar la sentencia $x \rightarrow sel = NULL$ al grafo de entrada. Estos nuevos grafos se obtienen añadiendo el nuevo enlace desde el nodo apuntado por x, n_x y el nodo apuntado por y, n_y por el selector sel.

El conjunto de multiselectores del grafo no se ve afectado por este nuevo selector, por tanto:

- $MSL(rsg'_i) = MSL(rsg_i)$

Por otro lado, el nuevo enlace cambia las propiedades del nodo n_y para reflejar la existencia del mismo. El cálculo de la nueva propiedad $SHARED$ debe tener en cuenta ahora la existencia de enlaces por medio de multiselectores sobre el nodo, y no sólo por selectores. Si existe algún otro selector o multiselector apuntando a n_y, su propiedad $SHARED$ pasará a valer *true* puesto que ahora también es apuntado por sel desde n_x.

- $SHARED(n_y) = 1$ si $\exists n_i \in N(rsg'_i),\ |\ (< n_i, sel_i, n_y > \in NL(rsg'_i) \wedge sel_i \neq sel)\ \vee\ (< n_i, ms_i, ins_j, mc_k, n_y > \in MSL(rsg'_i))$

La propiedad $STRUCTURE$ de los nodos n_x y n_y, y de los nodos conectados con ellos, pasa a ser la misma puesto que ahora existe un enlace por sel entre ambos nodos. Para ello se debe utilizar la nueva definición del conjunto $C(n, rsg)$ presentada anteriormente.

C.1.3 Sentencia $[y = x \rightarrow sel]$

La acción de esta sentencia es que la variable puntero y apunte a los nodos apuntados por $x \rightarrow sel$, previo enfoque de dicho enlace. Por tanto, la semántica abstracta de esta sentencia crea una serie de grafos rsg'_i a partir de los grafos rsg_i resultantes del enfoque de $x \rightarrow sel$ en el grafo de entrada rsg.

El conjunto de multiselectores de los grafos no se ve afectado, ya que esta sentencia no modifica ninguna conexión de los mismos. Por tanto:

- $MSL(rsg'_i) = MSL(rsg_i)$

Si el nodo apuntado por x, n_x, apunta por sel al nodo n, ahora n será apuntado directamente por la variable puntero y, lo que va ha provocar la inclusión de nuevos elementos en los conjuntos $SPATH$ de los nodos referenciados por n. Los multiselectores, amplían los elementos que pueden aparecer en dichos $SPATH$ puesto que ahora n puede apuntar a otros nodos mediante un multiselector. Por tanto, $\forall n \in N(rsg'_i)$:

$$SPATH(n) = SPATH(n)\ \cup\ \{< y, \emptyset >\ |\ < y, n > \in PL(rsg'_i)\}\ \cup$$
$$\{< y, sel_i >\ |\ < y, n_y > \in PL(rsg'_i)\ \wedge\ < n_y, sel_i, n > \in NL(rsg'_i)\}\ \cup$$
$$\{< y, sel_j >\ |\ < y, n_y > \in PL(rsg'_i)\ \wedge\ < n_y, sel_j, ins, mc, n > \in MSL(rsg'_i)\}$$

Así queda descrita la nueva semántica abstracta de las seis sentencias simples que manejan punteros, para que sea capaz de manejar grafos con multiselectores.

Aparte de las sentencias normales, en el apéndice B.7 introdujimos una serie de *pseudoinstrucciones* que utilizaban información adicional del código para acercar las posibles configuraciones de memoria que pueden aparecer en una sentencia a las que realmente aparecerán en tiempo de ejecución. La semántica abstracta de dichas *pseudoinstrucciones* no se ve afectada por la inclusión de los multiselectores en los grafos. tan solo la *pseudoinstrucción* $FORCE(x \rightarrow sel == NULL)$, crea un grafo modificado, rsg' a partir del de entrada rsg. Hay que añadir que el conjunto de multiselectores del nuevo grafo es igual que el del original, $MSL(rsg') = MSL(rsg)$.

A continuación presentamos la semántica abstracta de las tres nuevas sentencias que utilizan multiselectores, $x \rightarrow sel[i] = NULL$, $x \rightarrow sel[i] = y$ y $y = x \rightarrow sel[i]$, que como veremos es muy similar a la de su correspondiente basada en selectores normales, gracias al proceso de "pre-enfoque". Además, también hemos creado una *pseudoinstrucción* relacionada con los multiselectores, $FORCE(x \rightarrow sel[i] == NULL)$, equivalente a la de selectores simples $FORCE(x \rightarrow sel == NULL)$.

C.2 Sentencia $[x \rightarrow sel[i] = NULL]$

Esta sentencia va a eliminar el enlace que exista desde la porción de memoria apuntada por x en la posición i del array de punteros sel. Vemos, que la acción es exactamente igual a la llevada a cabo por la sentencia $x \rightarrow sel = NULL$, una vez determinado el valor de i. Por tanto, a la hora de crear la semántica abstracta de esta sentencia, las operaciones a realizar serán análogas a la semántica abstracta de $x \rightarrow sel = NULL$, pero previamente aislando los enlaces que puede tener la posición i del array sel en las configuraciones de memoria representadas por el grafo a analizar.

A este proceso de aislar los enlaces correspondientes a la posición i, lo hemos denominado "pre-enfoque" y lo hemos descrito en la sección 4.3.2. Como hemos visto consta de dos operaciones: *división por clases de multireferencias* para separar en distintos grafos aquellos conjuntos de enlaces que no pueden coexistir en las mismas configuraciones de memoria; *instanciación* del multiselector, creando una *instancia simple* que represente la posición i del array y todos aquellos enlaces que pueden aparecer en la misma, en las distintas configuraciones de memoria. Los enlaces de esta instancia pueden ser tratados ya como selectores normales, puesto que representan un único enlace en cada configuración de memoria representada por el grafo.

Al igual que en la sentencia $x \rightarrow sel = NULL$ definimos el conjunto de grafos enfocados, $F(rsg, x, sel, i)$, como paso previo a aplicar los cambios específicos de la sentencia, vamos a definir ahora el conjunto de grafos "pre-enfocados", como el conjunto de grafos obtenidos tras la aplicación de las operaciones del "pre-enfoque" y posteriormente las del "enfoque", sobre un determinado grafo de entrada para una sentencia con un multiselector.

Definimos el conjunto de grafos "pre-enfocados" y "enfocados", $PF(rsg, x, sel, i)$ de un grafo de entrada rsg, para el enlace $x \rightarrow sel[i]$, como:

$PF(rsg, x, sel, i) = \{rsg_i |$

- $[rsg_i = MATERIALIZE\_NODE(rsg_j, x, sel, i), \forall rsg_j \in PPR(rsg, x, sel, i) \wedge$
  $< x, n_x > \in PL(rsg_j) \wedge < n_x, sel, ins, mc, n > \in MSL(rsg_j) \wedge i \in ivs(ins) \wedge$
  $(\nexists pv \in P | < pv, n > \in PL(rsg_j))] \vee$

- $[\forall rsg_i \in MATERIALIZE\_RSG(rsg_j, x, sel, i), \forall rsg_j \in PPR(rsg, x, sel, i) \wedge$
  $< x, n_x > \in PL(rsg_j) \wedge \ < n_x, sel, ins, mc, n > \in MSL(rsg_j) \ \wedge \ i \in ivs(ins) \wedge$
  $(\exists pv \in P | < pv, n > \in PL(rsg_j))]\}$

donde el conjunto PPR es el conjunto de grafos *podados* resultantes de la operación de *división* de los grafos obtenidos tras el "pre-enfoque" de rsg.

$$PPR(rsg, x, sel, i) = \{rsg_j \mid rsg_j = PRUNE(rsg'_j), \ \forall \ rsg'_j \in DV(rsg, x, sel, i)\}$$

Este conjunto PPR se ha definido utilizando el conjunto de grafos *divididos*, DV, para el cual será necesario calcular previamente el conjunto de grafos instanciados, INS, provenientes de la división por clases de multireferencias.

$$DV(rsg, x, sel, i) = \{rsg_i \mid rsg_i = DIVIDE(rsg_j, x, sel, i), \forall rsg_j \in INS(rsg, x, sel, i)\}$$

$$INS(rsg, x, sel, i) = \{rsg_i \mid rsg_i = INSTANCE(x, sel, i, st, rsg_j), \\ \forall rsg_j \in MC\_DIVIDE(x, sel, rsg)\}$$

De esta forma queda definido el conjunto de grafos enfocados PF sobre el enlace $x \rightarrow sel[i]$. Si revisamos su construcción, nos damos cuenta que se aplican las operaciones de "pre-enfoque" y posteriormente las de "enfoque" sobre el grafo de entrada. Primero, este grafo se *divide por clases de multireferencias* (MC_DIVIDE). Los grafos resultantes son *instanciados* ($INSTANCE$), con lo que la fase de "pre-enfoque" termina. A continuación, cada uno de estos grafos instanciados, es *dividido* ($DIVIDE$) para separar los distintos enlaces, y los grafos resultantes de esta división son *podados* ($PRUNE$). Por último, en cada grafo resultante, se *materializa un grafo* ($MATERIALIZE\_RSG$) o se *materializa un nodo* ($MATERIALIZE\_NODE$), según sea o no apuntado el nodo destino por una variable puntero.

Terminado el enfoque para los enlaces con multiselectores, ya es posible definir la semántica abstracta de la sentencia $x \rightarrow sel[i] = NULL$:

$$ST_{[x \rightarrow sel[i]=NULL]}(rsg) = \{rsg'_i \mid \forall rsg_i \in PF(rsg, x, sel, i)\}$$

Los grafos rsg'_i son el resultado de modificar de la siguiente forma los grafos obtenidos tras el enfoque (PF):

- $N(rsg'_i) = N(rsg_i)$
- $PL(rsg'_i) = PL(rsg_i)$
- $NL(rsg'_i) = NL(rsg_i)$
- $MSL(rsg'_i) = MSL(rsg_i) \setminus \{< n_x, sel, ins, mc, n > \in ML(rsg_i) \mid \ < x, n_x > \in PL(rsg_i)$
  $\wedge \ i \in ivs(ins)\} \cup \{< n_x, sel, ins, mc, NULL > \ | \ < x, n_x > \in PL(rsg_i) \wedge i \in ivs(ins)\}$

Lo único que se modifica es el conjunto de multiselectores, eliminando el que hay desde el nodo apuntado por x, n_x por el multiselector sel y la instancia ins que contiene a la $ivar$ i, al nodo n, sustituyéndolo por un enlace $NULL$. La desaparición de este enlace provoca modificaciones en las propiedades de los nodos fuente, n_x, y destino, n, del mismo. Las nuevas propiedades de n son:

- $SPATH(n) = SPATH(n) \setminus \{< x, sel > \mid \nexists < n_x, sel, ins', mc, n > \in MSL(rsg_i')\}$
- $SELINset(n) = SELINset(n) \setminus \{sel \mid SHSEL(n, sel) = 0 \vee (\nexists n_1 \in N(rsg_i') \wedge < n_1, sel, ins, mc, n > \in MSL(rsg_i'))\}$
- $PosSELINset(n) = PosSELINset(n) \setminus \{sel \mid SHSEL(n, sel) = 0 \vee (\nexists n_1 \in N(rsg_i') \wedge < n_1, sel, ins, mc, n > \in MSL(rsg_i'))\} \cup \{sel \mid SHSEL(n, sel) = 1 \wedge (\exists n_1 \in N(rsg_i') \wedge < n_1, sel, ins, mc, n > \in MSL(rsg_i'))\}$
- $SHARED(n) = 0$ si $\nexists n_i, n_j \in N(rsg_i') \mid$
  $(< n_i, sel_i, n >, < n_j, sel_j, n > \in NL(rsg_i') \wedge sel_i \neq sel_j \in S) \vee$
  $(< n_i, sel_i, ins_i, mc_i, n >, < n_j, sel_j, ins_j, mc_j, n > \in MSL(rsg_i') \wedge sel_i \neq sel_j \in MS) \vee$
  $(< n_i, sel_i, n > \in NL(rsg_i') \wedge < n_j, sel_j, ins_j, mc_j, n > \in MSL(rsg_i'))$
- $SHSEL(n, sel) = 0$ si $[\nexists n_i \in N(rsg_i') \mid < n_i, sel, ins, mc, n > \in MSL(rsg_i')] \vee [\exists_1 n_i \in N(rsg_i') \mid < pvar, n_i > \in PL(rsg_i') \wedge \exists_1 < n_i, sel, ins, mc, n > \in MSL(rsg_i') \wedge \exists iv \in IV(rsg), iv \in ivs(ins)]$

Los cambios en las propiedades del nodo n son similares a los producidos por la sentencia $x \rightarrow sel$, pero haciendo ahora referencia a la desaparición de un enlace por un multiselector. La principal diferencia es en el nuevo valor de $SHSEL(n, sel)$. La justificación de las modificaciones de estas propiedades puede ser consultada en el apéndice B. Una diferencia destacable con la descripción presentada allí, es que el nodo n puede dejar de ser *shared* por el multiselector sel, si ya no es apuntado por ningún enlace de dicho multiselector, o tan solo existe un nodo n_1, apuntado por una variable puntero, y un enlace por el multiselector sel, que apunta a n. La diferencia es que la instancia a la que pertenece el enlace debe de ser una instancia simple (tiene que tener una *ivar* en su identificador), puesto que si es múltiple, este enlace por el multiselector puede representar varios enlaces desde la misma porción de memoria, y por tanto n debe seguir siendo *shared* por el multiselector sel.

Otra singularidad, es que a diferencia de $x \rightarrow sel$, ahora los *cyclelinks* de n no se modifican, puesto que ahora se elimina un multiselector que apunta a n, y como vimos al principio de este capítulo, un multiselector no tiene sentido que aparezca como segundo elemento (enlace de vuelta) de un *cyclelink*.

En cuanto a las propiedades del nodo n_x, las afectadas son:

- $SELOUTset(n_x) = SELOUTset(n_x) \setminus \{sel\}$
- $CYCLELINKS(n_x) = CYCLELINKS(n_x) \setminus \{< sel, sel_i > \mid \forall sel_i, \nexists < n_x, sel, ins, mc, n_d > \in MSL(rsg_i')\}$

Es decir, eliminamos el multiselector sel de los selectores de salida del nodo. Además si el nodo n_x ya no apunta a ningún otro nodo por el multiselector sel, entonces se eliminan los *cyclelinks* cuyo primer elemento sea sel.

Por último, al igual que en $x \rightarrow sel$, la eliminación de este multiselector, puede provocar la ruptura de una componente conexa, cambiando la propiedad $STRUCTURE$ de muchos nodos. Utilizando la definición del conjunto $C(n, rsg)$ de nodos del grafo rsg conectados con n (definición modificada en esta sección cuando se presentó la semántica abstracta nueva para la sentencia $x \rightarrow sel$) definimos la nueva propiedad $STRUCTURE$.

Si $C(n_x, rsg'_i) \cap C(n, rsg'_i) = \emptyset$ entonces $\forall n_i \in C(n_x, rsg'_i), STRUCTURE(n_i) = S_1 \ \wedge \ \forall n_j \in C(n, rsg'_i), STRUCTURE(n_j) = S_2$, donde S_1 y S_2 son los identificadores de dos estructuras nuevas.

De esta forma queda completamente descrita la semántica abstracta de la sentencia $x \rightarrow sel[i] = NULL$.

C.3 Sentencia $[x \rightarrow sel[i] = y]$

Esta sentencia va a crear un nuevo enlace entre el nodo apuntado por x, n_x, y el apuntado por y, n_y, mediante la posición i del array de punteros sel. Como ocurría con $x \rightarrow sel = y$, antes de la creación el nuevo enlace, se debe eliminar el que existiera anteriormente. Por ese motivo, antes de aplicar la semántica abstracta de esta sentencia, se aplica la de $x \rightarrow sel[i] = NULL$. Esto provoca que además de eliminar el enlace, se "pre-enfoque" y "enfoque" la posición i del multiselector, pudiendo aplicar las modificaciones de esta sentencia directamente.

La semántica abstracta de esta sentencia queda definida como:

$$ST_{[x \rightarrow sel[i]=y]}(rsg) = \{rsg'_i \mid \forall rsg_i \in ST_{[x \rightarrow sel[i]=NULL]}(rsg)\}$$

construyendo un conjunto de nuevos grafos rsg'_i, transformando los grafos rsg_i obtenidos tras aplicar la semántica abstracta de la sentencia $x \rightarrow sel[i] = NULL$. La construcción de los nuevos grafos debe reflejar la inclusión del nuevo multiselector:

- $N(rsg'_i) = N(rsg_i)$
- $PL(rsg'_i) = PL(rsg_i)$
- $NL(rsg'_i) = NL(rsg_i)$
- $MSL(rsg'_i) = MSL(rsg_i) \setminus \{< n_x, sel, ins_i, mc_i, NULL > \mid i \in ivs(ins_i) \wedge \ < x, n_x > \in PL(rsg_i)\} \ \cup \ \{< n_x, sel, ins_i, mc_i, n_y > \mid i \in ivs(ins_i) \ \wedge \ < x, n_x > \in PL(rsg_i) \ \wedge \ < y, n_y > \in PL(rsg_i)\}$

Vemos como la instancia que contiene en su identificador a i, sustituye su enlace $NULL$ por uno al nodo n_y. De nuevo la inserción de este nuevo enlace va a cambiar las propiedades relacionadas con la conectividad de los nodos n_x (tiene un enlace más por el multiselector sel) y n_y (es apuntado por un enlace con el multiselector sel).

Si n_x y n_y son los dos nodos de rsg'_i de modo que $<x, n_x > \in PL(rsg'_i)$ y $<y, n_y > \in PL(rsg'_i)$, las propiedades modificadas de n_x son:

- $SELOUTset(n_x) = SELOUTset(n_x) \ \cup \ \{sel \mid \nexists < n_x, sel, ins_j, mc_j, NULL > \in MSL(rsg'_i)\} \ \setminus \ \{sel \mid \exists < n_x, sel, ins_j, mc_j, NULL > \in MSL(rsg'_i)\}$

- $PosSELOUTset(n_x) = PosSELOUTset(n_x) \setminus \{sel \mid \nexists < n_x, sel, ins_j, mc_j, NULL > \in MSL(rsg'_i)\} \cup \{sel \mid \exists < n_x, sel, ins_j, mc_j, NULL > \in MSL(rsg'_i)\}$

- $CYCLELINKS(n_x) = CYCLELINKS(n_x) \cup \{< sel, sel_i > \mid (\exists < n_y, sel_i, n_x > \in NL(rsg'_i)) \wedge (SELOUT(n_y, sel_i) = 1) \wedge (\nexists n_2 \in N(rsg'_i), < n_y, sel_i, n_2 > \in NL(rsg'_i)) \wedge (\nexists < n_x, sel, ins', mc', n_d > \in MSL(rsg'_i) \wedge i \notin ivs(ins'))\} \setminus \{< sel, sel_j > \mid (< n_y, sel_j, n_x > \notin NL(rsg'_i)) \vee (\exists n_2 \in N(rsg'_i), < n_y, sel_j, n_2 > \in NL(rsg'_i))\}$

Tras la inserción del enlace por el multiselector *sel* en el nodo n_x, dicho selector formará parte de los *selectores de salida* del nodo (*SELOUTset*) si ya no existe ningún enlace *NULL* en el multiselector *sel* de n_x. Si por el contrario, existe algún enlace *NULL* en *sel*, sabemos que algunas posiciones de sel tienen enlaces y otras no, por lo tanto *sel* debe ser introducido en los *posibles selectores de salida* de n_x (*PosSELOUTset*).

La modificación de la propiedad *cyclelinks* para este caso es bastante más compleja que el de la sentencia $x \rightarrow sel = y$. Para introducir ahora un par del tipo $<sel, sel_i>$ se tienen que cumplir una serie de condiciones. Para introducir $<sel, sel_i>$ debería cumplirse que el nodo n_y, ahora apuntado por n_x y *sel*, apunte de nuevo a n_x por el selector sel_i y sólo a n_x. Hasta aquí todo normal, pero el problema surge cuando el nodo n_x apunta a otro u otros nodos por el multiselector *sel* (esto no es posible con selectores normales pero si con multiselectores puesto que representan múltiples enlaces). Si apunta a otros nodos por *sel*, entonces no se introduce el par $<sel, sel_i>$, aunque sea cierto para n_y. Para que se introduzca, dicho *cyclelink* debería satisfacerse para *sel* y el otro nodo, en cuyo caso ya estaría dentro del conjunto *CYCLELINKS*, por lo que no hay que volverlo a introducir. Si para el otro nodo y *sel* no se cumple el par $<sel, sel_i>$ (por ejemplo el otro nodo no apunta a n_x por sel_i), entonces tampoco hay que introducirlo ya que no se cumple para todos los enlaces por *sel*. Como vemos, en cualquier caso, si hay otro enlace por *sel*, entonces el conjunto de *CYCLELINKS* no se ve modificado (o ya existe el par o no se puede meter).

Pero no acaban ahí las modificaciones sobre los *cyclelinks* de n_x, sino que paradójicamente, esta instrucción que introduce un nuevo enlace, puede provocar la eliminación de parejas en los *CYCLELINKS* de n_x. Para la sentencia $x \rightarrow sel = y$, esto no sucede, puesto que al crear un enlace no se puede romper ningún *cyclelink* existente. La diferencia con la instrucción $x \rightarrow sel[i] = y$, es que esta crea un nuevo enlace por *sel* en la posición i, pero pueden existir más enlaces en otras posiciones. Esto implica que en n_x puede existir ya un *cyclelink* $<sel, sel_j>$ donde *sel* es el multiselector por el cual se está creando el nuevo enlace. Esta es la diferencia fundamental con la instrucción que utiliza un selector simple, al insertar un enlace por dicho selector es imposible que exista un *cyclelink* que comience con dicho selector, puesto que no puede existir ningún otro enlace por él aparte del que se está creando. Por tanto, se deben eliminar aquellas parejas $<sel, sel_j>$ donde el primer elemento es el multiselector *sel*, y el nodo n_y no apunta "sólo" a n_x por el segundo, sel_j. Esto sucederá cuando los demás enlaces existentes por el multiselector *sel* satisfagan el *cyclelink* $<sel, sel_j>$ pero el nuevo insertado sobre n_y no.

Por otro lado las propiedades de n_y deben reflejar en nuevo enlace de entrada por *sel* desde n_x, de la siguiente manera:

- $SELINset(n_y) = SELINset(n_y) \cup \{sel\}$

- $PosSELINset(n_y) = PosSELINset(n_y) \setminus \{sel\}$

- $SPATH(n_y) = SPATH(n_y) \cup \{< x, sel >\}$
- $SHARED(n_y) = 1$ si $\exists n_i \in N(rsg_i),\ |\ [< n_i, sel_i, n_y > \in NL(rsg_i)]\ \vee\ [< n_i, sel_i,\ ins,\ mc,\ n_y > \in MSL(rsg_i)\ \wedge\ sel_i \neq sel]$
- $SHSEL(n_y, sel) = 1$ si $\exists n_i \in N(rsg_i)\ |\ < n_i, sel, ins, mc, n_y > \in NL(rsg_i)$

Las propiedades de n_y cambian de una manera más obvia, así, definitivamente las porciones de memoria representadas por n_y son apuntadas por el multiselector *sel* (se introduce en *SELINset* y se elimina de *PosSELINset*). Además como ahora dichas porciones son apuntadas por algún enlace desde $x \rightarrow sel[i]$, hay que introducir $<x, sel>$ en los *simple paths* de n_y. Por último si el nodo era apuntado por otros selectores o multiselectores distintos de *sel* pasará a ser *shared* ($SHARED(n_y) = 1$), y si lo era por el propio multiselector *sel*, al serlo de nuevo ahora, pasa a ser *shared* por *sel* ($SHSEL(n_y, sel) = 1$).

Al igual que con la sentencia $x \rightarrow sel$, ahora usamos la nueva definición del conjunto $C(n, rsg)$, de nodos de rsg conectados a n, para cambiar el atributo $STRUCTURE$ de los nodos conectados a n_x y n_y, puesto que ahora forman parte (si no lo eran) de la misma componente conexa.

$$\forall n_i \in C(n_x, rsg_i'), STRUCTURE(n_i) = STRUCTURE(n_x)$$

C.4 Sentencia $[y = x \rightarrow sel[i]]$

Esta sentencia tiene que hacer que y apunte a aquellas porciones de memoria referenciadas por la posición i del array de punteros *sel* de las porciones de memoria apuntadas por x. En el dominio de los grafos, tenemos que localizar aquellos selectores que pueden ser los que hay en la posición i del multiselector *sel* del nodo n_x apuntado por x, o sea, tenemos que "pre-enfocar". A continuación, tenemos que "enfocar" todos esos posibles enlaces para que puedan ser tratados individualmente, y por último, hacer que la variable y apunte a los nodos destino de dichos enlaces.

Para las operaciones de "pre-enfocado" y "enfocado" vamos a utilizar la definición del conjunto $PF(rsg, x, sel, i)$ presentada en la semántica abstracta de la sentencia $x \rightarrow sel[i] = NULL$. Además, como ya se explicó cuando se presentó la semántica abstracta de $y = x \rightarrow sel$, antes de modificar el grafo, hay que aplicar la semántica de $y = NULL$, puesto que la nueva asignación de y así lo precisa.

Por tanto definimos:

$$ST_{[y=x\rightarrow sel[i]]}(rsg) = \{rsg_i' \mid \forall rsg_i \in PF(ST_{[y=NULL]}(rsg), x, sel, i)\}$$

El conjunto de grafos modificados rsg_i' por la semántica abstracta de esta sentencia, se obtiene de los "enfocados" rsg_i de la siguiente forma:

- $N(rsg_i') = N(rsg_i)$
- $PL(rsg_i') = PL(rsg_i) \cup \{< y, n > \ |\ < x, n_x > \in PL(rsg_i)\ \wedge\ < n_x, sel, ins, mc, n > \in NL(rsg_i)\ \wedge\ i \in ivs(ins)\}$
- $NL(rsg_i') = NL(rsg_i)$

- $MSL(rsg_i') = MSL(rsg_i')$

Se añade un enlace desde la variable y al nodo apuntado desde n_x por sel y con la instancia que represente a la posición i del array ($i \in ivs(ins)$).

La única propiedad a la que afecta esta sentencia es a los *simple paths* de los nodos apuntados por el nodo al que ahora apunta y, puesto que ahora existe un camino de longitud uno desde la variable puntero y. Por supuesto, en los *simple paths* de dicho nodo ahora aparecerá $<y, \emptyset>$ puesto que y apunta directamente al nodo.

Por tanto, $\forall n \in N(rsg_i')$:

$$SPATH(n) = SPATH(n) \cup \{< y, \emptyset > \mid < y, n > \in PL(rsg_i')\} \cup$$
$$\{< y, sel_i > \mid < y, n_y > \in PL(rsg_i') \wedge < n_y, sel_i, n > \in NL(rsg_i')\} \cup$$
$$\{< y, sel_j > \mid < y, n_y > \in PL(rsg_i') \wedge < n_y, sel_j, ins, mc, n > \in MSL(rsg_i')\}$$

Así terminamos la definición de la semántica abstracta de las nuevas instrucciones que maneja multiselectores. A continuación presentamos la semántica abstracta de la *pseudoinstrucción* $FORCE(x \rightarrow sel[i] == NULL)$.

C.5 Pseudoinstrucción $FORCE(x \rightarrow sel[i] == NULL)$

Con esta *pseudoinstrucción* se pretende indicar que en las configuraciones de memoria que se pueden dar en ese punto del programa, las posición i del array sel de la porción de memoria apuntada por x debe de tener el valor $NULL$. En el dominio de los grafos, en un mismo grafo pueden estar representadas configuraciones de memoria que cumplan la restricción y otras que no la cumplan. Por tanto, la semántica abstracta de esta sentencia deberá eliminar del grafo, en la medida de lo posible, aquellos nodos y enlaces que representan porciones y enlaces de configuraciones de memoria que no satisfacen la restricción.

Para llevar esto a cabo, la idea es, de una manera similar a $FORCE(x \rightarrow sel == NULL)$, eliminar dicho enlace, y posteriormente *podar* el grafo, de manera que los nodos y enlaces que ahora no cumplen sus propiedades es porque formaban parte de una configuración de memoria en la que existía dicho enlace, y son eliminados. Para la instrucción que utiliza el selector normal, el enlace del nodo apuntado por x por sel puede borrarse directamente pues identifica los enlaces a los que hace referencia la *pseudoinstrucción*. Sin embargo, con los multiselectores, como ya sabemos, esto no es así, puesto que el multiselector representa múltiples enlaces desde la misma porción de memoria. Para poder eliminar el enlace y hacer la posterior poda, en el multiselector sel del nodo apuntado por x debe existir una instancia simple que represente las posiciones que puede tomar la *ivar* i (i debe aparecer en el identificador de la instancia). Si existe, se pueden borrar los enlaces que posea dicha instancia puesto que se puede tratar como un selector simple. Sin embargo si dicha instancia no existe en el multiselector, no se puede hacer nada, y el resultado de aplicar al *pseudoinstrucción* sobre un grafo será el propio grafo.

Teniendo todo esto en cuenta, definimos la semántica abstracta de esta *pseudoinstrucción* como:

$$ST_{[FORCE(x \rightarrow sel[i] == NULL)]}(rsg) = PRUNE(rsg')$$

donde rsg' se construye a partir de rsg de la siguiente manera:

- $N(rsg') = N(rsg)$
- $PL(rsg') = PL(rsg)$
- $NL(rsg') = NL(rsg)$
- $MSL(rsg') = MSL(rsg) \setminus \{< n_x, sel, ins, mc, n > \mid \forall < n_x, sel, ins, mc, n > \in MSL(rsg), < x, n_x > \in PL(rsg) \ \wedge \ i \in ivs(ins)\}$

Queda así completamente definida la semántica abstracta de esta *pseudoinstrucción.*

Apéndice D: Ejemplo de RSG completo

En este apéndice vamos a presentar el único RSG obtenido en el RSRSG de la última sentencia del código que genera la estructura ejemplo utilizada en el capítulo 3 sección 3.2.1.

Recordamos que se trata de una estructura en la que una variable puntero S apunta a una lista doblemente enlazada por los selectores *nxt* y *prv*. Cada elemento de esta lista apunta a su vez a un árbol binario. Por último las hojas de estos árboles apuntan por el selector *list* a otras listas doblemente enlazadas. En la figura D.1 volvemos a presentar una visión esquemática de esta estructura.

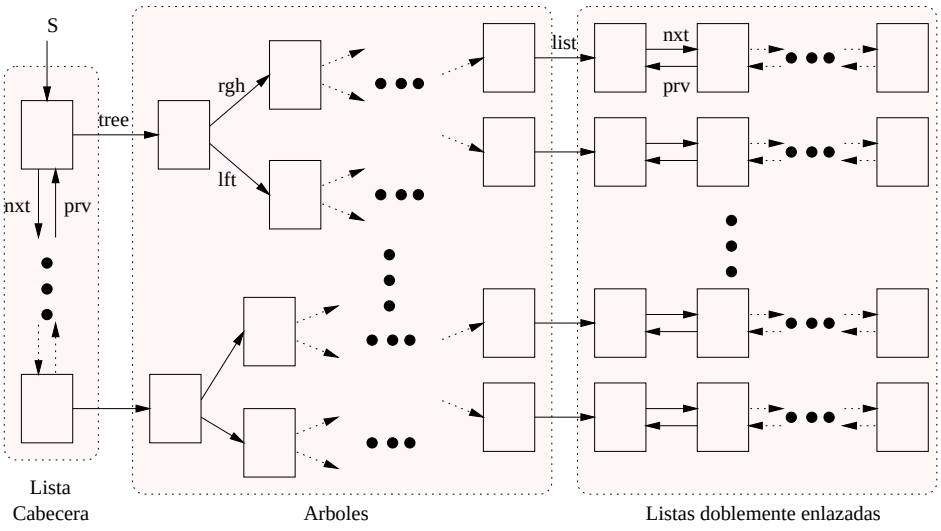


Figura D.1: Ejemplo de estructura de datos dinámica compleja.

En la figura D.2 presentamos el RSG obtenido por el compilador tras el análisis del código, que representa la estructura de datos de la figura D.1. Podemos observar que existen bastantes nodos y enlaces entre los mismos. En el interior de cada nodo aparece un número que lo identifica, sus *simple paths* si posee alguno y el conjunto de *cycle links*. El color amarillo de algunos nodos indica que tienen su propiedad $SHARED = true$.

Los nodos 1, 2, 3, 4 y 5 representan las porciones de memoria de la lista cabecera. Por ejemplo los nodos 3 y 4 representan las últimas porciones de la lista, pero difieren en el conjunto

de *simple paths*. Vemos como los nodos 2 y 5 que representan a los elementos centrales de la lista cabecera son *shared*, puesto que dichos elementos son referenciados a la vez por *nxt* desde el elemento anterior y por *prv* desde el posterior.

Los nodos del 6 al 17 representan las porciones de memoria de los árboles. El por qué de la existencia de tantos nodos es porque hay muchas combinaciones distintas de selectores de entrada/salida a las porciones de memoria (*reference patterns*). Así, por ejemplo, el nodo 10 representa a porciones de memoria apuntadas por *tree* que no tienen ningún hijo. El nodo 11 representa a las apuntadas por *tree* con hijos a la derecha (*rgh*) y a la izquierda (*lft*). Los nodos 12 y 13 representan porciones que sólo tienen hijos por la derecha o izquierda respectivamente. Otro ejemplo a destacar son los nodos 14 y 15 que representan a los elementos centrales de los árboles apuntados por el selector *rgh* o *lft* respectivamente. Por último los nodos 16 y 17 representan a las hojas de los árboles apuntadas por *rgh* o *lft* respectivamente.

Los nodos del 18 al 21 representan las listas doblemente enlazadas apuntadas desde los árboles. En concreto el nodo 18 representa listas de tan solo un elemento. En cuanto a los restantes nodos (19, 20 y 21) representan las restantes listas de dos o más elementos. El nodo 21 es *shared* por las mismas razones expuestas anteriormente para los nodos 2 y 5. Sin embargo, en este caso el nodo 19 que representa a los primeros elementos de dichas listas, también es *shared*, ya que las localizaciones que representa son apuntadas a la vez por dos selectores distintos: *prv* desde el siguiente elemento de la lista y por *list* desde una hoja de algún árbol.

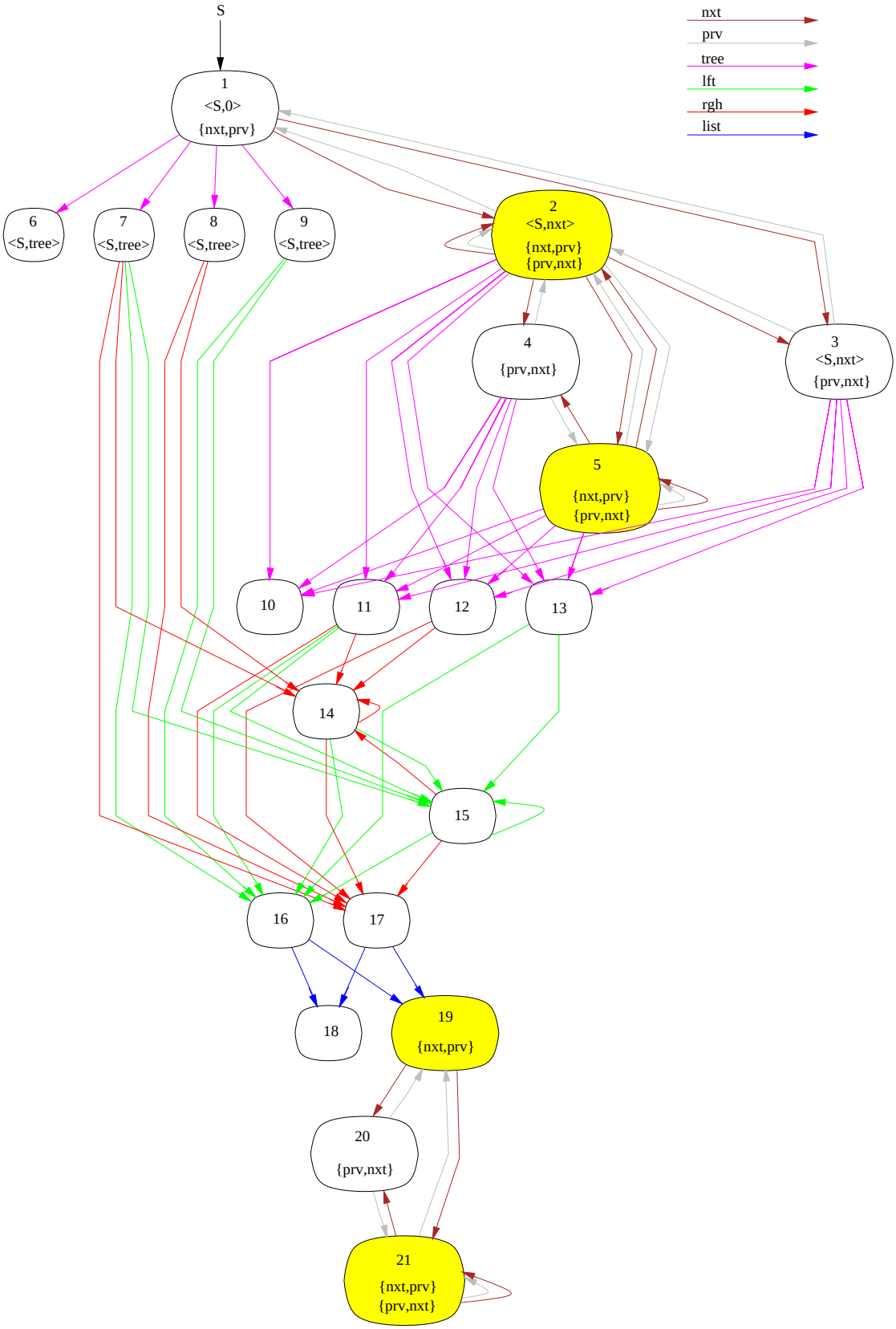


Figura D.2: RSG resultante para el código ejemplo.

Apéndice E: Códigos analizados

En este apéndice presentamos los códigos analizados por el compilador. Aparecen tanto la versión original como la transformada para que sólo aparezcan los seis tipos básicos de sentencias que manejan punteros además de las *pseudoinstrucciones FORCE* y *DeleteIndex*.

E.1 Códigos sin arrays de punteros

En esta sección presentamos los códigos basados en estructuras de datos que no utilizan arrays de punteros, cuyos resultados han sido presentados en el capítulo 3.

E.1.1 Código del programa ejemplo

```
/* Estructura Arbol */
typedef struct t1 {
  t1_data data;
  struct t1 *rgh, * lft ;
  struct t2 * list ;
}

/* Estructura Lista */
typedef struct t2 {
  t2_data data;
  struct t2 *nxt, * prv;
}

/* Estructura Lista Cabecera */
typedef struct t3 {
  t3_data data;
  struct t3 *nxt, * prv;
  struct t1 *tree;
}

void main() {
  struct t1 *ct_t, *il_aux, *il_new, *il_new2,
            *il_aux2, *tmptree1, *tmptree2;
  struct t2 *cl_l, *cl_aux, *cl_new, *il_auxl;
  struct t3 *S, *cd_aux, cd_new, *aux1, *aux2,
            *tmp;

  /* creacion de la estructura */
  S = malloc;
  ct_t = NULL;
  ct_t = malloc;
  while () {
  cl_l = NULL;
  /* creacion de una lista */
  cl_l = malloc;
  cl_aux = cl_l;
  while () {
    cl_new = malloc;
    cl_aux→nxt = cl_new;
    cl_new→prv = cl_aux;
    cl_aux = cl_new;
  }
  cl_aux = NULL;
  cl_new = NULL;
  il_aux = ct_t;
  /* creacion de un arbol insertando
     la lista */
  il_new = malloc;
  il_new→ list = cl_l;
  while () {
    FORCE il_new != NULL;
    if () {
      il_new2 = malloc;
      il_auxl = il_aux→ list ;
      il_aux→ list = NULL;
      il_new2→ list = il_auxl;
      il_auxl = NULL;
      if () {
        il_aux→rgh = il_new;
        il_aux→ lft = il_new2;
      }
      else {
        il_aux→rgh = il_new2;
        il_aux→ lft = il_new;
```

```
      }
      il_aux = NULL;
      il_new = NULL;
      il_new2 = NULL;
    }
    else {
      FORCE il_aux→list == NULL;
      if () {
        if () {
          FORCE il_aux→rgh == NULL;
          il_aux→rgh = il_new;
          il_aux = NULL;
          il_new = NULL;
        }
        else {
          il_aux2 = il_aux→rgh;
          il_aux = il_aux2;
          il_aux2 = NULL;
        }
      }
      else {
        if () {
          FORCE il_aux→lft == NULL;
          il_aux→ lft = il_new;
          il_aux = NULL;
          il_new = NULL;
        }
        else {
          il_aux2 = il_aux→ lft ;
          il_aux = il_aux2;
          il_aux2 = NULL;
        }
      }
    }
  }
  FORCE il_aux == NULL;
  FORCE il_new == NULL;
}
cl_l = NULL;
/* insertar arbol como primer elemento
   de la lista cabecera */
S→tree = ct_t;
ct_t = NULL;
cd_aux = S;
while () {
  cd_new = malloc;
  ct_t = NULL;
  ct_t = malloc;
  while () {
    cl_l = NULL;
    /* creacion de nuevas listas */
    cl_l = malloc;
    cl_aux = cl_l;
    while () {
      cl_new = malloc;
      cl_aux→nxt = cl_new;
      cl_new→prv = cl_aux;
      cl_aux = cl_new;
    }
    cl_aux = NULL;
    cl_new = NULL;
    il_aux = ct_t;
    /* creacion de arboles insertando
       las listas */
    il_new = malloc;
    il_new→ list = cl_l;
    while () {
      FORCE il_new != NULL;
      if () {
        il_new2 = malloc;
        il_auxl = il_aux→ list ;
        il_aux→ list = NULL;
        il_new2→ list = il_auxl;
        il_auxl = NULL;
        if () {
          il_aux→rgh = il_new;
          il_aux→ lft = il_new2;
        }
        else {
          il_aux→rgh = il_new2;
          il_aux→ lft = il_new;
        }
        il_aux = NULL;
        il_new = NULL;
        il_new2 = NULL;
      }
      else {
        FORCE il_aux→list == NULL;
        if () {
          if () {
            FORCE il_aux→rgh == NULL;
            il_aux→rgh = il_new;
            il_aux = NULL;
            il_new = NULL;
          }
          else {
            il_aux2 = il_aux→rgh;
            il_aux = il_aux2;
            il_aux2 = NULL;
          }
        }
        else {
          if () {
            FORCE il_aux→lft == NULL;
            il_aux→ lft = il_new;
            il_aux = NULL;
            il_new = NULL;
          }
          else {
            il_aux2 = il_aux→ lft ;
            il_aux = il_aux2;
            il_aux2 = NULL;
          }
        }
      }
    }
    FORCE il_aux == NULL;
    FORCE il_new == NULL;
  }
  cl_l = NULL;
  /* insertar arboles en lista cabecera */
  cd_new→tree = ct_t;
  ct_t = NULL;
  cd_aux→nxt = cd_new;
  cd_new→prv = cd_aux;
  cd_aux = cd_new;
}
cd_new = NULL;
cd_aux = NULL;
/* recorrido sobre lista cabecera */
aux1 = S;
while () {
  tmp = aux1→nxt;
  aux1 = tmp;
}
tmp = NULL;
/* segundo recorrido sobre lista cabecera */
aux2 = aux1→nxt;
while () {
  tmp = aux2→nxt;
  aux2 = tmp;
}
tmp = NULL;
/* intercambio de arboles entre elementos
```

```
    de la lista cabecera */
  tmptree1 = aux1→tree;
  tmptree2 = aux2→tree;
  aux1→tree = tmptree2;
  aux2→tree = tmptree1;
  tmptree1 = NULL;
  tmptree2 = NULL;
  aux1 = NULL;
  aux2 = NULL;
}
```

E.1.2 Código multiplicación matriz dispersa por vector

```
/* Estructura Lista Cabecera Matriz */
typedef struct t1 {
  int rowindex;
  struct t1 *nxt, *prv;
  struct t2 *row;
}

/* Estructura Filas Matriz y Vectores */
typedef struct t2 {
  double data;
  int index;
  struct t2 *nxt, *prv;
}

void main() {
  struct t1 *M, *auxf, *newf;
  struct t2 *r, *v, *auxf1, *auxf2, *auxv, *new,
            *auxr;
  char eof1 = 0, eof2 = 0;
  double prod;

  auxf = NULL;
  /* creacion y llenado de la matriz M */
  while (!eof1) {
    auxf1 = (struct t2 *)malloc(
              sizeof(struct t2));
    Leer_datos(&(auxf1→data), &(auxf1→index),
               &eof2);
    auxf2 = auxf1;
    while (!eof2) {
      new = (struct t2 *)malloc(
              sizeof(struct t2));
      Leer_datos(&(new→data), &(new→index),
                 &eof2);
      new→prv = auxf1;
      auxf1→nxt = new;
      auxf1 = new;
    }
    newf = (struct t1 *)malloc(
             sizeof(struct t1));
    Leer_rowindex(&(newf→rowindex), &eof1);
    newf→row = auxf2;
    newf→nxt = auxf;
    if (auxf != NULL) {
      auxf→prv = newf;
    }
    auxf = newf;
  }
  M = auxf;
  /* creacion y llenado del vector v */
  v = (struct t2 *)malloc(
        sizeof(struct t2));
  Leer_datos(&(v→data), &(v→index),
             &eof2);
  auxf1 = v;
  while (!eof2) {
    new = (struct t2 *)malloc(
            sizeof(struct t2));
    Leer_datos(&(new→data), &(new→index),
               &eof2);
    new→prv = auxf1;
    auxf1→nxt = new;
    auxf1 = new;
  }
  auxf = M;
  /* multiplicacion y creacion del vector r */
  r = (struct t2 *)malloc(
        sizeof(struct t2));
  auxr = r;
  while (auxf != NULL) {
    prod = 0;
    auxf2 = auxf→row;
    auxv = v;
    while (auxf2 != NULL) {
      while ((auxv != NULL) &&
             (auxv→index < auxf2→index)) {
        auxv = auxv→nxt;
      }
      if ((auxv != NULL) &&
          (auxv→index == auxf2→index)) {
        prod += auxv→data * auxf2→data;
      }
      auxf2 = auxf2→nxt;
    }
    if (prod != 0) {
      new = (struct t2 *)malloc(
              sizeof(struct t2));
      new→data = prod;
      new→index = auxf→rowindex;
      auxr→nxt = new;
      new→prv = auxr;
      auxr = new;
    }
    auxf = auxf→nxt;
  }
  auxr = r;
  r = auxr→nxt;
  r→prv = NULL;
  free(auxr);
}
```

E.1.3 Código multiplicación matriz dispersa por vector analizado

```
/* Estructura Lista Cabecera Matriz */
typedef struct t1 {
  t1_data data;
  struct t1 *nxt, *prv;
  struct t2 *row;
}

/* Estructura Filas Matriz y Vectores */
typedef struct t2 {
  t2_data data;
```

```
  struct t2 *nxt, *prv;
}

void main() {
  struct t1 *M, *auxf, *tmpf, *newf;
  struct t2 *r, *v, *auxf1, *auxf2, *auxv, *tmp,
            *new, *auxr;

  auxf = NULL;
  while () {
    auxf1 = malloc;
    auxf2 = auxf1;
    while () {
      new = malloc;
      new→prv = auxf1;
      auxf1→nxt = new;
      auxf1 = new;
    }
    newf = malloc;
    newf→row = auxf2;
    auxf1 = NULL;
    auxf2 = NULL;
    new = NULL;
    newf→nxt = auxf;
    if () {
      FORCE auxf != NULL;
      auxf→prv = newf;
    }
    else {
      FORCE auxf == NULL;
    }
    auxf = newf;
  }
  newf = NULL;
  M = auxf;
  auxf = NULL;
  v = malloc;
  auxf1 = v;
  while () {
    new = malloc;
    new→prv = auxf1;
    auxf1→nxt = new;
    auxf1 = new;
  }
  new = NULL;
  auxf1 = NULL;
  auxf = M;
  r = malloc;
  auxr = r;
  while () {
    FORCE auxf != NULL;
    auxf2 = auxf→row;
    auxv = v;
    while () {
      FORCE auxf2 != NULL;
      while () {
        FORCE auxv != NULL;
        tmp = auxv→nxt;
        auxv = tmp;
        tmp = NULL;
      }
      if () {
        FORCE auxf2 != NULL;
        FORCE auxv != NULL;
      }
      else {
        NOP
      }
      tmp = auxf2→nxt;
      auxf2 = tmp;
      tmp = NULL;
    }
    FORCE auxf2 == NULL;
    if () {
      new = malloc;
      auxr→nxt = new;
      new→prv = auxr;
      auxr = new;
      new = NULL;
    }
    else {
      NOP
    }
    auxv = NULL;
    tmpf = auxf→nxt;
    auxf = tmpf;
    tmpf = NULL;
  }
  FORCE auxf == NULL;
  auxr = r;
  r = auxr→nxt;
  r→prv = NULL;
  auxr = NULL;
}
```

E.1.4 Código multiplicación matriz dispersa por matriz dispersa

```
/* Estructura Lista Cabecera Matriz */
typedef struct t1 {
  int rowindex;
  struct t1 *nxt, *prv;
  struct t2 *row;
}

/* Estructura Filas Matriz */
typedef struct t2 {
  double data;
  int index;
  struct t2 *nxt, *prv;
}

void main() {
  struct t1 *A, *B, *C, *auxf, *newf, *m,
            *auxfil , *auxfil2 ;
  struct t2 *r, *auxf1, *auxf2, *auxv, *v, *new,
            *auxr;
  char eof1 = 0; eof2 = 0;
  double prod;

  /* creacion y llenado de la matriz A */
  while (!eof1) {
    auxf1 = (struct t2 *)malloc(
              sizeof(struct t2));
    Leer_datos(&(auxf1→data), &(auxf1→index),
               &eof2);
    auxf2 = auxf1;
    while (!eof2) {
      new = (struct t2 *)malloc(
              sizeof(struct t2));
      Leer_datos(&(new→data), &(new→index),
                 &eof2);
      new→prv = auxf1;
      auxf1→nxt = new;
      auxf1 = new;
    }
```

```
    newf = (struct t1 *)malloc(
             sizeof(struct t1));
    Leer_rowindex(&(newf->rowindex), &eof1);
    newf->row = auxf2;
    newf->nxt = auxf;
    if (auxf != NULL) {
      auxf->prv = newf;
    }
    auxf = newf;
  }
  A = auxf;
  /* creacion y llenado de la matriz B */
  eof1 = 0;
  while (!eof1) {
    auxf1 = (struct t2 *)malloc(
             sizeof(struct t2));
    Leer_datos(&(auxf1->data), &(auxf1->index),
               &eof2);
    auxf2 = auxf1;
    while (!eof2) {
      new = (struct t2 *)malloc(
              sizeof(struct t2));
      Leer_datos(&(new->data), &(new->index),
                 &eof2);
      new->prv = auxf1;
      auxf1->nxt = new;
      auxf1 = new;
    }
    newf = (struct t1 *)malloc(
             sizeof(struct t1));
    Leer_rowindex(&(newf->rowindex), &eof1);
    newf->row = auxf2;
    newf->nxt = auxf;
    if (auxf != NULL) {
      auxf->prv = newf;
    }
    auxf = newf;
  }
  B = auxf;
  /* multiplicacion AxB y creacion matriz C */
  m = B;
  auxfil = A;
  while (auxfil != NULL) {
    prod = 0;
    v = auxfil->row;
    auxf = m;
    r = (struct t2 *)malloc(
          sizeof(struct t2));
    auxr = r;
    while (auxf != NULL) {
      auxf2 = auxf->row;
      auxv = v;
      while (auxf2 != NULL) {
        while ((auxv != NULL) &&
               (auxv->index < auxf2->index)) {
          auxv = auxv->nxt;
        }
        if ((auxv != NULL) &&
            (auxv->index == auxf2->index)) {
          prod += auxv->data * auxf2->data;
        }
        auxf2 = auxf2->nxt;
      }
      if (prod != 0) {
        new = (struct t2 *)malloc(
                sizeof(struct t2));
        new->data = prod;
        new->index = auxf->rowindex;
        auxr->nxt = new;
        new->prv = auxr;
        auxr = new;
      }
      auxf = auxf->nxt;
    }
    auxr = r;
    r = auxr->nxt;
    r->prv = NULL;
    free(auxr);
    /* insercion nueva fila en matriz C */
    if (r != NULL) {
      newf = (struct t1 *)malloc(
               sizeof(struct t1));
      newf->row = r;
      newf->rowindex = auxfil->rowindex;
      r = NULL;
      if (auxfil2 != NULL) {
        auxfil2->nxt = newf;
        newf->prv = auxfil2;
        auxfil2 = newf;
      }
      else {
        C = newf;
        auxfil2 = newf;
      }
    }
    auxfil = auxfil->nxt;
  }
}
```

E.1.5 Código multiplicación matriz dispersa por matriz dispersa analizado

```
/* Estructura Lista Cabecera Matriz */
typedef struct t1 {
  t1_data data;
  struct t1 *nxt, *prv;
  struct t2 *row;
}

/* Estructura Filas Matriz */
typedef struct t2 {
  t2_data data;
  struct t2 *nxt, *prv;
}

void main() {
  struct t1 *A, *B, *C, *auxf, *tmpf, *newf,
            *m, *auxfil, *auxfil2;
  struct t2 *r, *auxf1, *auxf2, *auxv, *v,
            *tmp, *new, *auxr;

  while () {
    auxf1 = malloc;
    auxf2 = auxf1;
    while () {
      new = malloc;
      new->prv = auxf1;
      auxf1->nxt = new;
      auxf1 = new;
    }
    newf = malloc;
    newf->row = auxf2;
    auxf1 = NULL;
    auxf2 = NULL;
    new = NULL;
    newf->nxt = auxf;
```

```
    if () {
      FORCE auxf != NULL;
      auxf→prv = newf;
    }
    else {
      FORCE auxf == NULL;
    }
    auxf = newf;
  }
  newf = NULL;
  A = auxf;
  auxf = NULL;
  while () {
    auxf1 = malloc;
    auxf2 = auxf1;
    while () {
      new = malloc;
      new→prv = auxf1;
      auxf1→nxt = new;
      auxf1 = new;
    }
    newf = malloc;
    newf→row = auxf2;
    auxf1 = NULL;
    auxf2 = NULL;
    new = NULL;
    newf→nxt = auxf;
    if () {
      FORCE auxf != NULL;
      auxf→prv = newf;
    }
    else {
      FORCE auxf == NULL;
    }
    auxf = newf;
  }
  newf = NULL;
  B = auxf;
  auxf = NULL;
  m = B;
  auxfil = A;
  while () {
    FORCE auxfil != NULL;
    v = auxfil→row;
    auxf = m;
    r = malloc;
    auxr = r;
    while () {
      FORCE auxf != NULL;
      auxf2 = auxf→row;
      auxv = v;
      while () {
        FORCE auxf2 != NULL;
        while () {
          FORCE auxv != NULL;
          tmp = auxv→nxt;
          auxv = tmp;
          tmp = NULL;
        }
        if () {
          FORCE auxf2 != NULL;
          FORCE auxv != NULL;
        }
        else {
          NOP
        }
        tmp = auxf2→nxt;
        auxf2 = tmp;
        tmp = NULL;
      }
      FORCE auxf2 == NULL;
      if () {
        new = malloc;
        auxr→nxt = new;
        new→prv = auxr;
        auxr = new;
        new = NULL;
      }
      else {
        NOP
      }
      auxv = NULL;
      tmpf = auxf→nxt;
      auxf = tmpf;
      tmpf = NULL;
    }
    FORCE auxf == NULL;
    v = NULL;
    auxr = r;
    r = auxr→nxt;
    r→prv = NULL;
    auxr = NULL;
    if () {
      FORCE r != NULL;
      newf = malloc;
      newf→row = r;
      r = NULL;
      if () {
        FORCE auxfil2 != NULL;
        auxfil2→ nxt = newf;
        newf→prv = auxfil2 ;
        auxfil2 = newf;
        newf = NULL;
      }
      else {
        FORCE auxfil2 == NULL;
        C = newf;
        auxfil2 = newf;
        newf = NULL;
      }
    }
    else {
      FORCE r == NULL;
    }
    tmpf = auxfil→nxt;
    auxfil = tmpf;
    tmpf = NULL;
  }
  FORCE auxfil == NULL;
  auxfil2 = NULL;
  m = NULL;
}
```

E.1.6 Código factorización LU de una matriz dispersa

```
/* Estructura Lista Cabecera Matriz */
typedef struct t1 {
  int colindex;
  struct t1 *nxt, *prv;
  struct t2 *col;
}

/* Estructura Columnas Matriz */
typedef struct t2 {
  double data;
```

```
  int index;
  struct t2 *nxt, *prv;
}

void main() {
  struct t1 *A, *cpiv, *auxc, *newc;
  struct t2 *auxf1, *auxf2, *auxf3, *pv, *tmp,
            *new;
  char eof1 = 0, eof2 = 0;
  double pivot, product;
  int pivind, ind;

  /* creacion y llenado de la matriz */
  while (!eof1) {
    auxf1 = (struct t2 *)malloc(
              sizeof(struct t2));
    Leer_datos(&(auxf1->data), &(auxf1->index),
               &eof2);
    auxf2 = auxf1;
    while (!eof2) {
      new = (struct t2 *)malloc(
              sizeof(struct t2));
      Leer_datos(&(new->data), &(new->index),
                 &eof2);
      new->prv = auxf1;
      auxf1->nxt = new;
      auxf1 = new;
    }
    newc = (struct t1 *)malloc(
             sizeof(struct t1));
    Leer_colindex(&(newc->colindex), &eof1);
    newc->col = auxf2;
    newc->nxt = auxc;
    if (auxc != NULL) {
      auxc->prv = newc;
    }
    auxc = newc;
  }
  A = auxc;
  cpiv = A;
  /* pivind = indice del pivote */
  pivind = 0;
  /* for (k) */
  while (cpiv != NULL) {
    /* se busca en la columna del pivot, un
       elemento con indice de fila igual a
       pivind, este es el pivot (pv) */
    pv = cpiv->col;
    while ((pv != NULL) &&
           (pv->index < pivind)) {
      pv = pv->nxt;
    }
    if ((pv == NULL) ||
        (pv->index != pivind))
      exit(1);
    auxc = cpiv->nxt;
    while (auxc != NULL) {
      /* en las columnas posteriores a la del
         pivot, se buscan elementos en la fila
         del pivot */
      auxf1 = auxc->col;
      while ((auxf1 != NULL) &&
             (auxf1->index < pv->index)) {
        auxf1 = auxf1->nxt;
      }
      if (auxf1 != NULL) {
        if ((auxf1->index == pivind) &&
            (BESTPIV(auxf1->data, pv->data))) {
          /* Si existe elemento en la fila del
             pivot y es un mejor pivot, se hace
             "pivoting" de columnas */
          tmp = cpiv->col;
          new = auxc->col;
          cpiv->col = new;
          auxc->col = tmp;
          tmp = pv;
          pv = auxf1;
          auxf1 = tmp;
        }
      }
      auxc = auxc->nxt;
    }
    pivot = 1/pv->data;
    /* se recorren los elementos de la columna
       del pivot por debajo de este,
       actualizando su valor */
    auxf2 = pv->nxt;
    while (auxf2 != NULL) {
      auxf2->data *= pivot;
      auxf2 = auxf2->nxt;
    }
    auxc = cpiv->nxt;
    /* for (j) */
    while (auxc != NULL) {
      /* se recorren los elementos en la fila
         del pivot a su derecha (columnas
         siguientes a la del pivot auxc)*/
      auxf1 = auxc->col;
      while ((auxf1 != NULL) &&
             (auxf1->index < pv->index)) {
        auxf1 = auxf1->nxt;
      }
      if ((auxf1 != NULL) &&
          (auxf1->index == pv->index)) {
        /* existe elemento en la misma fila del
           pivot */
        auxf2 = pv->nxt;
        /* for (i) */
        while (auxf2 != NULL) {
          /* se recorren los elementos de la
             columna del pivot por debajo de
             este */
          product = -(auxf1->data *
                      auxf2->data);
          ind = auxf2->index;
          auxf3 = auxf1;
          /* fill-in */
          while ((auxf3->nxt != NULL) &&
                 (auxf3->nxt->index < ind)) {
            auxf3 = auxf3->nxt;
          }
          if ((auxf3->nxt != NULL) &&
              (auxf3->nxt->ind == ind)) {
            /* si existe el elemento (i,j) se
               actualiza su valor */
            auxf3->nxt->valor += product;
          }
          else {
            /* si no existe elemento (i,j) se
               inserta */
            if (auxf3->nxt != NULL) {
              /* insertar en medio de la
                 columna */
              new = (struct t2 *)malloc(
                      sizeof(struct t2));
              new->index = ind;
              new->data = product;
              new->nxt = auxf3->nxt;
              new->prv = auxf3;
              new->nxt->prv = new;
              auxf3->nxt = new;
            }
            else {
              /* insertar al final de la
```

```
        columna */
      new = (struct t2 *)malloc(
            sizeof(struct t2));
      new→index = ind;
      new→data = product;
      auxf3→nxt = new;
      new→prv = auxf3;
    }
  }
```

```
      auxf2 = auxf2→nxt;
    }
  }
  auxc = auxc→nxt;
 }
 cpiv = cpiv→nxt;
 pivindex++;
 }
}
```

E.1.7 Código factorización LU de una matriz dispersa analizado

```
/* Estructura Lista Cabecera Matriz */
typedef struct t1 {
  t1_data data;
  struct t1 *nxt, *prv;
  struct t2 *col;
}

/* Estructura Columnas Matriz */
typedef struct t2 {
  t2_data data;
  struct t2 *nxt, *prv;
}

void main() {
  struct t1 *A, *cpiv, *auxc, *tmpc, *newc;
  struct t2 *auxf1, *auxf2, *auxf3, *pv, *tmp,
            *new;

  while () {
    auxf1 = malloc;
    auxf2 = auxf1;
    while () {
      new = malloc;
      new→prv = auxf1;
      auxf1→nxt = new;
      auxf1 = new;
    }
    newc = malloc;
    newc→col = auxf2;
    auxf1 = NULL;
    auxf2 = NULL;
    new = NULL;
    newc→nxt = auxc;
    if () {
      FORCE auxc != NULL;
      auxc→prv = newc;
    }
    else {
      FORCE auxc == NULL;
    }
    auxc = newc;
  }
  newc = NULL;
  A = auxc;
  auxc = NULL;
  cpiv = A;
  while () {
    FORCE cpiv != NULL;
    pv = cpiv→col;
    while () {
      FORCE pv != NULL;
      tmp = pv→nxt;
      pv = tmp;
    }
    tmp = NULL;
    FORCE pv != NULL;
    auxc = cpiv→nxt;
```

```
  while () {
    FORCE auxc != NULL;
    auxf1 = auxc→col;
    while () {
      FORCE auxf1 != NULL;
      tmp = auxf1→nxt;
      auxf1 = tmp;
    }
    tmp = NULL;
    if () {
      FORCE auxf1 != NULL;
      if () {
        tmp = cpiv→col;
        new = auxc→col;
        cpiv→col = new;
        auxc→col = tmp;
        new = NULL;
        tmp = pv;
        pv = auxf1;
        auxf1 = tmp;
        tmp = NULL;
      }
      else {
        NOP
      }
    }
    tmp = auxc;
    auxc = tmp→nxt;
    tmp = NULL;
  }
  auxf2 = pv→nxt;
  while () {
    FORCE auxf2 != NULL;
    tmp = auxf2→nxt;
    auxf2 = tmp;
  }
  FORCE auxf2 == NULL;
  tmp = NULL;
  auxc = cpiv→nxt;
  while () {
    FORCE auxc != NULL;
    auxf1 = auxc→col;
    while () {
      FORCE auxf1 != NULL;
      tmp = auxf1→nxt;
      auxf1 = tmp;
    }
    tmp = NULL;
    if () {
      FORCE auxf1 != NULL;
      auxf2 = pv→nxt;
      while () {
        FORCE auxf2 != NULL;
        auxf3 = auxf1;
        while () {
          tmp = auxf3→nxt;
          auxf3 = tmp;
```

```
        }
        FORCE auxf3 != NULL;
        if () {
          NOP
        }
        else {
          if () {
            new = malloc;
            tmp = auxf3→nxt;
            auxf3→nxt = new;
            new→nxt = tmp;
            tmp→prv = new;
            new→prv = auxf3;
            tmp = NULL;
            auxf3 = NULL;
            new = NULL;
          }
          else {
            FORCE auxf3→nxt == NULL;
            new = malloc;
            auxf3→nxt = new;
            new→prv = auxf3;
            auxf3 = NULL;
            new = NULL;
          }
          }
          tmp = auxf2→nxt;
          auxf2 = tmp;
          tmp = NULL;
        }
        FORCE auxf2 == NULL;
      }
      else {
        NOP
      }
      auxf1 = NULL;
      auxf2 = NULL;
      auxf3 = NULL;
      tmpc = auxc→nxt;
      auxc = tmpc;
      tmpc = NULL;
    }
    FORCE auxc == NULL;
    tmpc = cpiv→nxt;
    cpiv = tmpc;
    tmpc = NULL;
    pv = NULL;
  }
  FORCE cpiv == NULL;
}
```

E.1.8 Código simulación Barnes-Hut

```
typedef double vector [3];

/* Estructura Lista de Cuerpos */
typedef struct t1 {
  double mass;
  vector pos;
  struct t1 *nxt;
}

/* Estructura Octree */
typedef struct t2 {
  double mass;
  vector pos;
  int level;
  int index;
  struct t2 *child, *nxt;
  struct t1 *body;
}

void Make_tree(Root, Lbodies)
struct t2 **Root;
struct t1 *Lbodies;
{
  struct t1 *mt_aux, *lb_p, *lb_auxb;
  struct t2 *mt_list, *mt_list2, *mt_new,
            *lb_q, *lb_aux, *lb_aux2,
            *lb_c;
  int count, kidindex;

  /* se crea e inicializa nodo raiz del arbol */
  Root = (struct t2 *)malloc(
         sizeof(struct t2));
  Init_root(Root);
  /* se crean sus ocho hijos, inicializando sus
     datos con respecto a la posicion que ocupan
     como hijos de Root */
  mt_list = (struct t2 *)malloc(
            sizeof(struct t2));
  mt_list→index = 0;
  mt_list2 = mt_list;
  count = 1;
  while (count < 8) {
    mt_new = (struct t2 *)malloc(
             sizeof(struct t2));
    mt_new→index = count;
    mt_new→level = Root→level+1;
    mt_list2→nxt = mt_new;
    mt_list2 = mt_new;
    count ++;
  }
  Root→child = mt_list;
  mt_aux = Lbodies;
  /* se recorren todos los cuerpos para ser
     insertados en el arbol */
  while (mt_aux != NULL) {
    lb_p = mt_aux;
    lb_q = Root;
    while (lb_q != NULL) {
      lb_aux = lb_q→child;
      /* calcular el indice para el cuerpo
         apuntado por lb_p dentro del nodo
         lb_q */
      kidindex = SUBINDEX(lb_q, lb_p);
      if (lb_aux != NULL) {
        /* el nodo actual de arbol, lb_aux,
           tiene hijos */
        while (lb_aux→index < kidindex) {
          lb_aux = lb_aux→nxt;
        }
        lb_q = lb_aux;
        /* el cuerpo tiene que ser insertado en
           el subarbol apuntado por lb_q */
      }
      else {
        /* el nodo actual del arbol, lb_q, no
           tiene hijos */
        if (lb_q→body == NULL) {
          /* el nodo actual del arbol no apunta
             a ningun cuerpo. Se inserta el
             cuerpo apuntado por lb_p en dicho
             nodo */
          lb_q→body = lb_p;
```

```
            lb_q = NULL;
          }
          else {
            /* el nodo actual del arbol apunta a
               un cuerpo. Se tiene que subdividir
               el nodo e insertar en estas
               subdivisiones el cuerpo apuntado
               por lb_q→body y el nuevo
               apuntado por lb_p */
            lb_auxb = lb_q→body;
            lb_q→body = NULL;
            lb_c = (struct t2 *)malloc(
                    sizeof(struct t2));
            lb_c→index = 0;
            lb_c→level = lb_q→level+1;
            count = 1;
            lb_aux = lb_c;
            while (count < 8) {
              lb_aux2 = (struct t2 *)malloc(
                        sizeof(struct t2));
              lb_aux2→index = count;
              lb_aux2→level = lb_q→level+1;
              lb_aux→nxt = lb_aux2;
              lb_aux = lb_aux2;
              count ++;
            }
            lb_q→child = lb_c;
            kidindex = SUBINDEX(lb_q, lb_auxb);
            while (lb_c→index < kidindex) {
              lb_c = lb_c→nxt;
            }
            lb_c→body = lb_auxb;
          }
        }
      }
      /* siguiente cuerpo */
      mt_aux = lb_p→nxt;
    }
  }

  void Hack_cofm(node)
  struct t2 *node;
  {
    struct t2 *hc_aux;
    vector tmpv;

    if (node→body == NULL) {
      /* si el nodo no apunta a un cuerpo, se
         inicializa su masa y posicion */
      node→mass = 0;
      CLRV(node→pos);
      hc_aux = node→child;
      while (hc_aux != NULL) {
        /* se recorren todos sus hijos calculando
           sus masas y posiciones con una llamada
           recursiva */
        Hack_cofm(hc_aux);
        /* en base a la masa y posicion del hijo
           actual, se van acumulando masa y
           posicion del padre */
        node→mass += hc_aux→mass;
        MULVS(tmpv, hc_aux→pos, hc_aux→mass);
        ADDV(node→pos, node→pos, tmpv);
      }
      DIVVS(node→pos, node→pos, node→mass);
    }
    else {
      /* el nodo apunta a un cuerpo, por tanto
         no hay que calcular masa y posicion */
      node→mass = node→body→mass;
      CPYV(node→pos, node→body→pos);
    }
  }

  void Hack_grav(node, body)
  struct t2 *node;
  struct t1 *body;
  {
    struct t2 *hc_aux;

    /* hay que determinar si el cuerpo esta lo
       suficientemente alejado del nodo
       representado por node como para usar su
       masa total y centro de masas para calcular
       la fuerza sobre body */
    if (SUBDIVP(node, body)) {
      /* no esta suficientemente lejos, hay que
         subdividir */
      if (node→body == NULL) {
        /* el nodo no apunta directamente a un
           cuerpo, por tanto se avanza
           recursivamente sobre sus ocho hijos */
        hc_aux = node→child;
        while (hc_aux != NULL) {
          Hack_grav(hc_aux, body);
        }
      }
      else {
        /* el nodo apunta a un cuerpo, por tanto
           se utilizan los datos de este cuerpo
           para calcular la fuerza y modificar
           posicion del cuerpo apuntado por body
           */
        GRAV_SUB(body, node);
      }
    }
    else {
      /* el nodo esta suficientemente lejos, se
         toman su masa total y centro de masas */
      GRAV_SUB(body, node);
    }
  }

  void main() {
    struct t1 *Lbodies, *auxb, *sr_aux, *sr_n;
    struct t2 *Root;
    char eof;


    Lbodies = (struct t1 *)malloc(
              sizeof(struct t1));
    sr_aux = Lbodies;
    Leer_datos(&(sr_aux→mass), &(sr_b→pos),
               &eof);
    while (!eof) {
      sr_n = (struct t1 *)malloc(
             sizeof(struct t1));
      Leer_datos(&(sr_n→mass), &(sr_n→pos),
                 &eof);
      sr_aux→nxt = sr_n;
      sr_aux = sr_n;
    }

    /* construir el octree apuntado por Root
       insertando los cuerpos almacenados en
       la lista Lbodies */
    Make_tree(&Root, Lbodies);

    /* calcular el centro de masas y masa total
       para los nodos del arbol */
    Hack_cofm(Root);

    /* para cada cuerpo se recorre el arbol para
       calcular las fuerzas sobre el cuerpo */
```

```
  auxb = Lbodies;
  while (auxb != NULL) {
    Hack_grav(Root, auxb);
    auxb = auxb→nxt;
  }
}
```

E.1.9 Código simulación Barnes-Hut analizado

```
/* Estructura Lista de Cuerpos */
typedef struct t1 {
  t1_data data;
  struct t1 *nxt;
}

/* Estructura Octree */
typedef struct t2 {
  t2_data data;
  struct t2 *child, *nxt;
  struct t1 *body;
}

/* Estructura Pila */
typedef struct t3 {
  t3_data data;
  struct t3 *nxt;
  struct t2 *node;
}

void main() {
  struct t1 *Lbodies, *auxb, *sr_aux,
            *sr_n, *mt_aux, *lb_p, *lb_auxb,
            *hg_p;
  struct t2 *Root, *lb_q, *lb_aux, *lb_c,
            *hc_p, *hc_q, *hc_aux, *ts_q,
            *ts_aux1, *ts_aux2, *lb_aux2,
            *hc_aux2, *mt_list, *mt_list2,
            *mt_new;
  struct t3 *Stack, *hc_end, *hc_new;


/* creacion de la lista de cuerpos */
  Lbodies = malloc;
  sr_aux = Lbodies;
  while () {
    sr_n = malloc;
    sr_aux→nxt = sr_n;
    sr_aux = sr_n;
  }
  sr_aux = NULL;
  sr_n = NULL;
/* creacion del arbol insertando cada cuerpo
   Make_tree() */
  Root = malloc;
  mt_list = malloc;
  mt_list2 = mt_list;
  while () {
    mt_new = malloc;
    mt_list2→nxt = mt_new;
    mt_list2 = mt_new;
  }
  Root→child = mt_list;
  mt_list = NULL;
  mt_list2 = NULL;
  mt_new = NULL;
  mt_aux = Lbodies;
  while () {
    FORCE mt_aux != NULL;
    lb_p = mt_aux;
    mt_aux = NULL;
    lb_q = Root;
    while () {
      FORCE lb_q != NULL;
      lb_aux = lb_q→child;
      if () {
        FORCE lb_aux != NULL;
        while () {
          lb_aux2 = lb_aux→nxt;
          lb_aux = lb_aux2;
          lb_aux2 = NULL;
        }
        FORCE lb_aux != NULL;
        lb_q = lb_aux;
        lb_aux = NULL;
      }
      else {
        FORCE lb_aux == NULL;
        if () {
          FORCE lb_q→body == NULL;
          lb_q→body = lb_p;
          lb_q = NULL;
        }
        else {
          lb_auxb = lb_q→body;
          lb_q→body = NULL;
          lb_c = malloc;
          lb_aux = lb_c;
          while () {
            lb_aux2 = malloc;
            lb_aux→nxt = lb_aux2;
            lb_aux = lb_aux2;
          }
          lb_aux = NULL;
          lb_aux2 = NULL;
          lb_q→child = lb_c;
          while () {
            lb_aux = lb_c→nxt;
            lb_c = lb_aux;
          }
          lb_aux = NULL;
          FORCE lb_c != NULL;
          lb_c→body = lb_auxb;
          lb_c = NULL;
          lb_auxb = NULL;
        }
      }
      NOP
    }
    FORCE lb_q == NULL;
    mt_aux = lb_p→nxt;
  }
  FORCE mt_aux == NULL;
  lb_p = NULL;
/* recorrido del arbol para el calculo del centro
   de masas y masa total utilizando la pila.
   Hack_cofm() */
  hc_p = Root;
  Stack= malloc;
  Stack→node = hc_p;
  while () {
    FORCE Stack != NULL;
    hc_q = Stack→node;
    auxb = hc_q→body;
    if () {
      /* el nodo apunta directamente
```

```
        a un cuerpo. Se saca nodo
        de la pila */
      FORCE auxb != NULL;
      Stack→node = NULL;
      hc_new = Stack→nxt;
      Stack = hc_new;
      hc_new = NULL;
    }
    else {
      FORCE auxb == NULL;
      if () {
        /* los hijos del nodo no
           han sido visitados, se
           introducen en la pila */
        hc_aux = hc_q→child;
        while () {
          FORCE hc_aux != NULL;
          hc_new = malloc;
          hc_new→node = hc_aux;
          hc_new→nxt = Stack;
          Stack = hc_new;
          hc_new = NULL;
          hc_q = hc_aux→nxt;
          hc_aux = hc_q;
          hc_q = NULL;
        }
        FORCE hc_aux == NULL;
        hc_q = NULL;
      }
      else {
        /* todos los hijos del nodo
           han sido visitados, se
           saca nodo de la pila */
        hc_aux = hc_q→child;
        while () {
          FORCE hc_aux != NULL;
          hc_q = hc_aux→nxt;
          hc_aux = hc_q;
          hc_q = NULL;
        }
        FORCE hc_aux == NULL;
        hc_q = NULL;
        Stack→node = NULL;
        hc_new = Stack→nxt;
        Stack = hc_new;
        hc_new = NULL;
      }
    }
  }
  FORCE Stack == NULL;
  hc_p = NULL;
/* para cada cuerpo, recorrer el arbol para
   calcular fuerzas, utilizando la pila.
   Hack_grav() */
  auxb = Lbodies;
  while () {
    FORCE auxb != NULL;
    hg_p = auxb;
    auxb = NULL;
    hc_p = Root;
    Stack = malloc;
    hc_end = Stack;
    hc_aux = hc_p→child;
    Stack→node = hc_aux;
    hc_aux = NULL;
    while () {
      FORCE Stack != NULL;
      /* sacar elemento de la pila */
      hc_q = Stack→node;
      Stack→node = NULL;
      /* recorrer sus hijos */
      while () {
        FORCE hc_q != NULL;
        hc_aux = hc_q→child;
        if () {
          /* no estan suficientemente lejos,
             se introducen en pila */
          FORCE hc_aux != NULL;
          hc_new = malloc;
          hc_new→node = hc_aux;
          hc_end→nxt = hc_new;
          hc_end = hc_new;
          hc_new = NULL;
        }
        else {
          /* estan lejos o apunta a un
             cuerpo */
          NOP
        }
        hc_aux = hc_q→nxt;
        hc_q = hc_aux;
        hc_aux = NULL;
      }
      FORCE hc_q == NULL;
      /* siguiente elemento de la pila */
      hc_new = Stack→nxt;
      Stack = hc_new;
      hc_new = NULL;
    }
    FORCE Stack == NULL;
    hc_end = NULL;
    hc_q = NULL;
    hc_p = NULL;
    auxb = hg_p→nxt;
  }
  FORCE auxb == NULL;
  hg_p = NULL;
}
```

E.2 Códigos con arrays de punteros

En esta sección presentamos los códigos basados en estructuras con arrays de punteros cuyos resultados han sido presentados en el capítulo 4.

E.2.1 Código multiplicación matriz dispersa por vector

```
/* Estructura Cabecera Matriz */
typedef struct t1 {
  struct t2 *row[n];
}

/* Estructura Filas Matriz y Vectores */
```

```
typedef struct t2 {
  double data;
  int index;
  struct t2 *nxt, *prv;
}

void main() {
  struct t1 *M;
  struct t2 *v, *r, *auxf1, *auxf2, *newf,
            *auxv, *auxr;
  char eof1 = 0, eof2 = 0;
  double prod;
  int I;

  /* creacion y llenado de la matriz M */
  M = (struct t1 *)malloc(
       sizeof(struct t1));
  while (!eof1) {
    auxf1 = (struct t2 *)malloc(
              sizeof(struct t2));
    Leer_datos(&(auxf1->data), &(auxf1->index),
               &eof2);
    auxf2 = auxf1;
    while (!eof2) {
      newf = (struct t2 *)malloc(
               sizeof(struct t2));
      Leer_datos(&(newf->data), &(newf->index),
                 &eof2);
      newf->prv = auxf1;
      auxf1->nxt = newf;
      auxf1 = newf;
    }
    Leer_rowindex(&I, &eof1);
    M->row[I] = auxf2;
  }

  /* creacion y llenado del vector v */
  v = (struct t2 *)malloc(
       sizeof(struct t2));
  Leer_datos(&(v->data), &(v->index),
             &eof2);
  auxf1 = v;
  while (!eof2) {
    newf = (struct t2 *)malloc(
             sizeof(struct t2));
    Leer_datos(&(newf->data), &(newf->index),
               &eof2);
    newf->prv = auxf1;
    auxf1->nxt = newf;
    auxf1 = newf;
  }

  /* multiplicacion y creacion del vector r */
  r = (struct t2 *)malloc(
       sizeof(struct t2));
  auxr = r;
  for (I=0; I<n; I++) {
    prod = 0;
    auxf1 = M->row[I];
    auxv = v;
    while (auxf1 != NULL) {
      while ((auxv != NULL) &&
             (auxv->index < auxf1->index) {
        auxv = auxv->nxt;
      }
      if ((auxv != NULL) &&
          (auxv->index == auxf1->index)) {
        prod += auxv->data * auxf1->data;
      }
      auxf1 = auxf1->nxt;
    }
    if (prod != 0) {
      newf = (struct t2 *)malloc(
               sizeof(struct t2));
      newf->data = prod;
      newf->index = I;
      auxr->nxt = newf;
      newf->prv = auxr;
      auxr = newf;
    }
  }
  auxr = r;
  r = auxr->nxt;
  r->prv = NULL;
  free(auxr);
}
```

E.2.2 Código multiplicación matriz dispersa por vector analizado

```
/* Estructura Cabecera Matriz */
typedef struct t1 {
  struct t2 *row[n];
}

/* Estructura Filas Matriz y Vectores */
typedef struct t2 {
  t2_data data;
  struct t2 *nxt, *prv;
}

void main() {
  struct t1 *M;
  struct t2 *v, *r, *auxf1, *auxf2, *newf,
            *auxv, *auxr, *tmp;

  M = malloc;
  while () {
    auxf1 = malloc;
    auxf2 = auxf1;
    while () {
      newf = malloc;
      newf->prv = auxf1;
      auxf1->nxt = newf;
      auxf1 = newf;
    }
    newf = NULL;
    auxf1 = NULL;
    DeleteIndex(I);
    M->row[I] = auxf2;
  }
  auxf2 = NULL;
  v = malloc;
  auxf1 = v;
  while () {
    newf = malloc;
    newf->prv = auxf1;
    auxf1->nxt = newf;
    auxf1 = newf;
  }
  newf = NULL;
  auxf1 = NULL;
  r = malloc;
  auxr = r;
  for () {
    DeleteIndex(I);
```

```
    auxf1 = M→row[I];
    auxv = v;
    while () {
      FORCE auxf1 != NULL;
      while () {
        FORCE auxv != NULL;
        tmp = auxv→nxt;
        auxv = tmp;
      }
      tmp = NULL;
      if () {
        FORCE auxf1 != NULL;
        FORCE auxv != NULL;
      }
      else {
        NOP
      }
      tmp = auxf1→nxt;
      auxf1 = tmp;
      tmp = NULL;
    }
    FORCE auxf1 == NULL;
    auxv = NULL;
    if () {
      newf = malloc;
      auxr→nxt = newf;
      newf→prv = auxr;
      auxr = newf;
      newf = NULL;
    }
    else {
      NOP
    }
  }
  DeleteIndex(I);
  DeleteIndex(va(I));
  auxr = r;
  r = auxr→nxt;
  r→prv = NULL;
  auxr = NULL;
}
```

E.2.3 Código multiplicación matriz dispersa por matriz dispersa

```
/* Estructura Cabecera Matriz */
typedef struct t1 {
  struct t2 *row[n];
}

/* Estructura Filas Matriz */
typedef struct t2 {
  double data;
  int index;
  struct t2 *nxt, *prv;
}

void main() {
  struct t1 *A, *B, *C;
  struct t2 *auxa, *auxb, *new, *r, *auxr;
  int I, J;
  double prod;
  char eof1 = 0, eof2 = 0;

 /* creacion y llenado de la matriz A */
  A = (struct t1 *)malloc(
        sizeof(struct t1));
  while (!eof1) {
    auxa = (struct t2 *)malloc(
             sizeof(struct t2));
    Leer_datos(&(auxa→data), &(auxa→index),
               &eof2);
    auxb = auxa;
    while (!eof2) {
      new = (struct t2 *)malloc(
              sizeof(struct t2));
      Leer_datos(&(new→data), &(new→index),
                 &eof2);
      new→prv = auxb;
      auxb→nxt = new;
      auxb = new;
    }
    Leer_rowindex(&I, &eof1);
    A→row[I] = auxa;
  }
  eof1 = 0;

 /* creacion y llenado de la matriz B */
  B = (struct t1 *)malloc(
        sizeof(struct t1));
  while (!eof1) {
    auxa = (struct t2 *)malloc(
             sizeof(struct t2));
    Leer_datos(&(auxa→data), &(auxa→index),
               &eof2);
    auxb = auxa;
    while (!eof2) {
      new = (struct t2 *)malloc(
              sizeof(struct t2));
      Leer_datos(&(new→data), &(new→index),
                 &eof2);
      new→prv = auxb;
      auxb→nxt = new;
      auxb = new;
    }
    Leer_rowindex(&I, &eof1);
    B→row[I] = auxa;
  }

 /* multiplicacion y creacion de la matriz C */
  auxr = NULL;
  C = (struct t1 *)malloc(
        sizeof(struct t1));
  for (I=0; I<n; I++) {
    for (J=0; J<n; J++) {
      prod = 0;
      auxa = A→row[I];
      auxb = B→row[J];
      while (auxa != NULL) {
        while ((auxb != NULL) &&
               (auxb→index < auxa→index)) {
          auxb = auxb→nxt;
        }
        if ((auxb != NULL) &&
            (auxb→index = auxa→index)) {
          prod += auxa→data * auxb→data;
        }
        auxa = auxa→nxt;
      }
      if (prod != 0) {
        if (auxr != NULL) {
          new = (struct t2 *)malloc(
                  sizeof(struct t2));
          new→data = prod;
          new→index = J;
```

```
          new→prv = auxr;
          auxr→nxt = new;
          auxr = new;
        }
        else {
          r = (struct t2 *)malloc(
                sizeof(struct t2));
          r→data = prod;
          r→index = J;
          auxr = r;
        }
      }
    }
    /* inserccion de una nueva
       fila en matriz C */
    C→row[I] = r;
    r = NULL;
    auxr = NULL;
  }
}
```

E.2.4 Código multiplicación matriz dispersa por matriz dispersa analizado

```
/* Estructura Cabecera Matriz */
typedef struct t1 {
  struct t2 *row[n];
}

/* Estructura Filas Matriz */
typedef struct t2 {
  t2_data data;
  struct t2 *nxt, *prv;
}

void main() {
  struct t1 *A, *B, *C;
  struct t2 *auxa, *auxb, *new, *tmp, *r, *auxr;

  A = malloc;
  while () {
    auxa = malloc;
    auxb = auxa;
    while () {
      new = malloc;
      new→prv = auxb;
      auxb→nxt = new;
      auxb = new;
    }
    new = NULL;
    auxb = NULL;
    DeleteIndex(I);
    A→row[I] = auxa;
  }
  auxa = NULL;
  B = malloc;
  while () {
    auxa = malloc;
    auxb = auxa;
    while () {
      new = malloc;
      new→prv = auxb;
      auxb→nxt = new;
      auxb = new;
    }
    new = NULL;
    auxb = NULL;
    DeleteIndex(I);
    B→row[I] = auxa;
  }
  auxa = NULL;
  C = malloc;
  for () {
    DeleteIndex(I);
    for () {
      DeleteIndex(J);
      auxa = A→row[I];
      auxb = B→row[J];
      while () {
        FORCE auxa != NULL;
        while () {
          FORCE auxb != NULL;
          tmp = auxb→nxt;
          auxb = tmp;
        }
        tmp = NULL;
        if () {
          FORCE auxb != NULL;
          FORCE auxa != NULL;
        }
        else {
          NOP
        }
        tmp = auxa→nxt;
        auxa = tmp;
        tmp = NULL;
      }
      FORCE auxa == NULL;
      auxb = NULL;
      if () {
        if () {
          FORCE auxr != NULL;
          new = malloc;
          new→prv = auxr;
          auxr→nxt = new;
          auxr = new;
          new = NULL;
        }
        else {
          FORCE auxr == NULL;
          r = malloc;
          auxr = r;
        }
      }
      else {
        NOP
      }
    }
    DeleteIndex(J);
    DeleteIndex(va(J));
    C→row[I] = r;
    r = NULL;
    auxr = NULL;
  }
  DeleteIndex(I);
  DeleteIndex(va(I));
}
```

E.2.5 Código factorización LU de una matriz dispersa

```
/* Estructura Cabecera Matriz */
typedef struct t1 {
  struct t2 *col[n];
}

/* Estructura Columnas Matriz */
typedef struct t2 {
  double data;
  int index;
  struct t2 *nxt, *prv;
}

void main() {
  struct t1 *A;
  struct t2 *auxa, *auxb, *new, *pv, *tmp1,
            *tmp2, *auxa2, *auxa3;
  char eof1 = 0, eof2 = 0;
  int I, J, K, ind;
  double pivot, product;


  /* creacion y llenado de la matriz */
  A = (struct t1 *)malloc(
       sizeof(struct t1));
  while (!eof1) {
    auxa = (struct t2 *)malloc(
             sizeof(struct t2));
    Leer_datos(&(auxa->data), &(auxa->index),
               &eof2);
    auxb = auxa;
    while (!eof2) {
      new = (struct t2 *)malloc(
              sizeof(struct t2));
      Leer_datos(&(new->data), &(new->index),
                 &eof2);
      new->prv = auxb;
      auxb->nxt = new;
      auxb = new;
    }
    Leer_colindex(&I, &eof1);
    A->col[I] = auxa;
  }
  for (K=0; K<n; K++) {
    /* se busca en la columna del pivot, un
       elemento con indice de fila igual a K,
       este es el pivot (pv) */
    pv = A->col[K];
    while ((pv != NULL) &&
           (pv->index < K)) {
      pv = pv->nxt;
    }
    if ((pv == NULL) ||
        (pv->index != K))
      exit(1);
    for (J=K+1; J<n; J++) {
      /* en las columnas posteriores a la del
         pivot, se buscan elementos en la fila
         del pivot */
      auxa3 = A->col[J];
      while ((auxa3 != NULL) &&
             (auxa3->index < K)) {
        auxa3 = auxa3->nxt;
      }
      if (auxa3 != NULL) {
        if ((auxa3->index == K) &&
            (BESTPIV(auxa3->data, pv->data)))
        {
          /* Si existe elemento en la fila del
             pivot y es un mejor pivot, se hace
             " pivoting " de columnas */
          tmp1 = A->col[K];
          tmp2 = A->col[J];
          A->col[K] = tmp2;
          A->col[J] = tmp1;
          pv = auxa3;
        }
      }
    }
    pivot = 1/pv->data;
    /* se recorren los elementos de la columna
       del pivot por debajo de este,
       actualizando su valor */
    auxa = pv->nxt;
    while (auxa != NULL) {
      auxa->data *= pivot;
      auxa = auxa->nxt;
    }
    for (J=K+1; J<n; J++) {
      /* se recorren los elementos en la fila
         del pivot a su derecha (columnas
         siguientes a la del pivot auxc)*/
      auxa2 = A->col[J];
      while ((auxa2 != NULL) &&
             (auxa2->index < K)) {
        auxa2 = auxa2->nxt;
      }
      if ((auxa2 != NULL) &&
          (auxa2->index == K)) {
        /* existe elemento en la misma fila del
           pivot */
        auxb = pv->nxt;
        /* for (i) */
        while (auxb != NULL) {
          /* se recorren los elementos de la
             columna del pivot por debajo de
             este */
          product = -(auxa2->data *
                      auxb->data);
          ind = auxb->index;
          tmp1 = auxa2;
          while ((tmp1->nxt != NULL) &&
                 (tmp1->nxt->index < ind)) {
            tmp1 = tmp1->nxt;
          }
          if ((tmp1->nxt != NULL) &&
              (tmp1->nxt->ind == ind)) {
            /* si existe el elemento (i,j) se
               actualiza su valor */
            tmp1->nxt->valor += product;
          }
          else {
            /* si no existe elemento (i,j) se
               inserta */
            tmp2 = tmp1->nxt;
            if (tmp2 != NULL) {
              /* insertar en medio de la
                 columna */
              new = (struct t2 *)malloc(
                      sizeof(struct t2));
              new->index = ind;
              new->data = product;
              tmp1->nxt = new;
              new->nxt = tmp2;
              tmp2->prv = new;
              new->prv = tmp1;
            }
            else {
              /* insertar al final de la
```

```
        columna */
      new = (struct t2 *)malloc(
            sizeof(struct t2));
      new→index = ind;
      new→data = product;
      tmp1→nxt = new;
      new→prv = tmp1;
    }
```

```
      }
      auxb = auxb→nxt;
    }
  }
 }
}
```

E.2.6 Código factorización LU de una matriz dispersa analizado

```
/* Estructura Cabecera Matriz */
typedef struct t1 {
  struct t2 *col[n];
}

/* Estructura Columnas Matriz */
typedef struct t2 {
  t2_data data;
  struct t2 *nxt, *prv;
}

void main() {
  struct t1 *A;
  struct t2 *auxa, *auxb, *new, *pv, *tmp1,
            *tmp2, *auxa2, *auxa3;

  A = malloc;
  while () {
    auxa = malloc;
    auxb = auxa;
    while () {
      new = malloc;
      new→prv = auxb;
      auxb→nxt = new;
      auxb = new;
    }
    new = NULL;
    auxb = NULL;
    DeleteIndex(I);
    A→col[I] = auxa;
    auxa = NULL;
  }
  for () {
    DeleteIndex(K);
    pv = A→col[K];
    while () {
      FORCE pv != NULL;
      auxa = pv→nxt;
      pv = auxa;
    }
    auxa = NULL;
    FORCE pv != NULL;
    for () {
      DeleteIndex(J);
      auxa3 = A→col[J];
      while () {
        FORCE auxa3 != NULL;
        auxb = auxa3→nxt;
        auxa3 = auxb;
      }
      auxb = NULL;
      if () {
        if () {
          FORCE auxa3 != NULL;
          tmp1 = A→col[K];
          tmp2 = A→col[J];
          A→col[K] = tmp2;
          A→col[J] = tmp1;
```

```
          tmp1 = NULL;
          tmp2 = NULL;
          pv = auxa3;
        }
        else {
          NOP
        }
      }
      else {
        NOP
      }
      auxa3 = NULL;
    }
    DeleteIndex(J);
    DeleteIndex(va(J));
    auxa = pv→nxt;
    while () {
      FORCE auxa != NULL;
      auxb = auxa→nxt;
      auxa = auxb;
    }
    FORCE auxa == NULL;
    auxb = NULL;
    for () {
      DeleteIndex(J);
      auxa2 = A→col[J];
      while () {
        FORCE auxa2 != NULL;
        tmp1 = auxa2→nxt;
        auxa2 = tmp1;
        tmp1 = NULL;
      }
      if () {
      {
        FORCE auxa2 != NULL;
        auxb = pv→nxt;
        while () {
          FORCE auxb != NULL;
          tmp1 = auxa2;
          while () {
            FORCE tmp1 != NULL;
            tmp2 = tmp1→nxt;
            tmp1 = tmp2;
          }
          tmp2 = NULL;
          FORCE tmp1 != NULL;
          if () {
            NOP
          }
          else {
            tmp2 = tmp1→nxt;
            if () {
              FORCE tmp2 != NULL;
              new = malloc;
              tmp1→nxt = new;
              new→nxt = tmp2;
              tmp2→prv = new;
              new→prv = tmp1;
```

```
        }
        else {
          FORCE tmp2 == NULL;
          new = malloc;
          tmp1→nxt = new;
          new→prv = tmp1;
        }
        tmp2 = NULL;
        tmp1 = NULL;
        new = NULL;
      }
      tmp1 = auxb→nxt;
      auxb = tmp1;
      tmp1 = NULL;
    }
    FORCE auxb == NULL;
    auxa2 = NULL;
    auxb = NULL;
  }
 }
 DeleteIndex(J);
 DeleteIndex(va(J));
 pv = NULL;
}
DeleteIndex(K);
DeleteIndex(va(K));
}
```

E.2.7 Código simulación Barnes-Hut

```
typedef double vector [3];

/* Estructura Lista de Cuerpos */
typedef struct t1 {
  double mass;
  vector pos;
  struct t1 *nxt;
}

/* Estructura Octree */
typedef struct t2 {
  double mass;
  vector pos;
  int level;
  struct t2 *child [8];
  struct t1 *body;
}

void Make_tree(Root, Lbodies)
struct t2 **Root;
struct t1 *Lbodies;
{
  struct t1 *auxb, *auxb2;
  struct t2 *auxt, *auxt2, *newt;
  int I;

  /* se crea e inicializa nodo raiz del arbol */
  Root = (struct t2 *)malloc(
          sizeof(struct t2));
  Init_root(Root);
  auxb = Lbodies;
  /* cada cuerpo de la lista es insertado en el
     arbol apuntado por Root */
  while (auxb != NULL) {
    auxt = Root;
    while (auxt != NULL) {
      auxb2 = auxt→body;
      if (auxb2 != NULL) {
        /* el nodo actual auxt, apunta a un
           cuerpo directamente. Dicho cuerpo
           y el nuevo cuerpo tendran que ser
           insertados en distintas
           subdivisiones de auxt */
        auxt→body = NULL;
        I = SUBINDEX(auxt, auxb2);
        newt = (struct t2 *)malloc(
                sizeof(struct t2));
        newt→level = auxt→level+1;
        newt→body = auxb2;
        auxt→child[I] = newt;
      }
      else {
        /* el nodo actual auxt no apunta a un
           cuerpo. Se busca la subdivision
           correspondiente al cuerpo actual,
           auxb, y se inserta alli */
        I = SUBINDEX(auxt, auxb);
        auxt2 = auxt→child[I];
        if (auxt2 == NULL) {
          /* la subdivision correspondiente no
             posee nodo del arbol. Se crea uno
             nuevo y se inserta auxb como cuerpo
             */
          newt = (struct t2 *)malloc(
                  sizeof(struct t2));
          newt→level = auxt→level+1;
          newt→body = auxb;
          auxt→child[I] = newt;
          auxt = NULL;
        }
        else {
          /* en la subdivision correspondiente
             existe un nodo. Se avanza auxt en
             dicha direccion */
          auxt = auxt2;
        }
      }
    }
    auxb = auxb→nxt;
  }
}

void Hack_cofm(node)
struct t2 *node;
{
  vector tmpv;
  int I;

  if (node→body == NULL) {
    /* si el nodo no apunta a un cuerpo, se
       inicializa su masa y posicion */
    node→mass = 0;
    CLRV(node→pos);
    for (I=0; I<8; I++) {
      /* se recorren todos sus hijos calculando
         sus masas y posiciones con una llamada
         recursiva */
      if (node→child[I] != NULL) {
        Hack_cofm(node→child[I]);
        /* en base a la masa y posicion del
           hijo actual, se van acumulando masa
           y posicion del padre */
        node→mass += node→child[I]→mass;
        MULVS(tmpv, node→child[I]→pos,
```

```
            node→child[I]→mass);
          ADDV(node→pos, node→pos, tmpv);
        }
      }
      DIVVS(node→pos, node→pos, node→mass);
    }
    else {
      /* el nodo apunta a un cuerpo, por tanto
         no hay que calcular masa y posicion */
      node→mass = node→body→mass;
      CPYV(node→pos, node→body→pos);
    }
}

void Hack_grav(node, body)
struct t2 *node;
struct t1 *body;
{
  struct t2 *hc_aux;
  int I;

  /* hay que determinar si el cuerpo esta lo
     suficientemente alejado del nodo
     representado por node como para usar su
     masa total y centro de masas para calcular
     la fuerza sobre body */
  if (SUBDIVP(node, body)) {
    /* no esta suficientemente lejos, hay que
       subdividir */
    if (node→body == NULL) {
      /* el nodo no apunta directamente a un
         cuerpo, por tanto se avanza
         recursivamente sobre sus ocho hijos */
      for (I=0; I<8; I++) {
        hc_aux = node→child[I];
        if (hc_aux != NULL) {
          Hack_grav(hc_aux, body);
        }
      }
    }
    else {
      /* el nodo apunta a un cuerpo, por tanto
         se utilizan los datos de este cuerpo
         para calcular la fuerza y modificar
         posicion del cuerpo apuntado por body
         */
      GRAV_SUB(body, node);
    }
  }
  else {
    /* el nodo esta suficientemente lejos, se
       toman su masa total y centro de masas */
    GRAV_SUB(body, node);
  }
}

void main() {
  struct t1 *Lbodies, *auxb, *sr_aux, *sr_n;
  struct t2 *Root;
  char eof;


  Lbodies = (struct t1 *)malloc(
              sizeof(struct t1));
  sr_aux = Lbodies;
  Leer_datos(&(sr_aux→mass), &(sr_aux→pos),
             &eof);
  while (!eof) {
    sr_n = (struct t1 *)malloc(
             sizeof(struct t1));
    Leer_datos(&(sr_n→mass), &(sr_n→pos),
               &eof);
    sr_aux→nxt = sr_n;
    sr_aux = sr_n;
  }

  /* construir el octree apuntado por Root
     insertando los cuerpos almacenados en
     la lista Lbodies */
  Make_tree(&Root, Lbodies);

  /* calcular el centro de masas y masa total
     para los nodos del arbol */
  Hack_cofm(Root);

  /* para cada cuerpo se recorre el arbol para
     calcular las fuerzas sobre el cuerpo */
  auxb = Lbodies;
  while (auxb != NULL) {
    Hack_grav(Root, auxb);
    auxb = auxb→nxt;
  }
}
```

E.2.8 Código simulación Barnes-Hut analizado

```
/* Estructura Lista Cuerpos */
typedef struct t1 {
  t1_data;
  struct t1 *nxt;
}

/* Estructura Octree */
typedef struct t2 {
  t2_data data;
  struct t2 *child [8];
  struct t1 *body;
}

/* Estructura Pila */
typedef struct t3 {
  t3_data;
  struct t3 *nxt;
  struct t2 *node;
}
```

```
void main() {
  struct t1 *Lbodies, *sr_aux, *sr_n, *auxb,
            *auxb2, *newb;
  struct t2 *Root, *auxt, *auxt2, *newt;
  struct t3 *Stack, *auxs, *news, *auxs2;

/* creacion de la lista de cuerpos */
  Lbodies = malloc;
  sr_aux = Lbodies;
  while () {
    sr_n = malloc;
    sr_aux→nxt = sr_n;
    sr_aux = sr_n;
  }
  sr_aux = NULL;
  sr_n = NULL;
/* creacion del arbol insertando cada cuerpo
   Make_tree() */
```

```
Root = malloc;
auxb = Lbodies;
while () {
  FORCE auxb != NULL;
  auxt = Root;
  while () {
    FORCE auxt != NULL;
    auxb2 = auxt→body;
    if () {
      FORCE auxb2 != NULL;
      auxt→body = NULL;
      DeleteIndex(I);
      newt = malloc;
      newt→body = auxb2;
      auxb2 = NULL;
      auxt→child[I] = newt;
      newt = NULL;
    }
    else {
      FORCE auxb2 == NULL;
      DeleteIndex(I);
      auxt2 = auxt→child[I];
      if () {
        FORCE auxt2 == NULL;
        newt = malloc;
        newt→body = auxb;
        auxt→child[I] = newt;
        newt = NULL;
        auxt = NULL;
      }
      else {
        FORCE auxt2 != NULL;
        auxt = auxt2;
        auxt2 = NULL;
      }
    }
    DeleteIndex(I);
  }
  FORCE auxt == NULL;
  auxb2 = auxb→nxt;
  auxb = auxb2;
  auxb2 = NULL;
}
FORCE auxb == NULL;
/* recorrido del arbol para el calculo del
   centro de masas y masa total utilizando
   la pila. Hack_cofm() */
Stack = malloc;
auxs = Stack;
auxs→node = Root;
while () {
  FORCE auxs != NULL;
  auxt = auxs→node;
  auxb = auxt→body;
  if () {
    /* el nodo apunta a un cuerpo,
       se saca de la pila */
    FORCE auxb != NULL;
    auxs→node = NULL;
    auxs2 = auxs→nxt;
    auxs = auxs2;
    auxs2 = NULL;
  }
  else {
    /* no apunta a un cuerpo */
    FORCE auxb == NULL;
    if () {
      /* sus hijos no han sido visitados,
         se introducen en la pila */
      for () {
        DeleteIndex(I);
        news = malloc;
        auxt2 = auxt→child[I];
        news→node = auxt2;
        news→nxt = auxs;
        auxs = news;
        news = NULL;
        auxt2 = NULL;
        auxt = NULL;
      }
      DeleteIndex(I);
      DeleteIndex(va(I));
    }
    else {
      /* sus hijos han sido visitados,
         se saca de la pila */
      for () {
        DeleteIndex(I);
      }
      DeleteIndex(I);
      DeleteIndex(va(I));
      auxs→node = NULL;
      auxs2 = auxs→nxt;
      auxs = auxs2;
      auxs2 = NULL;
    }
  }
}
FORCE auxs == NULL;
/* para cada cuerpo, recorrer el arbol para
   calcular fuerzas, utilizando la pila
   Hack_grav() */
auxb = Lbodies;
while () {
  FORCE auxb != NULL;
  Stack = malloc;
  auxs = Stack;
  auxs→node = Root;
  while () {
    FORCE Stack != NULL;
    /* se saca nodo de la pila */
    auxt = Stack→node;
    if () {
      /* no es un cuerpo */
      for () {
        DeleteIndex(I);
        auxt2 = auxt→child[I];
        if () {
          /* no esta suficientemente
             lejos, se inserta en pila */
          FORCE auxt2 != NULL;
          news = malloc;
          news→node = auxt2;
          auxt2 = NULL;
          auxs→nxt = news;
          auxs = news;
          news = NULL;
        }
        else {
          /* esta suficientemente
             lejos */
          NOP
        }
      }
      DeleteIndex(I);
      DeleteIndex(va(I));
    }
    else {
      /* es un cuerpo */
      NOP
    }
    news = Stack→nxt;
    Stack = news;
    news = NULL;
```

```
    }
    FORCE Stack == NULL;
    auxs = NULL;
    auxt = NULL;
    auxb2 = auxb→nxt;
      auxb = auxb2;
      auxb2 = NULL;
    }
    FORCE auxb == NULL;
}
```

Bibliografía

[1] R. Asenjo. *Factorización LU de Matrices Dispersas en Multiprocesadores.* PhD thesis, Dept. de Arquitectura de Computadores, Univ. Málaga. España, 1997.

[2] U. Banerjee. *Dependence Analysis for Supercomputing.* Norwell, Mass. Kluwer Academic Publishers, 1988.

[3] J. Banning. An efficient way to find the side effects of procedure calls and the aliases of variables. In *Conference Record of the Sixth Annual ACM Symposium on the Principles of Programming Languages*, San Antonio, TX, 1979.

[4] J. Barnes and P. Hut. A hierarchical O(n· log n) force calculation algorithm. *Nature*, 324, December 1986.

[5] R. Barret, M. Berry, T. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. van der Vorst. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods.* Siam Press, 1994.

[6] J. Barth. A practical interprocedural data flow analysis algorithm. *Communications of the ACM*, 21(9):724–736, 1978.

[7] W. Blume, R. Doallo, R. Eigenmann, J. Grout, J. Hoeflinger, T. Lawrence, J. Lee, D. Padua, Y. Paek, B. Pottenger, L. Rauchwerger, and P. Tu. Parallel programming with Polaris. *Computer*, 29(12):78–82, 1996.

[8] W. Blume, R. Eigenmann, J. Hoeflinger, D. Padua, P. Petersen, L. Rauchwerger, and P. Tu. Automatic detection of parallelism: A grand challenge for high-performance computing. *IEEE parallel and distributed technology: systems and applications*, 2(3):37–47, Fall 1994.

[9] D. Chase, M. Wegman, and F. Zadek. Analysis of pointers and structures. In *SIGPLAN'90 Conference on Programming Language Design and Implementation*, pages 296–310, 1990.

[10] J. Choi, M. Burke, and P. Carini. Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side effects. In *Conference Record of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 232–245, Charleston, South Carolina, 1993.

[11] K. Cooper. Analyzing aliases of reference formal parameters. In *Conference Record of the Twelfth Annual ACM Symposium on the Principles of Programming Languages*, New Orleans, LA, 1985.

[12] K. Cooper and K. Kennedy. Fast interprocedural alias analysis. In *Sixteenth Annual ACM Symposium on the Principles of Programming Languages*, Austin, TX, 1989.

[13] F. Corbera, R. Asenjo, and E. Zapata. New shape analysis for automatic parallelization of c codes. In *ACM International Conference on Supercomputing*, pages 220–227, Rhodes, Greece, 1999.

[14] F. Corbera, R. Asenjo, and E. Zapata. Accurate shape analysis for recursive data structures. In *13th Int'l. Workshop on Languages and Compilers for Parallel Computing*, IBM T.J. Watson Res. Ctr., New York, 2000.

[15] F. Corbera, R. Asenjo, and E. Zapata. Un paso en el proceso de paralelización automática de códigos c. In *XI Jornadas de Paralelismo*, pages 53–58, Granada, Septiembre 2000.

[16] F. Corbera, R. Asenjo, and E. Zapata. Progressive shape analysis for real C codes. In *International Conference on Parallel Processing 2001*, Valencia, Spain, 2001.

[17] F. Corbera, R. Asenjo, and E. Zapata. New shape analysis and interprocedural techniques for automátic parallelization of C codes. *International Juournal in Parallel Programming*, En prensa.

[18] F. Corbera, R. Asenjo, and E. Zapata. Towards the automatic parallelization of C codes. *IEEE Trans. on Parallel and Distributed Systems*, Enviado.

[19] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Fourth ACM Symposium on Principles of Programming Language*, pages 238–252, Los Angeles, California, 1977. ACM Press, New York.

[20] A. Deutsch. On determining lifetime and aliasing of dynamically allocated data in higherorder functional specifications. In *17th Annual ACM Symposium on the Principles of Programming Languages*, pages 157–168, San Francisco, 1990.

[21] A. Deutsch. Storeless model of aliasing and its abstractions using finite representations of right-regular equivalence relations. In *IEEE 1992 International Conference on Computer Languages*, pages 2–13, San Francisco, 1992.

[22] A. Deutsch. Interprocedural may-alias analysis for pointers: beyond k-limiting. *ACM SIGPLAN Notices*, 29(6):230–241, 1994.

[23] I. Duff, A. Erisman, and J. Reid. *Direct Methods for Sparse Matrices*. Oxford University Press, 1986.

[24] I. Duff and J. Reid. MA48, a Fortran code for direct solution of sparse unsymmetric linear systems of equations. Technical Report RAL-93-072, Rutherford Appleton Lab., UK, 1993.

[25] M. Emami. *A practical interprocedural alias analysis for an optimizing/parallelizing C compiler*. PhD thesis, School of Computer Science, McGill University, 1993.

[26] M. Emami, R. Ghiya, and L. Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers, 1994.

[27] R. Ghiya. *Putting Pointer Analysis to Work*. PhD thesis, School of Computer Science, McGill University, Montreal, May 1998.

[28] R. Ghiya and L. Hendren. Connection analysis: A practical interprocedural heap analysis for c. *International Journal of Parallel Programming*, 24(6):547–578, 1996.

[29] R. Ghiya and L. Hendren. Is it a tree, a dag, or a cyclic graph? a shape analysis for heap-directed pointers in C. In *23rd ACM Symposium on Principles of Programming Languages*, pages 1–15, 1996.

[30] R. Ghiya and L. Hendren. Putting pointer analysis to work. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 121–133, 1998.

[31] R. Ghiya, L. Hendren, and Y. Zhu. Detecting parallelism in C programs with recursive data structures. In *1998 International Conference on Compiler Construction*, March 1998.

[32] M. Girkar, M. Haghighata, C. Lee, B. Leung, and D. A. Schouten. *Parafrase-2 manual*. CSRD, Univ. of Illinois, 1992.

[33] M. González. *Distribución Automática de Datos en Multiprocesadores*. PhD thesis, Dept. de Arquitectura de Computadores, Univ. Málaga. España, 2000.

[34] S. Graphics. *IRIS Power C, User's Guide*. SGI, Inc., Mountain View, CA, 1996.

[35] S. Graphics. *IRIS Power Fortran, User's Guide*. SGI, Inc., Mountain View, CA, 1996.

[36] E. Gutiérrez, R. Asenjo, O. Plata, and E. Zapata. Automatic parallelization of irregular applications. *J. Parallel Computing*, 26(13-14):1709–1738, December 2000.

[37] M. Hall, J. Anderson, S. Amarasinghe, B. Murphy, S. Liao, E. Bugnion, and M. Lam. Maximizing multiprocessor performance with the SUIF compiler. *IEEE Computer*, 29(12):84–89, 1996.

[38] L. Hendren. *Parallelizing Programs with Recursive Data Structures*. PhD thesis, Cornell University, 1990.

[39] L. Hendren and A. Nicolau. Parallelizing programs with recursive data structures. *IEEE Trans. on Parallel and Distributed Computing*, 1(1):35–47, 1990.

[40] L. Hendren, C. Donawa, M. Emami, G. Gao, Justiani, and B. Sridharan. Designing the McCAT compiler based on a family of structured intermediate representations. In *Proceedings of the 5th International Workshop on Languages and Compilers for Parallel Computing*, pages 406–420. Springer-Verlag, LNCS 757, 1993.

[41] L. Hendren, J. Hummel, and A. Nicolau. Abstractions for recursive pointer data structures: Improving the analysis and transformation of imperative programs. In *Conference on Programming Language Design and Implementation (PLDI)*, volume 27, pages 249–260, New York, NY, 1992. ACM Press.

[42] J. Hoeflinger and Y. Paek. The access region test. In *Twelfth International Workshop on Languages and Compilers for Parallel Computing (LCPC'99)*, University of California, San Diego, La Jolla, CA USA, 1989.

[43] C. Hogue, D. Galgani, G. Ackley, B. Johnson, B. Nelson, C. Vu, and M. Itzkowitz. *MIPSpro^{TM} Fortran 77 Programmer's Guide*. Silicon Graphics, Inc., 1994-96. Document number 007-3572-002.

[44] S. Horwitz, P. Pfeiffer, and T. Reps. Dependence analysis for pointer variables. *ACM SIGPLAN Notices*, 24(7):28–40, 1989.

[45] P. Hudak. A semantic model of reference counting and its abstraction. In *1986 ACM Conference on LISP and functional programming*, pages 351–363, 1986.

[46] J. Hummel, L. Hendren, and A. Nicolau. Abstract description of pointer data structures: An approach for improving the analysis and optimization of imperative programs. *ACM Letters on Programming Languages and Systems*, 1(3):243–260, September 1992.

[47] J. Hummel, L. Hendren, and A. Nicolau. A general data dependence test for dynamic, pointer-based data structures. *ACM SIGPLAN Notices*, 29(6):218–229, 1994.

[48] J. Hummel, L. Hendren, and A. Nicolau. A language for conveying the aliasing properties of dynamic, pointer-based data structures. In *8th International Parallel Processing Symposium*, pages 208–216, Cancun, Mexico, 1994.

[49] Y. Hwang and J. Saltz. Identifying DEF/USE information of statements that construct and traverse dynamic recursive data structures. In *10th International Workshop on Languages and Compilers for Parallel Computing*, pages 131–145, 1997.

[50] K. Inoue, H. Seki, and H. Yagi. Analysis of functional programs to detect run-time garbage cells. *ACM Transactions on Programming Languages and Systems*, 10(4):555–578, 1988.

[51] N. Jones and S. Muchnick. *Flow Analysis and Optimization of Lis-like Structures*, chapter Flow Analysis: Theory and Applications,Chapter 4, pages 102–131. Prentice Hall, 1981.

[52] N. Jones and S. Muchnick. A flexible approach to interprocedural data flow analysis and programs with recursive data structures. In *ACM Symposium on Principles of Programming Languages*, pages 66–74, New York, 1982. ACM Press.

[53] S. Kleene. *Introduction to Metamathematics*. North-Holland, second edition, 1987.

[54] W. Landi. Undecidability of static analysis. *ACM Letters on Programming Languages and Systems*, 1(4), December 1992.

[55] W. Landi and B. Ryder. A safe approximation algorithm for interprocedural pointer aliasing. In *SIGPLAN'92 Conference on Programing Language Design and Implementation*, pages 235–248, 1992.

[56] J. Larus and P. Hilfinger. Detecting conflicts between structure accesses. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 21–33, New York, 1988. ACM Press.

[57] T. Lev-Ami, T. Reps, M. Sagiv, and R. Wilhelm. Putting static analysis to work for verification: a case study. In *International Symposium on Software Testing and Analysis, ISSTA'00*, 2000.

[58] T. Lev-Ami and M. Sagiv. TVLA: A system for implementing static analyses. In *Static Analysis Symposium*, pages 280–301, 2000.

[59] A. Matsumoto and T. T. D.S. Han. Alias analysis of pointers in Pascal and Fortran 90: Dependence analysis between pointer references. *Acta Informatica*, (33):99–130, 1996.

[60] S. Muchnick. *Advanced Compiler Design & Implementation*. Morgan Kaufmann, 1997.

[61] E. Myers. A precise inter-procedural data flow algorithm. In *Conference record of the 8th ACM Symposium on Principles of Programming Languages (POPL)*, pages 219–230, 1981.

[62] J. Plevyak, A. Chien, and V. Karamcheti. Analysis of dynamic structures for efficient parallel execution. In *Languages and Compilers for Parallel Computing*, pages 37–57, Berlin, 1993. Sprienger-Verlag.

[63] C. Polychronopoulos, M. Girkar, M. Haghighat, C. Lee, B. Leung, and D. Schouten. The structure of Parafrase-2: An advanced parallelizing compiler for C and fortran. In *Workshop on Languages and Compilers for Parallel Computing*, pages 423–453, August 1989.

[64] E. Ruf. Context-insensitive alias analysis reconsidered. *ACM SIGPLAN Notices*, 30(6):13–22, 1995.

[65] C. Ruggieri and T. Murtagh. Lifetime analysis of dynamically allocated objects. In *Conference Record of the Fifteenth Annual ACM Symposium on Principles of Programming Languages*, pages 285–293, San Diego, California, 1988.

[66] M. Sagiv, T. Reps, and R. Wilhelm. Solving shape-analysis problems in laguages with destructive updating. In *23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 16–31, St. Petersburg, Florida, January 1996.

[67] M. Sagiv, T. Reps, and R. Wilhelm. Solving shape-analysis problems in laguages with destructive updating. *ACM Transactions on Programming Languages and Systems, 20(1)*, pages 1–50, January 1998.

[68] M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. In *Symposium on Principles of Programming Languages*, pages 105–118, 1999.

[69] M. Shapiro and S. Horwitz. Fast and accurate flow insensitive points-to analysis, 1997.

[70] J. Singh, W. Weber, and A. Gupta. Splash: Stanford parallel applications for shared memory. *Computer Architecture News*, 20(1):5–44, March 1992.

[71] B. Steensgaard. Points-to analysis in almost linear time. In *23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 32–41, St. Petersburg, Florida, 1996.

[72] E. Wang and P. Hilfinger. Analysis of recursive types in Lisp-like languages. In *ACM Conference on LISP and Functional Programming*, pages 216–225, 1992.

[73] R. Wilson and M. Lam. Efficient context-sensitive pointer analysis for C programs. In *ACM SIGPLAN'95 Conference on Programming Language Design and Implementation*, pages 18–21, La Jolla, California, 1995.

[74] M. Wolfe. Optimizing supercompilers for supercomputers. Technical Report UIUCDCS-R-82-1105, Dept. Computer Science, 1982.

[75] S. Woo, M. Ohara, E. Torrie, J. Singh, and A. Gupta. The splash-2 programs: Characterization and methodological considerations. In *22nd Annual International Symposium on Computer Architecture*, pages 24–36, June 1995.

[76] H. Zima and B. Chapman. *Supercompilers for Parallel and Vector Computers.* ACM Press, 1990.