Departamento de Arquitectura de Computadores
Universidad de Málaga

TESIS DOCTORAL

Aplicando Cadenas Def-Use de Estructuras de Datos Dinámicas para la Optimización de Programas

Ana Rosa Castillo González

Málaga, 2009

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

Dra. María Ángeles González Navarro
Profesora Titular del Departamento
de Arquitectura de Computadores
de la Universidad de Málaga

CERTIFICAN:

Que la memoria titulada *Aplicando Cadenas Def-Use de Estructuras de Datos Dinámicas para la Optimización de Programas*, ha sido realizada por Dña. Ana Rosa Castillo González 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 Doctora en Ingeniería de Telecomunicación.

Málaga, a 28 de Agosto de 2009

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

Fdo: Dra. María Ángeles González Navarro
Codirectora de la Tesis Doctoral

Agradecimientos

En este largo período de formación que acaba de mi vida, han participado, de una forma u otra, no pocas personas. Por orden cronológico, en primera instancia agradezco al director del departamento el haberme admitido todo este tiempo en la familia DAC, donde tan bien me han tratado, y haberme dado la oportunidad de llegar hasta aquí. A mis tutores oficiales, su gran apoyo, ayuda, comprensión y mucha paciencia; también al que considero mi tutor extra-oficial, quién me ha ofrecido siempre su amable y valiosa colaboración.

Respecto a mis compañeros... quisiera mencionar especialmente a aquellos cuya compañía más he disfrutado, aquellos con los que he reído y llorado... Daros las gracias por los divertidos momentos vividos, que hacían más llevadera la rutina y, en ocasiones, menos pesada la carga. Me siento al mismo tiempo triste y afortunada, pues pierdo a unos excelentes compañeros, pero gano espero a unos, aún mejores, amigos. Ojalá sigamos acumulando recuerdos tan buenos como los que me habéis dejado en la memoria.

Mi familia y amigos que siempre están ahí a mi lado, dando su apoyo incondicional, su calor humano... no puedo más que seguir teniendo fe en ellos, y tratar de devolver ese amor que nunca me habéis dejado de dar y esa confianza que me inspiráis cada día. Sois una componente tan importante, que estoy segura que sin vosotros no estaría ahora donde me encuentro.

A esa persona especial, simplemente decirle que las palabras no son suficientes para expresar todo lo que me ha aportado en esta dura etapa, y me sigue aportando en mi vida, en todos los niveles. La separación física no puede romper aquello que se siente como parte de ti. Ya no soy yo, ni eres tú, sino que somos al mismo tiempo. Me siento más fuerte a tu lado y sé que contigo, todo saldrá adelante, donde sea y como sea.

De todos y cada uno de vosotros he aprendido algo y sigo aprendiendo...

...GRACIAS.

Índice

Índice de figuras VII

Índice de tablas IX

1.- Introducción **1**

1.1. Introducción . . . 1

1.2. Objetivos . . . 7

1.3. Organización de la tesis . . . 7

2.- Conceptos y marco de trabajo **9**

2.1. Cadenas de definición y uso . . . 9

2.2. Control Flow Graph (CFG) . . . 11

2.2.1. Dominancia . . . 12

2.3. Representación SSA (Static Single Assignment) . . . 13

2.3.1. Primera etapa: evaluación de los conjuntos de dominancia . . . 15

2.3.2. Segunda etapa: introducción de funciones *phi* . . . 15

2.3.3. Tercera etapa: renombrado de las variables del código . . . 17

2.3.4. Optimización . . . 19

2.4. Extensión interprocedural (ISSA) . . . 21

2.5. Análisis de Forma . . . 21

2.5.1. Sumarización y materialización . . . 23

2.6. Patrones de recorrido y caminos de acceso . . . 24

2.7. Análisis de Dependencias . . . 25

2.7.1. Dependencias de control . . . 25

2.7.2. Dependencias de datos . . . 25

3.- Análisis de cadenas de definición y uso **27**

3.1. Introducción . . . 27

3.2. Algoritmo DU link . . . 28
3.2.1. Preproceso del código de entrada previo a la aplicación del algoritmo . . . 28
3.2.2. Descripción del algoritmo *DU-link* . . . 29
Etapa 1: Análisis Intraprocedural . . . 29
Etapa 2: Construcción del IFG . . . 35
Etapa 3: Propagación en el IFG . . . 43
Etapa 4: Cómputo de cadenas DU interprocedurales . . . 45
3.2.3. Complejidad . . . 46
3.3. Cadenas de definición y uso ISSA . . . 46
3.3.1. Complejidad . . . 48
3.4. Resultados Experimentales . . . 48
3.5. Conclusiones . . . 52

4.- Code Slicing **53**
4.1. Aceleración del Análisis de Forma . . . 53
4.1.1. Introducción . . . 53
4.1.2. Funcionamiento . . . 54
4.2. Slicing del código . . . 61
4.2.1. Slicing estático . . . 62
4.2.2. Algoritmo . . . 63
4.3. Complejidad . . . 66
4.4. Resultados Experimentales . . . 67
4.5. Conclusiones . . . 68

5.- Análisis de dependencias **69**
5.1. Introducción . . . 69
5.2. Detección de los Punteros de Inducción . . . 71
5.3. Paths . . . 74
5.3.1. Representación de los paths . . . 75
5.3.2. Cálculo de los Access Paths . . . 76
5.4. Grafos de forma . . . 80
5.5. Test de dependencias . . . 81
5.5.1. Algoritmo básico . . . 81
Etapa 1: Construcción de los grupos de conflictos . . . 82
Etapa 2: Creación de paths y detección de conflictos . . . 83
5.5.2. Algoritmo extendido . . . 91

5.5.3. Ejemplo práctico 93
LCD para el bucle L1 95
LCD para el bucle L2 98
Generación de los *paths* 98
Despliegue de *ce* 99
Despliegue de *pre* 99
Despliegue de *nav* 101
Despliegue de *tail* 102
5.6. Complejidad 103
5.7. Resultados Experimentales 103
5.8. Conclusiones 107

6.- Conclusiones 109
6.1. Conclusiones 109
6.2. Posibles líneas futuras 111

Bibliografía 113

Índice de figuras

1.1. Ejemplos de estructuras dinámicas. 2
1.2. Etapas del análisis completo. 5
2.1. Ejemplo para detección de cadenas DU escalares. 10
2.2. Código de un programa simple. 10
2.3. Programa simple escrito en C. 10
2.4. CFG del programa de la Fig. 2.2. 11
2.5. Algoritmo para calcular el árbol de dominancia. 13
2.6. Árbol de dominancia para el CFG de la Fig. 2.4. 14
2.7. Algoritmo para el cálculo de las fronteras de dominancia. 15
2.8. Algoritmo para la inserción de sentencias phi. 16
2.9. Código de la Fig. 2.2 tras aplicar el algoritmo de inserción. 17
2.10. Algoritmo para el renombramiento de variables. 18
2.11. Código de la Fig. 2.2 tras aplicar el algoritmo de renombrado. 19
2.12. Algoritmo para la eliminación de funciones phi redundantes. 20
2.13. Forma SSA definitiva del código de la Fig. 2.2. 20
2.14. (a) Código de un programa simple; (b) Forma ISSA del código. 22
2.15. Código ejemplo para el análisis de forma. 22
2.16. (a) Lista;(b) Programa;(c) Expresiones de camino 24
2.17. Dependencia de control 25
2.18. Dependencia de flujo 25
2.19. Antidependencia 25
2.20. Dependencia de salida 26
2.21. Dependencia de entrada 26
2.22. Dependencia acarreada por lazo 26
3.1. Programa simple escrito en C. 28
3.2. Pasos por los que debe pasar el código de entrada. 29
3.3. Pasos en el algoritmo DU link. 29

3.4. Algoritmo implementado para el análisis intraprocedural. 35
3.5. Estructura de datos del código simple de la Fig. 3.6. 36
3.6. Ejemplo de código simple para mostrar las etapas del algoritmo. 36
3.7. Ejemplo de propagación de usos virtuales en un código simple 37
3.8. IFG del programa de la Fig. 3.6 38
3.9. Pasos para la construcción de los enlaces IFG. 39
3.10. Ejemplo de creación de enlace interreaching. 40
3.11. Ejemplo para explicar los caminos reales. 41
3.12. Algoritmo para el cálculo de los enlaces *Interreaching*. 42
3.13. Algoritmo para el cálculo de la función de transferencia de un procedimiento. 43
3.14. Algoritmo para la propagación IFG de tuplas. 44
3.15. FT de la función Fipdef 45
3.16. Código de la Fig. 3.6 en representación ISSA. 47
3.17. Estructuras de datos de los códigos. 49
3.18. Estructura de datos del código Em3d. 50
4.1. Vista jerárquica de los elementos de un grafo de forma. 55
4.2. Análisis de un bucle hasta que alcanza un punto fijo en los grafos. 55
4.3. (a) Sumarización para compactar la estructura; (b) materialización para enfocar regiones actualmente accedidas. 56
4.4. Una lista simplemente enlazada de cuatro elementos en el dominio concreto. 57
4.5. Lista simplemente enlazada en ambas representaciones del dominio concreto y abstracto. . 58
4.6. Aporte de precisión de otros atributos en la abstracción del heap. 59
4.7. Código que genera la matriz dinámica. 60
4.8. Estructura dinámica creada en el código ejemplo que representa una matriz. 60
4.9. Grafo que representa dicha estructura dinámica obtenido por el analizador de forma. . . . 61
4.10. Ejemplo para ilustrar la técnica de *slicing*: (a) Código original; (b) Código podado. 62
4.11. Fases dentro del proceso de poda. 63
4.12. Algoritmo para el filtrado de sentencias. 64
4.13. Etapas dentro del análisis. 64
4.14. Código del programa Treeadd. 66
4.15. Código Treeadd podado. 66
4.16. Medidas de tiempo del slicing para los códigos. 67
4.17. Reducción del tamaño de los códigos. 68
5.1. (a) Ejemplo de puntero de inducción en bucle (b) Bucle transformado para solucionar la dependencia debida a p. 72

5.2. (a) Hay puntero de inducción;(b) No hay puntero de inducción. 72
5.3. Algoritmo para la detección de punteros de inducción. 73
5.4. Código ejemplo de recorrido (a) con un navegador, (b) con dos navegadores. 74
5.5. Estructura de datos para una matrix dinámica. 74
5.6. Función para el cálculo de los *Access Paths*. 77
5.7. Recorrido de una lista. 78
5.8. Anidamiento de dos niveles 78
5.9. Grafo de forma del código presentado en la Fig.5.4(a). 81
5.10. Algoritmo para la detección de dependencias. 82
5.11. Función `Create_Conflict_Groups`. 83
5.12. Función para la detección de conflictos de primer nivel. 83
5.13. Sitios visitados por la navegación del ejemplo. 84
5.14. Grafo de forma del código empleado para el ejemplo. 85
5.15. Representación de los sitios abstractos visitados en el grafo de forma. 86
5.16. Función para la detección de conflictos de segundo nivel. 88
5.17. Función de detección de conflictos de tercer nivel. 89
5.18. Función para la detección de conflictos de cuarto nivel. 89
5.19. Función `Check_ComEst` 91
5.20. Función `Check_Conflict_Paths`. 91
5.21. Función `Check_Conflict_Site`. 92
5.22. Función `Check_Conflict_Site_pre`. 93
5.23. Función `Check_Ciclic_nav`. 93
5.24. Función `Check_Conflict_Site_nav`. 94
5.25. Función `Unroll_Path`. 94
5.26. (a) Código con doble bucle anidado. (b) Recorridos e inducciones sobre la estructura real. . 95
5.27. Sitio obtenido al desplegar (a) *ce* y (b) *nav* para ambos paths en L1. 96
5.28. Representación sobre la estructura real de los *Unrolled Paths* que provocan un `LCD`. 98
5.29. (a) Elementos concretos representados por los `unrolled paths` UP_{w4} y UP_{r4}. (b) Sitios abstractos que forman ambos `unrolled paths` UP_{w4} y UP_{r4} ($< SL7, N2 >$ y $< SL2, N2 >$). 102
5.30. Etapas del análisis completo. 105
5.31. Representación de la aceleración en el análisis de dependencias. 106
6.1. Etapas del análisis completo. 110

Índice de tablas

3.1. Ejemplo de propagación de tuplas de uso y definición 33
3.2. Ejemplo de detección de cadena DU intraprocedural. 37
3.3. Relación entre los nodos IFG y los nodos CFG 37
3.4. Clasificación de los enlaces IFG 39
3.5. Fases de propagación del código de la Fig. 3.6 45
3.6. Información acerca de cada código analizado 51
3.7. Medidas de tiempo en segundos por etapas del algoritmo DU-link 51
3.8. Dimensiones IFG y cantidad de tuplas 51
4.1. Casos difíciles para el análisis de forma. 61
4.2. Medidas de la aceleración conseguida en el análisis de forma. 67
5.1. Definiciones de paths y access paths. 75
5.2. Reglas de Chequeo de Conflictos (RCC). 90
5.3. Parámetros y tiempos para el algoritmo `LCDs_Detection`. 104
5.4. Medidas del análisis completo. 105
5.5. Comparativa de tiempos entre ambos análisis de dependencias. 106
5.6. Aceleración en el análisis de dependencias. 106

1 Introducción

1.1. Introducción

Abordar el problema de la optimización de códigos irregulares tiene una importancia clave, respaldada por la estimación de que al menos un 50 % de todos los programas científicos son irregulares y que más del 50 % de todos los ciclos de reloj son consumidos en supercomputadores por ese tipo de códigos. Sin embargo, los compiladores actuales son poco efectivos en la optimización de este tipo de códigos para su ejecución en las modernas arquitecturas de computadores multi-core/multi-procesador. Esta limitación de los compiladores actuales se debe principalmente a que no son capaces de extraer del código fuente la información necesaria para explotar aspectos como: cuáles son las fuentes de paralelismo; cómo hacer uso de la localidad en estructuras de datos dinámicas; y cómo implementar otras optimizaciones clave que minimicen el tiempo de ejecución de las aplicaciones y aprovechen toda la potencialidad de la arquitectura en que se ejecutan. Esa información que necesitamos extraer de estos códigos basados en estructuras dinámicas es principalmente una descripción de qué posiciones de memoria se leen o escriben en cada sentencia del código, y para ello es necesario capturar automáticamente las topologías y propiedades de las estructuras de datos presentes en el código.

En particular, las estructuras de datos basadas en memoria dinámica y que se acceden con punteros están más allá del ámbito de la mayoría de compiladores actuales. Son ineficientes cuando se trata de optimizar aplicaciones basadas en punteros para los multiprocesadores modernos [1].

En las estructuras dinámicas de datos, puesto que se construyen dinámicamente en tiempo de ejecución, sus localizaciones en memoria pueden tener cualquier forma y conexión. Por lo tanto, un programa que trata con estructuras de datos dinámicas lleva a cabo en tiempo de ejecución la distribución de las localizaciones de memoria. Estas localizaciones son accesibles y conectadas a través de punteros. Más precisamente, los punteros dirigidos al *heap* almacenados en el *stack*, o simplemente punteros, son usados para acceder a la estructura, y los punteros dirigidos al *heap* almacenados en el *heap*, o punteros al *heap*, son usados para interconectar elementos ubicados en el *heap*. Las estructuras de datos dinámicas son accedidas por punteros al *heap*. A menudo, estas estructuras son además *recursivas*, en el sentido, de que cada elemento del *heap* puede apuntar a otros elementos del *heap*, formando estructuras como listas enlazadas, *Grafos Dirigidos Acíclicos* (DAG, *Direct Acyclic Graph*), o árboles (ver Fig.1.1). Estas estructuras son comúnmente utilizadas

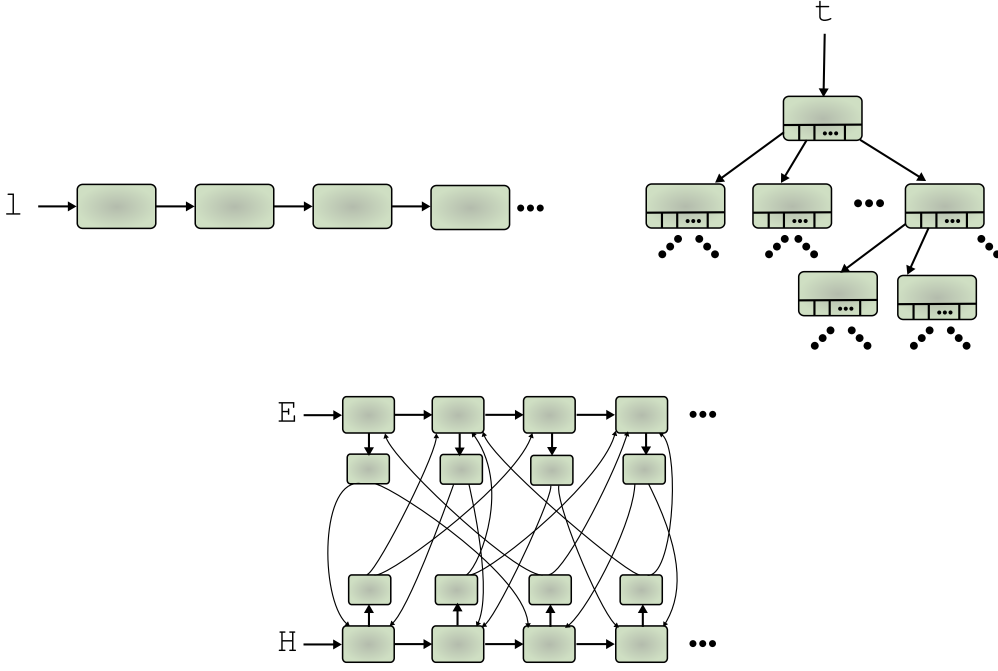


Figura 1.1: Ejemplos de estructuras dinámicas.

en aplicaciones irregulares basadas en punteros, y suponen importantes desafíos para los pases de compilación de los compiladores actuales, debido al problema de los alias. Además, todos los algoritmos basados en el flujo de datos que tratan con estructuras recursivas de datos deben tener en cuenta a priori la naturaleza no finita de estas estructuras, que pueden dinámicamente crecer hacia un tamaño indeterminado, mientras que el tamaño de las estructuras estáticas se fija en tiempo de compilación. Algunos autores solucionan este problema con el uso del proceso conocido como *k-limiting* [2], [3], [4]. Otra complicación añadida supone la identificación de variables vivas en estas estructuras, ya que muchas variables no son visibles directamente a través de las llamadas a funciones, *call sites*, sino sólo a través de la información de alias.

El problema de calcular los alias inducidos por los punteros debe resolverse de forma que los compiladores puedan desambigüar referencias de memoria sin problemas. El conocimiento estático de los alias de punteros es un elemento clave para llevar a cabo optimizaciones relacionadas

con el paralelismo y la localidad. Sin este conocimiento, es necesario tomar decisiones converva-tivas en cuanto a los accesos de los punteros, que pueden repercutir en la precisión y eficiencia de cualquier análisis que requiera esta información, como por ejemplo un compilador optimizador. Además está demostrado que un incremento de precisión en análisis complejos como análisis de punteros o *slicing* de programas, no sólo provoca que los siguientes análisis sean también más precisos, sino que sean más rápidos [5] [6].

Dependiendo del tipo de aplicación puede ser necesaria mayor o menor precisión en el análisis que se lleva a cabo. Lo que sí es cierto, es que en lenguajes modernos como C, C++ y Java la presencia de punteros demanda un análisis interprocedural con información específica que otras técnicas anteriores más antiguas enfocadas a lenguajes como FORTRAN no pueden proporcionar. Existen trabajos que aplican análisis de punteros sólo para localizaciones en el *stack* [7], representando las variables locales con nombres simbólicos que se acceden de forma indirecta pero no son visibles en el procedimiento actual. Los nombres simbólicos permiten que el algoritmo reutilice la información de punteros calculada para un procedimiento de forma local y válida para varios contextos de llamadas, si la configuración de los alias para las entradas no cambia en esos contextos. Wilson y Lam [8] desarrollaron esta idea extendiendo el algoritmo para representar a variables globales y localizaciones de memoria que pueden accederse a través de dereferencias de los parámetros formales y punteros globales en un procedimiento. Otros [4] ofrecen un análisis interprocedural eficiente basado en el uso de grafos con información paramétrica de los punteros pero insensible al flujo, además de aumentar la complejidad del algoritmo en presencia de recursión. Otros trabajos más recientes aplican análisis de punteros a lenguajes modernos como Java [9], ofreciendo una técnica escalable sensible al contexto que también puede evitar el análisis de código irrelevante. Sin embargo, no admiten métodos recursivos. Estas técnicas sólo estudian los punteros al *stack*, por lo que fracasan a la hora de encontrar el paralelismo oculto debido a los accesos al *heap*, ya que no son capaces de analizar este otro tipo de punteros. Normalmente existe un compromiso entre el nivel de exactitud del análisis del *heap* y la rapidez y escalabilidad de los mismos [10], [11]. Sin embargo, algunas técnicas son capaces de extraer información valiosa acerca de la forma en que la memoria dinámica es usada en los programas basados en punteros [12], [13].

Estudiando algunas de estas técnicas, descubrimos la contribución que pueden hacer las llamadas cadenas de definición y uso en este ámbito. Las cadenas de definición y uso (DU, def-use) han sido comúnmente construidas por los compiladores para expresar las dependencias entre las definiciones y usos de las variables en los programas [14]. Estas cadenas son usadas en análisis estático de código como el análisis de flujo de datos [15]. El conocimiento de las cadenas DU en un programa o subprograma, es un prerequisito para muchas optimizaciones de compiladores, como la propagación de constantes y la eliminación de subexpresiones comunes, ya que permite el flujo de información entre definiciones y usos sin que tenga que pasar por sentencias no relacionadas. Además pueden emplearse para desarrollar técnicas de paralelización [16], depuración, y en técnicas como el *slicing* estático y dinámico de programas, e incluso en algunas herramientas de ingeniería del software (*checkers*, *testers*)[17][18].

Existen diversas técnicas convencionales de análisis de flujo de datos para calcular las cadenas DU en procedimientos individuales (*reaching definitions, upwards exposed uses, copy-propagation...*) y también se han propuesto algunos algoritmos interprocedurales [19], [12], [20]. Un analisis interprocedural determina el flujo de datos a través de los límites de los procedimientos,

aportando información sobre parámetros de referencia y de cómo éstos son llamados, usados y modificados por los distintos *call-sites*. Por lo tanto este análisis resulta de utilidad en la optimización, generación y paralelización de código.

Cuando tratamos con punteros al *heap*, por ejemplo en programas que usan estructuras dinámicas, la mayoría de algoritmos clásicos para el cálculo de cadenas DU no son válidos, por lo que se necesitan técnicas específicas [21]. En estos casos, el análisis interprocedural es un componente crítico a la hora de desambigüar referencias de memoria de forma precisa. A lo largo de los años, la meta del análisis ha evolucionado desde la detección de alias entre los parámetros formales de programas en Fortran, hacia los alias entre referencias de punteros multi-nivel en C, C++ y Java. Sin embargo, el análisis interprocedural no está totalmente maduro debido a la dificultad de acomodar los algoritmos a programas reales de gran tamaño. Entre otras, algunas de las características deseables de un análisis interprocedural serían:

- Sensible al flujo: hay que considerar los efectos de la asignación de punteros con respecto al orden de las sentencias. Entre los beneficios está el que una asignación posterior puede matar a definiciones anteriores del mismo puntero.

- Sensible al contexto: para diferenciar entre diferentes contextos de llamada desde un mismo proceso llamante, *callee*. Normalmente el uso de algún grafo de control de flujo interprocedural ayuda a cumplir esta propiedad.

- Representación de caminos: un algoritmo estático a menudo necesita de una notación abstracta para representar las localizaciones de memoria que son accedidas en tiempo de ejecución. Los *access paths* representan de forma simple cómo son accedidas las localizaciones de memoria física desde una variable inicial. Además se demuestra que el número de nombres que representan a objetos recursivos del *heap* pueden acotarse y la localización de objetos del *heap* acíclicos puede ser desambigüada usando *access paths* [22].

Existe también, por tanto, aportación de las cadenas DU en ámbitos como las localizaciones en el *heap* y variables puntero ([2],[21]) gracias a análisis de este tipo. En el contexto de las estructuras dinámicas de datos recursivas, las definiciones de las estructuras de datos modifican el contenido de las localizaciones, pero también pueden modificar la configuración (conexiones) de la estructura. Aplicando pues las cadenas DU, podemos extraer información de las dependencias entre las construcciones y referencias de las estructuras de datos recursivas. Si además incluimos la llamada representación SSA (*static single assignment*)[23],[24], las cadenas DU son explícitas ya que cada cadena contiene un elemento único [16]. Para el cálculo de las cadenas DU en un programa, se requiere un análisis de variables vivas, de forma que pueda hacerse un seguimiento de todas las variables a lo largo del código. Estas variables vivas ya comentamos que eran importantes cuando se analiza el flujo de datos en códigos con estructuras recursivas. En este sentido la representación SSA puede ser de gran ayuda al proporcionar un renombramiento único de las variables que facilita esta tarea de seguimiento. Es por ello que la representación SSA facilita la realización de algunas optimizaciones en compiladores [25],[26],[16],[27]. En particular, aporta una gran contribución en algunas técnicas de análisis de punteros, como la representación de información del flujo de datos.

Partiendo de estas ideas y buscando la forma de optimizar programas que tratan con códigos irregulares, estudiamos el diseño de una técnica específica para el cálculo de las cadenas DU debidas a los punteros al *heap*. Nuestra propuesta [28],[29] está basada en la técnica desarrollada por Hwang y Saltz [30] para identificar cadenas DU interprocedurales en programas que definen y recorren estructuras dinámicas de datos. Nuestro trabajo con cadenas DU pretende contribuir con aportaciones para análisis de dependencias [31], análisis de efectos laterales y análisis de forma [32], esenciales en optimizaciones o paralelización de programas con estructuras dinámicas de datos recursivas. Concretamente en nuestro departamento, se trabajó en el desarrollo y perfeccionamiento de un analizador de códigos para que en tiempo de compilación, capturara de forma precisa y en un tiempo razonable las estructuras de datos usadas en aplicaciones irregulares y/o basadas en estructuras de datos dinámicas [33]. Esta herramienta se basa en un análisis de forma que es capaz de capturar estructuras de datos muy complejas en códigos escritos en C, tales como arrays de punteros, listas de árboles, listas de listas, y otras combinaciones de estructuras recursivas. El análisis de forma basado en grafos es una técnica sensible al flujo, contexto y campo de la estructura. Es un análisis basado en ejecución simbólica o interpretación abstracta de las sentencias. Los grafos de forma abstraen las configuraciones de memoria que encontramos en el programa analizado, es decir, se usan abstracciones de forma expresadas como grafos para modelar el *heap*. Concretamente, el algoritmo implementado para este análisis se basa en la interpretación abstracta (ejecución simbólica) de cada secuencia del código sobre el conjunto de grafos que representan el estado del *heap* en ese punto del programa, y, por lo tanto, es un análisis que depende del número de sentencias del código y estados del *heap* que se pueden modelar. Como consecuencia, es una técnica habitualmente mucho más costosa que otros enfoques al análisis del *heap* [34]. La información DU de sentencias al *heap* puede ser utilizada para optimizar y mejorar la eficacia de este analizador a través del diseño de un preproceso tipo *code slicing*, tal como se muestra en la Fig. 1.2 que explicaremos a continuación. De esta forma tratamos no sólo de reducir el coste del analizador en determinados programas, sino de ampliar el número de casos posibles a tratar gracias a la técnica propuesta.

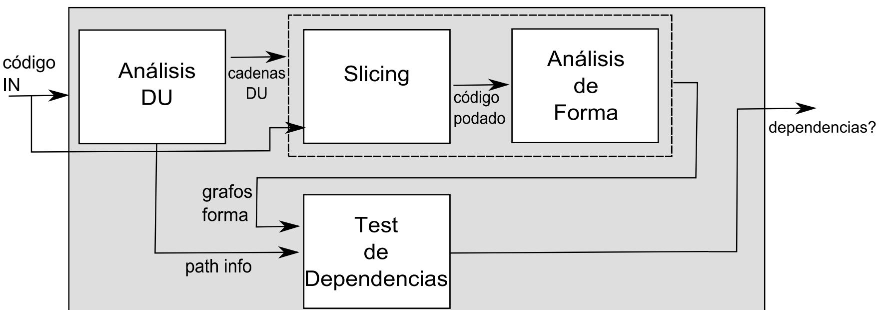


Figura 1.2: Etapas del análisis completo.

La meta final de nuestro grupo de investigación es la paralelización automática de códigos secuenciales [35] que usan estructuras dinámicas de datos, optimizando los necesarios pasos de análisis que deben llevarse a cabo para determinar si un programa puede paralelizarse o no [36]. Para tal propósito, necesitamos conocer si el programa presenta o no dependencias (salida en la Fig. 1.2). Este test de dependencias [31] está basado en análisis de conflictos de caminos de acceso

sobre grafos de forma. En la caja correspondiente de la Fig. 1.2, aparecen precisamente dos entradas: una para la información de los grafos y otra para la información de los caminos de acceso. Los grafos de forma provienen del correspondiente análisis de forma, en particular un análisis que ha sido optimizado gracias a un pase de *code slicing* basado en la información proporcionada por una herramienta de análisis basada en cadenas DU. Esta herramienta además, como vemos en la figura, proporciona la información de los caminos de acceso necesaria para el citado test de dependencias.

Por este motivo, también presentamos en este trabajo un test de dependencias para códigos basados en estructuras dinámicas de datos, herramienta que, como hemos dicho, aparece en el esquema global de la Fig. 1.2. Los resultados experimentales llevados a cabo sobre un test de dependencias previo nos han señalado el alto coste computacional de dicho test como análisis cliente del análisis de forma, debido a la interpretación abstracta. De hecho, los algoritmos tienen una complejidad exponencial (que depende del número de variables vivas), mientras que el nuevo análisis de dependencias que proponemos presenta complejidad polinómica. Por este motivo, una finalidad que perseguimos con esta propuesta es mejorar los resultados obtenidos por un antiguo test de dependencias desarrollado en nuestro grupo de investigación [37].

Varios trabajos han sido propuestos en el desarrollo de técnicas de compilación para la detección de dependencias entre escalares o variables puntero, pero no entre accesos de memoria del *heap*. La principal razón de los pocos trabajos existentes en este área es la falta de soporte a través de un apropiado análisis del *heap*. Otros tests de dependencias para accesos al *heap* basados en análisis de forma han sido desarrollados, pero fracasan a la hora de desambigüar las referencias del *heap* cuando la estructura contiene ciclos o hay modificaciones de la estructura [38] [39]. En otro trabajo reciente [40], el análisis del *heap* está enfocado hacia colecciones de librerías en Java preparadas manualmente. En resumen, la aplicación de estas otras técnicas no resulta tan general como la de nuestro enfoque.

En particular, nuestro test de dependencias se centra en la detección de las dependencias acarreadas por lazo en bucles o funciones a llamadas recursivas que recorren estructuras dinámicas de datos basadas en el *heap*. El método funciona a partir del intervalo donde va a llevarse a cabo el análisis de dependencias, seleccionando los grafos de forma que representan al *heap* en ese intervalo, así como los llamados caminos de acceso , *access paths*, para las sentencias que acceden al *heap* en el intervalo. A través de esta información de los *paths* y los grafos de forma, se desarrolla el test de dependencias basado en la proyección de cada pareja de *paths* sobre los grafos de forma y comprobando si existen conflictos entre los sitios abstractos visitados por estos *paths*. La detección de conflictos no está basada en la identificación de subexpresiones comunes [41] [42]. En su lugar, descomponemos cada *path* en una secuencia de *subpaths* (componentes de entrada, navegación y cola) y usamos información precisa que proporciona el análisis de forma, para identificar los sitios (nodos en el grafo alcanzado por un puntero o un selector) visitados por cada *subpath*. Se lleva a cabo un proceso recursivo de detección de conflictos entre los sitios alcanzados por cada *subpath*, finalizando cuando el último componente de los *paths* ha sido analizado.

La herramienta Cetus [43] ha sido de gran utilidad para la implementación de estas técnicas. Cetus fue desarrollado en la Universidad de Purdue como una infraestructura de compilador para transformaciones fuente a fuente de programas. Proporciona una representación intermedia del código a alto nivel. Permite de forma flexible la incorporación de nuevos pases de compilación, siendo especialmente útil para comprobar nuevas técnicas de compilación que requieren un alto

nivel de abstracción respecto al programa de entrada. Existen evidencias de la utilidad [44] de esta herramienta en este aspecto. Además Cetus está escrito en Java, lo que suponía una ventaja a la hora de la interacción de la técnica con la herramienta de análisis de forma, también desarrollada en Java. Por todos estos motivos, seleccionamos Cetus como base para comenzar el desarrollo de nuestro trabajo.

1.2. Objetivos

Los principales objetivos que se plantean son:

- Implementación de un algoritmo para la detección de cadenas DU en programas recursivos e irregulares. Dicho algoritmo extrae todas las cadenas DU interprocedurales de las sentencias que acceden al *heap* en programas con estructuras dinámicas de datos, además de otro tipo de información sobre el recorrido de dichas estructuras.
- Aplicabilidad de este algoritmo para la optimización de programas desarrollando técnicas de compilación para:
  - *code slicing*: desarrollando una técnica de *code slicing* orientado a la reducción de tiempos de nuestra herramienta de análisis de forma.
- Implementación de un novedoso *test de dependencias de datos* basado en análisis de conflictos de caminos de acceso proyectados sobre grafos de forma.

1.3. Organización de la tesis

El resto del contenido de esta tesis puede agruparse en los siguientes capítulos:

- En el capítulo 2 presentamos algunos conceptos fundamentales para entender mejor el marco y las herramientas de trabajo.
- El capítulo 3 aborda de lleno el funcionamiento de la técnica implementada para el cálculo de las cadenas de definición y uso. En particular presentaremos el algoritmo con detalle y veremos algunos resultados experimentales.
- El capítulo 4 contiene alguna de las aplicaciones desarrolladas a partir de la técnica presentada en el capítulo 3, especialmente la técnica de *code slicing*. Describiremos las características de estas técnicas y algunas medidas obtenidas para las mismas.
- En el capítulo 5 se brinda especial protagonismo al test de dependencias desarrollado, asistido por la información DU proveniente de la técnica presentada en el capítulo 3. También se mostrarán resultados numéricos de los experimentos llevados a cabo y se compararán con los resultados de un test de dependecias anterior en nuestro grupo de investigación.
- Finalmente en el capítulo 6 discutiremos las principales conclusiones alcanzadas con este trabajo y algunas de las posibles líneas futuras de investigación.

2 Conceptos y marco de trabajo

En este capítulo presentamos algunos de los conceptos básicos más relevantes para entender el marco de trabajo y las diferentes técnicas presentadas.

2.1. Cadenas de definición y uso

Una definición de una variable es una sentencia que asigna o puede asignar un valor a dicha variable. Un uso de una variable es una referencia de dicha variable en una sentencia que lee o puede leer el valor de esa variable. Las cadenas de definición y uso (DU) han sido comúnmente construidas por los compiladores para expresar las dependencias entre las definiciones y usos de las variables en los programas. Es decir, una cadena DU establece una relación entre el punto donde un valor es creado y el punto donde dicho valor es usado. De este modo se crea un camino entre los puntos del programa en que se define una variable y los usos alcanzables por esa variable. En el ejemplo siguiente, Fig. 2.1, encontraríamos las cadenas DU para `x` entre `S1-S2` y `S1-S3`. El uso en `S4` es alcanzable sólo desde la definición `S3` de `x`, estableciéndose otra cadena DU entre `S3-S4`.

El cálculo de las cadenas DU en un programa, requiere un análisis de variables vivas, de forma que pueda hacerse un seguimiento de todas las variables a lo largo del código. En este sentido la representación SSA puede ser de gran ayuda al proporcionarnos un renombramiento único de las

```
S1: x = 0;
S2: y = x;
S3: x = x + 1;
S4: z = x + y;
```

Figura 2.1: Ejemplo para detección de cadenas DU escalares.

variables que facilita esta tarea. Asimismo también se requiere alguna representación para poder trazar el flujo de datos en el programa.

```
S1:    clave = 1;
S2:    for (i=1;i<NMAX;i=i+1){
S3:       if (mod(i,2)){
S4:          if (clave){
S5:             x = 0;
S6:          }else{
S7:             x = 1;
S8:             clave = 0;
S9:          }
S10:      }else{
S11:         x = 23;
S12:      }
S13:   }
```

Figura 2.2: Código de un programa simple.

En estructuras recursivas de datos, estos conceptos se amplían. Por ejemplo, al hablar de cadenas DU debidas a punteros, podemos distinguir entre los punteros que apuntan al *stack* y los punteros que apuntan al *heap*. En la Fig. 2.3 se presenta un código simple basado en una estructura recursiva *node*, que no es más que una lista simplemente enlazada a través de un campo tipo puntero *nxt*. Distinguimos en `S1` la definición de un puntero al *stack*, *z*, y en `S2` la definición de un puntero al *heap*, `x→nxt`. Por lo tanto, una definición de una estructura, es una sentencia que asigna o puede asignar un valor a una localización que puede ser accedida a través del recorrido de los enlaces comenzando desde un puntero. Del mismo modo, un uso de una estructura de datos recursiva, por ejemplo `y→nxt` en `S3`, es una sentencia que lee o puede leer el valor de una localización que puede ser accedida a través del recorrido de los enlaces desde un puntero.

```
        void f(struct node *x, struct node *y){
                struct node *z, *t;
S1:         z = (struct node *) malloc(sizeof(struct node));
S2:         x->nxt = z;
S3:         t = y->nxt;
        }
```

Figura 2.3: Programa simple escrito en C.

2.2. Control Flow Graph (CFG)

Las sentencias de un procedimiento pueden agruparse en ciertos bloques básicos, conjuntos de sentencias donde el control entra en la primera sentencia del bloque y sale únicamente por la última sentencia del mismo. La información del flujo de control de un procedimiento se puede representar por un grafo dirigido llamado *Control Flow Graph (CFG)*, cuyos nodos, *N*, son los bloques básicos de un procedimiento junto con dos bloques adicionales, *Entry* y *Exit*. Además dicho grafo contiene una serie de enlaces, *E*, entre dichos nodos, *CFG = [N ∪ {Entry, Exit}, E]*. Todos los nodos son alcanzables desde el nodo *Entry*. En la Fig. 2.4 tenemos el CFG construído para el código en la Fig. 2.2.

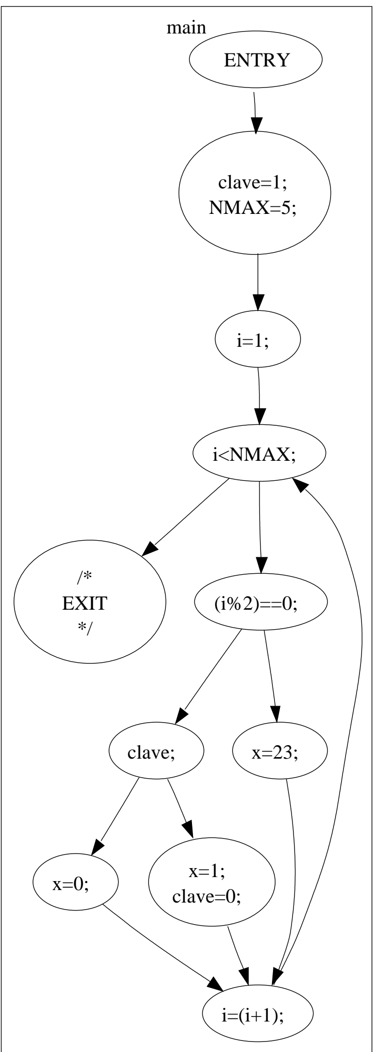


Figura 2.4: CFG del programa de la Fig. 2.2.

En el CFG se aplican términos como nodo *predecesor*, nodo *sucesor* y *camino*. En la Fig. 2.4 el nodo último que contiene la sentencia `i=i+1`, por ejemplo, tiene tres nodos predecesores y un nodo sucesor. El camino entre dos nodos `A` y `B` del CFG está formado por los nodos y enlaces que hay que recorrer para llegar al nodo `B` desde el nodo `A`.

2.2.1. Dominancia

Aspectos como la dominancia son tenidos en cuenta para determinados análisis estáticos que trabajan con el CFG. Se dice que un nodo `A` domina a otro nodo `B` en un CFG, cuando existe al menos un camino desde el nodo raíz hasta el nodo `B`, que pasa por el nodo `A`. Cuando todos los caminos desde el nodo raíz hacia el nodo `B` pasan obligatoriamente por `A` hablamos de dominancia *estricta*. En caso contrario se habla de dominancia *no estricta* o parcial. La relación de dominancia tiene propiedades: antisimétrica, reflexiva y transitiva. En el ejemplo de la figura Fig. 2.4, el nodo que contiene la sentencia `i=1` domina estrictamente al resto de nodos. En cambio el nodo que contiene la sentencia `x=23` dominaría sólo parcialmente al nodo último `i=i+1`. El *dominador inmediato* de un nodo `y` es el dominador estricto más cercano en cualquier camino desde el nodo raíz hacia el nodo `y`. El dominador inmediato de un nodo es único. Puede generarse un árbol de dominancia, *Dominator Tree*, a partir un CFG dado donde se reflejen las relaciones de dominancia entre los nodos CFG. En este caso se aplican conceptos de parentesco al hablar de nodos padre y nodos hijo. En un árbol de dominancia, un nodo `n` es el hijo de su dominador inmediato.

Nosotros hemos implementado el algoritmo de la Fig. 2.5 para calcular el árbol de dominancia a partir del CFG de un código de entrada. Este árbol de dominancia aporta la información de dominancia necesaria en la técnica que hemos implementado para la llamada transformación SSA, *Static Single Assignment*, de un código de entrada y que presentamos en el siguiente apartado de este capítulo.

El algoritmo de la Fig. 2.5, *idomTree*, para la construcción del árbol de dominancia, primero calcula el conjunto de dominadores para cada nodo, gracias a la función *findDominators*. Esta función inicializa todos los nodos CFG, salvo el nodo raíz o *entry*, con el mismo conjunto inicial de partida. Luego, recorre el CFG en orden ascendente, comenzando desde el nodo raiz, para calcular de forma incremental el conjunto de dominadores, `Dominators`, de cada nodo. Es decir, se calcula dicho conjunto para un nodo a partir de la intersección de los conjuntos de dominadores de sus nodos predecesores. Por ejemplo, en la Fig. 2.4, el conjunto de dominadores del nodo `i=i+1`, llamémosle `N`, se calcularía a partir de sus tres nodos predecesores (los nombramos `P1`, `P2` y `P3`) de la forma siguiente: $Dominators[N] = Dominators[P1] \cap Dominators[P2] \cap Dominators[P3]$. De ahí que se calculen estos conjuntos de forma ordenada desde el nodo raíz del CFG hacia abajo. Por último, después de añadir el propio nodo al conjunto `Dominators`, se actualiza la información en la tabla correspondiente.

Si aplicamos este algoritmo al CFG de la Fig. 2.4, el árbol de dominancia que obtendríamos sería el de la Fig. 2.6. Observamos como el nodo que contiene la condición de control `clave` de la sentencia `S4` tiene dos hijos, uno por cada rama de la estructura `if`, que domina totalmente. En cambio, este mismo nodo no domina totalmente, y por tanto, no es padre del nodo CFG que contiene la sentencia `i=i+1`, condición de avance del bucle `for`.

Observamos como la forma que adopta un árbol de dominancia es la de un árbol binario, con nodos padre y nodos hijo, tal como habíamos explicado. También se comprueba como el nodo raiz del CFG es siempre un dominador del resto de nodos del CFG.

```
algorithm idomTree(CFG)
begin
    /* initialization */
    for each node N ∈ CFG do
        Doms = findDominators(N)
    end
    /* extract inmediate dominators */
    for each node N ∈ CFG do
        if N has one predecessor then
            idom(N) = predecessor node
        else
            /* find the closest dominator using Doms set */
            idom(N) = closest dominator node
        end
    end
end
algorithm findDominators(CFG)
begin
    /* initialization */
    Dominators(root) = root
    for each CFG node n ∈ N − root do
        Dominators(n) = N
    end
    while(change)
        /* traversing CFG in increasing level */
        for each CFG node n ∈ N − root do
            T = N
            /* take predecesors of the node */
            for each predecessor P ∈ Preds(n) do
                T = T ∩ Dominators(P)
            end
            D = n ∪ T
            if D ≠ Dominators(n) then
                change = true
                Dominators(n) = D
            end
        end
    end
end
```

Figura 2.5: Algoritmo para calcular el árbol de dominancia.

2.3. Representación SSA (Static Single Assignment)

Encontramos la primera definición de la llamada representación SSA en el trabajo de Cytron [23]. La representación SSA original de un programa garantiza que cada definición de una variable tiene un identificador único dentro del procedimiento al que pertenece. Esto se consigue mediante el renombramiento de cada variable, cada vez que se define, a través de un subíndice numérico, comenzando por el valor `0` para la primera definición de la variable, e incrementando dicho valor con cada definición nueva encontrada para esa variable.

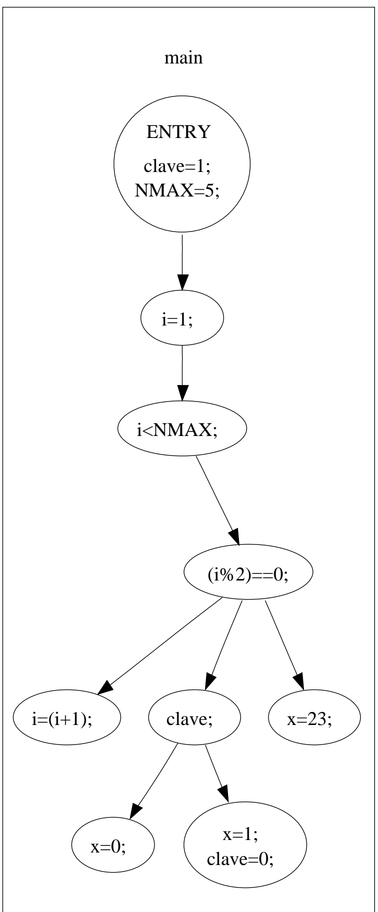


Figura 2.6: Árbol de dominancia para el CFG de la Fig. 2.4.

Existen diversas aplicaciones y técnicas basadas en la forma SSA. Destacamos los trabajos inciales de Wolfe et al.[16] y Padua et al.[25] que utilizaron la forma SSA en el ámbito del paralelismo. También los trabajos de Hendren et al. [26] que propusieron una extensión interesante de la forma SSA para punteros. Sin embargo, su nueva representación no tiene sentencias *phi*, de forma que no podría ser usada para capturar cadenas de definición y uso. Las sentencias *phi* son usadas en la forma SSA en determinados nodos del CFG para unificar las distintas definiciones de una variable que alcanzan un mismo punto del programa.

Para efectuar la transformación de un código a esta representación es necesario disponer de algún grafo de flujo del mismo. En nuestro caso vamos a tomar, como punto de partida, el CFG disponible en nuestra plataforma de trabajo, que es la que proporciona Cetus [44]. Este grafo ha sido extendido con más propiedades para la técnica que explicaremos en el siguiente capítulo.

La implementación de la representación SSA en Cetus se ha conseguido basándonos en el algoritmo original de Cytron y mejorándolo [45]. Este algoritmo requiere tres pasos: (i) evaluación de los conjuntos de dominancia DF (*Dominance Frontiers*), (ii) introducción de funciones *phi*, y (iii) renombrado de las variables del código.

2.3.1. Primera etapa: evaluación de los conjuntos de dominancia

La frontera de dominancia de un nodo `X`, *Dominance Frontier DF*, es el conjunto de nodos `Y` tal que `X` domina a un predecesor inmediato de `Y` pero no domina estrictamente a `Y`.

```
algorithm DomFrontier (idomTree)
input:
    idomTree: dominance tree
begin
    /* traverse the CFG in reverse order */
    for each nodeN ∈ CFG do
        DF(N) = 0
        /* extract nodes dominated by N */
        doms = idomTree(N)
        /* first part: filter inmediate dominators */
        for each X ∈ Succ(N) do
            if X ∉ doms then DF(N) = DF(N) ∪ X
        end
        /* second part: extracting nodes from DF of children */
        for each Y ∈ doms do
            takeDF(Y)
            for each Z ∈ DF(Y) do
                if Z ∉ doms then DF(N) = DF(N) ∪ Z
            end
        end
    end
end
```

Figura 2.7: Algoritmo para el cálculo de las fronteras de dominancia.

El algoritmo de la Fig. 2.7, *DomFrontier*, recibe a la entrada, el árbol de dominancia para cada procedimiento del programa. Estos árboles se han calculado previamente siguiendo el algoritmo de la Fig. 2.5. En la primera parte del algoritmo, se añaden algunos de los sucesores de cada nodo CFG que contribuyen a la frontera de dominancia. En la segunda parte, contribuyen también al conjunto `DF` las fronteras de dominancia de los hijos del nodo CFG en cuestión. Por esta razón hay que hacer un recorrido inverso del CFG, porque para calcular la `DF` de un nodo padre, deben haberse calculado antes los conjuntos `DF` de sus nodos hijo.

2.3.2. Segunda etapa: introducción de funciones *phi*

Después de calcular las fronteras de dominancia, pasamos a la inserción de las llamadas sentencias *phi*. Las sentencias *phi* se introducen al comienzo de cada nodo del CFG, son necesarias cuando diferentes definiciones de una variable alcanzan un mismo punto desde distintos caminos. Todas las versiones de una misma variable se engloban entonces por lo que se conoce como función *phi* y dicha variable es renombrada para unificarlas.

Para la inserción de las funciones o sentencias `phi` se ha seguido el algoritmo de la Fig. 2.8. El algoritmo recibe tanto el grafo de control de flujo como el conjunto de las fronteras de dominancia,

```
algorithm PlacePhiStmts (CFG,DF)
input:
    CFG = [N ∪ Entry, Exit, E]: control flow graph of the procedure
    DF : set of dominance frontiers
begin
    /* initialization */
    for each node N ∈ CFG do
        HasAlready(N) = 0
        Work(N) = 0
    end
    W = 0
    for each variable V do
        IterCount = IterCount + 1
        for each no ∈ A(V) do
            Work(no) = IterCount
            W = W ∪ no
        end
        while Wnotempty do
            take X from W
            for each d ∈ DF(X) do
                if HasAlready(d)<IterCount then
                    /* place phi-statement at d */
                    HasAlready(d) = IterCount
                    if Work(d) <IterCount then
                        Work(d) = IterCount
                        W = W ∪ d
                    end
                end
            end
        end
    end
end
```

Figura 2.8: Algoritmo para la inserción de sentencias phi.

`DF`, calculado anteriormente. Existen dos conjuntos que se inicializan para cada nodo CFG. El conjunto `Work` indica los nodos que ya han sido añadidos o visitados en la iteración actual sobre las variables. El conjunto `HasAlready` indica si una sentencia *phi* ha sido insertada para un nodo en concreto. Primero se hace un recorrido a través de los nodos del CFG, y luego a través de las variables. El conjunto `A(V)` contiene todos los nodos que definen a la variable `V`. El conjunto `W` contiene la lista de nodos CFG que ya han sido procesados. En cada iteración este conjunto se inicializa con el conjunto `A(V)` que contiene todos los nodos con definiciones de la variable `V`. Se utiliza un flag numérico para poder comparar con la cuenta de cada iteración actual. El algoritmo termina cuando la lista de nodos a procesar, `W`, está vacía.

En la Fig. 2.9 se muestra cómo quedaría el código de la Fig. 2.2 después de aplicar esta etapa del algoritmo SSA. A priori, siempre se insertan más funciones *phi* de las estrictamente necesarias. Esto se debe a que la forma SSA implementada es la conocida como *minimal SSA* y no garantiza necesariamente que el número de funciones *phi* generadas sea el mínimo. Por esta razón, siempre

```
S1:   clave = 1;
S2:   for (i=1;i<NMAX;i=i+1){
S3:      NMAX = phi (NMAX, NMAX);
S4:      clave = phi (clave, clave);
S5:      x = phi (x, x);
S6:      i = phi (i, i);
S7:      if (mod(i,2)==0){
S8:         if (clave){
S9:            x = 0;
S10:        }else{
S11:           x = 1;
S12:           clave = 0;
S13:        }
S14:     }else{
S15:        x = 23;
S16:     }
S17:     x = phi (x, x, x);
S18:     clave = phi (clave, clave, clave);
S19:  }
```

Figura 2.9: Código de la Fig. 2.2 tras aplicar el algoritmo de inserción.

es necesario un pase final de optimización que disminuya el número de funciones *phi*.

2.3.3. Tercera etapa: renombrado de las variables del código

Después de tener todas las funciones *phi* en el código, viene la etapa de renombrado de todas las variables del procedimiento. El algoritmo que hemos utilizado se muestra en la Fig. 2.10. Se trata de un algoritmo recursivo, que comienza analizando el nodo raíz del CFG y luego realiza llamadas recursivas a través de los nodos hijos en el árbol de dominancia. El procedimiento `SEARCH`, que se llama para cada nodo del árbol de dominancia, itera sobre las sentencias del nodo buscando las variables que son leídas y escritas dentro de cada sentencia. Utiliza unas pilas de lectura, $C(v)$, y escritura, $S(v)$, para cada variable, que han sido inicializadas previamente. Cada vez que una variable es renombrada, el índice es actualizado en cada una de estas pilas. Las sentencias *phi* son tratadas de forma separada. Son renombradas desde el CFG en vez de usar el árbol de dominancia. En el algoritmo original SSA, estas funciones eran renombradas tanto desde el árbol de dominancia como desde el CFG. Nosotros eliminamos esta redundancia en el renombrado, que conduce a más iteraciones para llegar al punto fijo del algoritmo, y decidimos renombrar las sentencias *phi* a partir de la información que proporcionan los nodos CFG predecesores sobre las variables implicadas en la sentencia. Esta mejora puede ser significativa si tenemos en cuenta que el número de funciones *phi* insertadas en un programa puede llegar a ser elevado en determinados códigos. Cuando salimos de la llamada recursiva y se realiza un recorrido hacia atrás en el CFG, es importante llevar un control de las variables que son escritas en cada rama, para mantener la coherencia de las lecturas en el programa.

En la Fig. 2.11 se muestra cómo quedaría el código de la Fig. 2.2 después de aplicar esta nueva etapa del algoritmo SSA. Observamos cómo algunas funciones *phi*, en `S3`, `S5` y `S18`, contienen argumentos redundantes. Esto es debido a que la transformación SSA mínima, como ya comentamos, puede insertar más sentencias *phi* de las necesarias, nunca menos. Por lo tanto, es necesaria

```
algorithm Rename(CFG,V,DT)
input:
    CFG = [N ∪ {Entry, Exit}, E]: control flow graph of the procedure
    V : set of variables in the procedure
    DT : dominator tree of the procedure
begin
    /* iteration over variables */
    for each variable v ∈ V do
        C(v) ← 0
        S(v) ← EmptyStack
    end
    call SEARCH(root node)
end

algorithm SEARCH (node X):
begin
    /* iteration over statements of X */
    for each statement A ∈ node X do
        if A is not a phi-function then
            /* read variables */
            for each variable v ∈ RHS(A) do
                rename v by v_i where i = Top(S(v))
            end
            /* written variables *)
            for each variable v ∈ LHS(A) do
                i ← C(v)
                rename v by v_i
                push i into S(v)
                C(v) ← i + 1
            end
        end
    end
    /* traversing sucessors in the CFG */
    for each node Y ∈ Succ(X) do
        j = position of X in predecessors list of Y
        for each function phi F ∈ Y do
            rename j − th operand v by v_i in RHS(F)
                where i = Top(S(v))
        end
    end
    /* traversing children in the dominator tree,DT */
    for each node Y ∈ Children(X) do
        /* ordered list by level */
        call SEARCH(Y)
    end
    /* restoration of variables
        in backward traverse through CFG */
    for each written variable v ∈ WrittenVar(X) do
        pop S(v)
    end
end
```

Figura 2.10: Algoritmo para el renombramiento de variables.

una etapa de optimización que elimine estas sentencias phi redundantes.

2.3.4. Optimización

Para solucionar este problema, se diseñó una optimización que elimina estas funciones *phi* sobrantes. El algoritmo, ver Fig. 2.12, funciona primero recorriendo el CFG para examinar las sentencias *phi* y comprobando si encajan con alguno de los patrones de redundancia que se describen a continuación. En caso afirmativo, la sentencia *phi* es eliminada y es necesario llevar a cabo un nuevo renombramiento de las variables. El motivo del renombramiento es que al eliminar funciones *phi* decrementamos el subíndice de las variables que se ven afectadas por esa función eliminada. Esto puede observarse comparando las figuras 2.11 y 2.13 para la variable *x*. Los patrones de redundancia a detectar serían:

- $x_2 = phi(x_1, x_1, ...)$
- $x_2 = phi(x, x_1, x...)$
- $x_2 = phi(..., x_2, ...)$

El primer patrón sería aquel en el que todos los argumentos tienen el mismo índice SSA, el segundo en el que hay algunos argumentos redundantes no renombrados y el tercero aquel en el que no hay más de una nueva definición de la variable. Por ejemplo, en la Fig. 2.11 la sentencia S3 seguiría el patrón tercero, la sentencia S5 encaja con el patrón segundo, mientras que la sentencia S18 sigue el primer patrón.

```
S1:  clave_0 = 1;
S2:  for (i_0=1;i_1<NMAX_0;i_2=(i_1+1)){
S3:       NMAX_1 = phi (NMAX_0, NMAX_1);
S4:       clave_1 = phi (clave_0, clave_3);
S5:       x_0 = phi (x, x_4);
S6:       i_1 = phi (i_0, i_2);
S7:       if (mod(i_1,2)==0){
S8:            if (clave_1){
S9:                 x_1 = 0;
S10:           }else{
S11:                x_2 = 1;
S12:                clave_2 = 0;
S13:           }
S14:      }else{
S15:           x_3 = 23;
S16:      }
S17:      x_4 = phi (x_3, x_1, x_2);
S18:      clave_3 = phi (clave_1, clave_1, clave_2);
S19: }
```

Figura 2.11: Código de la Fig. 2.2 tras aplicar el algoritmo de renombrado.

Este proceso de eliminación de sentencias *phi* redundantes se repite de forma iterativa hasta que no hay cambios en las sentencias del grafo CFG. El alcance del punto fijo está garantizado puesto

que no se introducen nuevas sentencias en el programa, sino que se va reduciendo el número de sentencias en cada pase.

algorithm DeleteExtraPhi (CFG)
input:
$CFG = [N \cup Entry, Exit, E]$: control flow graph of the procedure
begin
/* traverse CFG nodes in increasing order */
for each $nodeN \in CFG$ **do**
for each $statementS \in N$ **do**
if S is Phi-Statement **then**
/* check if it is redundant */
is-re = **isRedundant(S)**
if $is - re$ delete(S)
end
end
end
end
algorithm boolean isRedundant (Ph)
input:
$Ph : Phi - Statement\ to\ analyze$
begin
/* check pattern of the statement */
if $Ph\ has\ a\ redundant\ pattern$ **then**
return true
end end

Figura 2.12: Algoritmo para la eliminación de funciones phi redundantes.

```
S1:     clave_0 = 1;
S2:     for (i_0=1;i_1<NMAX_0;i_2=i_1+1){
S3:       clave_1=phi(clave_0,clave_3);
S4:       i_1 = phi(i_0, i_2);
S5:       if (mod(i_1,2)==0){
S6:           if (clave_1){
S7:              x_0 = 0;
S8:           }else{
S9:              x_1 = 1;
S10:             clave_2 = 0;
S11:          }
S12:      }else{
S13:          x_2 = 23;
S14:      }
S15:     x_3 = phi(x_2, x_1, x_0);
S16:     clave_3=phi(clave_1,clave_2);
S17:    }
```

Figura 2.13: Forma SSA definitiva del código de la Fig. 2.2.

En la figura Fig. 2.13 presentamos la transformación final del código de la Fig. 2.2 con la representación SSA. Observamos como han desaparecido algunas funciones `phi`, o argumentos,

de las que antes habían sido identificadas como redundantes, y en consecuencia ha cambiado el renombrado de algunas variables.

2.4. Extensión interprocedural (ISSA)

La forma SSA puede extenderse desde el ámbito local de cada procedimiento al ámbito global que abarca todo el programa. Entonces se llama *Interprocedural Static Single Assignment*, ISSA. La finalidad de la forma ISSA es que cada variable tenga un identificador único, no sólo dentro de cada método sino en todo el programa. Existen diversas propuestas para transformar el código en esta nueva representación partiendo de la forma SSA clásica. La idea más extendida, consiste en la incorporación de unas sentencias especiales en algunos puntos del programa:

- Después de cada llamada a función. El propósito de este tipo de sentencias es el mapeo entre los parámetros formales y reales cuando se regresa del procedimiento que es llamado. Para los parámetros formales, necesitamos el último indentificador usado para los mismos.

- Al inicio de cada procedimiento que es llamado. La funcionalidad es justo la inversa a la discutida en el párrafo anterior. Se condensan en una misma sentencia todos los parámetros reales (uno por cada llamada al procedimiento desde otros puntos del programa), conectándolos con el parámetro formal correspondiente. Tienen una apariencia similar a las funciones *phi* clásicas de la forma SSA. Nosotros las hemos llamado *sentencias phi Interprocedurales*, *Iphi*.

Nos basamos en la técnica más ampliamente utilizada [30], para implementar la representación ISSA a partir de la extensión de la forma SSA. Tal como se ha explicado anteriormente, la técnica se basa en la introducción de nuevas sentencias en determinados puntos del programa. De este modo, hemos incorporado una serie de métodos dentro del proceso de transformación SSA, encargados de recolectar la información sobre los parámetros reales y formales de cada procedimiento. Esta información se utiliza para la creación de las nuevas sentencias que nosotros llamamos *sentencias interprocedurales*.

En la figura Fig. 2.14(a) y Fig. 2.14(b), se muestra un código simple escrito en C y en su correspondiente forma ISSA. Las sentencias `S4`, `S5`, `S6`, `S8`, `S9`, `S18`, `S19` y `S20` serían del primer tipo. Las sentencias `S10`, `S11`, `S15` y `S16` del segundo, sentencias *Iphi*. Como se observa, las variables poseen dos subíndices con esta representación, el primero corresponde al renombramiento de la forma SSA clásica y el segundo un identificador del procedimiento al que pertenece esa variable.

2.5. Análisis de Forma

El análisis de forma es una técnica de análisis estático que en tiempo de compilación obtiene información detallada acerca del *heap* en programas basados en punteros. Esto se hace extrayendo información acerca de la *forma* o la conectividad de los elementos del *heap*.

```
int main (void){
   node *a, *b, *c;
   a=malloc();
   b=malloc();
   c=f(a,b);
   g(a,b);
}
node f (node *x,node *y){
   x->nxt=y;
   x=y;
   return x;
}
void g (node *s,node *t){
   node *v;
   v=f(s,t);
}
```

(a)

```
int main (void){
S1:   a_0_1=malloc();
S2:   b_0_1=malloc();
S3:   c_0_1=f(a_0_1, b_0_1);
S4:   c_0_1=x_1_0;
S5:   a_1_1=x_1_0;
S6:   b_1_1=y_0_0;
S7:   g(a_1_1,b_1_1);
S8:   a_2_1=s_1_2;
S9:   b_2_1=t_1_2;
}
node f (node *x, node *y){
S10:  x_0_0=Iphi(a_0_1, s_0_2);
S11:  y_0_0=Iphi(b_0_1, t_0_2);
S12:  x_0_0->nxt=y_0_0;
S13:  x_1_0=y_0_0;
S14:  return x_1_0;
}
void g (node *s, node *t){
S15:  s_0_2=Iphi(a_1_1);
S16:  t_0_2=Iphi(b_1_1);
S17:  v_0_1=f(s_0_2, t_0_2);
S18:  v_0_1=x_1_0;
S19:  s_1_2=x_1_0;
S20:  t_1_2=y_0_0;
```

(b)

Figura 2.14: (a) Código de un programa simple; (b) Forma ISSA del código.

```
Item *items[n];
for (int i=0; i<n; i++){
     S1:items[i] = newItem(...);
}
S2:process\_items(items);
for (int i=0; i<n; i++) {
     S3:delete\_items[i];
}
```

Figura 2.15: Código ejemplo para el análisis de forma.

Consideremos por ejemplo el programa de la Fig. 2.15. Este programa construye un array de objetos, los procesa de forma arbitraria y luego los borra. Suponiendo que la función *process_items* está libre de errores, resulta evidente que el programa es seguro: nunca referencia memoria liberada y borra todos los objetos que han sido construidos. Sin embargo, la mayoría de los análisis de punteros tienen dificultades en analizar este código de forma precisa. Para obtener información de los punteros, el análisis llevado a cabo debe ser capaz de nombrar los objetos del programa. En general, los programas pueden reservar un número indeterminado de objetos; pero para acabar el análisis, sólo puede tomarse un conjunto finito de nombres. Una aproximación típica consiste en dar a todos los objetos reservados en una misma línea del programa el mismo nombre. En el ejemplo anterior, todos los objetos construidos en la línea `S1` tendrían el mismo nombre. Además, cuando la sentencia de borrado es analizada por primera vez, el análisis determina que uno de

los objetos nombrados en S1 va a ser borrado. La segunda vez que se analiza esta sentencia (al estar dentro de un bucle) el análisis nos avisaría de un posible error: ya que es imposible distinguir los objetos dentro del array, podría darse el caso de estar borrando el mismo objeto que en el borrado anterior. Este falso error es un espúreo, y la meta del análisis de forma es evitar este tipo de imprecisiones.

2.5.1. Sumarización y materialización

El análisis de forma resuelve problemas del análisis de punteros usando un sistema de renombrado flexible para los objetos. En vez de dar a un objeto el mismo nombre a lo largo de todo el programa, los objetos pueden cambiar de nombre dependiendo de las acciones del programa. A veces, varios objetos diferentes con nombres distintos pueden ser englobados en un mismo nombre. A continuación, cuando el objeto sumarizado va a ser usado por el programa, éste puede materializarse, es decir, desplegarse en dos objetos con nombres distintos, uno representando un objeto sencillo y el otro representando el resto de objetos sumarizados. De esta forma, los objetos que están siendo usados por el programa son representados usando objetos materializados únicos, mientras que los objetos que no son usados permanecen sumarizados. El array de objetos del ejemplo anterior sería sumarizado de forma diferente según la sentencia de código en la que nos encontremos (S1, S2 o S3). En la sentencia `S1`, el array está aún parcialmente construido. Los elementos del array del `0... i-1` contienen objetos construidos. El elemento `i` del array va a ser construido, y los siguientes elementos están por inicializar. Un análisis de forma puede aproximar esta situación usando un sumario para el primer conjunto de elementos, una localización de memoria materializada para el elemento `i` y un sumario para el resto de las localizaciones todavía no inicializadas de la forma siguiente:

0 .. i-1	**i**	**i+1 .. n**
puntero al objeto construido(sumario)	no-inicializado	no-inicializado(sumario)

Después de que el bucle termina, en la sentencia `S2`, no hay necesidad de materializar nada. El análisis de forma determina en este punto que todos los elementos del array han sido inicializados.

0 .. n
puntero al objeto construido(sumario)

En la línea `S3`, sin embargo, el elemento del array `i` está en uso de nuevo. De nuevo el análisis divide el array en tres segmentos como en `S1`. Esta vez, sin embargo, el primer elemento antes de `i` ha sido eliminado y el resto de elementos son todavía válidos (asumiendo que la sentencia de borrado no se ha ejecutado todavía).

0 .. i-1	**i**	**i+1 .. n**
libre(sumario)	puntero a objeto construído	puntero a objeto construído(sumario)

En este caso, el análisis identifica que el puntero en el índice `i` no ha sido borrado todavía. Gracias a este aumento de precisión de los objetos, un análisis posterior no produciría ningún aviso de posible error.

La información derivada del análisis de forma en una aplicación basada en punteros puede usarse para varios propósitos como: (i) análisis de dependencias de datos, determinando si dos accesos pueden alcanzar la misma localización de memoria; (ii) explotación de localidad, capturando el modo en que se recorren las localizaciones de memoria para determinar cuando es probable que estén contiguas en memoria; (iii) verificación de programas, para proporcionar garantías de corrección en programas que manipulan el *heap*; y (iv) soporte al programador, para ayudar en la detección de un uso incorrecto de punteros o documentar estructuras de datos complejas.

2.6. Patrones de recorrido y caminos de acceso

Las expresiones para los caminos de acceso están compuestas por un lado del puntero que es referenciado y por otro los nombres de los campos conectados. Los campos accedidos están conectados a través del operador "$\rightarrow$". El puntero de la expresión del camino de acceso sería la entrada del camino y los campos la cadena del camino. Por ejemplo, para una estructura tipo `node` tenemos la expresión de camino de acceso $l \rightarrow nxt$, l sería la entrada al camino de acceso y `nxt` sería la cadena del camino. Podemos encontrarnos el mismo campo accedido varias veces en la expresión, en tal caso puede aplicarse el operador "$+$"para indicar que el campo se repite una o más veces. En el ejemplo anterior podríamos encontrarnos la expresión $l \rightarrow (nxt)^+$, lo que englobaría expresiones como $l \rightarrow nxt$, $l \rightarrow nxt \rightarrow nxt$, etc.

Para el cálculo de las expresiones de los caminos de acceso, se examinan las sentencias de asignación de punteros, a través de un recorrido hacia atrás del programa. En la Fig. 2.16 se presenta la secuencia de sentencias que recorren una lista simplemente enlazada y las expresiones de los caminos de acceso para cada sentencia. Las sentencias se examinan hacia atrás desde el final del programa y los caminos de acceso se van calculando siguiendo ese orden. El cálculo comenzaría en la sentencia `S3`, la expresión del camino de acceso $r, q \rightarrow n$ se pasaría a la sentencia predecesora de `S3`. En `S2`, se computa un nuevo camino de acceso $q, p \rightarrow n$. Al mismo tiempo, la expresión del camino $r, q \rightarrow n$ se transformaría debido a la sentencia `S2`, que define `q`. Quedaría pues la nueva expresión como $r, p \rightarrow n \rightarrow n$. Finalmente en `S1` se transformarán las expresiones correspondientes reemplazando `p` por `list`, como se muetra en la Fig. 2.16(c).

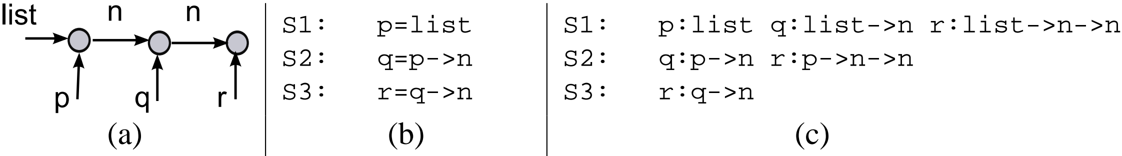


Figura 2.16: (a) Lista;(b) Programa;(c) Expresiones de camino

Este ejemplo demuestra que las expresiones de los caminos de acceso pueden mostrar las posiciones relativa de los punteros en estructuras dinámicas de datos. Además, las expresiones también representan el patrón de recorrido del programa, pudiendo ser utilizadas para determinar si dos expresiones de caminos pueden alcanzar las mismas localizaciones.

2.7. Análisis de Dependencias

Recordamos dos tipos de dependencias, las dependencias de control y las dependencias de datos.

2.7.1. Dependencias de control

Se establece una dependencia de control entre dos partes del programa, cuando la ejecución de una instrucción necesita de la ejecución previa de otra instrucción. Por ejemplo en la Fig. 2.17, la sentencia `S4` sólo se ejecuta si el predicado de la sentencia `S1` es falso. Por lo tanto, existe una dependencia de control entre la sentencia `S1` y `S4`.

```
S1:   if (x > 2) then{
S2:      z = y + 1;
S3:   }else{
S4:      y = 3;
   }
```

Figura 2.17: Dependencia de control

2.7.2. Dependencias de datos

Una dependencia de datos surge cuando dos sentencias acceden o modifican la misma fuente.

Dependencia de flujo
Una sentencia `S2` tiene una dependencia de flujo respecto a otra sentencia `S1`, si y sólo si, `S1` modifica una fuente que `S2` lee, y la ejecución de `S1` precede a la de `S2`. En la Fig. 2.18 tenemos un ejemplo de dependencia de flujo entre `S1` y `S2` tipo *RAW*, *Read After Write*.

```
S1:   x = 10;
S2:   y = x + 1;
```

Figura 2.18: Dependencia de flujo

Antidependencia
Una sentencia `S2` es antidependiente de otra `S1`, si y sólo si `S2` modifica una fuente que lee `S1` y la sentencia `S1` precede a `S2` en ejecución. En la Fig. 2.19 mostramos un ejemplo de antidependencia, *WAR, Write After Read*. La sentencia `S2` establece el valor de `y` pero la sentencia `S1` lee un valor previo de dicha variable.

```
S1:   x = y + 1;
S2:   y = 10;
```

Figura 2.19: Antidependencia

Dependencia de salida
Una sentencia `S2` tiene una dependencia de salida respecto a otra sentencia `S1`, si y sólo si, `S1` y `S2` modifican la misma fuente, y `S1` precede a `S2` en ejecución. En la Fig. 2.20 mostramos un

ejemplo de dependencia de salida, *WAW* (Write After Write). Tanto `S1` como `S2` escriben la misma variable `x`.

```
S1:   x = 10;
S2:   x = 20;
```

Figura 2.20: Dependencia de salida

Dependencia de entrada

Una sentencia `S2` tiene una dependencia de entrada respecto a otra sentencia `S1`, si y sólo si `S1` y `S2` leen de la misma fuente, y `S1` precede a `S2` en ejecución. En la Fig. 2.21, tanto `S1` como `S2` acceden a la variable `x`. Esta dependencia no es problemática, ya que no impide el reordenamiento de las instrucciones.

```
S1:   y = x + 3;
S2:   z = x + 5;
```

Figura 2.21: Dependencia de entrada

Dependencia acarreada por lazo

Dos sentencias dentro de un bucle tienen una dependencia acarreada por lazo, *LCD*, *Loop Carried Dependence*, si una localización de memoria que es accedida por una sentencia en una iteración dada, es accedida por otra sentencia en otra iteración futura, con uno de los accesos siendo de escritura. En el siguiente ejemplo existe una dependencia de flujo entre las iteraciones, *i* e *i+1*, del bucle.

```
Do i=1...n
   S1:   A(i+1) = A(i) * B(i);
end do
```

Figura 2.22: Dependencia acarreada por lazo

3

Análisis de cadenas de definición y uso

3.1. Introducción

En este capítulo presentamos un completo algoritmo para la recolección de cadenas DU debidas a punteros, en programas basados en estructuras dinámicas de datos. Se presentan dos técnicas diferentes de acuerdo al tipo de cadenas DU a detectar: por un lado las cadenas DU debidas a punteros que apuntan al *stack* y por otro las cadenas DU debidas a punteros que apuntan al *heap*.

- *Algoritmo DU link*: se trata de un algoritmo complejo, que consta de varias fases y que obtiene toda la información referente a los recorridos en las estructuras de datos dinámicas del programa. Calcula las cadenas DU intraprocedurales e interprocedurales debidas a punteros al *heap* con el formato *pointer→ field*.
- *Algoritmo DU ISSA*: basado en la representación ISSA del código, permite calcular todas las cadenas DU intraprocedurales e interprocedurales debidas a los punteros al *stack*.

Usamos el ejemplo de la figura Fig. 3.1 para indicar los tipos de cadenas DU que se detectarían con cada técnica. La técnica *DU link* detectaría en el procedimiento *main* la cadena DU entre

```
              void f(struct node *x, struct node *y){
                  struct node *z, *t;
S1:           z = (struct node *) malloc(sizeof(struct node));
S2:           x->nxt = z;
S3:           t = y->nxt;
              }

              void main(){
                  struct node *p, *r, *q;
S4:           p = (struct node *) malloc(sizeof(struct node));
S5:           q = (struct node *) malloc(sizeof(struct node));
S6:           p->nxt = q;
S7:           r = p->nxt;
S8:           f(r,q);
              }
```

Figura 3.1: Programa simple escrito en C.

las sentencias `S6` y `S7` debido a $p\rightarrow nxt$, y también detectaría la cadena DU entre las sentencias `S2` y `S3` del procedimiento `f`. Esta última cadena DU, `S2-S3`, no sería posible detectarla sin la información de alias entre ambos punteros que proviene del método `main`. Con la técnica *DU ISSA*, detectaríamos las cadenas DU `S1-S2` debida a `z`, y `S5-S6` debida a `q`.

Ambas técnicas han sido desarrolladas en Java usando la plataforma Cetus. En primer lugar vamos a explicar su funcionamiento y posteriormente presentaremos algunos experimentos con códigos reales.

3.2. Algoritmo DU link

Este algoritmo extiende la técnica de Hwang y Saltz [30] para el cálculo de cadenas DU intra e interprocedurales en códigos que crean y recorren estructuras dinámicas de datos.

El principal objetivo es encontrar para cada sentencia que define un puntero al *heap* en la estructura de datos, el conjunto de sentencias que usan el puntero definido por dicha sentencia. Se trata de un análisis de flujo de datos iterativo, que primero recolecta la información local tipo DU, así como la información de *alias* dentro de cada procedimiento. Luego, esta información local se propaga a través de un grafo especial interprocedural y finalmente se obtienen las cadenas DU del programa con un pase final interprocedural sobre cada procedimiento. Gracias a esta técnica, no sólo es posible detectar las cadenas DU intraprocedurales, sino también las interprocedurales debidas a los punteros al *heap*.

3.2.1. Preproceso del código de entrada previo a la aplicación del algoritmo

Antes de poner en marcha la técnica *DU-link*, necesitamos aplicar un preproceso al código de entrada. Resumimos pues todos los pasos por los que debe pasar el código de entrada, antes de aplicar el algoritmo *DU-link*, en el esquema de la Fig. 3.2.

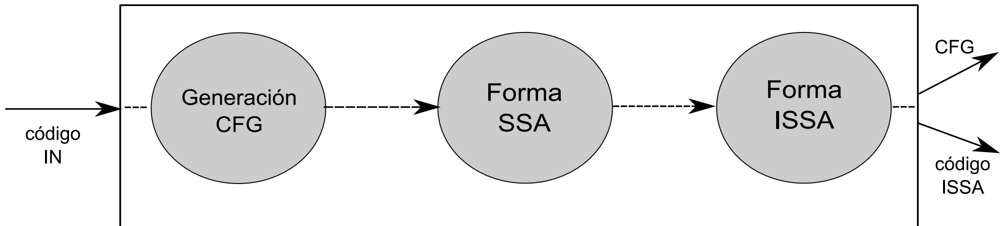


Figura 3.2: Pasos por los que debe pasar el código de entrada.

Hemos adaptado tanto nuestra plataforma de trabajo, Cetus, para que ésta genere cierta información que se necesita para aplicar la técnica *DU-link*. La adaptación de Cetus ha sido posible gracias a que se trata de una infraestructura extensible, de forma que hemos implementado un nuevo pase para la técnica *DU-link*. La información que se genera en estos pases es la siguiente:

- *Grafo de Control del Flujo*: Es necesario un grafo de control del flujo del programa. Hemos tomado el llamado Control Flow Graph (CFG), que la herramienta Cetus ya nos proporcionaba como punto de partida para comenzar a trabajar. Este CFG de Cetus ha sido extendido con nuevas propiedades de utilidad para la técnica *DU-link*, como por ejemplo el nivel de profundidad de cada bloque dentro del CFG. Además se añadieron nuevos parámetros necesarios para poder extender el grafo con la forma SSA y la forma ISSA.
- *ISSA*: Cada variable del código debe tener un identificador global en todo el programa. Recordamos del capítulo 2 que la forma SSA nos proporciona un nombre único para cada variable, pero sólo de forma local a cada procedimiento. Por lo tanto recurrimos a la representación ISSA, para cumplir con este requisito. La transformación del código a la representación ISSA se lleva a cabo siguiendo la técnica introducida en el capítulo 2. Primero se obtiene la forma SSA del código y luego se extiende a la forma ISSA.

3.2.2. Descripción del algoritmo *DU-link*

Tal como dijimos anteriormente, el algoritmo puede dividirse en cuatro fases bien diferenciadas, presentadas en la figura Fig. 3.3.

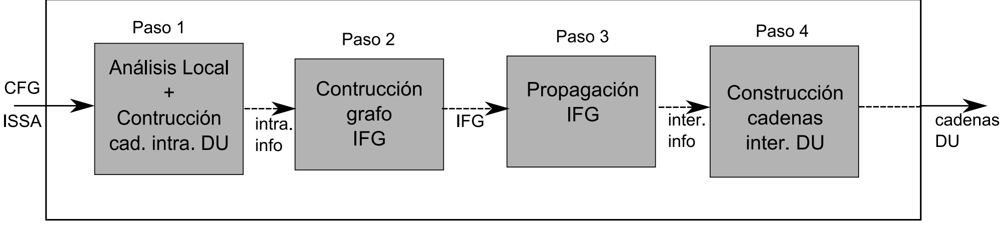


Figura 3.3: Pasos en el algoritmo DU link.

Etapa 1: Análisis Intraprocedural

En el paso 1 se realiza un análisis intraprocedural sobre cada procedimiento local del programa. Durante esta etapa, se recopila toda la información necesaria para la construcción del llamado *grafo*

interprocedural de flujo, IFG, al mismo tiempo que se capturan las cadenas DU intraprocedurales dentro de cada procedimiento.

Dentro del análisis intraprocedural cobran especial importancia las llamadas *tuplas*. Una *tupla* es un objeto asociado a una sentencia, y dentro de ésta, a una serie de variables que son bien definidas o bien referenciadas en la sentencia. Gracias a las tuplas, reunimos información sobre el tipo de acceso que se realiza en cada sentencia del programa. El conjunto de parámetros que se maneja en cada tupla, varía en función del tipo de sentencia, distinguiendo principalmente tres:

- definición: $< id\,block, id\;stmt, left\;expr, right\;expr >$
- uso: $< id\,block, id\;stmt, expr >$
- uso virtual: $< id\,block, id\;stmt, final\;expr, accessed\,pointer\;field, origin\;expr >$

Las tuplas contienen un par de campos *id block* e *id stmt* que son identificadores numéricos que nos sirven para encontrar la sentencia correspondiente en el CFG. Gracias a este identificador, podemos localizar la sentencia que contiene a una determinada expresión de una tupla de definición o de uso. Para el caso de las definiciones, se almacenan por separado las expresiones a cada lado (izquierda y derecha), *left expr,right expr*, de la asignación. Por ejemplo, para una sentencia tal que *q→ left=y* se crea una tupla de definición con la forma *<id block,id stmt,q→ left,y>*. En los usos se almacena sólo la expresión de la variable que se está leyendo. Por ejemplo, en una sentencia tipo *p=x→ next*, se crearía una tupla de uso tal que *<id block,id stmt,x→ next>*. Existe además un conjunto especial de usos, llamados *usos virtuales*. Se trata de usos ficticios creados a partir de los parámetros formales y reales de las funciones y llamadas a función. Son introducidos en determinados puntos del programa antes del análisis local que se lleva a cabo en la primera etapa del algoritmo, y una vez finalizado éste, nos dan la información de cómo son transformados estos parámetros formales y reales dentro de cada función. Concretamente se introducen en los nodos CFG de salida y las llamadas a función. Así pues, son creados expresamente para obtener información necesaria en la construcción de los enlaces IFG. Por eso existen dos campos diferenciados, uno con la expresión original introducida antes del análisis, y otro con la expresión final después de terminar éste. Sea el siguiente ejemplo:

```
function F (node *q){
    S0: p_0 = malloc();
    S1: p_0 -> next = q_0;
    S2: p_1 = p_0 -> next;
}
```

Como para éste y los sucesivos ejemplos no vamos a utilizar la información del identificador de bloque en las explicaciones, *id block*, hemos omitido este campo en las tuplas que presentaremos. En la sentencia `S2` se crearía una tupla de uso $< S2, p_0 \rightarrow next >$ y en la sentencia `S1` se crearía una de definición tal que $< S1, p_0 \rightarrow next, q_0 >$. Además tenemos que la variable `q` es un parámetro formal que apunta a una estructura dinámica. Se crearía un uso virtual para la variable `q` de la forma $< S2', q_0, next, q_0 >$ (donde `S2'` representa el nodo EXIT del procedimiento) para lo cual ha sido necesario determinar el índice SSA de dicha variable en el nodo CFG de salida de la función `F` y qué punteros al *heap* pueden ser expresados a través de esta variable. Si existiera

más de un posible puntero al *heap* para el mismo parámetro formal, se crearía un uso virtual por cada opción posible.

El algoritmo (ver Fig. 3.4) recibe, a la entrada, el CFG del procedimiento a analizar con el código ya transformado en la forma ISSA, y ofrece, a la salida, un listado de las cadenas DU intraprocedurales que se han detectado. El recorrido del CFG se efectúa a nivel de sentencias, línea 1 del algoritmo, definiendo tres tipos de conjuntos para cada una, tanto para los usos como para las definiciones: de salida ($UPEXP_{out}$ y $UPDEF_{out}$), intermedio ($UPEXP$ y $UPDEF$) y de entrada ($UPEXP_{in}$ y $UPDEF_{in}$). Para el caso de los usos virtuales, se crean sólo conjuntos de entrada y salida ($DummyUse_{in}$ y $DummyUse_{out}$). Se realiza un recorrido inverso en profundidad, línea 4 del algoritmo, desde el nodo de salida hacia el nodo de entrada del CFG. Cada conjunto de salida, ($UPEXP_{out}$, $UPDEF_{out}$ y $DummyUse_{out}$), recibe la información que se va propagando desde otras sentencias sucesoras, lo que se representa en las líneas 5 y 6 del algoritmo. Los conjuntos intermedios, ($UPEXP$ y $UPDEF$), procesan esta información añadiendo los nuevos usos o definiciones de la propia sentencia, líneas 8 y 9. Justo en este punto a parece una de las funciones que se encarga de transformar las tuplas recibidas, la función `Fs`.

Función Fs

La función `Fs` se encarga de efectuar las transformaciones oportunas debidas a cada sentencia de definición que se visita durante el recorrido. Transforma tanto a tuplas de definiciones, como de usos (virtuales y no virtuales). La función de transferencia que explicamos a continuación se ha especificado para transformar a tuplas de usos, pero puede aplicarse fácilmente a las tuplas de definición como se verá a continuación

$$Fs_{<\text{left}_e = right_e>} = \begin{cases} DUchain & \text{si } use_e = \text{left}_e \ \& \ \neg usovirtual \\ right_e & \text{si } use_e = \text{left}_e \ \& \ usovirtual \\ right_e \rightarrow f & \text{si } use_e = \text{left}_e \rightarrow \text{f} \\ use_e & \text{si } otherwise \end{cases}$$

Representamos a la expresión de la sentencia de definición que va a transformar a las tuplas como $\text{left}_e = right_e$, siendo left_e la expresión del lado izquierdo y $right_e$ la del lado derecho de la sentencia. Por otro lado, tenemos una tupla de uso cuya expresión es use_e. Esta expresión puede pertenecer bien a una tupla de uso normal o de uso virtual. Por esta razón, en la función de transferencia se muestran los distintos casos de coincidencia dependiendo de si es de un tipo o de otro. Si la expresión de la tupla de uso, use_e coincide exactamente con la expresión de la definición, $use_e = \text{left}_e$, entonces se detecta una cadena DU caso de ser una tupla de uso normal (primer caso de la función de transferencia), o bien simplemente se transforma la expresión en caso de tratarse de un uso virtual (segundo caso de la función de transferencia). Si la coincidencia es parcial (use_e no es exactamente igual a left_e) entonces simplemente se transforma la expresión de la tupla de uso, sea virtual o no (caso tercero en la función de transferencia). Por último, en el caso de que no haya coincidencia, se deja la tupla tal como estaba (cuarto caso en la función de transferencia). Cuando en vez de una tupla de uso, se aplica la función `Fs` sobre una tubla de definición, entonces tendríamos dos expresiones a analizar. Si definimos el formato de la expresión de la tupla de definición como $expression1 = expression2$, se aplicaría la misma función de transferencia descrita anteriormente a cada expresión por separado. En este caso, no se preguntaría si es uso virtual o no, puesto que se trata de una tupla de definición. Es necesaria una comprobación final después de transformar la tupla de definición para no permitir definiciones de tipo $a = a$.

Las definiciones que transforman pueden ser de tres tipos diferentes:

1. definición de un puntero al *heap*: $p_i \rightarrow \text{f} = n_j$
2. definición de un puntero en el *stack*: $p_i = q_j$ o $p_i = q_j \rightarrow f$
3. definición tipo phi: $p_i = phi(p_{i1}, p_{i2}, ...)$

Para las definiciones tipo 3, en caso de coincidencia simplemente se sustituye la tupla recibida por tantas copias como argumentos tenga la función *phi* en el lado derecho. En el ejemplo siguiente:

```
while (p_1!= NULL){
    S0: p_1 = phi (p_0, p_2);       usos: <S2,p_0->next>,<S2,p_2->next>
    S1: p_1 -> next = y_1;          usos: <S2,p_1->next>
    S2: p_2 = p_1 -> next;
}
```

Vemos como la tupla de uso creada en `S2`, alcanzaría la función *phi* y se desdoblaría en dos tuplas, una por cada argumento. La primera de ellas seguiría su propagación fuera del bucle y la otra dentro. Para las definiciones tipo 1 y 2, mostramos el mismo ejemplo anterior pero extendiendo algunas sentencias:

```
S0: q_0 -> next = y_0;          DU-chain detected S0-S4
S1: p_0 = q_0;                  usos: <S4,q_0->next>
    while (p_1!= NULL){
S2:     p_1 = phi (p_0, p_2);   usos: <S4,p_0->next>, <S4,p_2->next>
S3:     p_1 -> next = y_1;      usos: <S4,p_1->next>
S4:     p_2 = p_1 -> next;
    }
```

El uso de la sentencia `S4` alcanza la sentencia de definición en `S1`, que es una definición de tipo 2, al aplicar la función `Fs` y existir coincidencia parcial, la tupla de uso se transforma. Al alcanzar la sentencia `S0`, que es una definición de tipo 1, aplicaríamos de nuevo la función `Fs` y ahora al existir coincidencia total entre la tupla de uso y la tupla de definición que transforma, se detectaría una cadena DU.

Función Fupdef

En las líneas 11 y 12 del algoritmo de la Fig.3.4 aparece otra función nombrada como `Fupdef`. Antes de explicar su función de transferencia, vamos a tratar de justificar por qué se utiliza en el algoritmo otra función además de `Fs` para transformar las tuplas. Cuando el orden del recorrido de los enlaces difiere del orden de la construcción de los mismos, pueden aparecer casos donde la función `Fs` no sea suficiente para capturar todos los alias. Este problema se debe a que hay información de alias que puede no estar disponible cuando se aplica la función `Fs`, por lo que se hace necesaria la propagación también de las tuplas de definición y aplicar una función adicional, que llamamos `Fupdef` para que estas tuplas de definición también transformen a las otras tuplas. La aplicación de la función `Fupdef` vamos a entenderla mejor con el siguiente ejemplo:

```
S0: p_1 -> next = q_0;      usos: DU chain S0-S1, <S3,q_0->next>
S1: r_0 = p_1 -> next;      usos: <S1,p_1->next>, <S3,q_0->next>
S2: r_0 -> next = y_0;      usos: <S3,q_0->next>
S3: t_0 = q_0 -> next;      usos: <S3,q_0->next>
```

La tupla de uso en `S1` alcanza la tupla de definición en `S0` y se detecta la cadena DU. Sin embargo, el alias que existe entre las variables `r` y `q`, establecido a través de las sentencias `S0` y `S1`, no se detecta, y la cadena DU que existe entre `S2` y `S3` no es capturada. Si además de propagar las tuplas de uso, propagamos las tuplas de definición, y usamos la función `Fupdef` con las tuplas de definición propagadas, entonces ocurre lo que se muestra en la tabla 3.1.

Sentencia	**Usos**	**Definiciones**
$S0 : p_1 \rightarrow next = q_0;$	$< S3, q_0 \rightarrow next >$	$< S2, q_0 \rightarrow next = y_0 >$
$S1 : r_0 = p_1 \rightarrow next;$	$< S1, p_1 \rightarrow next >, < S3, q_0 \rightarrow next >$	$< S2, p_1- > next- > next = y_0 >$
$S2 : r_0 \rightarrow next = y_0;$	$< S3, q_0 \rightarrow next >$	$< S2, r_0 \rightarrow next = y_0 >$
$S3 : t_0 = q_0 \rightarrow next;$	$< S3, q_0 \rightarrow next >$	$-$

Tabla 3.1: Ejemplo de propagación de tuplas de uso y definición

Igual que en el caso anterior se detectaría la cadena DU, al aplicar la función `Fs`, en la sentencia `S0` entre `S0` y `S1`. Sin embargo, ahora se propagan también las tuplas de definición, concretamente la tupla de la sentencia `S2`. Esta tupla, alcanza la sentencia `S0` como $< S2, q_0 \rightarrow next = y_0 >$ y aquí es donde entra en juego el papel de la función `Fupdef`. Esta función tomaría la sentencia de definición `S2` propagada y aplicaría una transformación sobre las sentencias de uso también propagadas en ese punto del programa, del mismo modo que lo hace la función `Fs`. El resultado es que se detectaría una cadena DU entre `S2` y `S3` gracias a la función `Fupdef`.

$$Fupdef_{<\text{left}_e = right_e>} = \begin{cases} DUchain & \text{si } use_e = \text{left}_e \ \& \ tuple\text{_def} >> tuple_use \\ DUchain, use_e & \text{si } use_e = \text{left}_e \ \& \ tuple\text{_def} > tuple_use \\ right_e, use_e & \text{si } use_e = \text{left}_e \ \& \ tuple\text{_def} > tuple_use \\ right_e \rightarrow f & \text{si } use_e = \text{left}_e \rightarrow \text{f} \ \& \ tuple\text{_def} >> tuple_use \\ right_e \rightarrow \text{f} \ \& \ use_e & \text{si } use_e = \text{left}_e \rightarrow \text{f} \ \& \ tuple\text{_def} > tuple_use \\ use_e & \text{si } otherwise \end{cases}$$

La función de transferencia `Fupdef` realiza una transformación similar a `Fs`, pero aquí no se tratan los usos virtuales y, como novedad, se comprueba el tipo de dominancia entre la tupla de definición que va a hacer la transformación y la tupla a transformar (de uso o definición). Esta dominancia puede ser estricta ($>>$), o no ($>$). Representamos de nuevo a la expresión de la tupla de definición genérica como $\text{left}_e = right_e$, siendo left_e la expresión del lado izquierdo de la asignación y $right_e$ la del lado derecho. Se muestran las diferentes transformaciones que sufriría una tupla de uso conteniendo la expresión use_e. Si la coincidencia con la definición es total, entonces se detectaría una cadena DU. Si la coincidencia es parcial, no se trata de la definición exacta, entonces simplemente se transforma la expresión. En cualquiera de los dos casos anteriores, si la dominancia no es estricta entonces se mantiene además una tupla original intacta, sin transformar. En el caso de que no haya coincidencia, se deja la tupla tal como estaba. La transformación de las tuplas de definición se realiza utilizando esta misma función de transferencia, pero analizando por separado la expresión de la izquierda y la de la derecha en la asignación. De nuevo es necesaria una comprobación final después de transformar cada tupla de definición, para no permitir definiciones de tipo $a = a$.

Continuando con la explicación del algoritmo (Fig. 3.4), los conjuntos de entrada, $UPEXP_{in}$ y $UPDEF_{in}$, líneas 11 y 12 del algoritmo, son el resultado de eliminar aquella información que no debe seguir propagándose, por ejemplo los usos de aquellas variables que alcanzan su definición correspondiente. En el caso de los usos virtuales, sólo son eliminados, dentro de la función `Fs`, si encuentran una sentencia de definición que anule total o parcialmente la expresión.

El algoritmo es iterativo, por cada iteración se recorren todos los nodos del grafo, y se detiene una vez alcanzado el punto fijo, cuando los conjuntos de tuplas de entrada, $UPEXP_{in}$, $UPDEF_{in}$ y $DummyUse_{in}$, no cambian de una iteración a otra.

En la última parte del bucle, en las líneas 14-16 del algoritmo, se aplica una normalización para evitar bucles infinitos debido a los accesos a los campos recursivos, es decir, si tras varias iteraciones se encuentra alguna expresión del tipo *p→ f→ f→ f...* se normaliza la expresión con el uso de paréntesis p→(f)$^+$ para indicar que el campo `f` es accedido una o más veces. Clasificamos pues los punteros al *heap* en dos tipos, según apunten a un campo recursivo o no. De esta forma se asegura que las expresiones con campos recursivos van a ser siempre de longitud finita y, por tanto, que el algoritmo termina.

El cómputo de las cadenas DU intraprocedurales es dinámico y se realiza durante la propagación como se ha visto en el ejemplo anterior. Para entender mejor el comportamiento de ésta y las siguientes etapas vamos a utilizar la estructura de datos de la Fig. 3.5 y el código de la Fig. 3.6. El programa consta de dos funciones para la creación de la estructura de datos y dos funciones para el recorrido. Se trata de una estructura dinámica que apunta a una lista recursiva de nodos.

Explicamos a continuación cómo se llevaría a cabo la creación y transformación (en la primera iteración) de los usos virtuales para el procedimiento *Compute Lateral*. En la Fig. 3.7 vemos que el método tiene un parámetro formal *l* que es un puntero a la estructura recursiva lateral. Para este parámetro se crea el uso virtual $< S26', l, next_lateral, l >$, `S26'` es el identificador del nodo EXIT del procedimiento. Este uso virtual se propaga desde el nodo EXIT hacia el nodo ENTRY del procedimiento recorriendo todas las sentencias del código. En este caso, como vemos en la figura, no sufre transformación alguna. También tenemos en el procedimiento una llamada a función en la sentencia `S22` con el parámetro real `next`. El uso virtual que se crearía sería $< S22, next, next_lateral, next >$. Al propagarse hacia el nodo ENTRY alcanza la sentencia `S20` y la variable `next` se transforma en `l→ next_lateral`, por lo tanto, el uso virtual quedaría $< S22, l \rightarrow next_lateral, next_lateral, next >$. La expresión de origen sólo se transforma cuando se produce un avance a través de la propia variable `next`. Por ejemplo, para una definición del tipo `next→ next_lateral=...`.

Si ejecutamos el análisis intraprocedural descrito en este apartado sobre el código de la Fig. 3.6, detectaríamos la cadena DU intraprocedural entre las sentencias `S12` y `S13` debida al puntero `l→ next_lateral`. Representamos esta detección con la propagación de tuplas en la tabla 3.2. La tupla de uso creada en la sentencia `S13` alcanza la definición en `S12` y se detecta la cadena DU, por lo que la tupla de uso no continúa propagándose. La tupla de definición creada en `S12` si continúa propagándose.

Una vez termina el análisis intraprocedural, alcanzando el punto fijo tal como se ha explicado anteriormente, los conjuntos de tuplas de entrada contienen la información necesaria para las siguientes etapas del algoritmo.

```
algorithm ComputeDefUseChains (CFG)
input:
    CFG = [V, E, Entry, Exit]: control flow graph of the procedure
output:
    DefUse − Chains: List of DU chains detected in the procedure
    DefUse − Sets: The set of definitions and uses for each statement S of the procedure
    DummyUses: The set of dummy uses for each statement S of the procedure
begin
    /* initialization */
    DefUse_Chains = EmptyList
    /* creation of Dummy Uses for actual and formal parameters
    at certain points of the program */
1:  for each statement S in CFG do
2:      UPEXP_in[S] = UPDEF_in[S] = DefUse[S] = 0
    /* iteratively analyze a procedure */
3:      while changed
4:          for each node S ∈ V in reverse topological order do
            /* compute definition and use sets */
5:          UPEXP_out[S] = ⋃_{d∈succ(S)} UPEXP_in[d]
6:          UPDEF_out[S] = ⋃_{d∈succ(S)} UPDEF_in[d]
7:          DummyUse_out[S] = ⋃_{d∈succ(S)} DummyUse_in[d]

8:          UPEXP[S] = USE[S] ∪ F_S(UPEXP_out[S])
            /* store DU chains detected in DefUse_Chains*/
9:          UPDEF[S] = DEF[S] ∪ F_S(UPDEF_out[S])
10:         DummyUse_in[S] = DummyUse_in[S] ∪ F_S(DummyUse_out[S])

11:         UPDEF_in[S] = F_{UPDEF[S]}(UPDEF[S])
12:         UPEXP_in[S] = F_{UPEXP[S]}(UPEXP[S])
            /* store DU chains detected in DefUse_Chains */

            /* factor definition and use sets at loop header */
13:         if S is the header node of a loop then
14:             Factor(UPDEF_in[S])
15:             Factor(UPEXP_in[S])
16:             Factor(DummyUse_in[S])
            end
            /* delete output sets */
        end
        /* comparison of input sets between iterations */
    end
    /* annotations of final DU and dummy uses sets and DU chains detected */
end
```

Figura 3.4: Algoritmo implementado para el análisis intraprocedural.

Etapa 2: Construcción del IFG

Pasamos a describir el segundo paso del algoritmo tal como presentamos en la Fig. 3.3. Después de realizar el análisis intraprocedural sobre cada procedimiento del programa, tenemos por un lado

```
struct root{
    struct lateral *feeders;
};

struct lateral {
    struct lateral *next_lateral;
};
```

Figura 3.5: Estructura de datos del código simple de la Fig. 3.6.

```
struct root *build_tree(){
   register struct root *t;
   register struct lateral *l;

S1:   t=(struct root *)malloc();
S2:   l=build_lateral(20);
S3:   t->feeders=l;
S4:   l=NULL;

S5:   return t;
}

struct lateral *build_lateral(int num){
  register struct lateral *l;
  register struct lateral *next,*tmp;
  int val4;

S6:   l=NULL;
S7:   if (num == 0){
S8:   }else{
S9:      l=(struct lateral *) malloc();
S10:     val4=num-1;
S11:     next=build_lateral(val4);
S12:     l->next_lateral=next;
S13:     tmp=l->next_lateral;
S14:     next=NULL;
S15:  }
S16:  return l;
}
```

(a)

```
Compute_Tree(struct root *r){
   struct lateral *l;

S17:  l=r->feeders;
S18:  Compute_Lateral(l);

S19:  return;
}

Compute_Lateral(struct lateral *l){
   struct lateral *next;

S20:  next=l->next_lateral;
S21:  if (next!=NULL){
S22:     Compute_Lateral(next);
S23:  }else{
S24:  }

S25:  next=NULL;
S26:  return;
}

main(void){
   struct root *r;

S27:  r=build_tree();
S28:  Compute_Tree(r);
}
```

(b)

Figura 3.6: Ejemplo de código simple para mostrar las etapas del algoritmo.

todas las cadenas DU intraprocedurales y por otro los conjuntos de tuplas de definiciones y usos (virtuales y no virtuales) en cada sentencia del programa. Concretamente, los conjuntos de usos virtuales de entrada son leídos desde determinados puntos del CFG. En esta etapa, se extraen estas tuplas desde los nodos CFG ENTRY, y aquellos que contienen llamadas a función, para la construcción de los llamados enlaces *Reaching* como explicaremos más adelante en esta sección.

El grafo IFG se compone de un conjunto de nodos y enlaces. Hay cuatro tipos diferentes de nodos (ENTRY, EXIT, CALL y RETURN) y tres de enlaces (*Reaching*, *Binding* e *Interreaching*). Cada nodo IFG está relacionado con un nodo CFG de la forma en que se indica en la tabla 3.3.

```
Compute_Lateral(lateral *l)
{
   struct lateral *next;

S20:  next = l->next_lateral;
S21:  if (next != NULL){
S22:     Compute_Lateral(next);
S23:  }else{
S24:  }

S25:  next = NULL;
S26:  return;
S26':}
```

(a)

```
S19: <S26',l,next_lateral,l>,
   <S22,l->next_lateral,next_lateral,next>
S20: <S26',l,next_lateral,l>,
   <S22,next,next_lateral,next>
S21: <S26',l,next_lateral,l>,
   <S22,next,next_lateral,next>
S22: <S26',l,next_lateral,l>
S23: <S26',l,next_lateral,l>
S24: <S26',l,next_lateral,l>
S25: <S26',l,next_lateral,l>
S26': <S26',l,next_lateral,l>
```

(b)

Figura 3.7: Ejemplo de propagación de usos virtuales en un código simple

Sentencias	**Usos**	**Definiciones**
$S9 : l_1_1 = (struct\ lateral*)malloc();$	$-$	$< S12, 0, l_1_1 \rightarrow next_lateral, next_0_1 >$
$S10 : val4_0_1 = (num_0_1 - 1);$	$-$	$< S12, 0, l_1_1 \rightarrow next_lateral, next_0_1 >$
$S11 : next_0_1 = build_lateral(val4_0_1);$	$-$	$< S12, 0, l_1_1 \rightarrow next_lateral, next_0_1 >$
$S12 : l_1_1 \rightarrow next_lateral = next_0_1;$	DU chain	$< S12, 0, l_1_1 \rightarrow next_lateral, next_0_1 >$
$S13 : tmp_0_1 = l_1_1- > next_lateral;$	$< S13, 0, l_1_1 \rightarrow next_lateral >$	$-$
$S14 : next_1_1 = NULL;$		

Tabla 3.2: Ejemplo de detección de cadena DU intraprocedural.

nodo IFG	**nodo CFG**
ENTRY	ENTRY
EXIT	EXIT
CALL	NODO N con llamada a función
RETURN	NODO N con llamada a función

Tabla 3.3: Relación entre los nodos IFG y los nodos CFG

Para explicar los distintos tipos de enlaces vamos a usar la Fig. 3.8, que corresponde al grafo IFG del código de la Fig. 3.6. En la figura, podemos ver como cada enlace y cada nodo IFG tiene un identificador numérico asociado. Dentro de cada procedimiento, existe una tabla donde se asocia cada identificador numérico de cada enlace con su correspondiente función de transferencia. El identificador numérico de los nodos será usado más adelante para explicar la etapa de propagación a través del grafo.

La construcción de los nodos IFG se hace como se ha indicado anteriormente, creando un par de nodos ENTRY-EXIT por cada procedimiento y un par de nodos CALL-RETURN por cada llamada a función. De este modo podemos tener varios subgrafos IFG que componen el IFG global

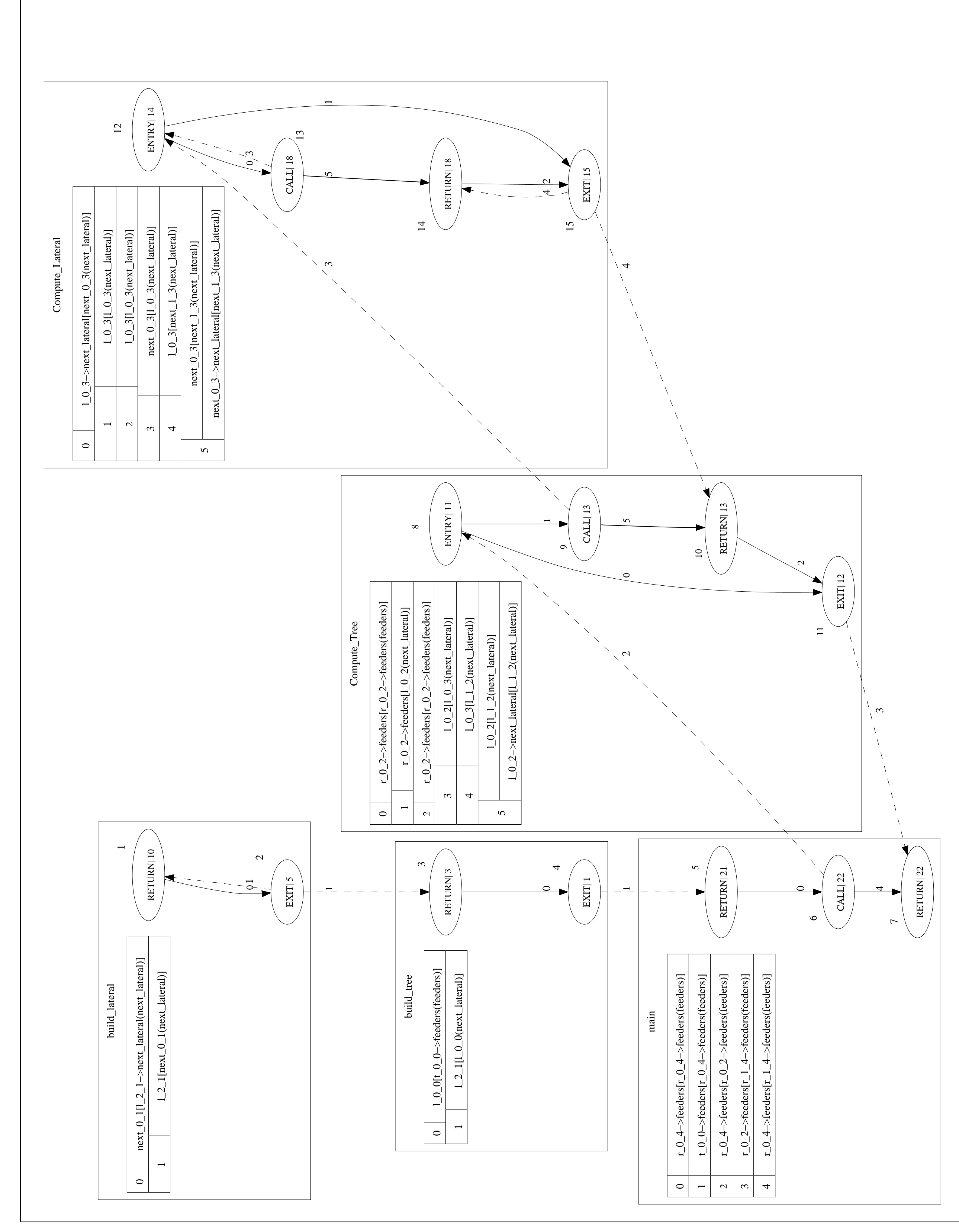


Figura 3.8: IFG del programa de la Fig. 3.6

al conectarse entre sí. Los nodos IFG no contienen sentencias sino conjuntos de tuplas de definición y de uso, de entrada y de salida.

Después de crear los nodos, se crean los enlaces siguiendo el orden indicado en la Fig. 3.9. Primero se construyen los enlaces tipo *Binding* para lo que hacen falta los parámetros formales y reales de cada procedimiento del programa. Luego se construyen los enlaces tipo *Reaching* haciendo uso de los usos virtuales calculados en la etapa anterior. Por último se crean los enlaces *Interreaching* para los que necesitamos los enlaces *Binding* y las funciones de transferencia, FT, de los procedimientos que son llamados.

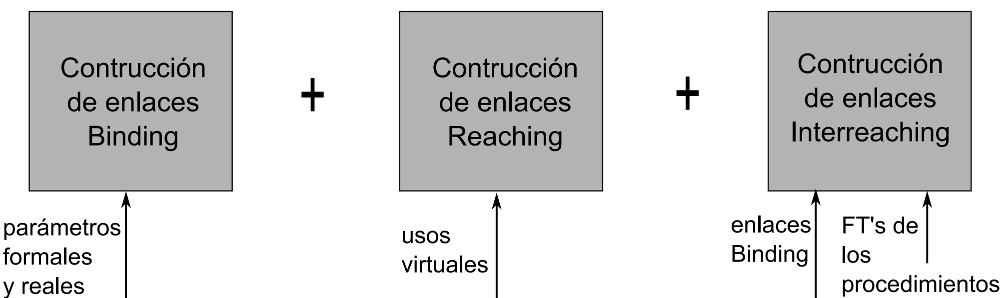


Figura 3.9: Pasos para la construcción de los enlaces IFG.

Los enlaces IFG conectan los nodos IFG de la forma indicada en la tabla 3.4. Cada enlace contiene una función de transferencia que nos indica la transformación que sufren las variables al atravesar dicho enlace desde un nodo hacia otro. Primeramente se construyen los enlaces tipo *Binding*, puesto que estos enlaces simplemente conectan la pareja de nodos CALL-RETURN con el procedimiento que es llamado desde un *call site*. Las funciones de transferencia de estos enlaces se calculan directamente a partir de los parámetros reales y formales del procedimiento que hace la llamada y del procedimiento que es llamado respectivamente. De este modo tenemos los enlaces CALL-*Binding* desde un nodo CALL a un nodo ENTRY, y los enlaces RETURN-*Binding* desde un nodo EXIT hacia un nodo RETURN. Por ejemplo, en la Fig. 3.8 el método *Compute_Tree*, grafo intermedio en la figura, hace una llamada con el parámetro real l_0_2 al método *Compute_Lateral* desde el nodo CALL-13 (Fig. 3.6) donde se convierte en l_0_3, por lo tanto, cualquier puntero al *heap* de la forma $l_0_3 \rightarrow next_lateral$ en *Compute_Lateral* se transformaría en $l_0_2 \rightarrow next_lateral$, lo que se indica con la función de transferencia $l_0_2[l_0_3(next_lateral)]$ en el correspondiente enlace CALL-*Binding*. La sintaxis de cada función de transferencia es tal que se separa por un lado la expresión de entrada (entre corchetes), y por otro la expresión de salida, fuera de los corchetes. Por lo tanto, podemos expresar de forma genérica el formato de una función de transferencia como $out_expression[in_expression(accessed_field)]$.

Enlace IFG	Conexiones NODO origen-NODO destino posibles
Reaching	ENTRY-CALL,RETURN-EXIT,RETURN-CALL,ENTRY-EXIT
Binding	CALL-ENTRY,EXIT-RETURN
Interreaching	CALL-RETURN

Tabla 3.4: Clasificación de los enlaces IFG

En la construcción de los enlaces tipo *Reaching* entran en juego los llamados usos virtuales. Recordamos que estos usos fueron introducidos en el análisis intraprocedural para extraer cómo

se transforman los parámetros reales y formales dentro de cada procedimiento. Concretamente se examina el conjunto final de usos virtuales que contienen el nodo raíz y las llamadas a función del CFG. Si un uso virtual ha alcanzado alguno de estos dos puntos desde el punto donde fue creado entonces quiere decir que existe al menos un camino donde ese uso no es destruido. Llevamos la información de los usos virtuales hacia sus nodos IFG correspondientes. Desde el nodo raíz CFG hacia el nodo ENTRY del IFG, y desde la llamada a función hacia el nodo RETURN del IFG. Si el conjunto de usos virtuales es el conjunto vacío, entonces quiere decir que no se debe crear ningún enlace tipo *Reaching* desde ese nodo IFG. En caso contrario, se crea un enlace desde el nodo ENTRY o RETURN por cada origen de usos virtuales alcanzable. Los usos virtuales participan en la creación de las funciones de transferencia de estos enlaces, reflejando la transformación que han sufrido a lo largo de ese camino.

Tomamos el mismo ejemplo que usamos para explicar la propagación de usos virtuales de la Fig. 3.7. Recordamos como al nodo raíz llegaban dos usos virtuales:

- $< S26', l, next_lateral, l >$, y
- $< S22, l \rightarrow next_lateral, next_lateral, next >$

desde puntos diferentes del programa. Esto provoca que haya dos enlaces tipo *Reaching* desde el nodo ENTRY-14, de la Fig. 3.8, el primer uso virtual se utiliza para construir la función de transferencia del enlace que va hacia el nodo EXIT-15, mientras que el segundo uso virtual es el responsable del enlace hacia el nodo CALL-18.

Los enlaces interreaching conectan cada par de nodos CALL-RETURN de una llamada a función dentro de un procedimiento. Se compone de la intersección de los enlaces tipo *Binding* y la función de transferencia del procedimiento que es llamado. Si existe función de transferencia del procedimiento entonces es que existe un posible camino a través del mismo en el que los parámetros formales pueden ser transformados o no, pero no son destruidos. En caso contrario no se crearía el enlace *Interreaching*. Por ejemplo, supongamos una llamada a una función `f` tipo `call(f)`, entonces para construir el enlace *Interreaching* correspondiente necesitaríamos las funciones de transferencia que se indican en la figura Fig. 3.10.

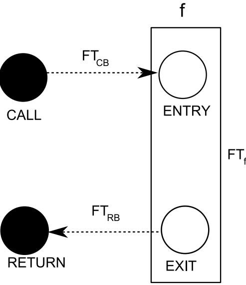


Figura 3.10: Ejemplo de creación de enlace interreaching.

De modo que la función de transferencia entre el nodo CALL y el nodo RETURN se construiría por la composición de $FT_{CB} \circ FT_f \circ FT_{RB}$.

Por ejemplo, el enlace interreaching entre el nodo CALL-13 y el nodo RETURN-13 de la figura Fig. 3.6 estaría compuesto por la composición de las siguientes funciones de transferencia (FT):

- FT del enlace *Binding*: $l_0_3[l_1_2(next_lateral)]$ desde el nodo EXIT-15, hacia el nodo RETURN-13.
- FT del procedimiento *Compute_Lateral*: la expresión $l_0_3[l_0_3(next_lateral)]$ y la expresión $l_0_3 \rightarrow next_lateral[l_0_3(next_lateral)]$ forman la función de transferencia.
- FT del enlace *Binding*: $l_0_2[l_0_3(next_lateral)]$ desde el nodo CALL-13 hacia el nodo ENTRY-14

Primero compondríamos la primera función de transferencia con la segunda, la expresión de salida l_0_3 de la primera coincide con la expresión de entrada de la segunda, y acceden al mismo campo $next_lateral$. El resultado de esta composición es $l_0_3[l_1_2(next_lateral)]$ y $l_0_3 \rightarrow next_lateral[l_1_2(next_lateral)]$. Por último, compondríamos este resultado con la última función de transferencia, obteniendo como expresiones finales $l_0_2[l_1_2(next_lateral)]$ y $l_0_2 \rightarrow next_lateral[l_1_2(next_lateral)]$.

Para calcular las funciones de transferencia de cada procedimiento, necesitamos conocer antes los llamados *caminos reales* del programa. Un *camino real* [2][46] representa una cadena de llamadas a procedimientos que se corresponde con una posible ejecución real del programa. En el programa de la Fig. 3.2.2 siguiente compuesto de tres procedimientos, los *caminos reales* del programa serían: `A-C-B`, `B` y `C-B`. El primer camino indica que si se realiza una llamada al procedimiento `A`, entonces también se va a producir una llamada al procedimiento `C` y luego al procedimiento `B`. Representan por lo tanto las cadenas reales de llamadas a procedimientos que podemos encontrarnos en un programa. En el ejemplo anterior no podríamos llamar al procedimiento `C` desde el procedimiento `B`, o tampoco es posible llamar al procedimiento `B` desde el procedimiento `A`. Por esta razón, no serían caminos reales las cadenas de llamadas `A-B` o `B-C`.

```
procedure A(){
   call C;
}
procedure B(){
}
procedure C(){
   call B;
}
```

Figura 3.11: Ejemplo para explicar los caminos reales.

Por lo tanto, a lo largo de cada camino se recorren nodos del grafo de llamadas y cada nodo en dicho grafo se corresponde con un procedimiento.

En la Fig. 3.12 se presenta el algoritmo utilizado para la construcción de los enlaces *Interreaching* del programa. Se utiliza el grafo de llamadas para recorrerlo en profundidad e ir calculando las funciones de transferencia de los procedimientos implicados por orden. Por esa razón se utiliza un método recursivo, *ComputeIERecursive*, que va profundizando en las llamadas hasta alcanzar el nivel más interno, y luego va resolviendo los enlaces *Interreaching* hacia el nivel

```
algorithm ComputeIE (CG)
input:
    CG: call graph
begin
    /* initialization */
    count = 1
    for each node N of CG do
        DFS(N)=0
    end
    /* traverse CG in depth-first-search order
    beginning at the root node */
    for each callee(e) from the root node do
        call ComputeIERecursive(callee(e))
        /* build an interreaching edge and compute the TF */
    end
end
algorithm ComputeIERecursive (N)
input:
    N: node of call graph
begin
    /* return if N has been visitied */
    if DFS(N) > 0 return DFS(N)
    /* recursively traverse the called procedures */
    DFS(N) = order = count++
    for each callee(e) from N node do
        put edge e active
        child = ComputeIERecursive(callee(e))
        if order > child then order = child
        if child > DFS(N) then
            /* build an Interreaching edge and compute the TF */
        end
    end
    /* return if N is part of recursive calls */
    if order <= DFS(N) then return order
    /* isolate N and possibly its callees and compute the TF */
    for each caller(e) of N node do
        put edge e inactive
    call ComputeTransferFunction(N)
    return order
    end
end
```

Figura 3.12: Algoritmo para el cálculo de los enlaces *Interreaching*.

más externo. La tabla `DFS` nos permite controlar los ciclos, ya sean debidos a procedimientos recursivos, o a bucles que pueden darse en la cadena de llamadas. El método *ComputeIERecursive* llama a la función *computeTransferFunction*, ver Fig. 3.13, para calcular la función de transferencia de cada procedimiento una vez terminada la construcción de todos los enlaces *interreaching* del mismo. Por ejemplo, en el código de la Fig. 3.6, para calcular la función de transferencia del enlace *Interreaching* entre los nodos etiquetados con los identificadores 9 y 10 dentro del método `Compute_Tree`, es necesario conocer la función de transferencia del procedimiento `Compute_Lateral`. Por lo tanto, cuando se llama al método *ComputeIERecursive*, éste llama a *computeTransferFunction* para calcular la función de transferencia de `Compute_Lateral`. Esta función primero activa los nodos y enlaces IFG que se corresponden con nodos y enlaces activos en el grafo de llamadas. Se utiliza el conjunto *ReachList[s]* para almacenar el resultado de la composición de funciones de transferencia en el nodo `s` del IFG . En este caso, nos devolvería que la función de transferencia está compuesta de dos expresiones $l_0_3[l_0_3(nex_lateral)]$,

```
algorithm ComputeTransferFunction (N)
input:
    N: node of call graph
output:
    Transfer Function of the Procedure or an empty set
begin
    active_CGEdges is initialized at ComputeIERecursive
    /* initialization of sets in active edges */
    for each active callee of N in active_CGEdges do
        put IFG nodes in active mode
        put IFG edges in active mode
        if node2 in the IFG edge e =< node1, node2 > is an EXIT node then
            ReachList[s] = TransFunc[e]
        else
            ReachList[s] = 0
    end
    /* compose edge transfer functions to compute the procedure transfer function */
    while changed do
        for each active IFG node d do
            for each active IFG edge e =< s, d > do
                ReachList[s] = ReachList[s] ∪ (ReachList[d] ∘ TransFunc[e])
            end
        end
    /* compute transfer function of procedures */
    for each active ENTRY node s do
        for each transfer function expOUT[expIN] in ReachList[s] do
            if expIN contains a formal parameter of the same procedure then
                TransFunc[s] = TransFunc[s] ∪ expOUT[expIN]
            end
        end
    end
end
```

Figura 3.13: Algoritmo para el cálculo de la función de transferencia de un procedimiento.

y $l_0_3 \rightarrow next_lateral[l_0_3(nex_lateral)]$, indicando que hay dos posibles caminos en el grafo, en uno no se transformaría la expresión y en el otro se produciría un avance a través del campo $next_lateral$. Una vez calculada la función de transferencia del procedimiento, se regresa al *call site* en *ComputeIERecursive* y se calcularía la función de transferencia del enlace *Interreaching* del método `Compute_Tree`. El resultado sería $l_0_2[l_1_2(nex_lateral)]$ y $l_0_2 \rightarrow next_lateral[l_1_2(nex_lateral)]$, como podemos ver en la Fig. 3.8.

Etapa 3: Propagación en el IFG

Pasamos a describir el tercer paso del algoritmo tal como presentamos en la figura Fig. 3.3. En este paso se realiza un volcado de tuplas de uso y definición desde ciertos nodos del CFG (nodos ENTRY y nodos que contengan una llamada a función) hacia ciertos nodos IFG (ENTRY y CALL) y comienza un proceso de propagación a lo largo de todo el IFG. Con esta propagación se simula el posible flujo de información que puede ocurrir entre distintos procedimientos del programa debido a las llamadas a función. El contexto en las llamadas a función está garantizado, ya que la propagación hacia atrás en el IFG se ejecuta de forma que cualquier camino que comienza en una llamada a función concreta siempre regresa al mismo punto de llamada. De esta manera se recorren en el IFG únicamente caminos reales de ejecución del programa.

El proceso de propagación se divide en dos etapas usando el mismo algoritmo de la figura Fig. 3.14. Cada fase incluye un filtrado específico a la hora de propagar las tuplas en el grafo IFG. En la primera fase, se excluyen los nodos EXIT del proceso de propagación, de manera que no se propagan tuplas a través de enlaces IFG cuyo origen o destino sea un nodo EXIT. Se simula el sentido del flujo de información desde los procedimientos que hacen las llamadas, hacia los procedimientos que son llamados (desde un nivel externo hacia un nivel interno). En la segunda fase, se excluyen los enlaces tipo *Call Binding* (conectan un nodo CALL con su correspondiente nodo ENTRY). De este modo, la información ahora fluye en el otro sentido, desde el procedimiento que es llamado hacia el procedimiento que hizo la llamada (de un nivel interno hacia uno externo). Además se excluyen los nodos ENTRY como receptores de tuplas, es decir, no se propagan tuplas hacia los nodos EXIT ya que estas tuplas no pueden salir a través de los enlaces *Call Binding* (los únicos que apuntan hacia nodos ENTRY). Con esta medida evitamos la propagación y acumulación de tuplas en los nodos ENTRY, ya que dicha información no va a ser utilizada posteriormente. Cuando termina el proceso de propagación, al alcanzar el punto fijo, sólo las tuplas contenidas en los nodos EXIT y los nodos RETURN son llevadas de vuelta hacia cada nodo CFG correspondiente.

procedure Propagate (V,E)
input:
  V: set of nodes
  E: set of edges
begin
  while changed
    for each node N1 of type V **do**
      for each node N2 that is sink of edge $e = < N1, N2 >$ in E **do**
        $IPUSE_{out}[N1] = IPUSE_{out}[N1] \cup F_e(IPUSE_{in}[N2])$
        $IPDEF_{out}[N1] = IPDEF_{out}[N1] \cup F_e(IPDEF_{in}[N2])$
      end
      $IPDEF[N1] = IPDEF_{out}[N1] \cup F_{IPDEF_{out[N1]}}(IPDEF_{out}[N1])$
      $IPUSE[N1] = IPUSE_{out}[N1] \cup F_{IPDEF_{out[N1]}}(IPUSE_{out}[N1])$
      $IPDEF_{in}[N1] = Factor(IPDEF[N1])$
      $IPUSE_{in}[N1] = Factor(IPUSE[N1])$
    end
  end
end

Figura 3.14: Algoritmo para la propagación IFG de tuplas.

El algoritmo de la Fig. 3.14 recibe a la entrada un conjunto diferente de nodos y enlaces según la fase de propagación en que nos encontremos, como hemos explicado anteriormente. Al ser una propagación hacia atrás, los nodos origen reciben las tuplas desde el nodo destino siendo éstas transformadas por la función de transferencia del enlace F_e. Luego se examina el conjunto de tuplas de definición, $IPDEF_{out}$, aplicando las transformaciones oportunas tanto sobre las propias tuplas de definición como sobre las tuplas de uso, que es lo que indicamos con la función $F_{IPDEF_{out}}$. La función de transferencia de esta función sería similar a la de `Fupdef` en el algoritmo intraprocedural, pero eliminando las comprobaciones de dominancia. Por lo tanto, su función de transferencia resultaría mucho más sencilla, tal como se muestra en la ecuación de la Fig.3.15.

Nodo IFG	Antes de la propagación	Fase 1 de propagación	Fase 2 de propagación
1	<S12,l_1_1→next_lateral,next_0_1>	<S12,l_1_1→next_lateral,next_0_1>	<S20,next_0_1→(next_lateral)+>, <S12,l_1_1→next_lateral,next_0_1>
2			<S17,l_2_1>,<S20,l_2_1→(next_lateral)+>, <S12,l_1_1→next_lateral,l_2_1>
3	<S3,t_0_0→feeders,l_0_0>	<S3,t_0_0→feeders,l_0_0>	<S17,l_0_0>, <S20,l_0_0→(next_lateral)+>, <S3,t_0_0→feeders,l_0_0>
4			<S17,t_0_0→feeders>, <S20,t_0_0→feeders→(next_lateral)+>
5		<S17,r_0_4→feeders>	<S20,r_0_4->feeders→(next_lateral)+>
6		<S17,r_0_4→feeders>	<S20,r_0_4->feeders→(next_lateral)+>
7			
8	<S17,r_0_2→feeders>	<S17,r_0_2→feeders>, <S20,r_0_2→feeders→(next_lateral)+>	<S17,r_0_2→feeders>, <S20,r_0_2→feeders→(next_lateral)+>
9		<S20,l_0_2→(next_lateral)+>	<S20,l_0_2→(next_lateral)+>
10			
11			
12	<S20,l_0_3→next_lateral>	<S20,l_0_3→(next_lateral)+>	<S20,l_0_3→(next_lateral)+>
13		<S20,next_0_3→(next_lateral)+>	<S20,next_0_3→(next_lateral)+>
14			
15			

Tabla 3.5: Fases de propagación del código de la Fig. 3.6

$$Fipdef_{<\text{left}_e = right_e>} = \begin{cases} right_e & \text{si } use_e = \text{left}_e \\ right_e \rightarrow \text{f} & \text{si } use_e = \text{left}_e \rightarrow \text{f} \\ use_e & \text{si } otherwise \end{cases}$$

Figura 3.15: FT de la función Fipdef

Todas las tuplas cuyas expresiones coincidan total o parcialmente con la expresión de la tupla de definición left_$e = right_e$ son transformadas y añadidas al conjunto intermedio $IPDEF$ si la tupla transformada es de definición, o al conjunto intermedio $IPUSE$ si la tupla es de uso. Por último, los conjuntos de entrada ($IPDEF_{in}$ y $IPUSE_{in}$) son el resultado de aplicar la normalización descrita en el algoritmo intraprocedural a los conjuntos intermedios finales. El punto fijo del algoritmo se alcanza cuando no cambian los conjuntos de entrada de una iteración a otra.

En la tabla 3.5 observamos las distintas tuplas de uso y definición que se propagan a lo largo de esta etapa para cada nodo IFG identificado con un número. En la primera columna, etiquetada como **Antes de la propagación**, se presentan las tuplas que se han depositado en cada nodo antes de empezar las fases de propagación. En la columna que hemos llamado **Fase 1 de propagación** se presentan las tuplas después de haber alcanzado el punto fijo. Por último, la tercera columna muestra el estado final con las tuplas que se han acumulado en cada nodo IFG, después de finalmente alcanzar el punto fijo tras la segunda fase de propagación.

Etapa 4: Cómputo de cadenas DU interprocedurales

Describimos el último paso del algoritmo tal como lo definimos en la Fig. 3.3. Una vez alcanzado el punto fijo en el IFG, las tuplas de uso y de definición son traídas desde los nodos EXIT

y CALL de cada subgrafo IFG, hacia su correspondiente nodo CFG. De este modo, tenemos ahora, en cada grafo CFG de cada procedimiento, la información interprocedural proveniente desde otros procedimientos. En esta etapa se aplica el mismo algoritmo intraprocedural que en la primera etapa, Fig. 3.4, pero eliminando el tratamiento de los usos virtuales. Gracias a este proceso, asociamos la información local con la información propagada, y es posible detectar las cadenas DU interprocedurales del programa. En el código de la Fig. 3.6 se detectarían por ejemplo, entre otras, las cadenas DU entre `S3-S17` y `S12-S20`.

3.2.3. Complejidad

Analizaremos la complejidad del algoritmo por etapas. La primera y la última etapa analizan las sentencias del programa a través de un proceso de propagación hacia atrás. Para un programa de N sentencias, habrá como máximo N definiciones y usos en total. Recordamos que las expresiones analizadas tenían el formato $pointer \rightarrow field$ y en el caso de accesos recursivos las expresiones pueden ser normalizadas $p \rightarrow (f)^+$ para que la longitud de los campos accedidos no crezca indefinidamente. Por lo tanto, la longitud de los campos recorridos $p.f1.f2....fk$ `(1<=k<=kmax)`, donde `kmax` representa el valor de la longitud máxima que se puede alcanzar, va a tener una longitud finita que va a depender de la configuración jerárquica de la estructura de datos declarada en el programa. Por ejemplo, en una lista simplemente enlazada, la longitud `kmax` sería 1, y en una lista doblemente enlazada, sería 2. El algoritmo usado en estas dos etapas tiene una técnica de flujo de datos iterativa, de forma que una tupla toma `d+2` iteraciones para alcanzar todos los destinos posibles, donde `d` es el nivel de anidamiento. Además son necesarias `3kmax` iteraciones más para calcular todos las expresiones de los caminos de acceso posibles [21]. Por lo tanto, la complejidad temporal total del algoritmo usado en estas dos etapas sería `d+3kmax+2`. En la etapa de construcción del IFG, cada procedimiento y cada llamada a función se modela por un par de nodos IFG. Si un programa contiene `P` procedimientos y `C` llamadas a función, entonces hay `2P+2C` nodos como máximo. El número total de enlaces tipo *binding* sería `2C` y el número total de enlaces tipo *interreaching* sería `C`. El número de enlaces tipo *reaching* depende del número de llamadas a funciones de cada procedimiento. Si el número de llamadas a función de un procedimiento es `c`, entonces el número máximo de enlaces tipo *reaching* del procedimiento sería $2c + c^2 + 1$: donde `2c` sería el número de enlaces *reaching* tipo ENTRY→CALL y RETURN→EXIT, tal como se vió en la tabla 3.4; c^2 es el número de enlaces *reaching* EXIT→CALL; y por último habría un enlace ENTRY→EXIT. Por último, la etapa de propagación presenta una complejidad cuadrática en función del número de nodos a recorrer $O(N^2)$.

3.3. Cadenas de definición y uso ISSA

Para completar la información DU del programa, añadimos una técnica adicional interprocedural centrada en los punteros al *stack*, basada en la representación ISSA, *Interprocedural Static Single Assignment*. De esta forma somos capaces de detectar el resto de cadenas DU debidas a punteros al *stack* no contempladas en el complejo algoritmo previamente descrito. Esta técnica recorre de nuevo la sentencias del programa extrayendo únicamente tuplas de dos tipos:

- definición: $< id\, block, id\, stmt, left\, expression, right\, expression >$

- uso: $< id\, block, id\, stmt, expression >$

Estas tuplas ya las presentamos en la sección 3.2.2, la diferencia es que ahora las expresiones contienen punteros al *stack* en vez de punteros al *heap*.

Después de este primer pase de creación de tuplas, se realiza un segundo pase donde se emparejan definiciones y usos directamente a partir de los nombres de las variables implicadas. En este listado de cadenas DU detectadas aparecen tanto sentencias reales como sentencias específicas que fueron introducidas en la transformación ISSA. Esta detección tan sencilla de cadenas DU es posible gracias a que la representación ISSA garantiza un nombre único para cada variable en todo el programa.

```
struct root *build_tree(){
   register struct root *t;
   register struct lateral *l;

S1:   t_0_0=(struct root *)malloc();
S2:   l_0_0=build_lateral(20);
S3:   l_0_0=l_2_1;
S4:   t_0_0->feeders=l_0_0;
S5:   l_1_0=NULL;
S6:   return t_0_0;
}

struct lateral *build_lateral(int num){
  register struct lateral *l;
  register struct lateral *next,*tmp;
  int val4;

S7:   num_0_1=val4_0_1;
S8:   l_0_1=NULL;
S9:   if (num_0_1==0){
S10:  }else{
S11:     l_1_1=(struct lateral *)malloc();
S12:     val4_0_1=(num_0_1 - 1);
S13:     next_0_1=build_lateral (val4_0_1);
S14:     next_0_1=l_2_1;
S15:     val4_1_1=num_0_1;
S16:     l_1_1->next_lateral=next_0_1;
S17:     tmp_0_1=l_1_1->next_lateral;
S18:     next_1_1=NULL;
S19:  }
S20:  l_2_1=phi(l_1_1,l_0_1);
S21:  return l_2_1;
}
```

(a)

```
Compute_Tree(struct root *r){
   struct lateral *l;

S22:  r_0_2=r_0_4;
S23:  l_0_2=r_0_2->feeders;
S24:  Compute_Lateral(l_0_2);
S25:  l_1_2=l_0_3;
S26:  return;
}

Compute_Lateral(struct lateral *l){
   struct lateral *next;

S27:  l_0_3=Iphi(next_0_3,l_0_2);
S28:  next_0_3=l_0_3->next_lateral;
S29:  if (next_0_3!=NULL){
S30:     Compute_Lateral(next_0_3);
S31:     next_1_3=l_0_3;
S32:  }else{
S33:  }

S34:  next_2_3 =phi(next_1_3, next_0_3);
S35:  next_3_3 = NULL;
S36:  return;
}

main(void){
   struct root *r;

S37:  r_0_4 = build_tree ();
S38:  r_0_4 = t_0_0;
S39:  Compute_Tree (r_0_4);
S40:  r_1_4 = r_0_2;
}
```

(b)

Figura 3.16: Código de la Fig. 3.6 en representación ISSA.

En el código de la Fig. 3.16 presentamos la forma ISSA del ejemplo que hemos tomado en este capítulo, Fig. 3.6, para explicar cada una de las etapas del algoritmo para extraer las cadenas DU. En este último paso, se detectarían entre otras cadenas DU intraprocedurales como `S1-S4` y `S28-S30`, así como cadenas interprocedurales `S23-S27` y `S38-S22`. Vemos que todas las sentencias adicionales que se han añadido para la representación ISSA, también son tenidas en cuenta.

En realidad la recopilación de tuplas del primer pase podría hacerse de forma paralela aprovechando el recorrido que realiza la técnica anterior. Sin embargo, hemos preferido mantener la

independencia de ambas técnicas para poder aplicar criterios separadamente según los requisitos del cliente, es decir, dependiendo del tipo de aplicación puede que sólo sean necesarias las cadenas DU debidas a punteros al *heap* o puede que también sean necesarias las cadenas debidas a punteros al *stack*.

3.3.1. Complejidad

El recorrido que se hace del programa es lineal en función del número de sentencias del mismo, no siendo necesario alcanzar ningún punto fijo como en el algoritmo DU link. Supongamos un programa de `S` sentencias en total, analizando cada etapa por separado, en la primera fase se recorrerían estas `S` sentencias para la creación de tuplas. Luego, en la segunda fase no se recorren todas las sentencias del programa, sino la lista de tuplas creadas en la primera fase, por lo tanto el número de elementos a recorrer, `n`, va a ser siempre menor que el número de sentencias del programa, `nS`. En conclusión, la cota máxima de complejidad para este algoritmo sería `2S`.

3.4. Resultados Experimentales

Hemos considerado siete programas (cuatro extraídos de la suite de benchmarks Olden [47]) para las pruebas de este análisis completo. Pasamos a describir cada uno de ellos de forma independiente.

1. **Reverse**. El programa primero llama a un procedimiento recursivo para crear la lista cíclica y luego llama a otro procedimiento recursivo para invertirla. La estructura de datos de este código por tanto sería la de una lista simplemente enlazada, como se muestra en la Fig. 3.17(a).

2. **Matrix x Vector(s)**. Dada una matriz dispersa `M` y un vector disperso `V`, este programa calcula la multiplicación de ambos. Las estructuras dispersas son aquellas donde la mayoría de sus elementos son cero. Por esta razón, resulta de gran utilidad usar punteros en este tipo de estructuras para almacenar únicamente los elementos distintos de cero. Se representan como listas de punteros con alguna información adicional de la localización de los elementos dentro de la estructura. La estructura de datos de la matriz sería la que podemos ver en la Fig. 3.17(b). Por ejemplo, para la matriz dispersa, cada elemento tiene su valor dentro de cada nodo del grafo, además de información adicional sobre su fila y columna. Esta representación es diferente a la representación clásica como array bidimensional, donde todos los elementos nulos consumirían almacenamiento en memoria. Este benchmark es un ejemplo de los kernels numéricos habituales que manipulan matrices dispersas. El producto en sí mismo consiste en tres bucles anidados.

3. **Matrix x Matrix(s)**. Dadas dos matrices dispersas `M1` y `M2`, este programa genera la matriz producto a la salida `M3=M1xM2`. La salida de la matriz es creada durante el recorrido de las matrices `M1` y `M2`. La representación usada para las matrices es la misma que la del programa anterior, la estructura de datos sería también la que podemos ver en la Fig. 3.17(b), pero en vez de una matrix, `M` y un vector `V`, tendríamos dos matrices `M`. Empleamos la misma

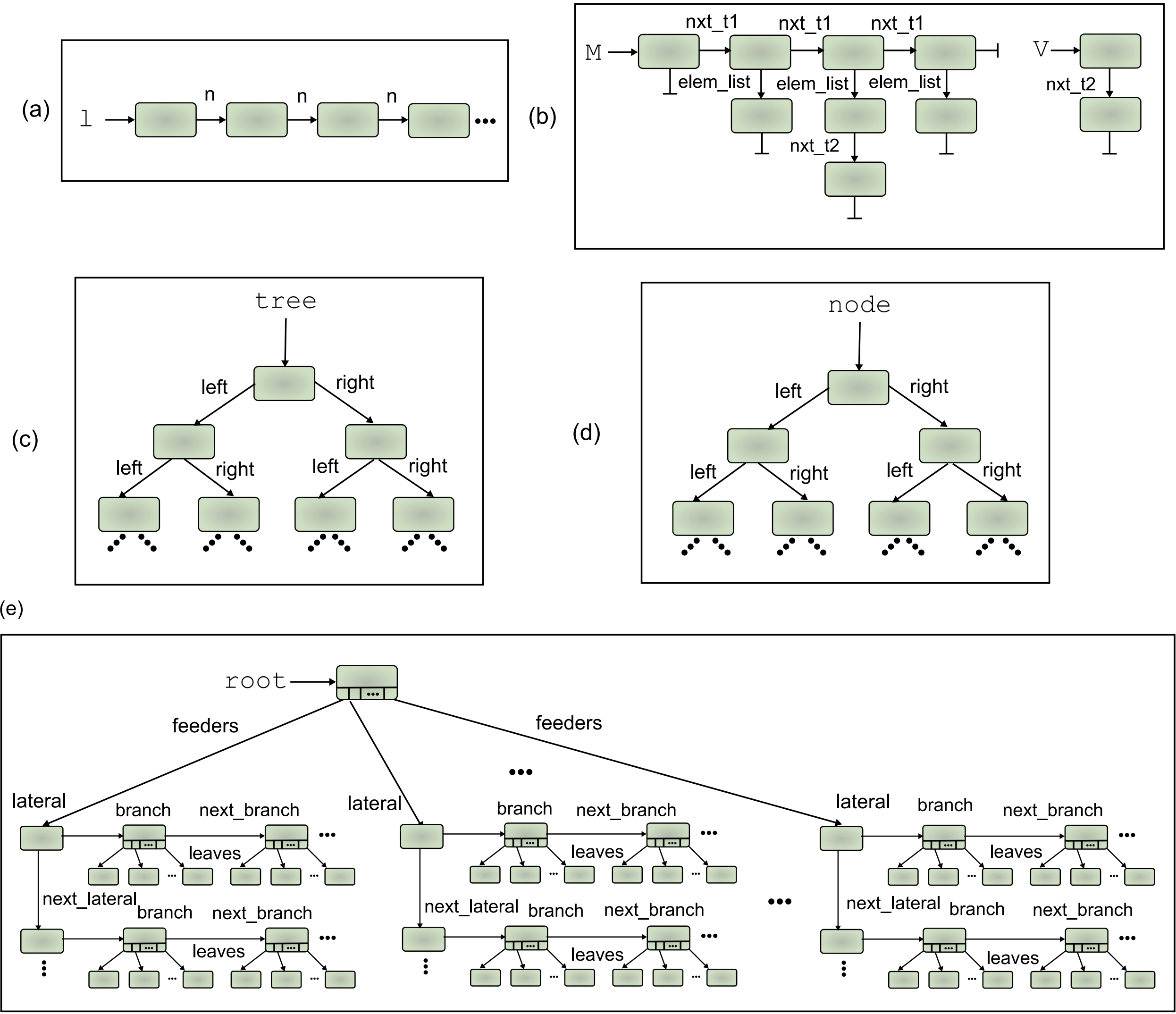


Figura 3.17: Estructuras de datos de los códigos.

estructura de listas simplemente enlazadas para las matrices dispersas. Este benchmark se muestra para mostrar el efecto de añadir más complejidad a la estructura de datos y el flujo de control del programa en relación con el benchmark `Matrix x Vector`, ya que este producto comprende tres matrices y cuatro bucles anidados.

4. **Em3d**. Este programa está extraído de la suite de Olden, crea dos listas simplemente enlazadas para el campo magnético y el campo eléctrico, y luego enlaza cada elemento en una lista con varios elementos en otra lista, creando un grafo bipartito. La estructura de datos se ha representado en la Fig. 3.18. Este ejemplo ha sido usado ampliamente en la literatura relacionada con las estructuras dinámicas debido a su complicada estructura de interconexión.

5. **TreeAdd**. Este programa proviene también de la suite Olden. La estructura de datos sería la de un árbol binario como podemos ver en la Fig. 3.17(c). Recorre un árbol y calcula la suma de valores de los nodos del árbol. Primero llama a un procedimiento recursivo que crea un

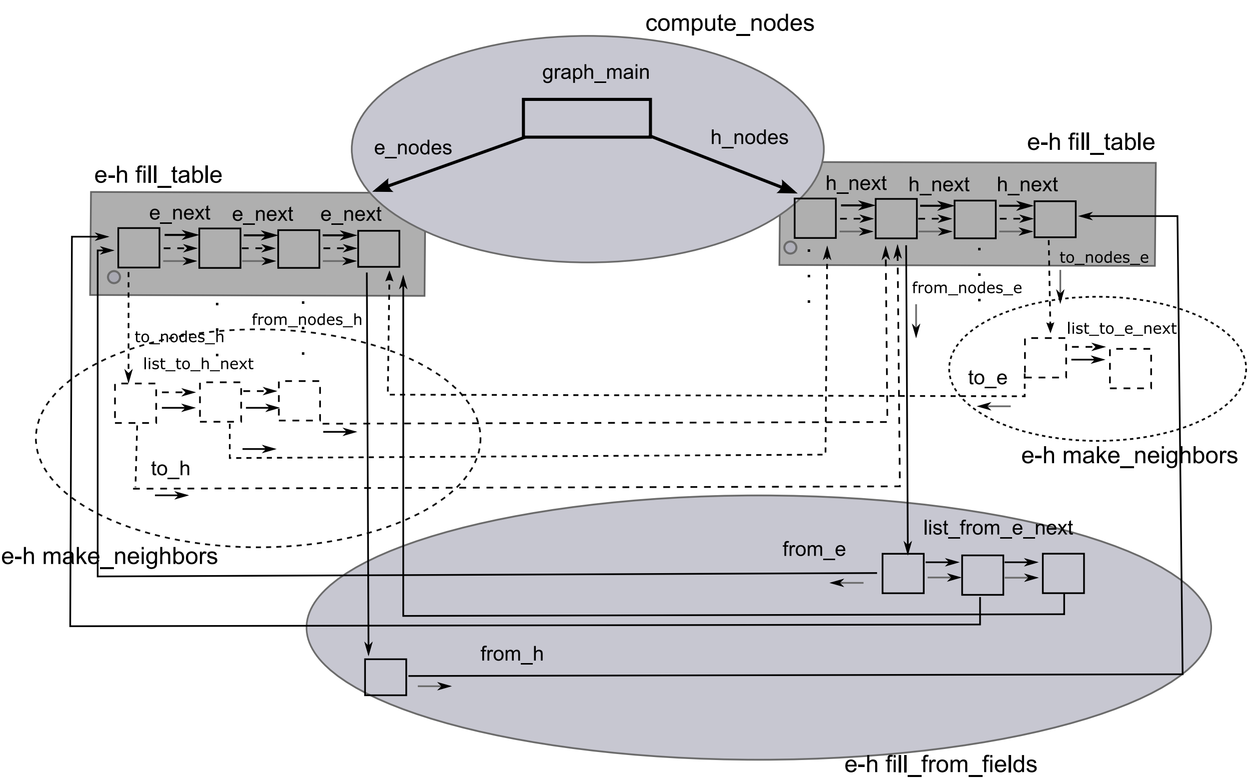


Figura 3.18: Estructura de datos del código Em3d.

árbol binario balanceado, y luego llama a otro procedimiento recursivo que recorre el árbol añadiendo las sumas de los subárboles.

6. **Power**. Este programa de Olden crea y luego recorre una estructura multinivel. Como vemos en la Fig. 3.17(e), el elemento raiz apunta a un número indeterminado de elementos laterales a través de un array de punteros. Cada uno de los elementos laterales conecta de forma recursiva a otros elementos del mismo tipo, formando listas simplemente enlazadas. Además cada elemento lateral apunta a una lista de otros elementos de diferente tipo, éstos a su vez apuntan a varios elementos tipo hoja a través de un array de punteros. Este programa muestra una compleja estructura en un flujo de control complejo de recursión anidada (funciones recursivas que llaman a otras funciones recursivas), a medida que se recorre cada nivel de la estructura.

7. **Bisort**. Otro benchmark de la suite Olden, Fig. 3.17(d), que crea y manipula un árbol binario. En esta ocasión, los elementos pueden ser intercambiados entre subárboles, preservando la forma del árbol tras el cambio. Este programa presenta otro caso de recursión anidada, con la complejidad adicional de dos llamadas recursivas por función recursiva (una por cada rama del árbol).

En la tabla 3.6 mostramos algunos datos medidos para estos códigos: el número de líneas, el número de procedimientos, el número de *call sites*, dentro de los *call sites* cuáles son recursivos, el

número de estructuras de datos que maneja, qué tipo de estructura presenta y el número de campos recursivos que maneja.

En la tabla 3.7 presentamos las tiempos medidos (en segundos) para cada una de las etapas del algoritmo DU-Link. Para los programas de multiplicación con matrices el análisis llevado a cabo ha sido sólo intraprocedural al contener el programa un único método. También se presenta la tabla 3.8 con los datos sobre el tamaño de los grafos en el análisis, número de tuplas de definición y uso manejadas durante la etapa de propagación interprocedural y cadenas DU detectadas tanto por el algoritmo DU link (intraprocedurales e interprocedurales), como por el algoritmo DU ISSA. Estos experimentos han sido ejecutados en una máquina Intel(R) Core^{TM} 2 CPU 1.86 GHz con 2 GB RAM.

Código	**nº líneas**	**nº procs**	**call sites**	**llamadas recursivas**	**nº estructuras**	**tipo de estructura**	**campos recursivos**
MatrixVector	155	1	0	0	2	lista simplemente enlazada	2
MatrixMatrix	200	1	0	0	2	lista simplemente enlazada	2
Reverse	61	4	7	7	1	lista simplemente enlazada	1
TreeAdd	42	3	6	6	1	árbol binario	2
Power	272	15	22	10	5	árbol jerárquico	2
Em3d	773	17	18	0	7	grafo bipartito	6
Bisort	468	10	16	16	1	árbol binario	2

Tabla 3.6: Información acerca de cada código analizado

Código	**SSA+intra**	**IFG**	**Propa**	**DU-link**	**DU-ISSA**	**Total**
Matrix x Vector	2.606	0.0	0.0	0.0	0.18	2.786
Matrix x Matrix	6.872	0.0	0.0	0.0	0.432	7.304
Reverse	0.546	0.102	0.040	0.258	0.25	1.200
TreeAdd	0.703	0.141	0.078	0.226	0.12	1.270
Power	6.765	3.156	0.031	2.457	3.01	16.03
Em3d	16.16	4.793	14.156	8.831	4.59	48.53
Bisort	2.703	0.953	0.156	2.517	0.44	6.77

Tabla 3.7: Medidas de tiempo en segundos por etapas del algoritmo DU-link

Código	**nodos**	**enlaces**	**TuplasDef**	**TuplasUso**	**cadenasIntra**	**cadenasInter**	**cadenasISSA**
Matrix-vector	0	0	0	0	0	0	139
Matrix-matrix	0	0	0	0	0	0	208
Reverse	22	24	20	21	0	7	45
TreeAdd	22	21	9	29	0	2	30
Power	88	43	11	100	0	9	133
Em3d	92	61	317	806	8	38	246
Bisort	68	68	21	231	0	6	86

Tabla 3.8: Dimensiones IFG y cantidad de tuplas

El número de nodos en el grafo IFG y el número de tuplas manejadas durante la etapa de propagación nos dan una idea de lo complejo que ha resultado el análisis con cada programa. Se aprecia que el código `em3d` ha sido el más complejo de analizar dado el tamaño del grafo y el gran número de tuplas en comparación con el resto de códigos analizados. En cuanto a las medidas de tiempo por etapas, observamos que la primera etapa, que corresponde a la transformación SSA del código, más el análisis intraprocedural, es la que más tiempo requiere en todos los casos. Esto se debe a que el análisis intraprocedural extrae toda la información requerida para las siguientes etapas del algoritmo, por tanto, influye directamente en esta etapa el número de tuplas generadas para el análisis, cuanto mayor más tiempo se tarda en alcanzar el punto fijo. Si el tamaño del grafo IFG es elevado la etapa de construcción del mismo puede requerir más tiempo, como ocurre en el caso del código `Power`. Si además tenemos un gran número de tuplas en la etapa de propagación interprocedural, entonces la etapa de propagación puede resultar más costosa, como sucede en el programa `Em3d`.

3.5. Conclusiones

En este capítulo hemos presentado la técnica para el análisis DU de códigos con estructuras dinámicas de datos. Recordamos que las estructuras dinámicas de datos son ampliamente usadas en códigos irregulares, códigos que suponen un gran reto para muchas aplicaciones con limitaciones tales como la recursividad, en ocasiones presente en dichos códigos. Gracias a esta técnica logramos extraer información valiosa que puede ser utilizada, como veremos en los siguientes capítulos, para la optimización de técnicas de compilación. Estas optimizaciones estarán orientadas a mejorar la eficiencia de dichas aplicaciones así como ampliará el número de casos que pueden tratar. El algoritmo que se ha presentado, está basado en el algoritmo de Hwang & Saltz [21] pero nosotros lo hemos extendido principalmente en dos aspectos:

- Aprovechar el recorrido de la primera etapa para extraer no sólo información DU sino también información de los *paths* y de los patrones de recorrido de las estructuras dinámicas.
- Ampliar el cálculo de cadenas DU para no sólo capturar dichas cadenas debidas a punteros al *heap* de naturaleza recursiva sino cualquier puntero al *heap*, sea el campo selector recursivo o no.
- Añadir también el cálculo de cadenas DU al *stack* debidas a la forma ISSA.

Además hemos optimizado el algoritmo original eliminando en la etapa tercera el recorrido de caminos del grafo que resultaban irrelevantes para la fase de propagación. El coste del análisis completo depende principalmente de dos factores: el número de tuplas generadas para el análisis y el tamaño del grafo IFG. Cuando alguno de estos parámetros, o los dos, son elevados, entonces el análisis es más costoso. Los resultados experimentales muestran tiempos para la mayoría de los casos de pocos segundos. En los siguientes capítulos presentaremos las aplicaciones que mostrarán la utilidad de esta técnica.

4

Code Slicing

4.1. Aceleración del Análisis de Forma

Primeramente vamos a introducir un poco el funcionamiento de la herramienta de análisis de forma que pretendemos optimizar. A continuación hablaremos de las dificultades que tratamos de resolver y presentaremos el algoritmo de *slicing* diseñado para tal fin. Por último presentaremos algunos resultados numéricos con las medidas realizadas y las principales conclusiones.

4.1.1. Introducción

El análisis de forma es una técnica de análisis del *heap* que considera información disponible en tiempo de compilación para arrojar información detallada acerca del *heap* en programas basados en punteros. Esto se hace extrayendo información acerca de la forma o la conectividad de los elementos del *heap*.

La información derivada del análisis de forma en una aplicación basada en punteros puede usarse para varios propósitos como: (i) análisis de dependencias de datos, determinando si dos accesos pueden alcanzar la misma localización de memoria; (ii) explotación de localidad, capturando el modo en que se recorren las localizaciones de memoria para determinar cuando es probable que

estén contiguas en memoria; (iii) verificación de programas, para proporcionar garantías de corrección en programas que manipulan el *heap*; y (iv) soporte al programador, para ayudar en la detección de un uso incorrecto de punteros o documentar estructuras de datos complejas.

En este enfoque de análisis de forma que pretendemos optimizar, se usan abstracciones de forma expresadas como grafos para modelar el *heap*. El análisis de forma basado en grafos es una técnica de análisis de punteros muy detallada, sensible al flujo, contexto y campo.

Este análisis de forma está basado en la construcción de *grafos de forma*. El propósito de un grafo de forma es representar las principales características de forma de las estructuras de datos dinámicas y recursivas. Estas características permiten identificar las estructuras como listas, o árboles, por ejemplo, incluyendo información acerca de la presencia o ausencia de ciclos, las localizaciones alcanzables desde un puntero, etc.

4.1.2. Funcionamiento

Los grafos de forma están constituidos por 3 elementos básicos que se combinan para formar conjuntos de enlaces como indica la vista jerárquica de la Fig. 4.1. En el nivel más bajo tenemos: (i) *punteros*, que se usan como puntos de acceso a las estructuras; (ii) *nodos*, que se usan para representar localizaciones de memoria alojadas en el *heap*; y (iii) *selectores*, o campos puntero, que se usan para enlazar nodos. Combinando estos elementos básicos, podemos crear dos tipos de relaciones: *enlaces de punteros* o *pointer links* (`pl`'s), que son enlaces entre punteros y nodos; y *enlaces de selectores* o *selector links* (`sl`'s), que son enlaces entre nodos a través de un selector. Finalmente, los `pl`'s y `sl`'s pueden combinarse para formar *conjuntos coexistentes de enlaces* o *coexistent links sets* (`cls`'s), que describen combinaciones de `pl`'s y `sl`'s que pueden existir *simultáneamente* en un nodo.

Los llamados *coexistent links sets* registran posibles patrones de conectividad entre elementos alojados en el *heap*. En general, proporcionan información de la probabilidad (información *may*) de que determinados patrones de conectividad puedan existir en el *heap*. En el caso de que exista más de un *cls* por nodo, entonces uno de ellos (y sólo uno) es representativo de las localizaciones de memoria abstraídas. Cuando hay sólo un *cls* por nodo, la información que proporciona es segura (información *must*). Los *coexistent links sets* también proporcionan información definida acerca de lo que no puede ser. En otras palabras, los patrones de conectividad que no han sido registrados en ningún *cls* para un nodo no pueden darse para las localizaciones de memoria abstraídas en el nodo.

El propósito del grafo de forma es representar las principales características de las estructuras dinámicas de datos. Estas características permiten identificar estructuras como listas o árboles por ejemplo, incluyendo información acerca de la presencia o ausencia de ciclos y el tipo de localizaciones alcanzables desde un puntero.

Cada sentencia puntero modifica un grafo de forma de entrada para producir un grafo de forma de salida, de manera que se representa de forma precisa el efecto de dicha sentencia en tiempo de ejecución. En la Fig. 4.2 se muestra, por ejemplo, cómo cambian los grafos cuando se analizan las sentencias que crean una lista simplemente enlazada. Una sentencia *malloc*, tal como `S1` y `S3`, produce la creación de un nuevo nodo, que abstrae una localización de memoria reservada en tiempo de ejecución. La sentencia `S4: p->nxt=a` se usa para conectar el nodo apuntado por `p` por el nodo apuntado por `a`. Las sentencias de aliasing, tales como `S2` y `S5`, producen un puntero

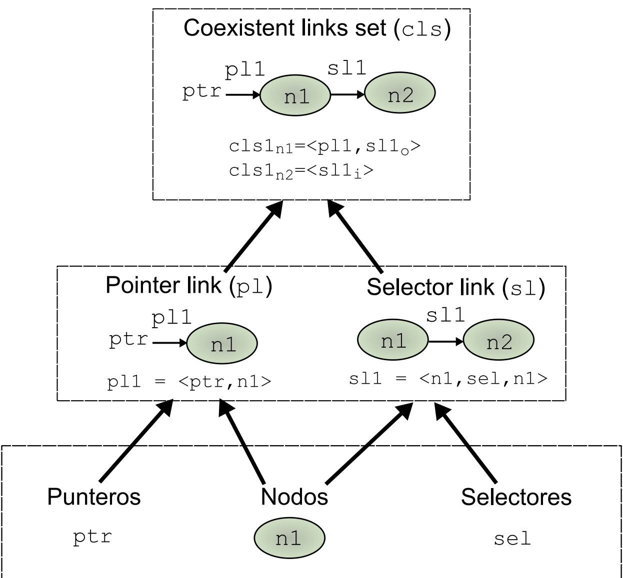


Figura 4.1: Vista jerárquica de los elementos de un grafo de forma.

que debe ser actualizado de forma que apunte al nodo apuntado por otro puntero. Cada tipo de sentencia puntero tiene asociado un comportamiento en el dominio del grafo de forma, que imita o simula el comportamiento de la sentencia en tiempo de ejecución. La forma en la que una sentencia puntero modifica los grafos de forma se define por sus semánticas abstractas.

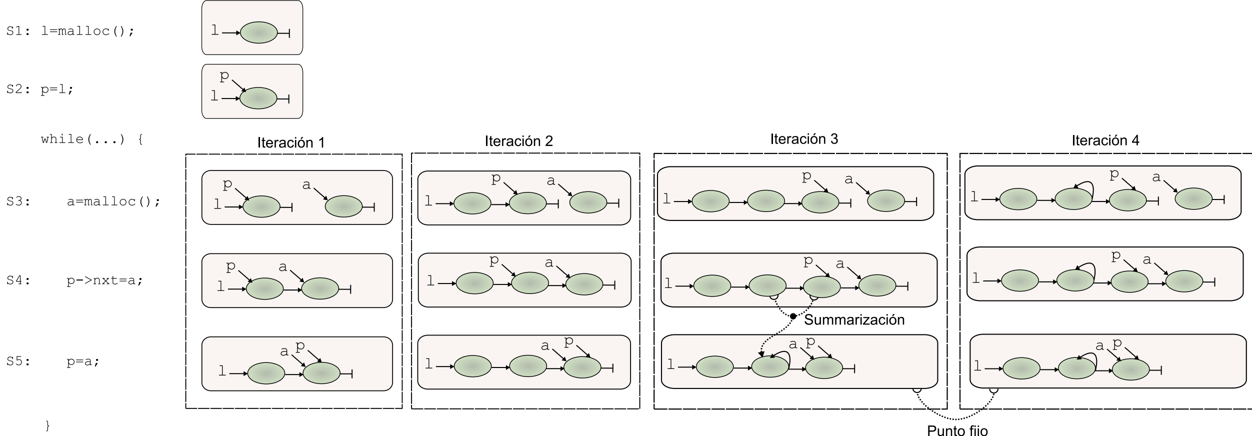


Figura 4.2: Análisis de un bucle hasta que alcanza un punto fijo en los grafos.

El algoritmo para el análisis de forma es un análisis iterativo sobre el flujo de datos. Las sentencias en el programa se ejecutan simbólicamente de manera iterativa, dirigidas por las ramas y bucles del programa. En este proceso, los grafos de forma cambian de acuerdo a la semántica abstracta de las sentencias analizadas. El proceso continúa hasta que los grafos de forma alcanzan un estado estacionario, donde la adición de interpretación abstracta no añade nueva información. A este estado es lo que se refiere el término *punto fijo*.

Estrechamente ligado a la noción de punto fijo del algoritmo, está la operación de *sumarización*. La sumarización, como ya explicamos en el capítulo 2, aquí se aplica como el proceso que mezcla nodos en grafos de forma cuando se estima que son *suficientemente similares*. La simili-

tud o *compatibilidad* de nodos se determina por las relaciones de alias de punteros y *propiedades* ajustables. El proceso de sumarización acota los grafos de forma, limitando el número de nodos que pueden tener. Adicionalmente, la sumarización previene el cambio sin fin de los grafos en el transcurso de la interpretación abstracta iterativa, permitiendo alcanzar la condición de punto fijo.

Por ejemplo, en la Fig. 4.2 la sumarización se produce al procesar `S5:p=a` en la iteración simbólica 3. Esto hace posible en el análisis alcanzar el punto fijo en la siguiente iteración simbólica, donde obtenemos el mismo grafo de forma. Nuevas iteraciones simbólicas en el bucle no producirían nueva información así que el análisis termina.

La sumarización implica pérdida de información a cambio de una representación acotada. También existe otra operación que actúa sobre nodos previamente sumarizados: la *materialización*. Esta operación puede ganar precisión donde los accesos a punteros llevan a cabo lo que se conoce como *strong update* ([48], [49]), descartando enlaces innecesarios en la mayoría de las situaciones. Sin embargo, grafos sumarizados muy conectados pueden hacer imposible que en la operación de materialización se recuperen exactamente los enlaces originales, dejando algunos enlaces conservativos.

La idea completa de la sumarización/materialización es la de plegar/desplegar la estructura en el grafo de forma, dependiendo de la parte de la estructura que está siendo accedida. La parte de la estructura que está siendo accedida por punteros se convierte en enfocada o desplegada, mientras que la parte de la estructura que no es directamente accesible por punteros se convierte en sumarizada o plegada. La Fig. 4.3(a) muestra un ejemplo de sumarización en una lista simplemente enlazada: cuando el puntero `p` tiene un alias con `a`, dos nodos en la mitad de la lista no son accesibles por punteros, y son sumarizados en lo que se llama *nodo sumario*. Por el contrario, en la Fig. 4.3(b) se muestra el recorrido de una lista simplemente enlazada donde un nodo sumario se despliega a través de la materialización de un nuevo nodo, que representa precisamente la localización de memoria apuntada por el puntero `p` en el momento de la ejecución del programa.

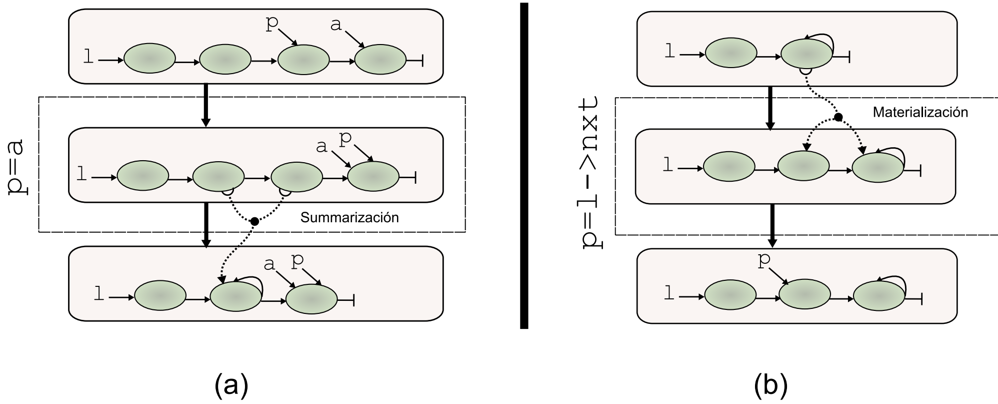


Figura 4.3: (a) Sumarización para compactar la estructura; (b) materialización para enfocar regiones actualmente accedidas.

El criterio básico para la fusión de nodos es sobretodo la sumarización de todos los nodos que no son apuntados por punteros, y de este modo, no son directamente accesibles por ellos. Sin embargo, el análisis también proporciona un conjunto de propiedades, que son evaluadas en las decisiones de sumarización en los casos donde el criterio básico es insuficiente para proporcionar la precisión requerida. Las propiedades son un instrumento clave para controlar cuán precisamente

los grafos de forma capturan las características de la configuración de memoria.

En algún punto durante el análisis, puede haber varios grafos diferentes para una misma sentencia, para capturar todas las posibles configuraciones de memoria que pueden alcanzar esa sentencia desde diferentes caminos de flujo. De hecho, se asocia no sólo un grafo simple, sino un grupo de ellos a cada sentencia y para cada iteración simbólica. Este grupo se denomina *RSSG*, *reduced set of shape graphs*.

Se distingue por un lado el dominio concreto, para las configuraciones de memoria, `mc`, en tiempo de ejecución, y el dominio abstracto, para las abstracciones del grafo de forma. La información del *heap* presentada en tiempo de ejecución pertenece al dominio concreto, y se describe como configuración de memoria. Esta información se abstrae del análisis en el dominio abstracto en la forma de grafos de forma. El dominio abstracto se basa en los grafos de forma como hemos dicho. El elemento base para la representación del *heap* abstracto es el nodo, `n`. En un grafo de forma, cada nodo puede representar un conjunto de localizaciones de memoria desde el dominio concreto, donde cada enlace puede representar una variable puntero o un conjunto de selectores con el mismo nombre.

En el dominio concreto, las localizaciones de memoria representan piezas individuales de memoria ubicadas en el *heap*. Se usan las etiquetas *PLC*, enlaces de punteros en el dominio concreto, y *SLC*, enlaces de selectores en el dominio concreto, para describir las relaciones entre punteros y localizaciones de memoria, y entre localizaciones de memoria a través de selectores respectivamente.

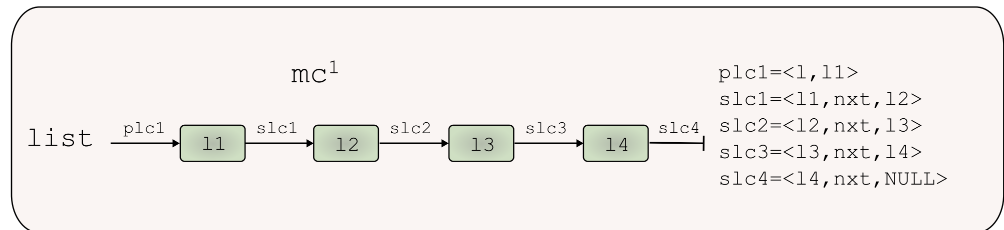


Figura 4.4: Una lista simplemente enlazada de cuatro elementos en el dominio concreto.

En la Fig. 4.4 se muestra explícitamente la información para los *PLC* y *SLC* en una lista enlazada de cuatro elementos. La tupla para *PLC* representa la relación entre la variable puntero y la localización de memoria apuntada por esta variable. En el ejemplo, `l` sería el puntero y `l1` la localización de memoria. De manera similar se explicaría la tupla utilizada para los *SLC*, donde se muestra la relación de punteros entre dos localizaciones a través de un selector. Por ejemplo, para `slc1=<l1,nxt,l2>` interpretaríamos que la localización `l1` está conectada con la localización `l2` a través del selector `nxt`.

Ahora llevamos esta información al dominio abstracto, tal como se muestra en la Fig. 4.5. El conjunto de todos los nodos en el grafo incluye un nodo especial llamado `NULL`, que indica que no hay localización. En un grafo, el número de nodos está limitado por la política de la sumarización. Esta política estipulaba que los nodos son distinguibles por el conjunto de variables puntero que les apuntan. Dos nodos se dice que son *compatibles*, si son indistinguibles en la representación. Concretamente, serán compatibles si son apuntados por el mismo conjunto de punteros. En particular, todas las localizaciones de memoria, que no son apuntadas por punteros, son representadas por un nodo sumario individual.

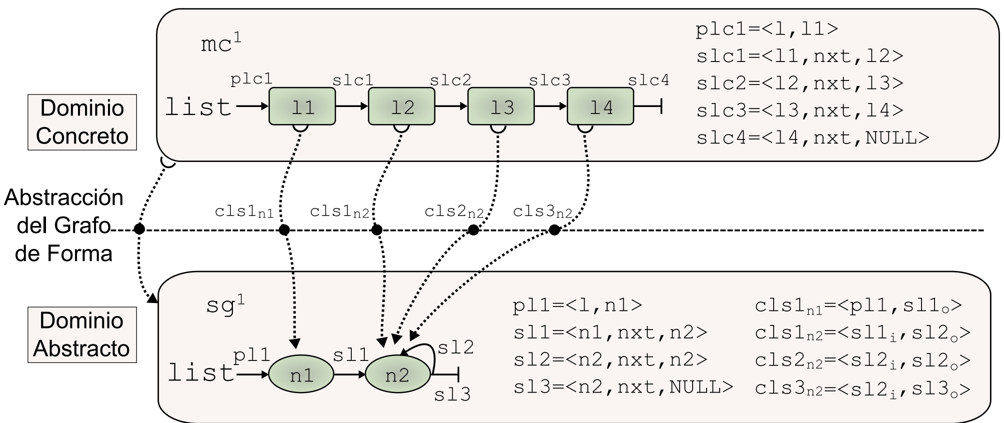


Figura 4.5: Lista simplemente enlazada en ambas representaciones del dominio concreto y abstracto.

Un grafo de forma se muestra como un conjunto de nodos, enlaces de punteros, enlaces de selectores y `coexistent links sets`. Estos elementos se ponen en el dominio abstracto para capturar la estructura de datos recursiva, tal como se muestra en la Fig. 4.5. Todas las tres localizaciones de memoria, `l2`, `l3` y `l4`, se traducen en `n2`, porque no están apuntadas por punteros. Del mismo modo, `l1` se convierte en `n1`, apuntado por el puntero `list`. Se observa como un enlace selector del dominio abstracto puede representar varias instancias de enlaces selectores del dominio concreto. Como en el caso de `sl2=<n2,nxt,n2>`, que representa a `slc2` y `slc3`.

A pesar de esta reducción en el número de elementos, el grafo de forma contiene, dentro de los llamados *cls*, toda la información presente en la configuración de memoria mc^1. Aunque el resultado de esta abstracción conservativa es que el grafo sg^1 representa una lista de *4 o más* elementos. Puede haber más de un `cls` para un nodo. Puesto que un nodo puede representar varias localizaciones de memoria, sus `cls` deben contener todas las posibilidades de enlaces existentes en estas localizaciones de memoria. Por ejemplo, los `cls` para el nodo `n2` (cls1$_{n2}$, cls2$_{n2}$, cls3$_{n3}$), representa los tres diferentes patrones de conexión para las localizaciones de memoria abstraídas para el nodo `n2`. Los atributos *incoming* (`i`) y *outgoing* (`o`) se usan dentro de los `cls` para expresar de forma precisa las posibles conexiones en la estructura.

Destacar que `sl2` en la Fig. 4.5 no implica ningún ciclo en la estructura. Por ejemplo, cls2$_{n2}$=<sl2$_i$,sl2$_o$> indica que `n2` puede representar una localización de memoria que es alcanzada desde otra localización abstraída por `n2`, y abandonada por otro destino también abstraído por `n2`, siendo el origen y el destino diferentes localizaciones de memoria (a pesar de estar representadas en el mismo nodo). En el caso de un ciclo directo, (por ejemplo, una localización de memoria que se apunta a sí misma), se introduce una nueva propiedad, *cyclic*(c). Ésta y otra nueva propiedad, *shared*(s), se usan para definir cómo un enlace de selector en particular se relaciona con los nodos que están enlazados con él. Estos atributos se emplean para capturar enlaces cíclicos (*cyclic links*) y enlaces compartidos (*sharing*).

En la Fig. 4.6 se muestra un ejemplo del funcionamiento de estas propiedades. Las localizaciones `l2` y `l3` se sumarizan en el nodo `n2`. Los enlaces de selector concretos `slc1` y `slc2` se traducen en `sl1` y `sl2` respectivamente, ya que se refieren a selectores diferentes (`nxt` y `prv`). Se aprecia que `sl1` y `sl2` aparecen en diferentes clsj$_{n2}$ de forma que no pueden coexistir, lo que precisamente captura el hecho de que los siguientes `nxt` o `prv` desde `n1` llevan a localiza-

ciones diferentes. Sin embargo, `slc3` y `slc4` (ambos usan `nxt`) son mapeados en `sl3`. De esta forma, $\texttt{sl3}_{s}$ en $\texttt{cls1}_{n3}$ indica que puede apuntarse a una localización representada por el nodo `n3` desde más de una localización diferente representada en el nodo `n2` por el mismo siguiente selector (`nxt`). Por otro lado, $\texttt{sl4}_{c}$ en $\texttt{cls1}_{n3}$ indica que la localización `l4` representada en `n3`, se apunta a sí misma. Se aprecia la diferencia con $\texttt{cls2}_{n2}$=<$\texttt{sl2}_{i}$,$\texttt{sl2}_{o}$> en la Fig. 4.5, lo que indica que una localización representada en el nodo `n2` está apuntando a una localización diferente representada en el mismo nodo.

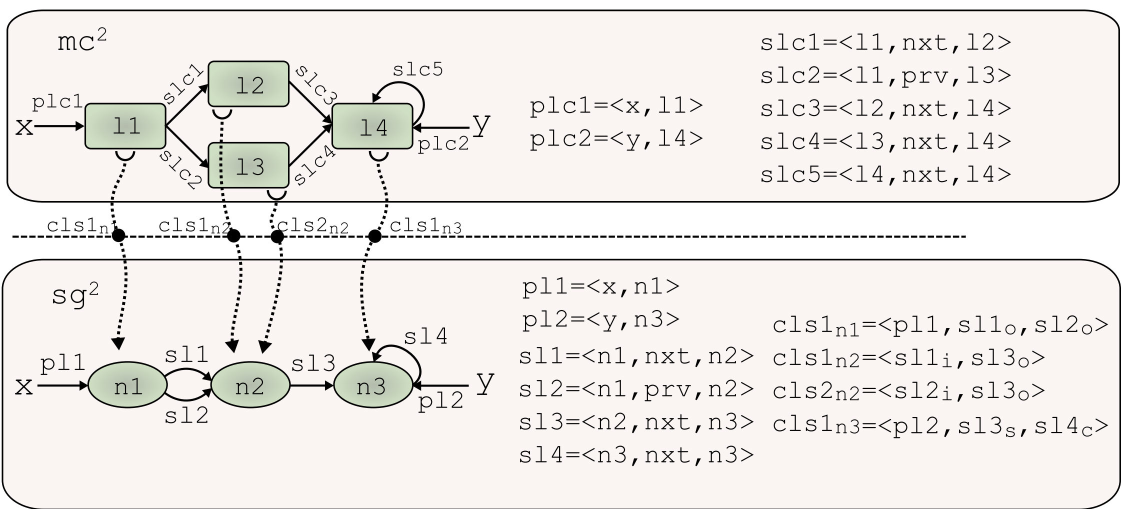


Figura 4.6: Aporte de precisión de otros atributos en la abstracción del heap.

El resultado final del análisis es el conjunto de grafos de forma que describen el estado del *heap* para cada sentencia y para los siguientes posibles caminos de flujo en el programa. Estos resultados son siempre conservativos, mostrando un super-conjunto de todos los posibles grafos de forma que representan el *heap* del programa.

La finalización del análisis está garantizada por la existencia de la sumarización de nodos y la unión de grafos: (i) nodos similares son sumarizados para plegar cada grafo de forma; y (ii) los grafos de forma con las mismas relaciones de alias entre punteros son unidos para plegar cada *RSSG*. Como el número de variables puntero en los tipos de datos recursivos declarados en un programa es fijo y conocido en tiempo de compilación, el número de grafos por sentencia está limitado por las diferencias y mutuamente exclusivas combinaciones de punteros sobre los nodos.

En la Fig. 4.7 mostramos un código ejemplo para la creación de una matriz basada en una estructura dinámica de datos. Dicha estructura está compuesta por una lista cabecera de tipo `t1` enlazada mediante el campo puntero `nxt_t1` y cuyos elementos apuntan a columnas por medio del campo puntero `elem_list`. Cada columna a su vez es otra lista de tipo `t2` enlazada mediante el campo puntero `nxt_t2`.

En la Fig. 4.8 podemos ver una representación gráfica de la matriz creada por el código. Tras analizar el código con la herramienta de análisis de forma, se obtendría el grafo de la Fig. 4.9. Este grafo recoge perfectamente todas las características fundamentales de la forma de la matriz dinámica.

Después de entender un poco el funcionamiento de la herramienta, y de ver el tipo de información, así como la salida que nos ofrece el análisis de forma, presentamos algunos de los casos más

```
struct t1 {  // tipo lista cabecera de columnas
     int header_index;
     int index;
     struct t1 *nxt_t1;      // siguiente columna
     struct t2 *elem_list;   // primer elemento de la columna
};
struct t1 *A, *auxH, *newH;
struct t2 {  // tipo lista columnas
     double data;
     int index;
     struct t2 *nxt_t2;  // siguiente elemento de la columna
};
struct t2 *auxE, *newE, *anchor;

int main () {
     int r, c;
     auxH = NULL;
     while (r < 2000){
          // creación de los elementos de la lista cabecera
          newH = ((struct t1 *) malloc (sizeof (struct t1)));
          if (auxH != NULL) {
               auxH->nxt_t1 = newH;  // enlazado de los elementos de la lista cabecera
          } else {
               A = newH;             // primer elemento de la lista cabecera
          }
          auxH = newH;
          auxE = NULL;
          while (c < 1000) {
               // creación de los elementos de las columnas
               newE = ((struct t2 *) malloc (sizeof (struct t2)));
               if (auxE != NULL) {
                    auxE->nxt_t2 = newE;  // enlazado de los elementos de las columnas
               } else {
                    anchor = newE;        // primer elemento de la columna
               }
               auxE = newE;
          }
          if (newE != NULL) {
               newE$->$nxt_t2 = NULL;
          }
          // enlazado de la columna con su elemento de la lista cabecera
          auxH$->$elem_list = anchor;
     }
     newH$->$nxt_t1 = NULL;
}
```

Figura 4.7: Código que genera la matriz dinámica.

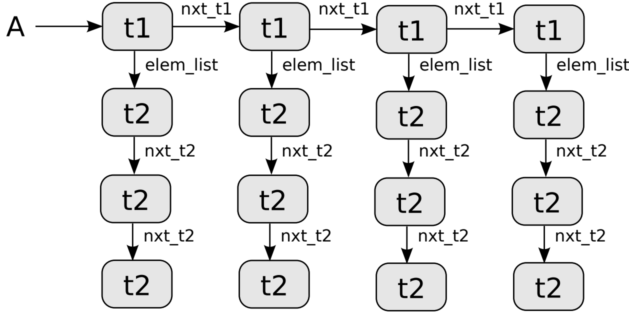


Figura 4.8: Estructura dinámica creada en el código ejemplo que representa una matriz.

costosos de analizar. Se muestran dos códigos en la tabla del cuadro 4.1 que ya fueron mostrados en el capítulo anterior, `Matrix x Matrix` y `Bisort`. La primera columna corresponde al tiempo, en segundos, que tarda en ejecutarse el análisis completo, y la siguiente columna presenta el número total de grafos de forma generados para ese código en el análisis. El tiempo de análisis depende en gran medida del número de grafos de forma generados. Además el número de grafos de forma generados depende del número de sentencias analizadas. Más sentencias provocan más grafos de forma para registrar los estados del *heap* en diferentes iteraciones simbólicas durante el análisis. El número de grafos generados para el código `Bisort` es mayor, pues se trata de un programa más grande con mayor número de procedimientos, además de contener más de una llamada

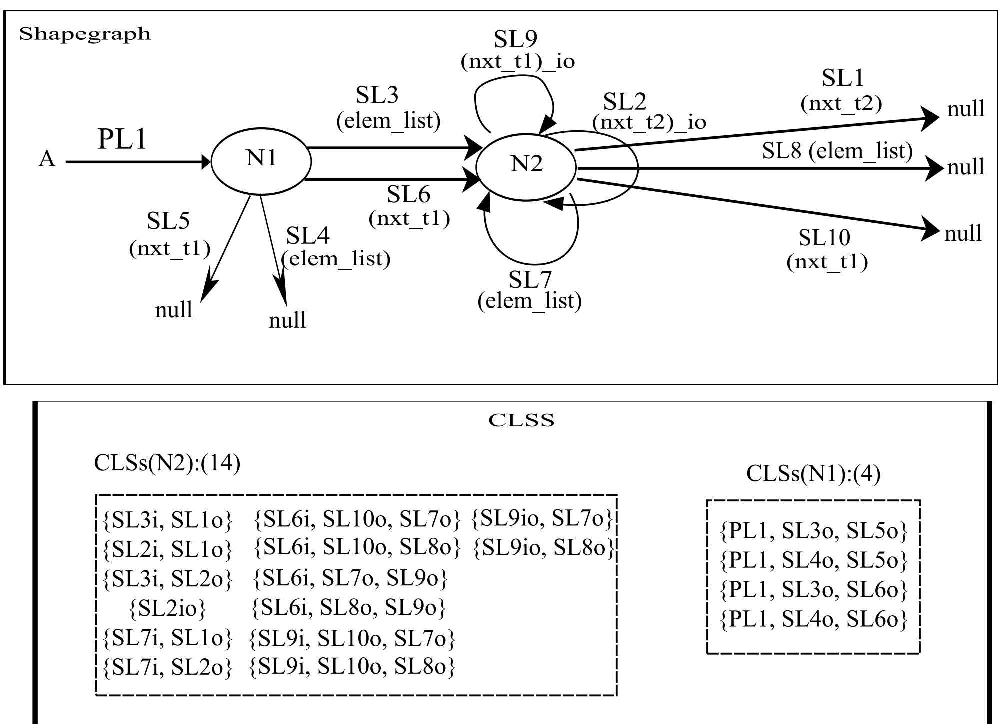


Figura 4.9: Grafo que representa dicha estructura dinámica obtenido por el analizador de forma.

recursiva por procedimiento. El punto fijo se tarda más en alcanzar, cuando hay más de un posible punto de retorno para los grafos de forma obtenidos en una sentencia `return` de una función recursiva.

Código	**Tiempo análisis**	**Nºgrafos forma**
Matrix x Matrix	26.220s	14,611
Bisort	53.138s	39,811

Tabla 4.1: Casos difíciles para el análisis de forma.

Una forma de mejorar la escalabilidad de esta herramienta, para mejorar los resultados como los que acaban de presentarse, y poder abordar el análisis de códigos con varios cientos de miles de sentencias, consiste en realizar un preproceso previo al análisis del código, que reduzca significativamente el número de sentencias a analizar, y, por lo tanto, el número de estados del *heap* que hay que mantener. Este preproceso filtraría sentencias irrelevantes del código y sólo seleccionaría para el análisis de forma posterior, aquellas sentencias que intervienen en la definición/modificación de las estructuras de datos utilizadas en el bucle o sección de código que se pretende analizar.

4.2. Slicing del código

El *Slicing* o *poda* es una técnica que extrae determinadas partes del código, relevantes para algún tipo de aplicación, de acuerdo a algún criterio. El trozo de código o *program slice* extrae

una porción del programa con algún significado semántico, basado en algún criterio de *slicing* seleccionado por el usuario.

Las técnicas de *slicing* han visto una rápida evolución desde la definición original de Mark Weiser [50]. Inicialmente, el *slicing* era sólo estático, es decir, se aplicaba a código fuente sin ningún otro tipo de información del mismo. Posteriormente se introdujo el concepto de *slicing* dinámico [51], que trabaja sobre una ejecución específica del programa (para una traza de ejecución dada). Incluso podemos encontrar técnicas de *slicing* que se aplican más recientemente sobre programación concurrente [52] y programación paralela [53].

4.2.1. Slicing estático

Basado en la definición original de Weiser, un trozo de programa estático, `S`, se compone de todas las sentencias en el programa `P` que pueden afectar al valor de una variable `v` en algún punto `p`. Este trozo se define de acuerdo a algún criterio de *slicing* $C = (x, V)$, donde `x` es una sentencia del programa `P` y `V` es un subconjunto de variables en `P`. El trozo estático incluye todas las sentencias que afectan a la variable `v` para un conjunto de todas las entradas posibles en el punto de interés (es decir, en la sentencia `x`). Los trozos estáticos se calculan encontrando conjuntos consecutivos de sentencias relevantes, de acuerdo a las dependencias de control y de datos.

En el ejemplo de la Fig. 4.10(a), si tomamos como criterio de *slicing*: (`write(sum),sum`), entonces el resultado sería el de la Fig. 4.10 (b).

```
int i;
int sum = 0;
int product = 1;
for(i = 0; i < N; ++i){
    sum = sum + i;
    product = product *i;
}
write(sum);
write(product);
```

(a)

```
int i;
int sum = 0;

for(i = 0; i < N; ++i) {
    sum = sum + i;

}
write(sum);
```

(b)

Figura 4.10: Ejemplo para ilustrar la técnica de *slicing*: (a) Código original; (b) Código podado.

En conclusión, la utilidad y eficacia del *slicing* se basa en la habilidad de facilitar tediosas tareas en la detección y prevención de errores como debugging, testing, paralelización, integración, seguridad del software, comprensión, mantenimiento del software y optimización ([54],[55]). Esto se consigue gracias a la extracción de las partes de interés del programa según el punto de vista del programador, eliminando sentencias irrelevantes y facilitando su comprensión.

El criterio de *slicing* determina el significado semántico de la porción de código que estamos extrayendo del programa, *program slice*. Nadie niega la utilidad de esta técnica para elaborar análisis que mejoren el entendimiento de los programas. Sin embargo, a veces el tamaño de los trozos de código que se extraen pueden resultar demasiado grandes para ser útiles. A medida que se avanza en el estudio del *slicing* de programas, hay cada vez más datos empíricos [55] sobre el tamaño de estos trozos, las herramientas de *slicing*, técnicas y aplicaciones de *slicing*. Así pues

la precisión del análisis llevado a cabo, influye en el tamaño de los trozos de código extraídos. Cuando trabajamos con programas basados en estructuras dinámicas, indudablemente los punteros introducen cierta imprecisión en el análisis. Existen propuestas ([56],[57]) sobre las mejoras que pueden conseguirse en el *slicing* de programas a partir de información dinámica de punteros. Sin embargo estas propuestas están basadas en *slicing* dinámico que requiere la ejecución del programa. Nosotros vamos a utilizar la información precisa que obtenemos a partir de las cadenas DU calculadas con el análisis del capítulo 3, para desarrollar una técnica de *slicing* que nos ayude a realizar un filtrado de sentencias del código a analizar por el análisis de forma presentado previamente. El criterio de *slicing* en nuestro caso, viene determinado por la finalidad última de paralelizar un programa. Para ello necesitamos encontrar las dependencias en determinados bucles o secciones de código conflictivas. Para el análisis de forma, las sentencias relevantes del programa serían aquellas que intervienen en la definición/modificación de las estructuras de datos utilizadas precisamente en estas regiones del programa.

4.2.2. Algoritmo

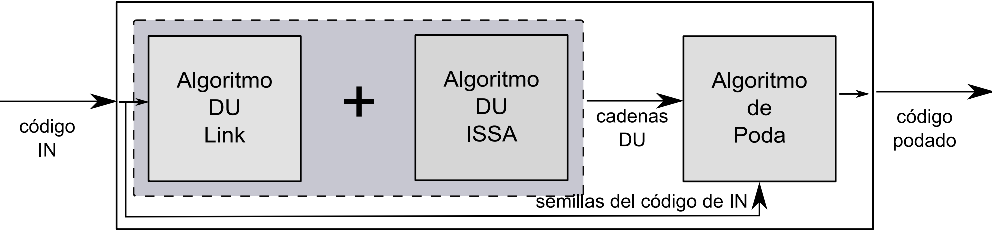


Figura 4.11: Fases dentro del proceso de poda.

El esquema del preproceso diseñado lo ampliamos en la Fig. 4.11. Incialmente el código fuente de entrada es analizado según las técnicas explicadas en el capítulo anterior, para la extracción de las cadenas DU del programa. Seguidamente, se llevaría a cabo el algoritmo de poda, que explicamos aquí, para realizar el filtrado de sentencias. Este algoritmo requiere que en el código fuente de entrada hayan sido marcadas una serie de sentencias *semilla* en aquellas regiones donde la estructura de datos se recorre y que conforman el criterio de *slicing*. Estas semillas nos permiten seleccionar las sentencias relacionadas con la creación y conexión de los elementos del *heap*. Esta selección de semillas se realiza de momento manualmente siguiendo el criterio siguiente: buscando en bucles o secciones de código computacionalmente intensivos de cara a la paralelización, las sentencias que potencialmente pueden producir dependencias. Podría desarrollarse un pase automático más genérico que a partir del código de entrada y el criterio de *slicing* especificado por el cliente, marcara estas semillas. Dejamos esta propuesta como mejora o posible tarea futura relacionada con el trabajo de esta tesis.

El proceso de poda se lleva a cabo en dos etapas, ver Fig. 4.13, de encadenamiento hacia atrás, es decir, buscando las definiciones a las que nos llevan los usos que vamos encontrando al recorrer las cadenas DU. En la primera etapa, se realiza una búsqueda de sentencias de definición de los punteros al *heap*, siguiendo el algoritmo de la Fig. 4.12, por lo que incorpora un filtrado

```
algorithm Poda (Seeds, DU-Lists)
input:
    Seeds: set of seed statements
    DU_Lists: list of DU chains detected
output:
    Stmts_out: set of selected statements
begin
    /* Extract Use expressions from seed statements */
    1: While Seeds is not empty
       2: for each statement S ∈ Seeds do
          % Extract Use in statement S
          3: Uses[S] = extractUses(S)
          /* Find definitions using DU-Lists */
          4: for each use U ∈ Uses[S] do
             5: Defs_List = findDefinitions(U, DU_Lists)
             /* Filter definitions */
             6: for each definiton d ∈ Defs_List do
                7: if d selected then
                   8: if d¬ ∈ Stmts_out then
                      9:% put d in Seeds
                      10:% put d in Stmts_out
                   end
                end
             end
          end
       end
    end
end
```

Figura 4.12: Algoritmo para el filtrado de sentencias.

de las sentencias que va encontrando. En la segunda etapa, esta búsqueda de sentencias sigue una versión más relajada del algoritmo anterior, eliminando el filtrado de sentencias que se realiza en la línea 7. Este proceso se ha dividido en dos etapas para eliminar todas las sentencias intermedias que se van encontrando entre las sentencias semilla, que son el punto de partida, y las primeras definiciones correspondientes de los punteros al *heap* implicados en dichas sentencias semilla. Estas sentencias intermedias que aparecen en la primera etapa de la poda, no son relevantes, no contienen información sobre la creación de la estructura de datos implicada en el recorrido de las sentencias semilla de partida y por eso se filtran.

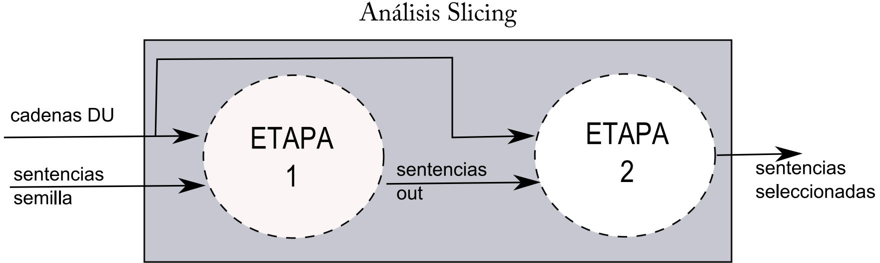


Figura 4.13: Etapas dentro del análisis.

El algoritmo, Fig. 4.12, recibe el conjunto *Seeds* de sentencias semilla que previamente han sido marcadas en el código de entrada. Mientras haya sentencias para analizar en este conjunto, el

algoritmo continúa ejecutándose. Se toma una sentencia del código y se extraen los usos (variables referenciadas en la misma) contenidos en ella. Por ejemplo, para una sentencia semilla tipo `tleft=t→ left`, se obtendrían dos usos: `t` y `t→ left`. Por lo tanto, se buscarían las definiciones para estos usos, a través de las cadenas DU disponibles en la lista *DU-Lists*, lista obtenida a partir del análisis presentado en el capítulo 3.1. El método *findDefinitions* realiza precisamente esta búsqueda, encontrando, si es posible, definiciones para estos usos. En caso de no encontrar ninguna definición, devolvería una lista vacía. Cada definición nueva encontrada, es primero examinada, quedándonos únicamente con las definiciones de los punteros al *heap*. Si esta definición no ha sido analizada anteriormente, se añade al conjunto *Seeds*. Si la definición ha sido analizada con anterioridad, quiere decir que hemos detectado un ciclo, y por lo tanto no se repite de nuevo el análisis de ese recorrido. Este control de ciclos (línea 8 del algoritmo) se lleva a cabo gracias a un conjunto, *Stmts_out*, que funciona como un registro del historial de sentencias visitadas. Cuando el conjunto *Seeds* no tiene más sentencias para analizar, entonces el algoritmo termina. Este conjunto de salida *Stmts_out* pasa a convertirse en el conjunto de entrada de la siguiente etapa de poda. En conclusión, esta primera etapa trata de alcanzar los puntos del programa donde están las definiciones de los enlaces de las estructuras, que son recorridas en las sentencias semilla marcadas inicialmente. La finalidad es eliminar cualesquiera otras sentencias implicadas en otros recorridos intermedios hasta alcanzar estos puntos del programa, porque no aportan información de la creación de la estructura.

El algoritmo de la segunda etapa es el mismo que el de la primera etapa, pero eliminando el filtrado de la línea 7. Aquí no es necesario hacer un filtrado de sentencias, puesto que partimos precisamente de las definiciones encontradas en la etapa anterior y no de las semillas iniciales de partida. Nos interesa encontrar todas las sentencias implicadas en la creación de las estructuras recorridas, sean del tipo que sean. Por lo tanto, aquí todas las sentencias de definición que se van encontrando pasan directamente al conjunto de salida, tras la comprobación habitual de ciclos en la línea 8 en el algoritmo de la Fig. 4.12. La meta de esta segunda etapa es la de recorrer ese tramo de programa que falta por analizar, desde los puntos intermedios alcanzados en la primera etapa, hasta las sentencias originales de creación de las estructuras.

El alcance del punto fijo de este análisis está asegurado puesto que el conjunto donde se realizan las búsquedas, *DU-Lists*, tiene un tamaño fijo, no cambia durante el proceso, y existe un control de ciclos. En definitiva, el conjunto de sentencias que se van analizando, *Seeds*, sí puede variar su tamaño durante el proceso, pero conforme se van extrayendo las sentencias, éstas no vuelven a ser visitadas, y por lo tanto el tamaño del conjunto disminuye hasta quedarse vacío. Finalmente, tenemos a la salida, *Stmts_out*, el conjunto de sentencias de interés que contienen toda la información acerca de dónde son creadas las estructuras de datos que son recorridas en las sentencias semillas originarias.

Mostramos el funcionamiento del algoritmo con un ejemplo a partir del código `TreeAdd` extraído de la suite Olden [47], que vemos en la figura Fig. 4.14 en su representación ISSA. El cuerpo del procedimiento `TreeAdd`, que contiene dos llamadas recursivas, sería el que nos interesaría paralelizar. Para detectar si es paralelizable o no, el test de dependencias debería comprobar si estas dos llamadas a `TreeAdd` son independientes. Por lo tanto, en este ejemplo las sentencias semillas que se marcarían en el código fuente serían `S25` y `S29` dentro del procedimiento `TreeAdd`. El resultado después de aplicar el algoritmo de poda, explicado anteriormente se muestra en la figura Fig. 4.15. Este código podado sería el que pasaría a la entrada del correspondiente análisis de

```
void main(){
S1:      root_0_3=TreeAlloc(level_0_3);
S2:   root_0_3=new_TreeAlloc_2_1;
S3:   TreeAdd(root_0_3);
S4:   return(0);
}
tree *TreeAlloc(int level){
S5: level_0_1=
Iphi(newlevel_0_1,newlevel_1_1,level_0_3);
S6: if (level_0_1==0){
S7:    new_0_1=NULL;
S8: }else{
S9:    newlevel_0_1=(level_0_1-1);
S10:    new_1_1=(struct tree *)malloc();
S11:    new_1_1->val=1;
S12:    left_0_1=TreeAlloc(newlevel_0_1);
S13:    left_0_1=new_2_1;
S14:    new_1_1->left=left_0_1;
S15:    left_1_1=NULL;
S16:    right_0_1=TreeAlloc(newlevel_0_1);
S17:    right_0_1=new_2_1;
S18:    new_1_1->right=right_0_1;
S19:    right_1_1=NULL;
 }
S20: return new_2_1;
}
```

```
void TreeAdd(tree *t){
S21: t_0_2=
         Iphi(tleft_0_2,tright_0_2,root_0_3);
S22: if (t_0_2==NULL){
S23:    totalval_0_2=0;
S24:  }else{
S25:    tleft_0_2=t_0_2->left;
S26:    leftval_0_2=TreeAdd(tleft_0_2);
S27:    leftval_0_2=totalval_2_2;
S28:    tleft_1_2=NULL;
S29:    tright_0_2=t_0_2->right;
S30:    rightval_0_2=TreeAdd(tright_0_2);
S31:    rightval_0_2=totalval_2_2;
S32:    tright_1_2=NULL;
S33:    value_0_2=t_0_2->val;
S34:    totalval_1_2=value+leftval
                  +rightval;
S35:    t_0_2->val=totalval_1_2;
  }
  totalval_2_2=
      phi(totalval_0_2,totalval_1_2);
  return;
}
```

Figura 4.14: Código del programa Treeadd.

forma.

```
void main(){
   ;
   return(0);
}
tree *TreeAlloc(int level){
 if (level==0){
    ;
 }else{
    new=(struct tree *)malloc();
    ;
    left=TreeAlloc(newlevel);
    new->left=left;
    ;
    right=TreeAlloc(newlevel);
    new->right=right;
    ;
 }
 return new;
}
```

```
void TreeAdd(tree *t){
 if (t==NULL){
    ;
  }else{
    ;
    ;
    ;
    ;
    ;
    ;
    ;
  }
  return;
}
```

Figura 4.15: Código Treeadd podado.

4.3. Complejidad

El algoritmo de poda de la Fig. 4.12 presenta una complejidad lineal, ya que es directamente proporcional al tamaño del conjunto de cadenas DU que recibe a la entrada. Llamamos `N` al

tamaño del conjunto de cadenas DU detectadas por el algoritmo DU, siendo el número total de sentencias de definición y uso implicadas en el conjunto. Cada etapa de poda realiza como mínimo una pasada sobre este conjunto, y como máximo `k_max` pasadas, siendo `k` el número de veces que se recorre el conjunto. Por lo tanto, el verdadero valor de `k` lo definimos comprendido entre 1 y `k_max`. El caso peor para `k_max` nunca puede ser `N` puesto que para cada etapa no se repiten las visitas de las sentencias Además estimamos que cada etapa tiene su valor `k_max` correspondiente. Por lo tanto, la cota superior de complejidad temporal del algoritmo podríamos situarla en *O(k_max1*N+k_max2*N)*.

4.4. Resultados Experimentales

Para los resultados experimentales, hemos considerado la mayoría de los programas que ya fueron presentados en el capítulo anterior. En la tabla 4.16 se muestran medidas de tiempo, en segundos, del análisis de poda completo sobre cada uno de estos códigos. Se observa que el análisis de poda no es un análisis costoso comparado con el análisis DU. En la Fig. 4.17 representamos la reducción del tamaño de los códigos, en cuanto a número de sentencias, conseguida para cada programa. El porcentaje medio de reducción conseguido se sitúa cerca del 50 %, superando este valor en la mayoría de casos.

Código	**Análisis Poda**
Matrix x Vector	0.182s
Matrix x Matrix	0.258s
TreeAdd	0.043s
Power	0.66s
Em3d	2.00s
Bisort	0.918s

Figura 4.16: Medidas de tiempo del slicing para los códigos.

Código	**Original**	**Podado**	**Aceleración**
Matrix x Vector	2.789s	0.798s	71.387 %
Matrix x Matrix	26.220s	1.355s	94.832 %
TreeAdd	4.698s	1.627s	65.368 %
Power	8.125s	0.481s	94.08 %
Em3d	34.16s	7.363s	78.445 %
Bisort	53.138s	1.802s	96.609 %

Tabla 4.2: Medidas de la aceleración conseguida en el análisis de forma.

En la siguiente tabla 4.2, mostramos las medidas de tiempo, en segundos, del análisis de forma según el código fuente de entrada que es analizado. Primero ejecutamos el análisis de forma sobre los códigos originales, resultados de la columna etiquetada como *Original*. En la siguiente columna se presentan los tiempos medidos para la ejecución del análisis de forma sobre el código

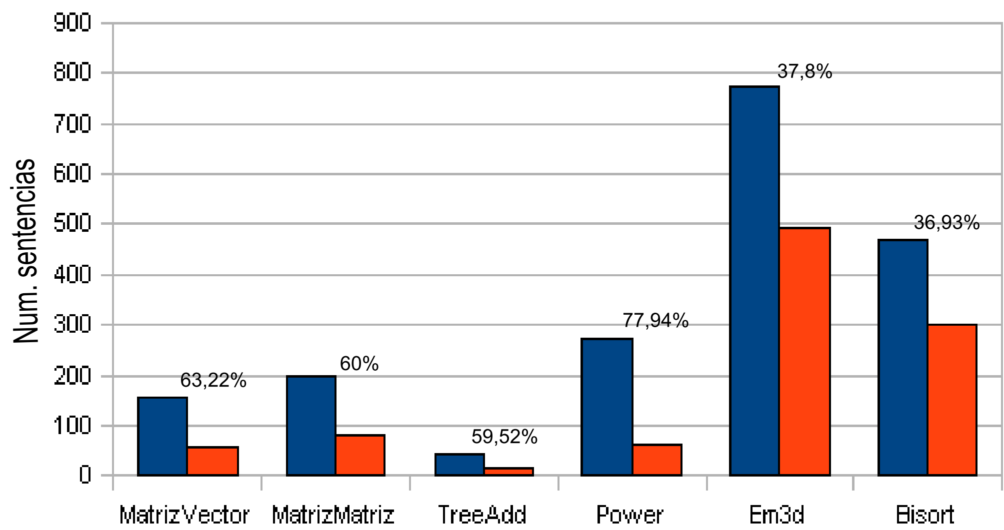


Figura 4.17: Reducción del tamaño de los códigos.

podado, tras realizar el *slicing*. Los resultados del análisis son exactamente iguales en ambos casos, indicando que la poda ha extraído correctamente la información precisa suficiente para alcanzar el mismo resultado. Por último, se miden las aceleraciones conseguidas, a partir de estos dos tiempos, en cada caso. Por término medio, el porcentaje de reducción es superior al 50 %.

4.5. Conclusiones

En este capítulo hemos presentado una posible aplicación a la información obtenida con el análisis DU sobre programas basados en estructuras dinámicas de datos. El algoritmo de *slicing* propuesto se ha diseñado para sacar el mayor rendimiento a la información aportada por el análisis DU. Concretamente se buscaba no sólo acelerar una herramienta de análisis de forma, sino aumentar el rango posible de casos a tratar por la misma. Al estar basada dicha herramienta en la ejecución simbólica de sentencias, y llevar a cabo un análisis complejo, podíamos encontrar casos de códigos con estructuras muy complejas o con un gran volumen de sentencias, directamente no abordables por esta herramienta. Gracias al proceso de *slicing* que presentamos, hemos aumentado el abanico de casos abordables para esta herramienta. Además, la complejidad tanto de la etapa del preproceso como de la etapa de *slicing* es polinómica, mientras que la complejidad del análisis de forma es exponencial. A partir de los resultados experimentales se observa que la aceleración que experimenta la herramienta de análisis de forma ha sido la esperada. El preproceso DU junto con el análisis de forma optimizado contribuyen también en el test de dependencias que presentamos en el siguiente capítulo.

5 Análisis de dependencias

En este capítulo contribuimos con un test de dependencias para códigos basados en estructuras dinámicas de datos. Este test de dependencias está basado en un nuevo análisis de conflictos de caminos de acceso sobre grafos de forma. La finalidad es mejorar los resultados obtenidos por un antiguo test de dependencias desarrollado en nuestro grupo de investigación [37].

Primero pasaremos a introducir un poco la motivación y funcionamiento general del test de dependencias. Luego explicaremos en detalle la información necesaria para poder aplicar esta técnica y la presentaremos formalmente. Mostraremos un ejemplo para entender cómo funciona el método, terminando con la presentación de algunos resultados experimentales y las principales conclusiones.

5.1. Introducción

Algunos de los trabajos previos en detección de dependencias combinan técnicas de análisis de dependencias con análisis de punteros [41], [58], [59], [60], [61], [62]. Horwitz y col. [58] desarrollaron un algoritmo para la detección de dependencias a través de la detección de interferencias en sitios comunes alcanzados. Larus y Hilfinger [59] proponen la identificación de conflictos de acceso a través de grafos de alias que usan expresiones de *paths* para nombrar las localizaciones. Hendren y Nicolau [41] usan matrices de *paths* para almacenar la información de conexionado entre los punteros y presentan una técnica para identificar interferencias entre los cómputos de programas con estructuras acíclicas. El énfasis de estas técnicas está en la identificación de dependencias a nivel de llamada a función, y no consideran la detección en el contexto de un bucle

como en el enfoque de nuestra propuesta. Johnson y Pingali [15] propusieron la construcción de un grafo específico de dependencias, *Dependence Flow Graph (DFG)*, que justifica la utilidad de las cadenas DU, así como la llamada representación SSA en la detección de dependencias de un programa pero no tratan con punteros al *heap*.

Otros autores ([61], [62], [39]) han propuesto tests de análisis de dependencias basados en análisis de forma en el contexto de bucles que recorren estructuras dinámicas de datos. Estos estudios resultan más cercanos a nuestro trabajo. Por ejemplo, Ghiya y Hendren [61] propusieron un test para la identificación de LCD's que se basa en la forma de la estructura de datos que es recorrida (árbol, DAG, o cíclica), así como en el cálculo de los *paths* de acceso para los punteros de las sentencias que son analizadas. Básicamente, su test identifica dependencias en programas con estructuras de datos tipo árbol o en bucles que recorren estructuras DAG cíclicas que han sido confirmadas por el programador como acíclicas, y donde los *paths* de acceso no contienen campos tipo puntero. La confirmación manual de bucles que recorren los DAG o las estructuras cíclicas es necesaria para la detección automática de LCD's. Otra limitación de este enfoque es que las estructuras de datos deben permanecer estáticas durante el recorrido de los datos dentro de los bucles analizados.

Para resolver algunas de las limitaciones previas Hwang y Saltz propusieron una nueva técnica para identificar LCD's en programas que recorren estructuras cíclicas [62], [39]. Este enfoque identifica de forma automática patrones de recorrido acíclico en estructuras categorizadas como cíclicas. Para este propósito, el algoritmo aisla los patrones de recorrido de la estructura completa de datos, y luego deduce la forma de estos patrones de recorrido (árbol, DAG o cíclica). Una vez extraída la información de la forma del patrón de recorrido, el análisis de dependencias se aplica para la detección de LCD's. En resumen su técnica identifica LCD's en programas que navegan estructuras cíclicas pero en un recorrido limpio tipo árbol. Por otro lado, su análisis puede sobrestimar la forma del recorrido cuando la estructura de datos es modificada durante el recorrido. En estas situaciones, el algoritmo de forma produce patrones de recorrido DAG o cíclicos, detectando dependencias. En otro trabajo reciente [40], el análisis del *heap* que ofrecen, está enfocado hacia colecciones de librerías en Java preparadas manualmente.

Nuestra propuesta permite analizar códigos C generales, que crean, recorren y modifican estructuras dinámicas complejas. A diferencia de la mayoría de aproximaciones, podemos analizar códigos con estructuras de datos que son modificadas durante el recorrido. Dicho test se basa en:

- un análisis estático de grafos de forma que captura información topológica sobre la conectividad entre localizaciones de memoria,
- una representación de *paths* basado en expresiones que describen cómo las sentencias acceden a las localizaciones de memoria.

La principal característica de este test es que lleva a cabo un análisis de conflictos basado en la descomposición de *paths* en una secuencia de *subpaths*, (entrada, navegación y cola), y en la proyección de dichos subpaths sobre los grafos de forma. De esta forma es posible identificar las localizaciones (nodos alcanzados en el grafo por un puntero o un selector) visitadas por cada *subpath*. Se lleva a cabo de forma recursiva este análisis de conflictos entre las localizaciones alcanzadas por cada *subpath*, y el proceso termina cuando el último componente de los *paths* ha sido comprobado.

Este test mejora las prestaciones de un análisis de dependencias previo [63], desarrollado por nuestro grupo, puesto que el nuevo método no necesita la interpretación abstracta de las sentencias para las que el análisis de conflictos se lleva a cabo. El nuevo test ya no necesita ejecutar simbólicamente todas las sentencias de los bucles o funciones a paralelizar y, por tanto, es de una complejidad sustancialmente menor al anterior. Se ha conseguido una reducción significativa de los tiempos. En realidad, la complejidad de este nuevo test de dependencias es polinómica, mientras que el anterior test tenía complejidad exponencial.

El objetivo del test es computar cuándo, para distintas sentencias, interfieren los paths de acceso de lectura y escritura. Para la implementación de este estudio de interferencias de los paths de acceso, hay que diseñar un algoritmo que sea capaz de proyectar (sobre los grafos que representan la forma de las estructuras de datos) el recorrido o navegación que está teniendo lugar, dentro del bucle, o sección de código, que estamos analizando. En caso de que podamos demostrar que ningún acceso de escritura alcanza o es alcanzado por algún acceso de lectura, entonces podremos afirmar que tal bucle o sección de código está libre de dependencias, y puede, por tanto, ser paralelizado [39]. Nuestro método trata de identificar dependencias de datos debidas a punteros que apuntan al *heap* en dos escenarios habituales: (i) bucles que recorren estructuras de datos dinámicas, identificando dependencias que pueden aparecer entre dos iteraciones del bucle, y ii) llamadas a funciones recursivas que recorren estructuras de datos dinámicas, identificando accesos en conflicto en diferentes llamadas recursivas. El nuevo test de dependencias se presenta en la sección 5.5, diferenciando el caso general, sección 5.5.1, para analizar códigos que no modifican la estructura de datos durante el recorrido, del caso donde sí existe modificación de la estructura, sección 5.5.2. Previamente, en las secciones 5.2, 5.3 y 5.4, presentamos la información que se ha de colectar para poder aplicar el test de dependencias.

5.2. Detección de los Punteros de Inducción

Se ha visto la utilidad de las variables de inducción para la detección de dependencias de datos [64]. Nosotros nos centramos específicamente en los llamados punteros de inducción, usados en bucles, o llamadas recursivas, para recorrer estructuras dinámicas de datos. Como veremos, serán fundamentales para la recolección de información de los paths.

En un programa, las variables puntero dentro de un bucle o en el cuerpo de una función recursiva pueden clasificarse en tres grupos: punteros globales, locales y punteros de iteración. Los punteros globales son punteros que se definen antes de entrar en el cuerpo de un bucle, o función, y no se redefinen durante la ejecución del mismo. Los punteros locales se definen en cada iteración y son sólo referenciadas en las iteraciones en las que son definidas. Por el contrario, los punteros de iteración pueden pasar su contenido entre iteraciones. Los punteros de iteración se dividen en dos tipos, los punteros de inducción y los que no son de inducción. Los *Punteros de Inducción*, también llamados *punteros de navegación*, son usados en bucles, o llamadas a función, para recorrer estructuras dinámicas de datos. Una variable de inducción se caracteriza porque es definida dentro del cuerpo de un bucle o función y luego incrementada o decrementada una cantidad fija en cada iteración. En el caso de variables puntero, este avance se produce en el recorrido de la estructura de datos, estableciendo una expresión de recorrido. Esta información, unida a la información de la forma de la estructura, nos permite detectar dependencias entre accesos. Naturalmente estos pun-

```
while (p!= NULL){
    p -> x = p -> y * 5;
    p = p -> next;
}
```

(a)

```
while (p!= NULL){
    for (i=0;(p!= NULL)&(i<max_proc);
            p=p->next)
    {   ptr_arr[i]=p;
        i=i+1;
    }
    forall (j=0;j<i;j++)
    {   p= ptr_arr[j];
        compute(p);
    }
}
```

(b)

Figura 5.1: (a) Ejemplo de puntero de inducción en bucle (b) Bucle transformado para solucionar la dependencia debida a p.

teros introducen también una dependencia entre iteraciones diferentes del bucle, lo que se conoce como *pointer-chasing problem* [65]. Sin embargo, hay técnicas para solucionarlo, partiendo de que no existan otras dependencias ([21], [38]). En el ejemplo de la Fig. 5.1(a), `p` sería el *puntero de inducción*, y en 5.1(b) mostramos un ejemplo de transformación de código que resuelve la dependencia debida a `p`.

La representación SSA nos ayuda a encontrar más fácilmente los punteros de inducción. Cada definición tipo función *phi* en la cabecera del bucle o función se corresponde con un puntero de iteración, mientras que cada definición en el cuerpo del bucle crea un puntero local. Para distinguir los punteros de inducción de los que no son de inducción, se calculan las expresiones de los caminos de acceso de los punteros. Los punteros de iteración son los que aparecen en las funciones *phi*, de manera que estas sentencias son las que deben ser analizadas. Para una sentencia *phi* tipo $p_i = phi(p_j, p_k)$, debe cumplirse que p_j sea el valor del puntero a la entrada del bucle o función, y p_k el valor de la iteración anterior. En el ejemplo de la figura Fig. 5.2(a) se produce un ciclo que empieza y acaba en la función *phi*, si observamos la sentencia `S3`, el puntero ptr_2 nos lleva a la sentencia `S4`, donde el puntero ptr_3 nos lleva de nuevo a la sentencia *phi* de partida, por lo tanto ptr_2 es un puntero de inducción y el camino de acceso para el puntero ptr_3 sería $ptr_2 \rightarrow next$. Sin embargo, en la figura Fig. 5.2(b), el puntero $list_2$ en la sentencia `S3` nos llevaría a la sentencia `S5` con $node_1 \rightarrow next$ y no sería un puntero de inducción.

```
S1:   ptr1=list;
S2:   do while (ptr2)
S3:      ptr2=phi(ptr1,ptr3);
         .
         .
S4:      ptr3=ptr2->next;
S5:   end
```

(a)

```
S1:   list1=null;
S2:   do i=1..N
S3:   list2=phi(list1,list3);
S4:      node1=new();
S5:      node1>next=list2;
S6:      list3=node1;
S7:   end
```

(b)

Figura 5.2: (a) Hay puntero de inducción;(b) No hay puntero de inducción.

La diferencia entre los que son y no son punteros de inducción se basa en que el valor que toma el puntero de inducción en la iteración anterior será referenciado, y asignado al valor del puntero de inducción en la iteración actual. En el caso de los punteros que no son de inducción,

el valor del puntero de la iteración anterior es referenciado pero no asignado al valor del puntero actual. El ejemplo de la figura Fig. 5.2(a) muestra como el valor del puntero de inducción ptr_2 en la iteración anterior se referencia, y en S4 se le asigna el valor ptr_3, que pasará a la siguiente iteración. Por otro lado, el valor en la iteración previa del puntero que no es de inducción, $list_2$, se referencia pero no se asigna a $list_3$. Encontramos ejemplos en la literatura que demuestran la utilidad de la representación SSA y de las cadenas DU para la detección de variables de inducción en bucles [66], [67]. Su detección en bucles nos ayuda a la hora de determinar las dependencias que se pueden encontrar en los mismos, aunque esta técnica se puede extender para detectar los punteros de inducción en el cuerpo de las funciones recursivas.

```
algorithm FindInductionPointers (P,DU)
input:
    P: set of phi-functions at loop headers
    DU: list of DU chains detected
output:
    Induction: set of induction pointers
    InductionPaths: set of paths for each induction pointer
begin
    /* analyze each Phi function */
    for each definition phi in P do
        /* extract referenced pointers in phi */
        Pointers = extractUses(phi)
        while ((not found)||(no more uses))
            /* find definitions using DU-Lists */
            /* get pointer p from Pointers */
            Def = findDefinition(p, DU)
            if Def equals defined pointer in phi then
                /* induction pointer found */
                /* put Def in Induction */
                update paths_temp
                put paths_temp in InductionPaths(Def)
            else
                newUses = extractUses(Def)
                /* put newUses in Pointers */
                /* refresh paths with traversed fields*/
                update paths_temp
                /* continue search process */
            end
        end
    end
end
```

Figura 5.3: Algoritmo para la detección de punteros de inducción.

En la figura Fig. 5.3 se muestra el algoritmo para la detección de los punteros de inducción. A la entrada recibe el conjunto de cadenas DU detectadas por el análisis descrito en el capítulo 3 y una lista de las funciones *phi* para los punteros recolectadas en las cabeceras de los bucles y funciones recursivas del programa. Recordamos el formato de las funciones *phi*, $p_n = phi(p_1, p_2, ..., p_k)$. Se examina cada función *phi* extrayendo los punteros que son referenciados $p_1, p_2, ..., p_k$. Para cada puntero, se lleva a cabo un proceso de búsqueda de su definición usando las cadenas DU disponibles. Si la definición encontrada coincide con la función *phi* de partida, entonces no se continúa la búsqueda, pues se ha encontrado un puntero de inducción. En caso contrario, a partir de esa definición el proceso sigue buscando parejas de usos-definiciones, hasta que encuentre dicha función *phi*, o no se encuentren más expresiones. De forma paralela, también se lleva un registro de los campos que son recorridos por estos punteros, y se almacenan en la variable temporal *paths_temp*.

Una vez encontrado el puntero de inducción, se actualiza la información de la navegación asociada a ese puntero. Al final tenemos un conjunto con los punteros de inducción encontrados en el programa y sus correspondientes expresiones de navegación. Esta información será usada en el análisis de dependencias. En el ejemplo de la Fig. 5.2(a) la expresión de navegación sería $(next)^*$.

5.3. Paths

Como ya hemos comentado, nuestro test de dependencias trata de descubrir LCDs basándose en la forma de la estructura de datos, y viendo qué sitios de dicha estructura de datos visitan las sentencias generadoras potenciales de LCDs. Para poder determinar estos sitios visitados, es fundamental expresar el recorrido que hacen las sentencias a través de la estructura de datos. Por ejemplo, en el código de la Fig. 5.4(a) se recorre la estructura de datos apuntada por `A` avanzando en cada paso del bucle por el selector nxt_t1. Hay dos accesos al campo *index*(uno de escritura, sentencia `SW`, y otro de lectura, sentencia `SR`), que podrían generar un LCD si un elemento pudiera ser visitado por dichas sentencias en el recorrido que se realiza en dicho bucle.

(a)	(b)

<pre> p = A;
  while (p != NULL) {
SR: i = p->index;
SW: p->index = i+1;
    p = p->nxt_t1;
  }</pre>
	<pre> p = A;
	  q = A;
	  while (p != NULL) {
	SR: i = p->index;
	SW: q->index = i+1;
	    p = p->nxt_t1;
	    if (cond)
	      q = q->nxt_t1;
	  }</pre>

Figura 5.4: Código ejemplo de recorrido (a) con un navegador, (b) con dos navegadores.

Este código está basado en la estructura de datos de la matriz dinámica que se tomó como ejemplo en el capítulo anterior para mostrar la generación de un grafo de forma, y que ahora recordamos en la Fig. 5.5.

```
struct t1 {  // tipo lista cabecera de columnas
     int header_index;
     int index;
     struct t1 *nxt_t1;      // siguiente columna
     struct t2 *elem_list;   // primer elemento de la columna
};

struct t2 {  // tipo lista columnas
     double data;
     int index;
     struct t2 *nxt_t2;  // siguiente elemento de la columna
};
```

Figura 5.5: Estructura de datos para una matrix dinámica.

Path:	$Path \in PATH$, $Path ::=< s, ce : (nce\|pre) : nav : tail >$ donde $s \in STMT$ y $ce\|nce\|pre\|nav\|tail$ son subpaths
Subpaths:	$subpath ::= step_1.step_2.\dots.step_m\|null$
Step:	$step_k ::= x_i \in PTR\|(sel_1\|sel_2\|\dots\|sel_m)^{(*\|+)}$ donde $sel_i \in SEL$
Access Path	$AP ::=< s, x_1.tailexp_1 : x_2.tailexp_2 : \dots : x_n.tailexp_n >$ donde $s \in STMT, x_1, x_2, \dots x_n \in PTR$ y $tailexp_1, tailexp_2, \dots tailexp_n$ son subpaths de la forma: $tailexp_i ::= step_1.step_2.\dots.step_m\|null,$ siendo $step_k ::= (sel_1\|sel_2\|\dots\|sel_m)$, donde $sel_i \in SEL$

Tabla 5.1: Definiciones de paths y access paths.

5.3.1. Representación de los paths

En la tabla 5.1 se muestran todas las definiciones para la representación de los paths. El *path* que consideramos en el análisis es una tupla que contiene la sentencia para la que se ha calculado el camino, y una expresión, llamada *path expression*, que consiste en una secuencia de subpaths ordenados delimitados por el operador ":". Cada uno de estos subpaths es una secuencia de campos conectados por el operador ".", o la expresión *null*, para representar que la correspondiente expresión del subcamino está vacía. Según sea el caso, un *campo* puede ser tanto una variable puntero como un campo selector. Las múltiples ocurrencias de un mismo campo selector se representan por "$+$", indicando al menos una ocurrencia, o "$*$", indicando cero o más. Estos operadores son empleados para la representación de expresiones regulares. Denominamos *PATH* al conjunto de todos los caminos de un programa. Concretamente, un camino *Path* $\in$ *PATH* se define como una tupla de cuatro elementos: $< S, PE : NAV : TAIL >$ donde S es una sentencia, PE es la expresión del *punto de entrada* compuesta de dos partes (ce y $(nce : pre)$), NAV es la expresión de la *navegación*, y $TAIL$ la de *cola*. Explicamos cada una de las partes de la expresión del path:

- *Punto de Entrada*: representa aquellos sitios en la estructura de datos donde puede comenzar el recorrido del bucle actual del cual depende el acceso. La expresión está formada por dos partes, *ce* y *(nce:pre)*. El campo *ce* representa lo que llamamos entrada de expresión común en el path. Por otra parte, *(nce:pre)* representa a dos subpaths mutuamente exclusivos: *nce* define la expresión de entrada no común en el path, y *pre* incluye la expresión de lo que llamamos *preámbulo*, que explicaremos más adelante. De este modo cuando uno de los dos es distinto de *null*, forzosamente el otro campo debe ser *null*, ya que no pueden darse al mismo tiempo ambas situaciones.

- *Navegación*: se define como la expresión de navegación para el recorrido del *path*. Informa de los sitios visitados a partir de los *puntos de entrada* anteriores siguiendo la expresión de navegación. Para esto, el subpath simplemente debe informar de la expresión de navegación (conjunto de selectores que se utilizan para avanzar de una iteración a la siguiente).

- *Cola*: la expresión de la cola en el path, expresa el camino relativo desde cada uno de los

sitios visitados en la navegación hasta el sitio sobre el cual se realiza el acceso (de escritura o de lectura).

Se distinguen varios tipos de puntos de entrada en un *path*: un punto de entrada no común, un punto de entrada común, seguido de un preámbulo, o un punto de entrada común seguido de un punto no común. La razón de esta clasificación es que buscamos los conflictos entre dos paths, de manera que podamos caracterizar cuando hay o no hay un punto de entrada común entre dos paths. El preámbulo aporta información relevante, se trata de una expresión del subpath que muestra el desfase entre el punto de entrada común y la navegación correspondiente. Para el ejemplo del código de la Fig. 5.4(a), el punto de entrada sería común, el sitio apuntado por la variable puntero `A`. Éste sería el conjunto de *Puntos de Entrada*, *pe*, al recorrido que se realiza en el bucle (en este caso sería $\{\texttt{A}\}$). La *navegación*, *nav*, en el ejemplo sería la expresión $(nxt_t1)^*$. En nuestro ejemplo, podemos observar que los accesos de escritura (SW) y de lectura (SR) se realizan sobre el mismo elemento que se visita en la navegación (se utiliza el propio navegador p o puntero de inducción para realizar los accesos). Por tanto, para ambos accesos del ejemplo el *tail* sería la cadena vacía. Finalmente, la tupla del path sería la misma para ambas sentencias de lectura y escritura, siendo $< A : (nxt_t1)^* : - >$.

5.3.2. Cálculo de los Access Paths

La información necesaria para construir los paths se extrae de los llamados *access paths*. Un *Access Path*, *AP*, es una tupla que contiene una sentencia, y una secuencia de expresiones de la forma $x_i.tailexpr_i$, que llamamos la expresión del access path, tal como se muestra en la tabla 5.1.

$AP ::=< s, x_1.tailexp_1 : x_2.tailexp_2 : \ldots : x_n.tailexp_n >$

donde x_i es una variable puntero y $tailexpr_i$, es un subpath que consiste bien en la expresión *null*, o bien una cadena de campos llamados *step*, $tailexpr_i ::= step_1.step_2....step_m$, siendo un campo en este caso un selector $step_k ::= (sel_1|sel_2|...|sel_m)$. En particular, el primer valor x_1 es un valor de entrada de un puntero, mientras que el resto de x_i para $i = 2...n$ son punteros de inducción. El número `n` depende del número de punteros de inducción usados para recorrer la estructura de datos en el acceso a la sentencia `s`. Esta expresión es una instancia de puntero seguida de una cadena de selectores o nombres de campo conectados por el operador ".", donde los operadores "$*$", "$+$"se usan para indicar múltiples ocurrencias del mismo selector. Veamos cómo se computan los *AP*.

El algoritmo de la Fig. 5.6 se basa en un análisis de intervalos. Para ello utilizamos un grafo de flujo de intervalos, IG=(V_{IG}, E_{IG}), donde V_{IG} es el conjunto de nodos, y E_{IG} el conjunto de enlaces. Para cada nodo $node \in V_{IG}$, $Loop_Level(node)$ es el nivel de anidamiento de bucle de $node$. En un anidamiento de bucles o en una cadena de llamadas a función, el algoritmo comienza en los niveles más internos del grafo de flujo de intervalos, calculando los paths para las sentencias que pertenecen al intervalo correspondiente y guardándolos en los correspondientes cabeceras de bucle o cabeceras de función. Luego, estos paths se propagan hacia los niveles más externos, hasta el bucle más externo del anidamiento, o hasta que se alcanza la llamada más externa de la cadena de llamadas.

La meta de la función `Access_Paths_Computation` es el cálculo de los access paths

```
fun Access_Paths_computation (IFG)
    foreach (Interval ⊂ IFG)
        [INDPTR_Interval, IPP_Interval = Find_Induction_Pointers(Interval)
        foreach s ∈ Interval
            AP_s=Initialize_Access_Path(s)
        endfor
    endfor
    foreach (Interval ⊂ IFG in reverse topological order)
        foreach (s ∈ Interval)
            AP_s=⋃_{d∈succ(s)} Transform_Path(AP_d)
        endfor
    endfor
    return(AP)
end
```

Figura 5.6: Función para el cálculo de los *Access Paths*.

para cada sentencia del código, propagando y expresando estos access paths por todos los puntos del programa. El método comienza con una llamada al método presentado anteriormente para el cálculo de los punteros de inducción (Fig. 5.3) aplicándolo por intervalos (ya sea en un bucle o en una cadena de llamadas a función). Esta función recordemos devuelve no sólo los punteros de inducción detectados sino también los paths para dichos punteros. El path de un puntero de inducción, IPP_i, se representa de forma parecida a los paths, con la tupla:

$IPP_i ::=< s_i, x_j.tailexpr_i, navexpr_i >$

En esta tupla, $s_i \in STMT_{Interval}$ es la sentencia donde se define el puntero de inducción (una función *phi* en la cabecera de un bucle o de una función); x_j es otro puntero de inducción o un puntero definido fuera del intervalo, mientras que $tailexpr_i$ es un subpath que consiste en una expresión *null* o una cadena de campos; $x_j.tailexpr_i$ representa la expresión de entrada, para el intervalo correspondiente, del subpath para el puntero de inducción definido en s_i; $navexpr_i$ es un subpath que representa la expresión de navegación asociada al puntero de inducción. Por ejemplo, en el código de la Fig. 5.7 las sentencias desde `s2` a `s9` representan un intervalo, donde el puntero de inducción p_2 es definido en una función *phi* en la cabecera del bucle `s3`, siendo `s3` un puntero que se define dentro del intervalo y p_1 se define fuera del intervalo. Siguiendo la cadena DU desde p_3 (que es el puntero que produce el ciclo) encontramos la expresión de navegación $(next)^+$, luego en la tupla *IPP* $navexpr_i = (next)^+$. Además, en este caso, encontraríamos $x_j.tailexpr_i = p_1$ al salir fuera del bucle. Finalmente, el path del puntero de inducción p_2 en la cabecera del bucle es $< s_3, p_1.-, (next)^+ >$.

Puede haber más de un puntero de inducción en el intervalo de análisis. Además, en un anidamiento de bucles o en llamadas recursivas, típicamente hay un recorrido o navegación en cada nivel, así que podría haber al menos un puntero de inducción por cada nivel, donde cada nivel implicaría un intervalo. En cualquier caso, cuando identificamos un puntero de inducción y su correspondiente expresión de navegación, éstos se asocian al intervalo del nivel de anidamiento bajo estudio. Por ejemplo, en la Fig. 5.8, hay dos niveles de anidamiento y dos intervalos: uno para el

s1: $p_1 = list_2$;
s2: while (p_2)
  s3: $p_2 = phi(p_1, p_3)$;
  si: $\vdots$
  s9: $p_3 = p_2.next$;
end

Figura 5.7: Recorrido de una lista.

bucle externo (s2-s10), y otro para el bucle interno (s5-s9). Aquí se identifican dos punteros de inducción: p_2 definido en s3 con su path $< s3, p_1.-, (next)^+ >$, y t_2 definido en s6 con su path $< s_6, t_1.-, (down)^+ >$.

```
s1: p_1=list_2;
s2:   while (p_2)
s3:      p_2=phi(p_1,p_3);
s4:      t_1=p_2.g ;
s5:      while (t_2)
s6:         t_2=phi(t_1,t_3);
s7:         t_2.val=0;
s8:         t_3=t_2.down;
s9:      end
s10:     p_3=p_2.next;
s11:  end
```

Figura 5.8: Anidamiento de dos niveles

El próximo paso del algoritmo de la Fig. 5.6 consiste en la inicialización de las expresiones de los access paths, *AP*, para cada sentencia del intervalo. La tupla *AP* que representa cada sentencia se inicializa con el identificador de la sentencia, s_j, y la expresión $p_j.sel_j$ que representa la instancia de acceso al *heap* en ese punto. Por ejemplo, el access path para la sentencia s_7 de la Fig. 5.8 sería $< s_7, t_2.val >$.

En el siguiente paso, todos los paths de acceso asociados a las sentencias del programa se examinan hacia atrás desde el final del programa hacia la entrada y son transformadas por la función Transform_Path. Para ello, las expresiones de los access paths se propagan hacia las sentencias predecesoras, y son convenientemente transformadas por ellas, hasta que alcanzan el bucle o cabecera de función del correspondiente intervalo. Los access paths son expresados en términos de los punteros de inducción como hemos dicho. De hecho, cada vez que un access path alcanza una función *phi* que define un puntero de inducción, y la expresión del path depende de ese puntero, entonces una nueva expresión de entrada se añade a la expresión del path, para indicar el valor de entrada para el correspondiente puntero de inducción. Por ejemplo, el path de acceso para la sentencia s_7, depende del puntero de inducción t_2:$< s_7, t_2.val >$. Cuando se propaga hacia atrás el path de acceso fuera del bucle, se añade una nueva componente al path $< s_7, t_1 : t_2.val >$ para indicar que t_2 depende del puntero t_1, y cuando alcanza la sentencia s_4, se transforma como $< s_7, p_2.g : t_2.val >$. Este *AP* se propaga de nuevo, y alcanza la cabecera del bucle exterior, en-

contrándose la función *phi* en la sentencia s_3 que define el puntero de inducción p_2 en función del puntero externo p_1. De nuevo, esa función *phi* añade un nuevo componente a la expresión del path: $< s_7, p_1 : p_2.g : t_2.val >$. El valor de p_1 representa la entrada para el puntero de inducción p_2, mientra que $p_2.g$ representa el valor de entrada para el puntero de inducción t_2. De este modo, cada sentencia de definición de un puntero de inducción (una función *phi*), introduce un nuevo valor de entrada para el intervalo anidado, de modo que el *AP* se expresa en términos de los punteros de inducción de los que depende.

Finalmente, la función `Access_Paths_Computation` devuelve las expresiones de los access paths para todos los puntos del programa. Un aspecto a tener en cuenta es que las expresiones de los access paths asociados a una sentencia dependen del punto del programa donde el path de acceso es examinado. Una expresión de un access path (así como la representación de nuestro path) puede representar varias localizaciones del *heap*. Por ejemplo, el access path $AP_1 =< s8, list_2.(next)^+ >$ representa las localizaciones del *heap* que son alcanzables por la sentencia $s8$: $list_2.next$, $list_2.next.next$, etc. Otro ejemplo lo encontramos en el access path $AP_2 =< s5, A_1.root.left.(left|right)^* >$, que representa las localizaciones del *heap* alcanzables desde la sentencia $s5$, pudiendo ser:

$A_1.root.left$, $A_1.root.left.left$, $A_1.root.left.right$, $A_1.root.left.left.left$, $A_1.root.left.left.right$, $A_1.root.left.right.left$, $A_1.root.left.right.right$, ...

En el formato de los paths que hemos explicado, podemos encontrar más de un campo con la forma $(sel_1|sel_2|\ldots|sel_m)^{*|+}$. Por ejemplo, un anidamiento de bucles corresponde con un anidamiento de intervalos en el que puede haber una navegación o recorrido por cada nivel. Debemos pues identificar la correspondiente expresión de navegación asociada a cada nivel de anidamiento. Llamamos expresión de navegación actual, a la correspondiente para el nivel de bucle que se está analizando. De este modo, el puntero y la secuencia de campos selectores que preceden a la actual expresión de navegación es lo que denominamos la *entrada*, mientras que la secuencia de campos selectores que sigue a la navegación actual sería lo que antes hemos definido como $tailexpr$.

Sean $AP_i =< s_i, entry_i.navigation_i.tailexpr_i >$ y $AP_j =< s_j, entry_j.navigation_j.tailexpr_i >$ dos access paths para los que queremos construir los paths $Path_i =< s_i, ce : (nce_i : pre_i) : nav_i : tail_i >$ y $Path_j =< s_i, ce : (nce_j : pre_j) : nav_j : tail_j >$. Vamos a utilizar esta nomenclatura, primero para ver el caso en el que no existe preámbulo y luego el caso para el que sí existiría preámbulo. Asignamos $nav_i = navigation_i$ y $nav_j = navigation_j$, también $tail_i = tailexpr_i$ y $tail_j = tailexpr_j$. Asumimos que $entry_i = step_1.step_2.\ldots.step_q$ y $entry_j = step'_1.step'_2.\ldots.step'_p$.

Primer caso: cuando encontramos un $step_k$ en $entry_i$ y un $step'_k$ en $entry_j$, tal que $step_i == step'_i$, para $i = 1 : k$, pero $step_{k+1} \neq step'_{k+1}$, entonces tenemos un punto de entrada común ce en ambos paths. Si $k > 1$, caso de punto de entrada común, debemos identificar si existe un preámbulo o algún subpath de entrada no común. Cuando $k = 1$, no hay punto de entrada común, lo que equivale a expresar $ce = NULL$. En este caso, tendríamos punto de entrada no común, los subpaths para nce serían las expresiones $nce_i = entry_i$ y $nce_j = entry_j$, y no existiría preámbulo ($pre_i = pre_j = NULL$) puesto que no pueden darse ambos casos nce y pre a la vez. De este modo, el subpath de entrada no común identifica qué parte de la expresión de entrada puede llevar a un punto de entrada común previo a la navegación en curso, indicada por la expresión de navegación.

Segundo caso: Si los campos en ambos paths coinciden hasta el campo $step_t$ en $entry_i =$

$step_1.\ldots.step_t.step_{t+1}\ldots.step_q$ y $entry_j = step'_1.\ldots.step'_t.step'_{t+1}\ldots.step'_p$, donde $step_i == step'_i$ para $i = 1 : t$, siendo $step_t = (sel_1|sel_2|\ldots|sel_m)^{*|+}$. Para $j = t+1 : q$, $step_j! = (sel_1|sel_2|\ldots|sel_m)^{*|+}$, y para $j = t+1 : p$, $step'_j! = (sel_1|sel_2|\ldots|sel_m)^{*|+}$, no son coincidentes. Entonces el subpath del punto de entrada común para ambas expresiones sería $ce = step_1.\ldots.step_t$, mientras que los subpahts del preámbulo serían $pre_i = step_{t+1}\ldots.step_q$ y $pre_j = step'_{t+1}\ldots.step'_p$. El subpath del preámbulo recordamos que nos indica qué parte de la expresión de entrada, para el caso de punto de entrada común, precede a la navegación y además esta expresión no está contenida en dicha expresión de navegación. Por ejemplo, para el código de la Fig. 5.26(a), que usaremos más adelante en un caso práctico del funcionamiento del análisis, si tomamos los access paths de las sentencias `SR` y `SW` para el bucle `L2`, existe un desfase previo debido a que el puntero `c2` (en `SR`) avanza por `nxt_t1` respecto al puntero `c1`, en `SW`. El path de la sentencia `SR` sería: $< A \rightarrow nxt_t1^* : nxt_t1 \rightarrow elem_list : nxt_t2^* : - >$; y para `SW`: $< A \rightarrow nxt_t1^* : elem_list : nxt_t2^* : - >$. El subpath del preámbulo para `SR` sería $elem_list$, y para `SW` $nxt_t1 \rightarrow elem_list$.

5.4. Grafos de forma

En el capítulo 4 explicamos el funcionamiento del análisis de forma y vimos en detalle cómo se interpretaban los correspondientes grafos de forma. Un grafo de forma es un conjunto de nodos, enlaces de punteros, enlaces de selectores y cls, `coexistent links sets`, que contienen información de la estructura de datos del programa. Recordamos que hablábamos de la relación entre el dominio concreto y el dominio abstracto. En un grafo de forma, cada nodo puede representar un conjunto de localizaciones de memoria del dominio concreto, mientras que cada enlace puede representar una variable puntero, o un conjunto de selectores, con el mismo nombre. Además, un enlace selector del dominio abstracto puede representar varias instancias de enlaces selectores del dominio concreto. Los llamados *coexistent links sets* comentábamos que permiten mantener la información de conectividad y *aliasing* que puede existir en el nodo abstracto, incluso cuando el nodo representa diferentes localizaciones de memoria con diferentes patrones de conexión.

Presentamos como ejemplo el grafo de la Fig. 5.9 que corresponde al código de la Fig. 5.4(a). Tenemos dos nodos, `N1` y `N2` y diez selectores que nos muestran las distintas localizaciones que podemos alcanzar partiendo del puntero inicial `A` y recorriendo los distintos enlaces selectores `SL`. Por ejemplo, partiendo del elemento `N1` y navegando por el enlace *elem_list*, podemos llegar bien a `N2` a través de `SL3`, o bien a `null` a través de `SL4`. También se muestra en una tabla los diferentes `cls` extraídos.

Este conocimiento de la forma de la estructura de datos accesible desde los punteros al *heap*, proporciona información crítica para diferenciar los accesos al *heap* y determinar si hay o no dependencias entre iteraciones de un bucle o distintas llamadas a función. Por lo tanto el test de dependencias se basa en la información de forma proporcionada por los grafos de forma. En el capítulo anterior presentamos una técnica que precisamente optimizaba los resultados del análisis de forma, gracias al proceso de *slicing* de los códigos de entrada. Gracias a la reducción del número de grafos de forma generados durante el análisis, se conseguía una reducción directa de los tiempos de ejecución, además de poder analizar más programas que antes eran inabordables.

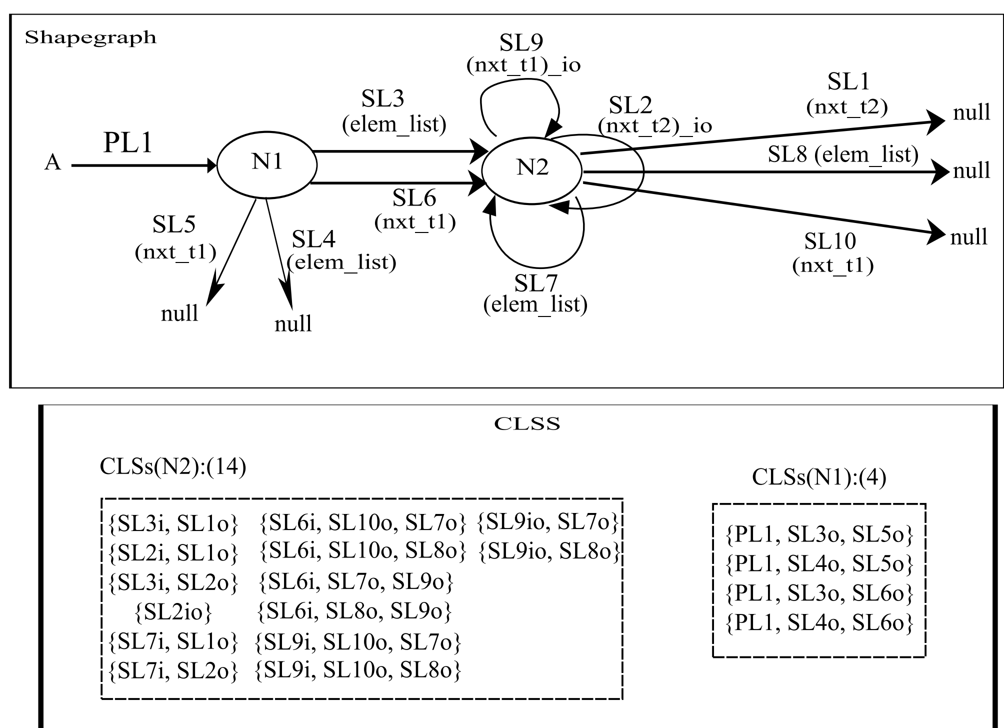


Figura 5.9: Grafo de forma del código presentado en la Fig.5.4(a).

5.5. Test de dependencias

La meta del algoritmo es la detección de dependencias acarreadas por lazo, *LCD*, en bucles o en funciones con llamadas recursivas, que recorren estructuras dinámicas de datos basadas en el *heap*. Dos sentencias en un bucle inducen una LCD si una localización de memoria accedida por una sentencia en una iteración dada, es accedida por otra sentencia en una iteración futura, con uno de los accesos siendo un acceso de escritura. En el caso de funciones con llamadas recursivas, se dice que dos *call sites* recursivos en el cuerpo de una función inducen un *LCD*, si una localización de memoria accedida por uno de los *call sites* es accedida por el otro *call site* con uno de los accesos siendo de escritura.

5.5.1. Algoritmo básico

En el algoritmo que vamos a describir, se asume que no hay modificación de la estructura de datos en el bucle o en la función durante el análisis. Este caso es el más común como veremos en el apartado de resultados experimentales. El caso más complejo, donde sí hay modificación de la estructura, se aborda y se explica en la sección 5.5.2.

Nuestro método, ver Fig. 5.10, trata de identificar si hay un LCD en un bucle o en una función. Almacena toda la información sobre las dependencias detectadas para los parámetros de entrada en la estructura $LCD_{Conflict}$ y la devuelve a la salida. Los parámetros de entrada son dos: $SG_{\bullet header}$ y $AP_{\bullet header}$. Definimos $SG_{\bullet header} \in SG$ como el conjunto de los grafos de forma que representa el estado del *heap* a la entrada de la cabecera del bucle (o la entrada de un *call site* de una función recursiva), y definimos $AP_{\bullet header} \in AP$ como el conjunto de los access paths relativos a sentencias

```
fun LCDs_Detection (SG•header, AP•header)
    1. CONFGROUP = Create_Conflict_Groups(AP•header);LCD_Conflict = 0;
    2. foreach ConfGroup_g ∈ CONFGROUP
          foreach AP_i =< s_i, entry_i.navigation_i.tail_i >∈ ConfGroup_g|s_i is a write stm.
             foreach AP_j =< s_j, entry_j.navigation_j.tail_j >∈ ConfGroup_g
                [Path_i, Path_j, ComNav]= Compute_Paths (AP_i, AP_j)
                foreach sg_n ∈ SG•header
                   LCD_Conflict_g(i, j)=Check_Conflict_ce(sg_n, Path_i, Path_j, ComNav)
                endfor
             endfor
          endfor
          LCD_Conflict = LCD_Conflict ∪ LCD_Conflict_g;
       endfor
    return LCD_Conflict
end
```

Figura 5.10: Algoritmo para la detección de dependencias.

del bucle o función, en concreto los access paths que llegan a la cabecera del bucle (o cuerpo de la función recursiva).

El algoritmo puede dividirse en dos etapas:

Etapa 1: Construcción de los grupos de conflictos

Primeramente, se crean los llamados *Conflict Groups*, y se inicializa el conjunto $LCD_{Conflict}$. Un grupo de conflicto, $ConfGroup_g$, es un conjunto de *access paths*, AP, que cumple dos condiciones:

- Todos los *access paths* que pertenecen a un *Conflict Group* representan sentencias que acceden al mismo tipo de datos. De modo que cada $AP_i \in ConfGroup_g$ representado tiene la forma $p \rightarrow g$, donde `p` es una variable puntero y `g` es un campo de datos.
- Al menos una de las sentencias de $AP_i \in ConfGroup_g$ es una sentencia de escritura, de la forma $p \rightarrow g = \ldots$.

En otras palabras, un $ConfGroup_g$ está relacionado con el conjunto de sentencias en un bucle (o función recursiva), que puede potencialmente llevar a un LCD, lo que sucede si: i) la sentencia hace un acceso de escritura, o ii) hay otras sentencias accediendo al mismo campo (`g`) y uno de los accesos es de escritura.

El método `Create_Conflict_Groups`, Fig. 5.11, crea los grupos de conflicto, usando como entrada el conjunto de expresiones de los AP disponibles a la entrada de la cabecera del bucle (o la cabecera de la función recursiva), $AP_{\bullet header}$. Es posible crear un grupo de conflictos con sólo un AP_i, donde s_i es una sentencia de escritura. Este grupo de conflictos nos ayuda a comprobar las dependencias de salida para la ejecución de s_i en diferentes iteraciones del bucle. El conjunto de grupos de conflicto que devuelve esta función es lo que nombramos como `CONFGROUP` en el algoritmo de la Fig. 5.10.

```
fun Create_Conflict_Groups(AP•header)
 CONFGROUP = ∅;
 foreach AP_i =< s_i, entry_i.navigation_i.tail_i >∈ AP•header
     if [(tail_i == sel_1.sel_2.....g, where g is a data field)
         if (∄DepGroup_g ∈ CONFGROUP and s_i is a write stmt.) then
             ConfGroup_g = {AP_i}; CONFGROUP = CONFGROUP ∪ {DepGroup_g};
         else
             ConfGroup_g = ConfGroup_g ∪ {AP_i};
         endif
     endif
 endfor
return CONFGROUP;
```

Figura 5.11: Función `Create_Conflict_Groups`.

Etapa 2: Creación de paths y detección de conflictos

Esta segunda etapa está indicada en el margen de la Fig. 5.10 con el número dos. Después de crear los grupos de conflictos, se buscan conflictos por parejas de access paths en cada grupo de conflicto, $AP_i, AP_j \in ConfGroup_g$, siendo la sentencia asociada AP_i una sentencia de escritura. Primero se calculan los paths $Path_i$ y $Path_j$, así como la variable booleana $ComNav$. Esta variable identifica si dos paths tienen *navegación común*. La *navegación común* indica si los dos accesos estudiados utilizan el mismo navegador (puntero de inducción). Si se cumple la *navegación común*, entonces sabemos que ambos accesos, en cada iteración del bucle o en cada llamada a función, parten del mismo elemento (sitio). Por el contrario, si no se cumple la `navegación común`, cada acceso puede partir de un elemento distinto. En caso afirmativo $ComNav = TRUE$, sino $ComNav = FALSE$.

```
fun Check_Conflict_ce (sg_n, Path_i, Path_j, ComNav)
    If (ce == null)
        ComEst = False;
        vs_k = null; VS = vs_k;
    else        ComEst = True;
        VS = Compute_Visited_Sites(sg_n, vs_⊥ , ce);
    endif
    foreach vs_k ∈ VS
        if (Check_Conflict_nce-pre(sg_n, vs_k, Path_i, vs_k, Path_j, ComNav, ComEst))
            return True;
        endif
    endfor
    return False;
    end
```

Figura 5.12: Función para la detección de conflictos de primer nivel.

Veamos cómo se computa $ComNav$ a partir de dos paths. Sean $Path_1 =< s_1, ce : (nce_1 : pre_1) : nav_1 : tail_1 >$ y $Path_2 =< s_2, ce : (nce_2 : pre_2) : nav_2 : tail_2 >$ dos paths genéricos. Decimos que estos dos paths tienen *navegación común*, $ComNav = True$, cuando se da alguna de las tres condiciones siguientes:

1. $nav_1 == nav_2$;
2. Los subpaths de navegación, nav_1 y nav_2 se calculan desde el mismo puntero de inducción;
3. Siendo $(sel_1|sel_2|\ldots|sel_m)^{*|+}$ el subpath de navegación y $tail_1 = T.f_1.f_2.\ldots$, $tail_2 = T'.f'_1.f'2.\ldots$ los correspondientes subpaths de cola. Entonces, los primeros campos selectores en las expresiones de cola no deben ser selectores que están en la expresión de navegación $T \neq sel_i, \forall i = 1 : m$ y $T' \neq sel_i, \forall i = 1 : m$.

A continuación el algoritmo busca un conflicto entre dos paths por cada grafo sg_n, que pertenece al conjunto de grafos disponible a la entrada de la cabecera del bucle (o la entrada de la cabecera de la función recursiva), $SG_{\bullet header}$. La función encargada de encontrar los conflictos se llama `Check_Conflict_ce` y es el método más importante de la técnica. En la tabla 5.1 se definen cuatro campos, por tanto vamos a hablar de cuatro niveles, cuatro versiones del método en función del nivel de subpath que se quiera chequear.

En la Fig. 5.12, se muestra el esquema de la función para la detección de conflictos de primer nivel. Primero se realiza un proceso donde se visitan todas las localizaciones alcanzables por cada subpath y se calcula, para cada una de estas localizaciones, si los siguientes subpaths provocan conflicto o no. Este proceso se lleva a cabo para cada grafo de entrada, sg_n. En este primer nivel, inicialmente se comprueba el subpath ce. Si hay una entrada común distinta de $null$, entonces gracias a la función `Compute_Visited_Sites` se calcula el conjunto de localizaciones visitadas por el subpath correspondiente (ce en este nivel).

Para entender mejor cómo se calculan los conjuntos de sitios visitados, vamos a retomar el código de la Fig. 5.4(a) para mostrar el funcionamiento. Recordamos que la expresión del access path era la misma para ambas sentencias de lectura y escritura: $< A : (nxt_t1)^* : - >$

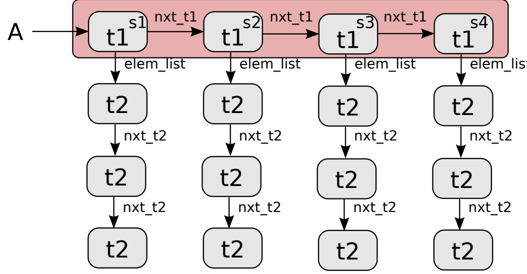


Figura 5.13: Sitios visitados por la navegación del ejemplo.

Reproducimos ahora de nuevo para mayor comodidad el grafo de forma de la estructura dinámica en la Fig. 5.14. A partir de ahora vamos a hablar de *sitios* para referirnos al dominio abstracto y de *elementos* para referirnos al dominio concreto. En la Fig. 5.13, podemos ver los elementos de la estructura de datos visitados en la navegación del path anterior (`s1, s2, s3, s4`). La proyección del subpath de entrada viene reflejado por el sitio apuntado por la variable puntero `A`, y la

proyección del subpath de navegación por el resto (s2, s3, s4). Así pues la zona que aparece sombreada sería la proyección del path sobre la estructura concreta.

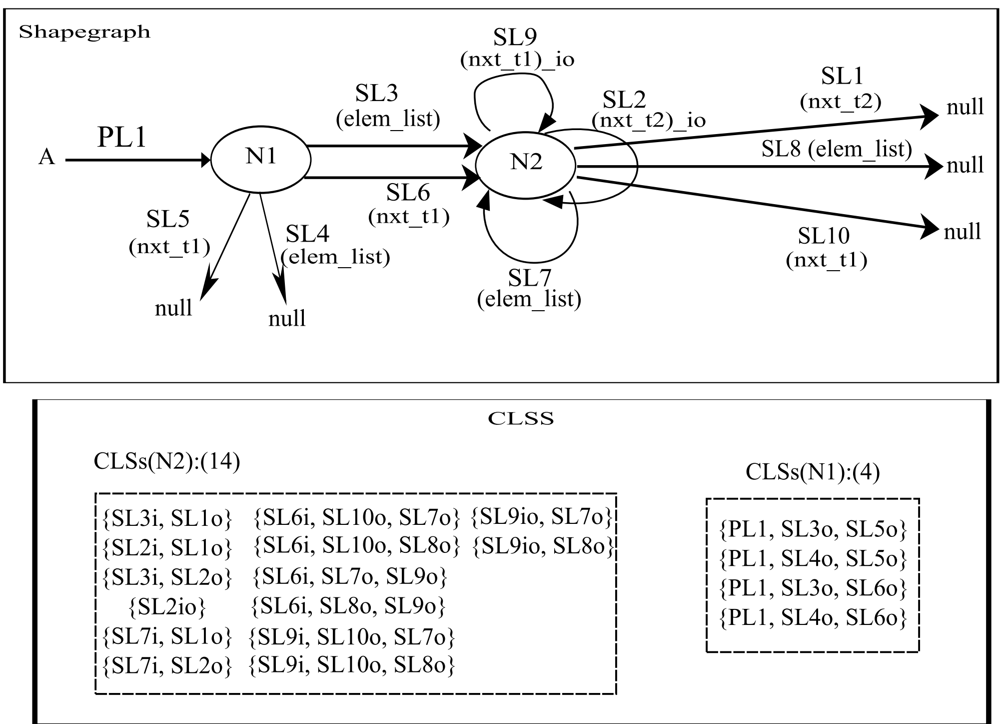


Figura 5.14: Grafo de forma del código empleado para el ejemplo.

La *proyección* de un path sobre una estructura concreta es bastante simple, puesto que todos los elementos aparecen separados, y dicha proyección equivale a hacer el mismo recorrido que hace el código analizado sobre al estructura real, siguiendo el enlace nxt_t1 desde el sitio apuntado por A. Sin embargo, la proyección de un path sobre un grafo que representa la estructura de datos que hay en un bucle es bastante más complicado. La razón fundamental es que los sitios visitados, en el dominio abstracto, ahora no se mantienen separados como en el caso anterior, sino que muchos de ellos son representados en el mismo nodo del grafo. Para representar un sitio en un grafo no basta con especificar el nodo que lo representa, pues sólo con esa información no podríamos distinguir entre todos los sitios representados por dicho nodo. Por eso definimos un *sitio abstracto* como la combinación de un nodo y un selector $< sel, N >$. Dicho selector sel es el enlace utilizado para alcanzar el sitio representado en el nodo N. El conjunto de sitios visitados por el access path $< A : (nxt_t1)^* : - >$ en el grafo de la figura 5.14 sería el siguiente :

- El sitio del que parte la navegación (el alcanzado por el *punto de entrada*) sería el apuntado por el puntero A. En el grafo, este enlace viene representado por $PL1$. Por tanto el sitio de comienzo de la navegación sería $vs1 =< PL1, N1 >$.

- A partir del sitio anterior $vs1$ se despliega la navegación. Los sitios visitados por la navegación serían pues los alcanzados a partir de $vs1$ siguiendo los enlaces por el selector nxt_t1. Los sitios visitados serían:

  - Como la expresión de navegación en el path es $(nxt_t1)^*$ que implica seguir 0 o más

veces el enlace por nxt_t1, el primer sitio visitado sería el propio $vs1$, siguiendo 0 veces el enlace nxt_t1.

- El siguiente sitio se obtendría a partir de $vs1$ siguiendo el enlace nxt_t1. Sabemos que $vs1$ representa a los sitios en `N1` a los cuales se ha accedido por $PL1$. Para ver a que otros sitios se llega desde éste por nxt_t1, primero localizamos los selectores de salida de `N1` por el campo puntero nxt_t1: $SL5, SL6$. Ahora basta ver cuáles de estos selectores de salida (o) pueden coexistir con el de entrada ($PL1$). Esta información precisamente es la que aparece en los CLS del grafo. Por tanto, hay que buscar CLS que contengan a la vez a $PL1$ y a $SL5o$ y/o $SL6o$. Observando el grafo, vemos que existen los CLS $\{PL1, SL3o, SL5o\}$, $\{PL1, SL4o, SL5o\}$, $\{PL1, SL3o, SL6o\}$ y $\{PL1, SL4o, SL6o\}$. Esto implica que los sitios apuntados por $SL5$ y $SL6$ representan sitios visitados a partir de $vs1$. Puesto que $SL5$ es un puntero a $null$, siguiendo ese selector no se visita ningún sitio nuevo. El sitio visitado desde $vs1$ por el selector $SL6$ sería: $vs2 =< SL6, N2 >$.
- Para calcular los siguientes sitios visitados repetimos el proceso del paso anterior. Tomamos de partida $vs2$, por tanto buscamos en el nodo `N2` selectores por el campo nxt_t1 ($SL9, SL10$), y que además aparezcan de salida (o) en algún CLS del nodo junto con $SL6$ de entrada (i) (que es el selector de entrada al sitio $vs2$). Como existen dichos CLS ($\{SL6i, SL10o, SL7o\}$, $\{SL6i, SL7o, SL9o\}$, ...), los siguientes sitios visitados serían los nodos apuntados por dichos selectores junto a los mismos. De nuevo, como $SL10$ apunta a $null$ no genera sitio. El único pues sería: $vs3 =< SL9, N2 >$.
- Repitiendo el proceso ahora desde $vs3$ de nuevo nos encontramos en el nodo `N2` y los selectores por nxt_t1, $SL9$ y $SL10$, (este último lo descartamos ya puesto que va a *null*). Buscamos CLS con $SL9i$ y $SL9o$. Existen varios que contienen $SL9io$ (lo que significa entrada y salida por $SL9$). Por tanto, el siguiente sitio visitado sería $< SL9, N2 >$ que es el sitio $vs3$, por tanto lo notaremos con $vs3^+$, puesto que se repite una o más veces. En este momento, ya podemos parar puesto que no vamos a encontrar nuevos sitios visitados.

En resumen, los sitios visitados en la navegación son: $vs1 =< PL1, N1 >$, $vs2 =< SL6, N2 >$ y $vs3^+ =< SL9, N2 >^+$ (estos sitios aparecen representados en la figura 5.15). La equivalencia entre estos sitios y los sitios concretos (5.13) es la siguiente: $vs1$ representa el sitio $s1$, $vs2$ representa a $s2$, y $vs3$ a la vez $s3$ y $s4$.

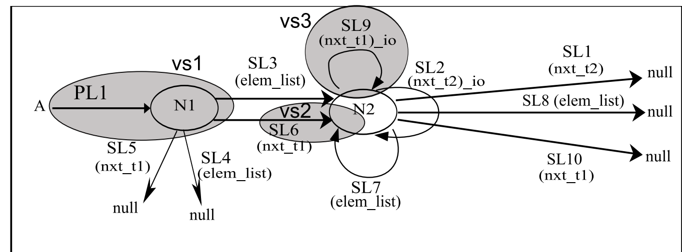


Figura 5.15: Representación de los sitios abstractos visitados en el grafo de forma.

Volviendo al algoritmo de la Fig. 5.12, el conjunto de sitios visitados que devuelve la función `Compute_Visited_Sites`, VS, representa las localizaciones de memoria abstraídas en el grafo sg_n, que son alcanzables desde la entrada común de los paths. Estas localizaciones son representadas como hemos dicho, como sitios abstractos, con el formato $< PL_i|SL_j, N_k >^+$. Cuando aparece el operador "$+$", significa que la tupla puede aparecer más de una vez, que el *sitio abstracto* (o la localización de memoria que representa) puede ser visitada por el subpath de entrada más de una vez. En caso de que no haya entrada común, entonces $Path_i.ce$ sería $null$ y el subpath ce no puede alcanzar o visitar los sitios del grafo. Esta situación se refleja con el parámetro de entrada $vs_k = null$.

Una vez que se han calculado los conjuntos de sitios visitados por el subpath ce, la función `Check_Conflict_nce-pre`, que mostramos en la Fig. 5.16, se encarga de buscar conflictos entre $Path_i$ y $Path_j$, para el siguiente subpath nce o pre. Estos subpaths representan la parte de la expresión del path entre la entrada común y la navegación actual, además son mutuamente exclusivos en un path. El subpath nce incluye una expresión de navegación, mientras que el subpath pre no. Un aspecto a destacar es que la función `Check_Conflict_nce-pre` tiene como entrada al parámetro vs_k, que indica el punto de inicio para comprobar conflictos en un sitio visitado por la entrada común ce. Hay dos parámetros más de entrada para esta función: i) $ComNav$, lo que indica que dos paths representan una navegación común, un parámetro que es establecido por la función `Compute_Paths`; y ii) $ComEst$ un parámetro que representa si dos subpaths del análisis alcanzan una estructura común o no. En este primer nivel, se visitan los sitios alcanzables por el subpath ce. Cuando no hay entrada común, entonces el subpath ce es $null$, y naturalmente este subpath no puede alcanzar una estructura común, asi que $ComEst = False$. Por el contrario, si ce no es $null$, entonces ambos subpaths alcanzan una estructura común, $ComEst = True$.

La función `Check_Conflict_nce-pre` de la Fig. 5.16 se utiliza para la búsqueda de conflictos en el segundo nivel de los paths, es decir, segundo nivel de campos accedidos en los paths como explicamos anteriormente. Se comprueba inicialmente si existe un subpath de preámbulo en alguno de los dos paths. En este caso, si pre_i o pre_j es distinto de $null$, entonces se usa la función `Check_ComEst` para averiguar si el subpath del preámbulo en ambos paths alcanza una estructura común, comenzando el recorrido desde el sitio visitado vs_k (recordamos que vs_k representa el sitio visitado por el subpath ce). En tal caso, $ComEst'$ estará a $True$. Luego, la función `Compute_Visited_Sites` calcula los conjuntos de sitios visitados por el subpath correspondiente (pre_i y pre_j a este nivel). A continuación, por cada par de sitios visitados (por el preámbulo), se comprueba con la función `Check_Conflict_nav`, Fig. 5.17, si los siguientes subpaths para $Path_i$ y $Path_j$ pueden provocar conflictos. Un proceso similar se lleva a cabo si existe un subpath de entrada no común en una de las expresiones de los paths, en tal caso la función `Check_ComEst` busca si el subpath de entrada no común, en ambos paths, puede alcanzar la misma estructura, y gracias a la función `Compute_Visited_Sites`, calcularía el conjunto de sitios visitados por el subpath nce correspondiente. Del mismo modo que antes, la función `Check_Conflict_nav` buscaría conflictos en el subpath nav.

La función `Check_Conflict_nav`, Fig. 5.17, comprueba los conflictos por parejas de sitios visitados, siguiendo el mismo esquema de la función de detección de conflictos del nivel anterior. En este caso, se añade una comprobación de navegación cíclica, reseteando o no la variable $ComEst'$ según sea el caso. El método `Check_Conflict_tail` de la Fig. 5.17, simplemente hace una llamada a la función `Check_ComEst`.

```
fun Check_Conflict_nce-pre (sg_n, vs_k, Path_i, vs_k, Path_j, ComNav, ComEst)
If (nce_i == nce_j == null)
     If (pre_i != null | pre_j != null)
          ComEst' = Check_ComEst(sg_n, vs_k, pre_i, vs_k, pre_j ,ComEst, Level = pre);
          VS1 = Compute_Visited_Sites(sg_n, vs_k, pre_i);
          VS2 = Compute_Visited_Sites(sg_n, vs_k, pre_j);
          foreach vs_p ∈ VS1
                 foreach vs_q ∈ VS2
                     if (Check_Conflict_nav(sg_n, vs_p, Path_i, vs_q, Path_j, ComNav, ComEst'))
                         return True;
                     endif
                 endfor
          endfor
          return False;
     else
          ComEst' = True;
          return Check_Conflict_nav(sg_n, vs_k, Path_i, vs_k, Path_j, ComNav, ComEst');
     endif
else
     ComEst' = Check_ComEst(sg_n, vs_k, nce_i, vs_k, nce_j ,ComEst, Level = nce);
     VS1 = Compute_Visited_Sites(sg_n, vs_k, nce_i);
     VS2 = Compute_Visited_Sites(sg_n, vs_k, nce_j);
     foreach vs_p ∈ VS1
          foreach vs_q ∈ VS2
                 if (Check_Conflict_nav(sg_n, vs_p, Path_i, vs_q, Path_j, ComNav, ComEst'))
                     return True;
                 endif
          endfor
     endfor
     return False;
endif
end
```

Figura 5.16: Función para la detección de conflictos de segundo nivel.

Básicamente, la función `Check_ComEst`, Fig. 5.19, desenrolla sobre el grafo de forma sg_n, los subpaths $subpath_i$ y $subpath_j$, comenzando con las localizaciones de entrada vs_k y vs_l respectivamente. A continuación se comparan por parejas si los caminos devuelven conflicto o no, llamando a la función `Check_Conflict_Paths`. $ComEst$ es una variable booleana que se utiliza para indicar si se ha alcanzado una estructura común. $PreLevel$ es una variable booleana que nos indica si algún path tiene preámbulo. En caso de existir conflicto, la función devuelve $True$ indicando que las localizaciones pueden alcanzar un punto común en la estructura.

La función `Check_Conflict_Paths`, Fig. 5.19, comprueba si dos *Unrolled Paths*, up_i y up_j, alcanzan una localización común en el grafo sg_n. Un *Unrolled Path*, `UP`, no es más que la secuencia de sitios abstractos que visita el path hasta el último acceso: `{vs1, vs2, ..., vsN}`. Obtendremos los `UP` de cada uno de los sitios visitados por la navegación, desplegados a partir del sitio de entrada común (*vspec*): $UP_{SR} = \{vsr1, vsr2, ..., vsrN\}$ y $UP_{SW} = \{vsw1, vsw2, ..., vswN\}$. Para chequear si los sitios abstractos $vsrN =< SELr, Nr >$ y $vswN =< SELw, Nw >$ (final del acceso) pueden representar el mismo sitio concreto, aplicamos las *Reglas de Chequeo de Conflictos (RCC)* presentadas en el cuadro 5.2. El proceso comienza con una llamada al método `Compute_Last_Site` que obtiene el último sitio visitado por ambos paths. A continuación, se emplean las llamadas *Reglas de Chequeo de Conflictos, RCC,* a través de los métodos `Check_Conflict_Site` (Fig. 5.21) y `Check_Conflict_Site_pre` (Fig. 5.22). Estos

```
fun Check_Conflict_nav (sg_n, vs_k, Path_i, vs_l, Path_j, ComNav, ComEst)
If (ComNav)
    If (! Check_Cyclic_nav(sg_n, vs_k, nav_i))
        ComEst'= False;
        VS = Compute_Visited_Sites(sg_n, vs_k, nav_i);
        foreach vs_p ∈ VS
            foreach vs_q ∈ VS s.t. p < q or p = q if vs_p is a site with operator +
                if (Check_Conflict_tail(sg_n, vs_p, Path_i, vs_q, Path_j, ComEst'))
                    return True;
                endif
            endfor
        endfor
        return False;
    else
        ComEst' = True;
        VS = Compute_Visited_Sites(sg_n, vs_k, nav_i);
        foreach vs_p ∈ VS
            foreach vs_q ∈ VS
                if (Check_Conflict_tail(sg_n, vs_p, Path_i, vs_q, Path_j, ComEst'))
                    return True;
                endif
            endfor
        endfor
        return False;
    endif
else
    ComEst' = Check_ComEst(sg_n, vs_k, nav_i, vs_l, nav_j ,ComEst, Level = nav);
    VS1 = Compute_Visited_Sites(sg_n, vs_k, nav_i);
    VS2 = Compute_Visited_Sites(sg_n, vs_l, nav_j);
    foreach vs_p ∈ VS1
        foreach vs_q ∈ VS2
            if (Check_Conflict_tail(sg_n, vs_p, Path_i, vs_q, Path_j, ComEst'))
                return True;
            endif
        endfor
    endfor
    return False;
endif
end
```

Figura 5.17: Función de detección de conflictos de tercer nivel.

```
fun Check_Conflict_tail (sg_n, vs_k, Path_i, vs_l, Path_j, ComEst)
    return Check_ComEst(sg_n, vs_k, tail_i, vs_l, tail_j ,ComEst, Level);
end
```

Figura 5.18: Función para la detección de conflictos de cuarto nivel.

métodos aplican las reglas a cada subpath correspondiente.

Las *RCC* son las reglas que se aplican a cada par de sitios visitados que se quieren analizar. Detectan posibles conflictos si ambos sitios llegan a una localización, parte de la estructura, común. Estas reglas se explican en la tabla 5.2. Para el caso concreto de comparación de dos *sites* pertenecientes al campo preámbulo, entonces se aplica una regla específica, de ahí, en el algoritmo de la Fig. 5.20, la llamada diferente a `Check_Conflict_Site_pre`.

Por lo tanto, ambas funciones devuelven $True$, si up_i (siendo vs_1 el último sitio visitado) alcanza la misma localización que up_j (siendo vs_2 el último sitio visitado).

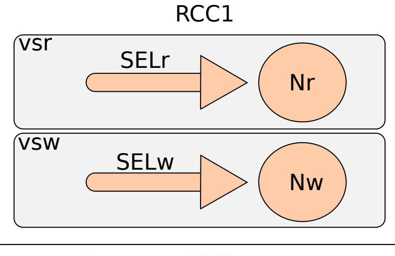
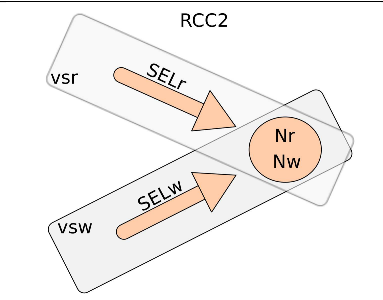
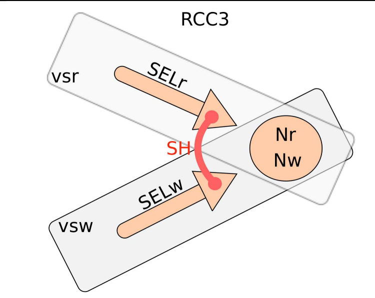
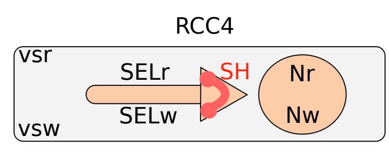
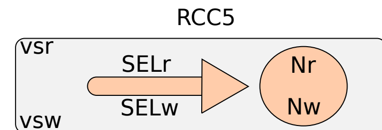
	Si el nodo de ambos sitios no es el mismo ($Nr \neq Nw$), el sitio concreto alcanzado al final de cada acceso tiene que ser distinto, por tanto no hay conflicto.
	Si el nodo es el mismo ($Nr = Nw$), el selector es distinto ($SELr \neq SELw$) y no existe en el nodo ningún CLS que contenga a ambos selectores a la vez con atributo de entrada (`SELr_i, SELw_i`), tampoco puede ser el mismo sitio concreto (no hay ningún sitio concreto representado en el nodo que pueda ser alcanzado por `SELr` y `SELw` a la vez). De nuevo, no hay conflicto pues el final de los accesos va a un sitio distinto.
	Si el nodo es el mismo ($Nr = Nw$), el selector es distinto ($SELr \neq SELw$) y existe en el nodo al menos un CLS que contiene a ambos selectores a la vez con atributo de entrada (`SELr_i, SELw_i`), entonces si que existe un sitio concreto representado en el nodo que puede ser alcanzado por ambos selectores, por lo que si que se puede producir un posible conflicto (ambos accesos puede acabar en el mismo sitio concreto).
	Si el nodo es el mismo ($Nr = Nw$), el selector es el mismo ($SELr = SELw$) y existe en el nodo al menos un CLS que contiene a dicho selector con atributo `shared` (`SELr_s`), de nuevo implica que existe un sitio concreto representado en el nodo que puede ser alcanzado por dos enlaces concretos representados por el selector, lo que da como resultado un posible conflicto.
	Si el nodo es el mismo ($Nr = Nw$), el selector es el mismo ($SELr = SELw$) y no existe en el nodo ningún CLS con selector con atributo `shared` (`SELr_s`) entonces no podemos determinar si hay conflicto o no. Para resolverlo, tenemos que ver cómo hemos llegado a este sitio (el sitio anterior en el `UP`). Si podemos determinar que el sitio anterior es distinto en ambos paths (no estructura común), vendríamos de sitios distintos y, por tanto, llegaríamos a sitios distintos. Si, por el contrario, se determina que el sitio anterior puede ser el mismo en ambos (estructura común), al seguir el mismo enlace por ambos paths ($SELr = SELw$), llegaríamos al mismo sitio y, por tanto, habría conflicto. Para saberlo, basta con aplicar de nuevo estas mismas reglas a los sitios anteriores de ambos `UP`s. Si ambos `UP` estuvieran formados exactamente por los mismos sitios abstractos y ninguno de los selectores utilizados fuera `shared`, aplicaríamos está última regla hasta que se terminaran los sitios en alguno de los paths. Entonces determinaríamos si hay o no conflicto dependiendo del punto de entrada desde el que se han desplegado los `UP`s. Si vienen desde el nivel del subpath previo con $ComEst = True$ habría conflicto, no habiéndolo en caso contrario.

Tabla 5.2: Reglas de Chequeo de Conflictos (RCC).

```
fun Check_ComEst (sg_n, vs_k, Subpath_i, vs_l, Subpath_j, ComEst, PreLevel)
UP1 = Unroll_Path (sg_n, vs_k, Subpath_i);
UP2 = Unroll_Path (sg_n, vs_l, Subpath_j);
foreach up_i ∈ CP1
     foreach up_j ∈ CP2
          if (Check_Conflict_Paths(sg_n, up_i, up_j, ComEst, PreLevel))
               return True;
          endif
     endfor
endfor
return False;
end
```

Figura 5.19: Función Check_ComEst

```
fun Check_Conflict_Paths (sg_n, up_i, up_j, ComEst, PreLevel)
     vs_1 = Compute_Last_Site (sg_n, up_i)
     vs_2 = Compute_Last_Site (sg_n, up_j)
     If (!PreLevel)
          return Check_Conflict_Site (sg_n, vs_1, up_i, vs_2, up_j, ComEst, True, True) ;
     else
          return Check_Conflict_Site_pre (sg_n, vs_1, up_i, vs_2, up_j) ;
     endif
end
```

Figura 5.20: Función Check_Conflict_Paths.

En conclusión, en esta etapa se detectan los posibles conflictos gracias a los distintos métodos presentados que realizan la detección de conflictos por niveles. Realmente se trata de una serie de funciones con llamadas en cascada, que van recorriendo los sitios visitados en cada campo de los paths correspondientes, en busca de conflictos.

5.5.2. Algoritmo extendido

El algoritmo presentado corresponde al caso más común en el que el código no presenta modificación de la estructura de datos durante el recorrido. Ahora abordamos el caso más complejo donde sí se produce esta modificación, es decir, programas donde al mismo tiempo que se recorre la estructura de datos, se hagan modificaciones en la misma. En tal caso, el método a seguir sería primero localizar dentro del bucle o la función recursiva aquellas sentencias que modifican la estructura de datos. Luego, extraeríamos el grafo de forma correspondiente para cada una de estas sentencias. Por último habría que ejecutar el algoritmo básico, Fig. 5.10, para cada uno de estos grafos y el conjunto de access paths de las sentencias que pueden provocar conflicto. Se tomarían las expresiones de los access paths que son visibles en los puntos donde se modifica la estructura, pues el grafo de forma va a ser distinto en cada una de las sentencias que modifican la estructura. En conclusión, el proceso a seguir sería el mismo que el que acabamos de describir en

```
fun Check_Conflict_Site (sg_n, vs_1, up_i, vs_2, up_j, ComEst, Flag_1, Flag_2)
If (vs_1 == null | vs_2 == null)
     return ComEst;
else
     If ((vs_1 is a site with operator +) and (Flag_1))
          If ((vs_2 is a site with operator +) and (Flag_2))
                If (Check_Conflict_Site (sg_n,vs_1,up_i,vs_2,up_j,ComEst,False,False))
                    return True;
                endif
                If (Check_Conflict_Site (sg_n,vs_1,up_i,Pred(vs_2),up_j,ComEst,False,True))
                    return True;
                endif
                If (Check_Conflict_Site (sg_n,Pred(vs_1),up_i,vs_2,up_j,ComEst,True,False))
                    return True;
                endif
                If (Check_Conflict_Site (sg_n,Pred(vs_1),up_i,Pred(vs_2),up_j,ComEst,True,True))
                    return True;
                endif
          else
                If (Check_Conflict_Site (sg_n,vs_1,up_i,vs_2,up_j,ComEst,False,True))
                    return True;
                endif
                If (Check_Conflict_Site (sg_n,Pred(vs_1),up_i,vs_2,up_j,ComEst,True,True))
                    return True;
                endif
          endif
     else
          If ((vs_2 is a site with operator +) and (Flag_2))
                If (Check_Conflict_Site (sg_n,vs_1,up_i,vs_2,up_j,ComEst,True,False))
                    return True;
                endif
                If (Check_Conflict_Site (sg_n,vs_1,up_i,Pred(vs_2),up_j,ComEst,True,True))
                    return True;
                endif
          else
                If (vs_1.n! = vs_2.n)
/* Case 1*/             return False;
                else
                    If (vs_1.sl! = vs_2.sl)
                        If (∃cls_k ∈ vs_1.n s.t. (vs_1.sl ∈ cls_k and vs_2.sl ∈ cls_k))
/* Case 3*/                     return True;
                        else
/* Case 2*/                     return False;
                        endif
                    else
                        If (∃cls_k ∈ vs_1.n s.t. (vs_1.sl ∈ cls_k and vs_1.sl has attribute s))
/* Case 4*/                     return True;
                        else
/* Case 5*/                     return Check_Conflict_Site (sg_n,Pred(vs_1),up_i, Pred(vs_2),up_j,ComEst,True,True))
                        endif
                    endif
                endif
          endif
     endif
endif
end
```

Figura 5.21: Función `Check_Conflict_Site`.

el apartado previo, simplemente se ejecuta una llamada del algoritmo básico por cada conjunto de parámetros de entrada, es decir, nuevos grafos de forma y access paths. En el apartado de resulta-

```
fun Check_Conflict_Site_pre (sg_n, vs_1, up_i, vs_2, up_j)
If (vs_1 == null and vs_2 == null)
    return True;
else
    If (vs_1 == null or vs_2 == null)
        return False;
    else
        If (vs_1.n! = vs_2.n)
            return False; /*Case1*/
        else
            If (vs_1.sl! = vs_2.sl)
                If (∃cls_k ∈ vs_1.n s.t. (vs_1.sl ∈ cls_k and vs_2.sl ∈ cls_k))
                    return True; /*Case3*/
                else
                    return False; /*Case2*/
                endif
            else
                If (∃cls_k ∈ vs_1.n s.t. (vs_1.sl ∈ cls_k and vs_1.sl has attribute s))
                    return True; /*Case4*/
                else
                    return Check_Conflict_Site_pre (sg_n, Pred(vs_1), up_i, Pred(vs_2), up_j))
                endif
            endif
        endif
    endif
endif
end
```

Figura 5.22: Función Check_Conflict_Site_pre.

```
/* Input: sg_n un grafo de forma, vs_k un sitio, nav_i subpath de navegación Path_i */
/* Output: True si existe conflicto entre vs_k y vs_l | False en otro caso */
fun Check_Ciclic_nav (sg_n, vs_k, nav_i)
VS = Compute_Visited_Sites(sg_n, vs_k, nav_i) ∪ vs_k ;
    foreach vs_p ∈ VS
        foreach vs_q ∈ VS
            If (Check_Conflict_Site_nav(sg_n, vs_p, vs_q))
                return True;
            endif
        endfor
    endfor
return False;
end
```

Figura 5.23: Función Check_Ciclic_nav.

dos experimentales, analizamos también un código que contiene modificación de la estructura de datos.

5.5.3. Ejemplo práctico

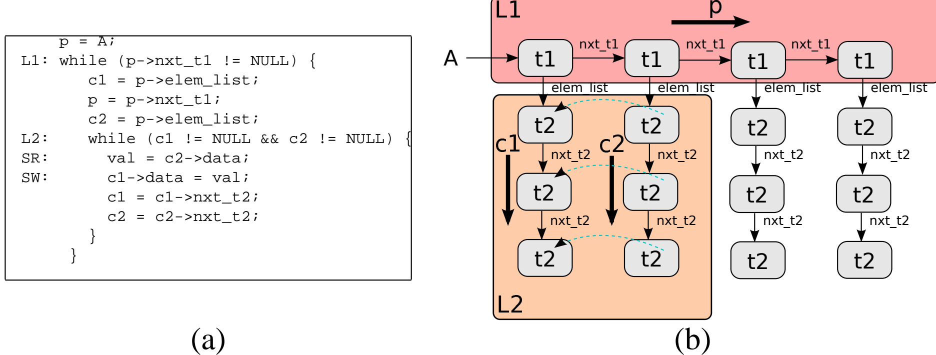
Para explicar todo el proceso del test de dependencias vamos a utilizar un código un poco más complejo con un doble bucle anidado como el mostrado en la Fig. 5.26(a). Este código recorre una matriz dinámica con la estructura de la Fig. 5.5, desplazando los valores de los elementos de las columnas hacia la izquierda ($a[i][j] = a[i][j + 1]$). En la figura 5.26(b), podemos ver la acción de los recorridos sobre la estructura.

```
/* Input: sg_n un grafo de forma, vs_k un sitio, vs_l un sitio */
/* Output: True si existe conflicto entre vs_k y vs_l | False en otro caso */
fun Check_Conflict_Site_nav (sg_n, vs_k, vs_l)
If (vs_k.n! = vs_l.n)
    return False;
else
    If (vs_k.sl! = vs_l.sl)
        If (∃cls_m ∈ vs_k.n s.t. (vs_k.sl ∈ cls_m and vs_l.sl ∈ cls_m))
            return True;
        else
            return False;
        endif
    else
        If (∃cls_m ∈ vs_k.n s.t. (vs_k.sl ∈ cls_m and vs_k.sl has attribute s))
            return True;
        else
            return False
        endif
    endif
endif
end
```

Figura 5.24: Función `Check_Conflict_Site_nav`.

```
/* Input: sg_n un grafo de forma, vs_k un sitio, Subpath_i = step_1.step_2.....step_m un subpath */
/* Output: UP = {up_k} un conjunto de unrolled paths, donde up_k = vs_1.vs_2.....vs_t */
fun Unroll_Path (sg_n, vs_k, Subpath_i)
cp_⊥ = vs_k;
W = < cp_⊥, step_1 > S;
CP = ∅;
while (W! = ∅)
    Remove < up_i, step_j > from W;
    vs_l= Compute_Last_Site (sg_n, up_i);
    VS1 = Compute_Visited_Sites(sg_n, vs_l, step_j);
    step'_j= Compute_Next_Step (Subpath_i, step_j);
    If ( step_j == (sel_r | sel_s | ...)* )
        If (step'_j! = null)
            Insert < up_i, step'_j > in W;
        else
            UP = UP ⋃ up_i;
        endif
    endif
    foreach vs_p ∈ VS1
        up'_i = up_i.vs_p;
        If ( step_j == (sel_r | sel_s | ...)^(*|+) )
            If ( vs_l.vs_p is not in up_i)
               Insert < up'_i, step_j > in W;
            else
               UP = UP ⋃ up'_i;
            endif
        endif
        If (step'_j! = null)
            Insert < up'_i, step'_j > in W;
        else
            UP = UP ⋃ up'_i;
        endif
    endfor
end
```

Figura 5.25: Función `Unroll_Path`.



(a) (b)

Figura 5.26: (a) Código con doble bucle anidado. (b) Recorridos e inducciones sobre la estructura real.

El primer bucle recorre los elementos de la lista cabecera con el puntero de inducción `p` (sólo hay un recorrido). El segundo bucle recorre a la vez dos columnas: la primera con el puntero de inducción `c1` partiendo de la columna apuntada desde elemento accedido por `p`, y la segunda con el puntero de inducción `c2` desde la columna apuntada desde el elemento accedido por $p \rightarrow nxt_t1$ (ver figura 5.26).

Vamos a mostrar el estudio de `LCDs` para el bucle `L1`. Lo primero que haremos será construir los *paths* de acceso de las sentencias `SR` y `SW` que son las que leen y escriben en un campo del mismo tipo y, por tanto, pueden generar LCDs. Como veremos a continuación, los paths de dichas sentencias son distintos para cada uno de los bucles. Una vez calculados los paths, se aplicará paso a paso el test de dependencias extrayendo, en cada caso, la información necesaria del grafo de forma para realizar los chequeos.

`LCD` para el bucle L1

El primer paso es la construcción de los paths para las sentencias a analizar teniendo en cuenta que el bucle de interés es `L1`. Recordamos el formato de los *paths*: $< ce : (nce \| pre) : nav : tail >$.

Generación de los *paths*

El punto de entrada común, ce, es cualquier acceso (con navegación o no) previo a la navegación del bucle analizado que es común a ambos accesos analizados. Es decir, ambos accesos han utilizado los mismos punteros en esta parte del acceso. Este ce puede existir o no.

El punto de entrada no común, con navegación (nce) o sin ella (pre), es cualquier acceso previo a la navegación del bucle analizado que es distinto para ambos accesos analizados. Es decir, son caminos que utilizan distintos selectores en cada uno de los accesos. Al igual que el anterior puede existir o no.

Un detalle a tener en cuenta ahora es que el concepto de *navegación común* asociado a nav, introducido anteriormente, solo puede aparecer cuando no existe *punto de entrada no común*, ya que si ambos accesos divergen antes de la navegación, es imposible que utilicen el mismo navegador en el bucle analizado (en otras palabras, como *navegación común* implica que ambos accesos utilizan el mismo puntero de inducción en el bucle analizado, es imposible que no utilicen los mismos selectores hasta llegar a la primera definición del puntero de inducción).

Así, los *paths* de las sentencias SR y SW quedarían de la siguiente manera:

- SR: $< A : - : nxt_t1^* : nxt_t1 \rightarrow elem_list \rightarrow nxt_t2^* >$
- SW: $< A : - : nxt_t1^* : elem_list \rightarrow nxt_t2^* >$

En este bucle $L1$ los dos accesos utilizan el mismo puntero de inducción p, por lo tanto, la primera asignación del puntero A es el ce, y la expresión de navegación del puntero de inducción, p, (nxt_t1) es nav. Sin embargo, aunque utilizan el mismo puntero de inducción para navegar en $L1$, el acceso SW avanza por nxt_t1 que es la expresión de navegación de dicho puntero de inducción. Esto hace que la navegación no sea común, puesto que este nxt_t1 es un desfase en dicha navegación. El resto del camino seguido para llegar al final del acceso es el $tail$.

Una vez obtenidos los paths y el grafo que representa la estructura pasamos a realizar el test de dependencias LCDs. El objetivo final es determinar si dos accesos de SR_i y SW_j, pertenecientes a dos iteraciones distintas ($i \neq j$) del bucle $L1$, pueden acceder al mismo elemento de la estructura de datos. Para ello, tenemos que proyectar los paths sobre el grafo y decidir si es posible este conflicto. Este chequeo se va a realizar en varias fases en las que se desplegará una sola parte del path.

Despliegue de *ce*

En primer lugar, proyectaremos la parte ce (si existe) del path, obteniendo los *sitios* que serán los puntos de partida de la siguiente fase. Puesto que ce es común a ambos paths, estos sitios alcanzados serán puntos de partida individuales para la siguiente fase, es decir, el trozo del subpath a desplegar a continuación para ambos paths partirá exactamente del mismo sitio. O sea, si el ce alcanza los sitios s1 y s2, los chequeos que se realizarían serían partiendo de s1 y desplegando el resto de ambos paths, y partiendo de s2 y desplegando el resto de ambos paths. No se chequearían conflictos desplegando un path desde s1 y el otro desde s2 puesto que no sería un caso real cuando el punto de entrada es común.

En nuestro ejemplo, el ce es A, por lo que si tomamos el grafo este enlace viene representado por $PL1$. Por tanto, el único sitio alcanzado por ce sería $vsce =< PL1, N1 >$. En la figura 5.27(a) podemos ver la representación de este sitio sobre el grafo.

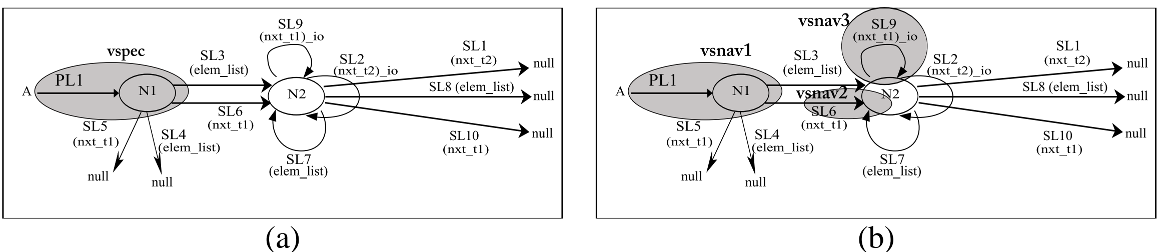


Figura 5.27: Sitio obtenido al desplegar (a) *ce* y (b) *nav* para ambos paths en L1.

Despliegue de *nav*

Puesto que no hay punto de entrada no común, ni preámbulo, $nce||pre$, en ninguno de los paths, ahora hay que desplegar las navegaciones a partir de *vsce*. Al ser una navegación no común, debemos comprobar si existen sitios que puedan ser visitados por ambas navegaciones. Si ocurre

esto, entonces el despliegue de la siguiente parte del path $tail$ podría empezar en el mismo *sitio* para ambos accesos (lo que denominamos ***Estructura Común,*** $ComEst = True$). Si no hay ningún sitio que pueda ser visitado por ambas navegaciones, entonces podemos asegurar que los $tails$ se desplegarán siempre desde *sitios* distintos ($ComEst = False$).

El campo nav de los pahts del ejemplo es nxt_t1^*. Lo desplegamos a partir de $vsce =< PL1, N1 >$. Los sitios obtenidos en ambas navegaciones son: $vsnav1 =< PL1, N1 >$, $vsnav2 =< SL6, N2 >$ y $vsnav3^+ =< SL9, N2 >^+$ (ver figura 5.27(b)). Ahora debemos comprobar si hay algún sitio que pueda ser visitado por ambas navegaciones. En el ejemplo, los *Unrolled Paths* de ambos accesos coinciden ya que la navegación es igual (aunque es *no común*) y serían estos:

- $UP_1 = [< PL1, N1 >]$ (sólo el sitio inicial),
- $UP_2 = [< PL1, N1 >, < SL6, N2 >]$ (el sitio inicial y un sitio más por nxt_t1),
- $UP_3 = [< PL1, N1 >, < SL6, N2 >, < SL9, N2 >^+]$ (el sitio inicial, el primer sitio por nxt_t1, y luego uno o más sitios de nuevo por nxt_t1)

Puesto que los UP son los mismos para ambas navegaciones, hay que aplicar las RCC para cada pareja de UP. Si chequeamos UP_1 contra él mismo, nos da conflicto, ya que al estar formado tan sólo por un sitio y este ser el mismo ($< PL1, N1 >$) aplicaríamos la regla ***RCC5***, que al no tener sitio previo da conflicto puesto que la entrada a las dos navegaciones es común ($ComEst = True$).

Por tanto, determinamos que las navegaciones pueden visitar los mismos sitios en iteraciones distintas del bucle, lo que implica que los $tails$ se pueden desplegar desde un sitio común.

Despliegue de *tail*

Ahora debemos desplegar el $tail$ de ambos paths, partiendo de los sitios visitados en cada una de las navegaciones, $vsnav1 =< PL1, N1 >$, $vsnav2 =< SL6, N2 >$ y $vsnav3^+ =< SL9, N2 >^+$ (cada uno visita por separado los mismos sitios). Es decir, tendremos que desplegar el $tail$ de cada path partiendo de todas las parejas de sitios de entrada que se pueden formar con los sitios anteriores ($vsnav1 \times vsnav1$, $vsnav1 \times vsnav2$, $vsnav1 \times vsnav3$, $vsnav2 \times vsnav1$, ...). El problema ahora es determinar si desplegando dichos $tails$ a partir de esos pares de sitios de entrada (con $ComEst = True$ detectada en la navegación), es posible llegar al mismo sitio concreto, lo que implicaría un conflicto `LCD`.

Para chequear esto vamos a utilizar de nuevo los *Unrolled Paths* (`UP`) que se obtiene de los $tails$ partiendo de las parejas de sitios de entrada, chequeándolos todos con las reglas RCC. Por ejemplo compararíamos:

- $tail$ de `SR` desplegado desde $vsnav1 =< PL1, N1 >$ (representaría el camino de los accesos de lectura (`SR`) de la primera iteración de $L1$):
  - $UP_{SR1} = [< SL6, N2 >, < SL7, N2 >]$ (sitio alcanzado desde el inicial siguiendo nxt_t1 ($SL6$) y a continuación $elem_list$ ($SL7$) como indica el $tail$ de `SR`).
  - $UP_{SR2} = [< SL6, N2 >, < SL7, N2 >, < SL2, N2 >^+]$ (sitio alcanzado desde el inicial siguiendo nxt_t1 ($SL6$), luego $elem_list$ ($SL7$) y a continuación una o más veces nxt_t2 ($SL2$) como indica el $tail$ de `SR`).

- $tail$ de `SW` desplegado desde $vsnav2 =< SL6, N2 >$ (representaría el camino de los accesos de escritura (`SW`) de la segunda iteración de $L1$):
  - $UP_{SW1} = [< SL7, N2 >]$ (sitio alcanzado desde el inicial siguiendo $elem_list$ ($SL7$))
  - $UP_{SW2} = [< SL7, N2 >, < SL2, N2 >^+]$ (sitio alcanzado desde el inicial siguiendo $elem_list$ ($SL7$) y a continuación 1 o más veces nxt_t2 ($SL2$) como indica el $tail$ de `SW`)

Cuando hacemos el chequeo de $UP_{SR1} = [< SL6, N2 >, < SL7, N2 >]$ y $UP_{SW1} = [< SL7, N2 >]$, aplicaríamos la regla $RCC5$ sobre el último sitio de ambos paths (puesto que coincide nodo y selector y este no es shared en ningún cls). Al ir hacia atrás, el camino UP_{SW1} se agota, por lo que devolvemos conflicto si partíamos de estructura común al desplegar los $tails$, como ocurre en este caso. Por tanto, me devolvería conflicto en el último sitio accedido por el $tail$ de `SR` y `SW`, por tanto hay `LCD`. En la figura 5.28 mostramos sobre la estructura real la representación de UP_{SR1} y UP_{SW1} partiendo de los sitios desde los cuales son desplegados ($vsnav1$ y $vsnav2$) y el `LCD` detectado.

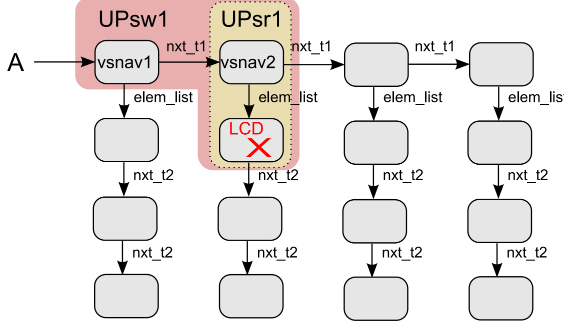


Figura 5.28: Representación sobre la estructura real de los *Unrolled Paths* que provocan un `LCD`.

`LCD` para el bucle L2

Vamos a realizar el test de dependencias ahora para el bucle `L2` del código de la Fig. 5.26(a) en el cual se recorren dos columnas de la matriz escribiendo los valores de la columna de la derecha en la de la izquierda. Las sentencias que pueden provocar las LCDs son las mismas (`SR, SW`) sin embargo tenemos que crear de nuevo los paths de ambas, puesto que, como ya hemos dicho, los paths no sólo dependen de los accesos de las sentencias sino del bucle a analizar así como de la relación entre ambas sentencias(`navegación común` ...).

Generación de los *paths*

Para el bucle `L2` claramente la navegación es `no común` puesto que cada acceso utiliza su propio puntero de inducción (`c1` y `c2`) para navegar la estructura de datos. Sin embargo si que tienen algo en común y es el punto de entrada y la navegación del nivel `L1` ya que ahí si que utilizan el mismo puntero de inducción (`p`). Sin embargo, la navegación de `L2` no se despliega desde el

mismo punto común proveniente de la navegación de L1, sino que c2 avanza por nxt_t1 antes de empezar la navegación. A los enlaces seguidos hasta la navegación de L2 los denominamos *preámbulo* y va a tener un tratamiento especial con respecto a lo visto anteriormente, a la hora de realizar el test de dependencias. Básicamente el *preámbulo* indica un desfase sin navegación, desde una navegación común (L1) hasta la navegación del bucle de estudio (L2). Es muy importante puesto recoge el hecho de que ambas navegaciones actuales que son distintas (no común) partieron de un punto de navegación común del bucle superior.

Por tanto los paths de las sentencias quedarían como siguen, siguiendo el formato: $< ce : (nce \| pre) : nav : tail >$.

- SR: $< A \rightarrow nxt_t1^* : nxt_t1 \rightarrow elem_list : nxt_t2^* : - >$
- SW: $< A \rightarrow nxt_t1^* : elem_list : nxt_t2^* : - >$

Despliegue de *ce*

Tenemos que encontrar los sitios abstractos visitados por esta navegación común y que serán puntos de partida comunes para el resto del acceso (para ambas sentencias). Desplegamos el path *ce* = $A \rightarrow nxt_t1^*$, que nos devuelve el siguiente conjunto de sitios visitados (es exactamente el mismo conjunto obtenido al desplegar la navegación para L1 y que se muestra en la Fig. 5.27(a) ya que se trata del mismo recorrido): $vsce1 =< PL1, N1 >$ (representa sitio apuntado por A, el primero del recorrido), $vsce2 =< SL6, N2 >$ (representa el segundo, el apuntado directamente por el sitio apuntado por A) y $vsce3^+ =< SL9, N2 >^+$ (representa el resto de sitios visitados).

Cada uno de estos sitios abstractos será el comienzo para el despliegue del resto de ambos paths, por tanto, para cada uno de ellos vamos a tener que repetir el proceso de búsqueda de conflictos. Recordemos que al ser estos puntos alcanzados por la parte *ce*, vamos a hacer un chequeo por cada punto (desplegamos el resto de los paths a partir de $vsce1$, $vsce2$ y $vsce3$) pero no vamos a mezclar puntos de comienzo distintos. Es decir, no vamos a chequear el despliegue del path SR desde un $vsce_i$ y el de SW desde otro $vsce_j$ distinto.

Despliegue de *pre*

Lo primero que hacemos al desplegar el *preámbulo* es chequear la propiedad de estructura común, $ComEst$, presentada anteriormente, que se le pasará a la navegación. Para ello vamos a chequear si hay o no conflicto con los unrolled paths de cada acceso y para cada sitio de entrada común, aplicando una ligera variación de las RCC específica para los preámbulos.

- Sitio abstracto $vsce1 =< PL1, N1 >$:
  - Preámbulo SW (*elem_list*): $UP_{w1} = [< SL3, N2 >]$
  - Preámbulo SR ($nxt_t1 \rightarrow elem_list$): $UP_{r1} = [< SL6, N2 >, < SL7, N2 >]$

  Al aplicar las RCC sobre los paths no da conflicto ya que en N2 no aparecen SL3 y SL7 de entrada en ningún CLS.

- Sitio abstracto $vsce2 =< SL6, N2 >$:
  - Preámbulo `SW` ($elem_list$): $UP_{w2} = [< SL7, N2 >]$
  - Preámbulo `SR` ($nxt_t1 \rightarrow elem_list$): $UP_{r2} = [< SL9, N2 >, < SL7, N2 >]$

  Al aplicar las `RCC` sobre los paths es cuando aplicamos una regla especial al tratarse del despliegue de un preámbulo. Al comparar los sitios finales ($< SL7, N2 >$) vemos que son idénticos por lo que aplicaríamos la regla `RCC5`. Al llevar hacia atrás UP_{w2} éste se agota en un camino pero en el otro no, así que devolvemos que no hay conflicto. Si se hubieran agotado ambos a la vez por la regla `RCC5` si se devolvería conflicto. Esto es así porque el preámbulo siempre parte de una navegación común, es decir, parte de un sitio común para ambos paths, y además no se navega en dicho preámbulo. Esto implica que si al desplegarlos, no son exactamente el mismo UP (uno se agota antes), es imposible que lleguen al mismo punto final puesto que partían de un mismo punto y han hecho un recorrido distinto. Por el contrario si los UP son idénticos, al partir de un sitio común, seguro que el sitio alcanzado también lo es (de ahí que se devuelva conflicto).

- Sitio abstracto $vsce3 =< SL9, N2 >$:
  - Preámbulo `SW` ($elem_list$): $UP_{w2} = [< SL7, N2 >]$
  - Preámbulo `SR` ($nxt_t1 \rightarrow elem_list$): $UP_{r2} = [< SL9, N2 >, < SL7, N2 >]$

  Es exactamente igual que el anterior, por lo que tampoco hay conflicto.

Vemos que todos los chequeos realizados nos han informado de que para la siguiente fase no hay estructura común, es decir, las navegaciones se van a desplegar desde sitios que pertenecen a trozos de estructura disjuntos (aunque comiencen ambas en el mismo sitio abstracto representará distintos elementos concretos). Partiendo de la condición de estructura no común, $ComEst = False$, para la fase de navegación, pasamos a determinar los sitios a partir de los cuales se deben desplegar las navegaciones. Estos puntos son los sitios finales visitados por los preámbulos, en definitiva los sitios finales de los UP calculados previamente. Así tenemos que los sitios visitados para cada uno de los puntos de entrada de esta fase son:

- Sitio abstracto $vsce1 =< PL1, N1 >$: $vspre_{w1} =< SL3, N2 >$ y $vspre_{r1} =< SL7, N2 >$
- Sitio abstracto $vsce2 =< SL6, N2 >$: $vspre_{w2} =< SL7, N2 >$ y $vspre_{r2} =< SL7, N2 >$
- Sitio abstracto $vsce3 =< SL9, N2 >$: $vspre_{w3} =< SL7, N2 >$ y $vspre_{r3} =< SL7, N2 >$

Ahora hay que desplegar las navegaciones a partir de cada par de puntos de entrada ($vspre_{r1} \times vspre_{w1}$ y $vspre_{r2} \times vspre_{w2}$). Para $vspre_{r3}$ y $vspre_{w3}$ no hace falta ya que son los mismos que el anterior. Aquí podemos observar la importancia de estructura no común ($ComEst = False$), ya que vamos a desplegar desde un mismo sitio abstracto ($vspre_{r2} = vspre_{w2}$) las dos navegaciones, pero podemos estar seguros que comienzan en dos elementos concretos distintos (representados por el mismo sitio abstracto).

Despliegue de *nav*

Vamos a desplegar las navegaciones desde dos pares de sitios de entrada ($vspre_{r1} \times vspre_{w1}$ y $vspre_{r2} \times vspre_{w2}$) y con el conocimiento de que la navegación es no común (información de la creación del path) y que la estructura recorrida en la navegación a partir de esos puntos también es no común (información de la fase anterior).

Al ser navegación no común, vamos a determinar si hay conflicto en los accesos de ambas navegaciones calculando los UP de cada acceso a partir de cada par de puntos de entrada y vamos a comprobar con las `RCC` si existe o no conflicto, es decir, si hay o no estructura común para la siguiente fase.

- Sitio abstracto $vspre_{w1} =< SL3, N2 >$ y navegación `SW` (nxt_t2^*):
  - $UP_{w1} = [< SL3, N2 >]$ (el sitio inicial sin navegar por `nxt_t2`)
  - $UP_{w2} = [< SL3, N2 >, < SL2, N2 >^+]$ (el resto de sitios alcanzados por `nxt_t2`)
- Sitio abstracto $vspre_{r1} =< SL7, N2 >$ y navegación `SR` (nxt_t2^*):
  - $UP_{r1} = [< SL7, N2 >]$ (el sitio inicial sin navegar por `nxt_t2`)
  - $UP_{r2} = [< SL7, N2 >, < SL2, N2 >^+]$ (el resto de sitios alcanzados por `nxt_t2`)

  Al aplicar las `RCC` a cada UP de `SR` con cada uno de `SW` obtenemos que no hay conflicto ($UP_{w1} \times UP_{r1}$ por la regla `RCC2`, $UP_{w1} \times UP_{r2}$ por `RCC2`, $UP_{w2} \times UP_{r1}$ por `RCC2` y $UP_{r2} \times UP_{w2}$ por `RCC5 ->RCC2`).
- Sitio abstracto $vspre_{w2} =< SL7, N2 >$ y navegación `SW` (nxt_t2^*):
  - $UP_{w3} = [< SL7, N2 >]$ (el sitio inicial sin navegar por `nxt_t2`)
  - $UP_{w4} = [< SL7, N2 >, < SL2, N2 >^+]$ (el resto de sitios alcanzados por `nxt_t2`)
- Sitio abstracto $vspre_{r2} =< SL7, N2 >$ y navegación `SR` (nxt_t2^*):
  - $UP_{r3} = [< SL7, N2 >]$ (el sitio inicial sin navegar por `nxt_t2`)
  - $UP_{r4} = [< SL7, N2 >, < SL2, N2 >^+]$ (el resto de sitios alcanzados por `nxt_t2`)

  Al aplicar las `RCC` a cada UP de `SR` con cada uno de los de `SW` obtenemos que no hay conflicto. En este caso es fundamental la condición de entrada de estructura no común, ya que puesto que muchas de las comparaciones son de dos UP idénticos, por lo que se aplica la regla `RCC5` hasta agotar los paths y devuelve que no hay conflicto.

  En la figura 5.29 podemos ver la representación de los UP_{r4} y UP_{w4} tanto concretos (a) como abstractos (b). Vemos que en el caso concreto es fácil apreciar que ambos recorridos visitan sitios totalmente distintos por lo que no hay conflicto. Sin embargo, en el caso del grafo (abstracto) esto ya no es tan obvio ya que ambos UP son exactamente iguales (formados por los dos mismos sitios abstractos $< SL7, N2 >$ y $< SL2, N2 >$). Por tanto, para determinar si el último punto de acceso de la navegación es visitado por ambos paths a la vez, debemos conocer si ambos UPs se han podido desplegar desde un mismo sitio

(`estructura común`). Si no se han desplegado desde un mismo sitio, los UPs abstractos (b) están representando dos recorridos concretos distintos (a) que han sido representados en el grafo por los mismos sitios abstractos. Vemos aquí la importancia de la información `estructura común` que se va calculando en cada fase y se pasa a la siguiente.

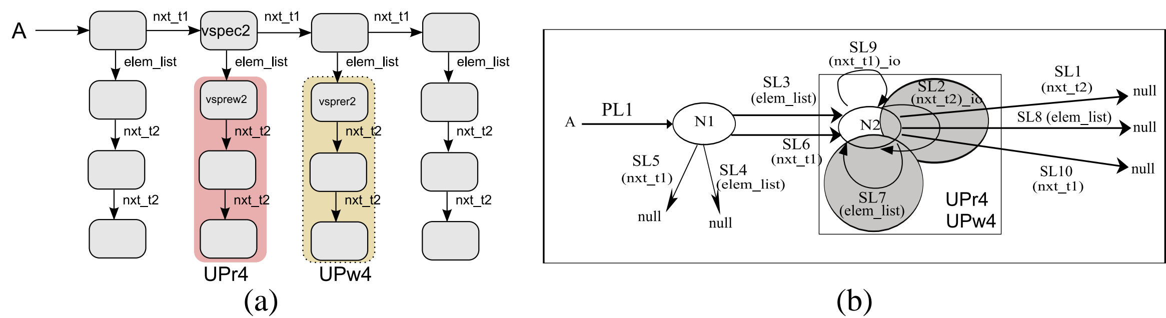


Figura 5.29: (a) Elementos concretos representados por los `unrolled paths` UP_{w4} y UP_{r4}. (b) Sitios abstractos que forman ambos `unrolled paths` UP_{w4} y UP_{r4} ($< SL7, N2 >$ y $< SL2, N2 >$).

Como vemos, no hay conflicto en los accesos de la navegación, por lo que podemos determinar que para la siguiente fase (*tail*) vamos a seguir partiendo de estructura no común.

Ahora determinamos los sitios alcanzados en estas navegaciones y que serán los puntos de entrada para desplegar los *tail*s. Los sitios visitados son los mismos calculados en los UPs anteriores.

- Sitios $vsnav_{w1} =< SL3, N2 >$ y $vsnav_{w2} =< SL2, N2 >^+$ (desde $vspre_{w1} =< SL3, N2 >$) y $vsnav_{r1} =< SL7, N2 >$ y $vsnav_{r2} =< SL2, N2 >^+$ (desde $vspre_{r1} =< SL7, N2 >$)
- Sitios $vsnav_{w3} =< SL7, N2 >$ y $vsnav_{w4} =< SL2, N2 >^+$ (desde $vspre_{w2} =< SL7, N2 >$) y $vsnav_{r3} =< SL7, N2 >$ y $vsnav_{r4} =< SL2, N2 >^+$ (desde $vspre_{r2} =< SL7, N2 >$)

Esos serán los sitios de entrada para desplegar el *tail*.

Despliegue de *tail*

Puesto que no hay *tail* en ninguno de los dos paths, lo único que hay que hacer es comprobar que no hay conflicto de acceso en las parejas de sitios de entrada, aplicando directamente las reglas `RCC` sobre dichos sitios (no hay que calcular ningún UP ya que no hay *tail*, o lo que es lo mismo, los UP son los propios sitios de entrada). Recordemos que venimos de estructura no común en la navegación, información que va a ser decisiva en esta fase.

Enfrentamos todos las parejas de puntos de entrada:

- `RCC`($vsnav_{w1} =< SL3, N2 >, vsnav_{r1} =< SL7, N2 >$) = no conflicto por `RCC2`
- `RCC`($vsnav_{w1} =< SL3, N2 >, vsnav_{r2} =< SL2, N2 >$) = no conflicto por `RCC2`
- `RCC`($vsnav_{w2} =< SL2, N2 >, vsnav_{r1} =< SL7, N2 >$) = no conflicto por `RCC2`

- `RCC`($vsnav_{w2} =< SL2, N2 >$, $vsnav_{r2} =< SL2, N2 >$) = no conflicto por `RCC5` y $ComEst = False$
- `RCC`($vsnav_{w3} =< SL7, N2 >$, $vsnav_{r3} =< SL7, N2 >$) = no conflicto por `RCC5` y $ComEst = False$
- `RCC`($vsnav_{w3} =< SL7, N2 >$, $vsnav_{r4} =< SL2, N2 >$) = no conflicto por `RCC2`
- `RCC`($vsnav_{w4} =< SL2, N2 >$, $vsnav_{r3} =< SL7, N2 >$) = no conflicto por `RCC2`
- `RCC`($vsnav_{w4} =< SL2, N2 >$, $vsnav_{r4} =< SL2, N2 >$) = no conflicto por `RCC5` y $ComEst = False$

Por tanto no hay ningún conflicto por ningún acceso, por lo que no hay LCD.

5.6. Complejidad

El algoritmo utilizado para la detección de dependencias depende de dos parámetros de entrada, los grafos de forma y los access paths. Cuanto mayor sea el número de grafos de forma a tener en cuenta en el análisis más costoso resulta. Sea `NG` el número de grafos de forma a considerar que depende del tipo de código a analizar; `NS` el número de sentencias a analizar; `NN` el número de sitios abstractos generados a partir de los access paths proyectados sobre los grafos de forma, que se compone de los sitios nombrados como `SL` y `PL` en los ejemplos presentados; y `D` el nivel de profundidad de anidamiento del bucle o el número de llamadas en el ciclo que contiene la llamada recursiva en el *call graph*. Entonces la complejidad del algoritmo vendría dada por la expresión:

$$O\left[D \times \left[\frac{NN \times NN}{2}\right] \times \left[\frac{NN \times NN}{2}\right] \times NN \times NG \times (\frac{NS \times NS}{2})\right]$$

La componente $\left[\frac{NN \times NN}{2}\right]$ se debe a la llamada del método que chequea los conflictos en primera instancia, `Check_Conflict_xx`, donde `xx` puede ser bien `ce`, `nce-pre`, `nav` o `tail`. La segunda componente $\left[\frac{NN \times NN}{2}\right]$ se debe al método `Check_ComEst`. Este método a su vez hace tantas llamadas al correspondiente método `Check_Conflict_Site` como sitios haya que chequear. Por último, este proceso debe repetirse por cada grafo, (`NG`), y por cada pareja de sentencias que pueden provocar conflicto ($\frac{NS \times NS}{2}$). Finalmente simplificando la expresión, obtenemos un término polinómico asintótico de la complejidad tal que $O < [NN^6 \times NG \times NS^2]$.

5.7. Resultados Experimentales

Para los resultados experimentales, hemos considerado la mayoría de los programas que ya fueron presentados en capítulos anteriores:

1. **Matrix x Vector**: código para la multiplicación de una matriz dispersa por un vector disperso.

2. **Matrix x Matrix**: multiplicación de dos matrices dispersas.

3. **TreeAdd**: programa recursivo para la creación y recorrido de un árbol binario.

4. **Power**: programa recursivo para la creación y recorrido de una estructura multinivel.

5. **Em3d**: programa que presenta una estructura dinámica compleja a través de la interconexión de dos listas simplemente enlazadas.

6. **Bisort**: otro programa recursivo que trabaja con árboles binarios.

En los dos programas de operaciones con matrices, se produce una modificación de la estructura de datos durante el recorrido. En el resto de casos se aplica el algoritmo básico presentado. En los casos donde se aplica el algoritmo extendido, el análisis de forma puede trabajar con más grafos de forma, por lo que en consecuencia, también pueden ser necesarios, por lo general, más grafos para el test de dependencias. En el código `Matrix x Matrix` hay modificación de la estructura de datos durante el recorrido, concretamente en 7 sentencias del programa, luego es necesario aplicar el algoritmo extendido, de la forma que se explicó en la sección 5.5.2 y, por tanto, han sido utilizados 7 grafos de forma diferentes en el análisis, mientras que para la multiplicación de matriz por vector, sólo es necesario un grafo. En el resto de casos, al no haber modificación, sólo ha sido necesario un grafo de forma representativo de la estructura a la entrada de cada bucle analizado.

En la tabla 5.3 mostramos los distintos parámetros que influyen en el test de dependencias. Por un lado tenemos el total de las sentencias (de lectura y escritura de datos) que son seleccionadas dentro de cada programa para llevar a cabo el análisis. Recordamos que a partir de estas sentencias son calculados los correspondientes paths. El número de grafos repercute directamente en el tiempo de análisis, por ello para el segundo código, al ser uno de los casos complejos, es el más costoso si atendemos a los tiempos de esta tabla. Las medidas de tiempo de la tabla han sido tomadas en una máquina Pentium 4 a 3Ghz y con 1GB de memoria RAM. En ningún caso son detectadas dependencias.

Código	**Sentencias**	**Conf.Groups**	**Grafos**	**Tiempo**
Matrix x Vector	6	3	1	6.3331ms
Matrix x Matrix	12	3	7	15.8049ms
TreeAdd	2	2	1	3.6508ms
Power	4	4	1	2.3269ms
Em3d	4	16	1	6.4742ms
Bisort	4	2	1	7.5609ms

Tabla 5.3: Parámetros y tiempos para el algoritmo `LCDs_Detection`.

Este test como ya explicamos mejora las prestaciones de un análisis de dependencias previo [63],[68], al no necesitar la interpretación abstracta de las sentencias para las que el análisis de conflictos se lleva a cabo. Para comparar equitativamente nuestro análisis con el anterior, debemos tener en cuenta cada una de las fases previas necesarias para llevarlo a cabo. En la Fig. 5.30, presentamos el esquema donde pueden verse cada una de las etapas por las que debe pasar el

código de entrada. Como ya comentamos en el capítulo 3, lo primero que se hace es un análisis de las cadenas DU del programa, siendo esta etapa la más costosa al ser el algoritmo más complejo de los aquí presentados. En la segunda etapa, se eliminan aquellas partes del código que no son relevantes de acuerdo al criterio de slicing, lo que repercute primero de forma directa en el posterior análisis de forma de la tercera etapa, y de forma indirecta en el análisis final de dependencias que usa los grafos de forma proporcionados por el análisis de forma. Gracias a este análisis previo combinado por una etapa de slicing y una etapa inicial de extracción de cadenas DU, ampliamos el rango de códigos y casos posibles que puede analizar una herramienta potente y compleja como es un análisis de forma. De esta forma conseguimos la optimización deseada, ampliando los límites de aplicación y reduciendo tiempos de ejecución.

En la tabla 5.4 referenciamos de nuevo todos los tiempos presentados en capítulos anteriores para cada una de las etapas comentadas. En la última columna tendríamos el tiempo total, fruto de la suma de todos los tiempos anteriores, tras el cual puede concretarse si el código de entrada presenta o no dependencias. Este tiempo será el que utilicemos para compararnos con el anterior test de dependencias.

Código	**DU-Analisis**	**Poda**	**A-Forma**	**DepAnalisis**	**Total**
Matrix x Vector	2.786s	0.182s	0.798s	6.3331ms	3.772s
Matrix x Matrix	7.304s	0.258s	1.355s	15.8049ms	8.933s
TreeAdd	1.270s	0.043s	1.627s	3.6508ms	2.943s
Power	16.03s	0.66s	0.481s	2.3269ms	17.173s
Em3d	48.53s	2.00s	7.363s	6.4742ms	57.899s
Bisort	6.77s	0.918s	1.802s	7.5609ms	9.497s

Tabla 5.4: Medidas del análisis completo.

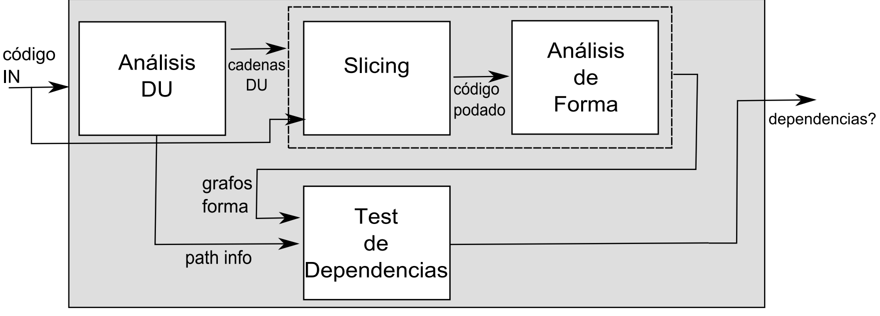


Figura 5.30: Etapas del análisis completo.

En la tabla 5.5 comparamos directamente los tiempos del test de dependencias anterior frente al nuevo test de dependencias. Observamos cómo los órdenes de magnitud son muy diferentes. El análisis de dependencias presentado en este capítulo está basado en la detección de conflictos mientras que el anterior análisis requiere interpretación abstracta. Sin embargo para entender mejor todo el proceso incluimos los tiempos totales en otra tabla.

Código	AnteriorTest	NuevoTest
Matrix x Vector	2m 16s	6.3331ms
Matrix x Matrix	8m 25s	15.8049ms
TreeAdd	25.12s	3.6508ms
Power	4.81s	2.3269ms
Em3d	1m 47s	6.4742ms
Bisort	43m 27s	7.5609ms

Tabla 5.5: Comparativa de tiempos entre ambos análisis de dependencias.

Código	Anterior	Ant+Poda	Nuevo	Aceleración
Matrix x Vector	2m 19.80s	2m 22.77s	3.772s	37.84
Matrix x Matrix	9m 13.38s	9m 20.94s	8.933s	62.79
TreeAdd	31.42s	32.73s	2.943s	11.12
Power	12.91s	29.6s	17.173s	1.72
Em3d	2m 5.88s	2m 56.41s	57.899s	3.04
Bisort	45m 6.88s	45m 14.56s	9.497s	285.83

Tabla 5.6: Aceleración en el análisis de dependencias.

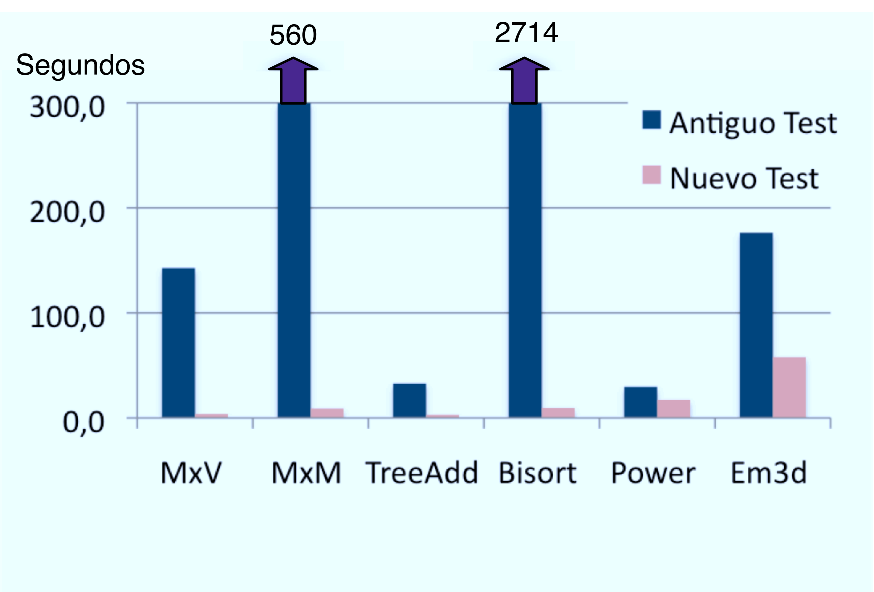


Figura 5.31: Representación de la aceleración en el análisis de dependencias.

En la tabla 5.6 se muestran los tiempos finales para comparar directamente el análisis de dependecias anterior con el análisis de este capítulo. También se muestra la Fig. 5.31 para resaltar de forma gráfica las diferencias entre ambos métodos. En el antiguo test, el *pruning* del código era manual, por lo que mostramos en la primera columna únicamente los tiempos de análisis de dependencias y, a continuación, en la segunda columna, estos tiempos sumando también nuestros tiempos de *pruning* para comparar ambos métodos en igualdad de condiciones (en la gráfica se

observa que la aportación del *pruning* es la misma en ambas columnas). A partir de la tabla 5.4, observamos como la mayor contribución para el nuevo test de dependencias proviene de la etapa de preproceso, donde se lleva a cabo el *slicing* del código y la obtención de los paths. Aunque el análisis de forma es un análisis costoso, vemos como para los códigos analizados, gracias a la etapa de *pruning* el coste de este análisis se ha visto bastante reducido. Por otra parte, la contribución del test de dependencias al tiempo total es insignificante. Precisamente, esta parte es la encargada básicamente de llevar a cabo la detección de análisis de conflictos, que hemos presentado en este capítulo. Otra conclusión que sacamos a partir de la tabla 5.6, es que en todos los casos el nuevo análisis de dependencias basado en detección de conflictos, se ha conseguido una reducción significativa en los tiempos de ejecución. En todos los casos la aceleración conseguida es elevada. La única excepción se presenta para el código `power`, donde los tiempos son similares. En realidad, la complejidad de este nuevo test de dependencias es polinómica, $O[NN^6]$ mientras que el anterior test tenía complejidad exponencial, $O[hv^{uv}]$.

5.8. Conclusiones

En este capítulo hemos presentado un novedoso análisis de dependencias de datos en el *heap* basándonos en información extraída del análisis presentado en el capítulo tres, y aprovechando la optimización aplicada al análisis de forma en el capítulo cuatro, gracias al proceso de *slicing*. Durante el análisis DU, se extrae información de los punteros de inducción y de los llamados *access paths* mientras se realiza el recorrido por las sentencias del código. Dados estos *access paths*, la idea es extraer de los grafos de forma, que representan la forma de las estructuras de datos dinámicas, la información de si es posible que esos recorridos representados por los *paths* puedan visitar un mismo sitio de la estructura en iteraciones distintas del bucle (o llamadas recursivas) bajo estudio. Así pues, a partir de la información proporcionada por el análisis previo del código, y de la información aportada por el análisis de forma, ha sido posible el desarrollo de un método de detección de dependencias que mejora y amplía los casos posibles a tratar por otro anterior test de dependencias. A partir de los resultados experimentales llevados a cabo, concluimos que la nueva herramienta de análisis de dependencias resulta mucho más eficiente (a la vista de las aceleraciones conseguidas), comparado con el test de dependencias anterior desarrollado en este departamento. Esto se ha conseguido porque evitamos la interpretación abstracta de las sentencias donde se lleva a cabo el análisis. La complejidad pasa de ser exponencial a polinómica, lo que hace este nuevo test apropiado para analizar códigos más grandes.

6 Conclusiones

6.1. Conclusiones

Nuestra principal meta radica en la optimización de códigos irregulares, dado su cada vez más importante uso en el ámbito científico, y el desafío que suponen para la mayoría de compiladores actuales. Concretamente nos hemos centrado en el estudio de códigos que crean y recorren estructuras dinámicas de datos.

En nuestro trabajo, hemos propuesto un análisis para extraer información relevante de los punteros al *heap*, puesto que muchas técnicas de compilación modernas no son capaces de abordar el análisis de este tipo de punteros. Nuestro objetivo final es la paralelización automática de códigos en los que aparecen punteros de este tipo. Además hemos probado la utilidad de la información que obtenemos de estos punteros, con una optimización sobre una herramienta de análisis de forma. Finalmente hemos presentado un novedoso test de dependencias para estructuras dinámicas de datos. Este test está basado en un algoritmo de análisis de conflictos, como puede verse en la Fig.6.1, que a su vez requiere información de dos fuentes: i) grafos abstractos de forma, usando el análisis de forma optimizado comentado anteriormente, y ii) expresiones de *path* aportadas por el análisis DU implementado. Gracias al test, podemos analizar códigos C de propósito general que crean, recorren y modifican estructuras dinámicas complejas, lo que no es posible de realizar con otras propuestas en la literatura.

Las principales contribuciones logradas con este trabajo han sido:

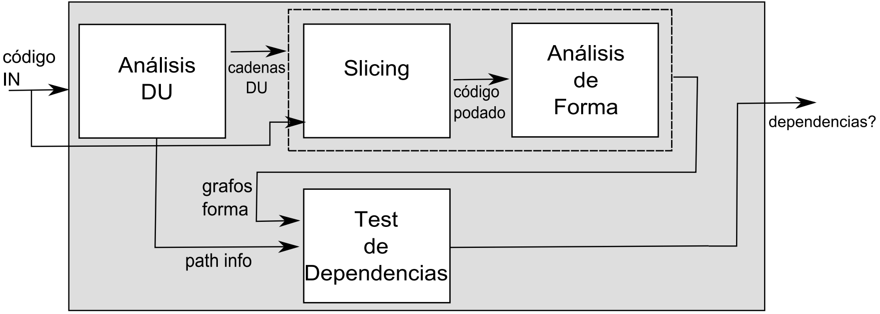


Figura 6.1: Etapas del análisis completo.

1. Hemos diseñado e implementado un análisis DU que permite extraer información relevante de los programas basados en estructuras dinámicas de datos. Para ello fue necesario adaptar el código de entrada a la correspondiente forma SSA, implementando un pase de compilación extra en la plataforma de trabajo CETUS. Se buscaron también mejoras respecto al algoritmo original de transformación SSA añadiendo optimizaciones para obtener una versión más eficiente. Para probar el análisis DU hemos partido de un conjunto conocido de programas de tamaño medio con estructuras complejas, conocido como Olden, encontrando resultados satisfactorios en cuanto al coste temporal de este análisis.

2. Probamos como aplicación del análisis DU la optimización de una herramienta de análisis de forma, a través del diseño e implementación de un pase de *slicing* previo al análisis de forma. Gracias al *code slicing* reducimos el número de sentencias a analizar, en nuestro caso, filtrando y seleccionando sólo aquellas que intervienen directamente en la creación de estructuras dinámicas, por lo que logramos optimizar de forma exitosa una herramienta de análisis de forma. Conseguimos una mejora en cuanto a la aceleración de esta herramienta y el aumento de los casos de estudio abordables.

3. Hemos diseñado e implementado un test de dependencias basado en un novedoso análisis de conflictos para programas con estructuras dinámicas de datos. La nueva herramienta de análisis de dependencias resulta mucho más eficiente comparado con un test de dependencias anterior desarrollado en este departamento. Esto se ha conseguido porque evitamos la interpretación abstracta de las sentencias donde se lleva a cabo el análisis. La complejidad pasa de ser exponencial a polinómica, lo que hace este nuevo test apropiado para analizar códigos más grandes. Mostramos evidencias de la mejora conseguida en el apartado de resultados experimentales, teniendo en cuenta todos los tiempos de las distintas etapas necesarias para ejecutar dicho análisis de dependencias.

Todos los algoritmos presentados en la tesis han sido implementados en Java e incorporados a nuestro *framework* en Cetus.

6.2. Posibles líneas futuras

Muchos autores argumentan que un aumento de la precisión o exactitud en el análisis puede no estar justificado en la mayoría de los casos a tratar, por lo que defienden más un análisis escalable o bajo demanda, donde podamos aplicar según el caso un análisis más exhaustivo o no. El coste de nuestro análisis no nos parece tan elevado teniendo en cuenta que en programas irregulares basados en estructuras dinámicas, una falta de precisión puede llevar a resultados demasiado difusos, por lo que las herramientas de análisis no son capaces de llegar a resultados concretos y satisfactorios. Esto se debe a que cuando tratamos con punteros dirigidos al *heap* o basados en el *heap* partimos de cierta incertidumbre, con lo que para llevar a cabo un análisis eficiente debemos trabajar con la información exacta. Sin embargo, hay que procurar no sobrepasar el límite en el que un mayor tiempo de análisis no aporta más nivel de detalle en la información obtenida. Vislumbramos varias posibles direcciones futuras en las que mejorar este trabajo, o abrir nuevas líneas de investigación.

- Aumentar el número de experimentos con códigos reales más grandes para probar las distintas herramientas y sus límites computacionales y de eficiencia. Este estudio podría ayudarnos a determinar cuándo se pueden simplificar las técnicas de análisis o cuándo es necesario aplicarlas con la máxima precisión posible. El objetivo sería diseñar un análisis bajo demanda, que en función de las características del código, ajustaría paramétricamente la precisión de nuestras herramientas DU y de análisis de forma.

- Extender la técnica DU para el tratamiento de arrays de punteros. Esta limitación también existe en otros trabajos relacionados, exigiendo un nivel más de complejidad en el algoritmo propuesto que exigiría quizás un ajuste completo.

- Diseñar un pase automático en la plataforma CETUS para la generación de código paralelo a partir de los resultados del test de dependencias. En este sentido se han realizado algunas pruebas experimentales preliminales con dos conocidas técnicas de paralelización para códigos irregulares, que son el objeto de nuestro estudio. Nuestra idea sería seleccionar la técnica más eficiente dados los casos tratados para mejorarla con una propuesta.

Bibliografía

[1] Hind M. Analysis: Haven't we solved this problem yet? In *ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (ACM'01)*, Snowbird, Utah, United States, 2001.

[2] Ryder B. G., Landi W. A., Stocks P. A., Zhang S., and Altucher R. A schema for interprocedural modification side-effect analysis with pointer aliasing. 23(1):105–186, 3 2001.

[3] Deutsch A. Interprocedural may-alias analysis for pointers: beyond k-limiting. In *PLDI '94: Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation*, pages 230–241, New York, NY, USA, 1994. ACM.

[4] Liang D and Harrold M.J. Lecture notes in computer science. pages 279–298, 2001.

[5] Shapiro M. and Horwitz S. The effects of the precision of pointer analysis. In *Lecture Notes in Computer Science*, volume 1302, pages 16–34, 1997.

[6] K. Jens. Context-sensitive matters, but context does not. In *Fourth IEEE International Workshop on Source Code Analysis and Manipulation.*, pages 29–35, 2004.

[7] Emami M., Ghiya R., and Hendren L. J. Context-sensitive interprocedural points-to analysis in the presence of function pointers. *SIGPLAN Not.*, 29(6):242–256, 1994.

[8] Wilson R.P. and Lam M.S. Efficient context-sensitive pointer analysis for C programs. In *Proc. ACM SIGPLAN'95 Conference on Programming Language Design and Implementation*, pages 1–12, La Jolla, California, June 1995.

[9] Sridharan M. and Bodik R. Refinement-based context-sensitive points-to analysis for java. *SIGPLAN Not.*, 41(6):387–400, 2006.

[10] Lattner C. and Adve V. Data structure analysis: A fast and scalable context-sensitive heap analysis, 2003.

[11] Nystrom E. M., Kim H.-S., and Hwu Wen mei W. Importance of heap specialization in pointer analysis. In *PASTE '04: Proceedings of the 5th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 43–48, New York, NY, USA, 2004. ACM.

[12] Harrold M.J. and Soffa M.L. Efficient computation of interprocedural definition-use chains. *ACM Trans. Program. Lang. Syst.*, 16(2):175–204, 1994.

[13] Landi W. and Ryder B. G. A safe approximate algorithm for interprocedural pointer aliasing. *SIGPLAN Not.*, 39(4):473–489, 2004.

[14] Harrold M.J. and Soffa M.L. Computation of interprocedural definition and use dependencies. *International Conference on Computer Languages*, pages 297–306, 1990.

[15] Johnson R. and Pingali K. Dependence-based program analysis. In *PLDI '93: Proceedings of the ACM SIGPLAN 1993 conference on Programming language design and implementation*, pages 78–89, New York, NY, USA, 1993. ACM.

[16] Stoltz E., Gerlek M.P., and Wolfe M. Extended ssa with factored use-def chains to support optimization and parallelism. *Software Technology, Proc. of the 27th Hawaii International Conference on System Sciences*, 2:43–52, 1994.

[17] Abdurazik A. and Offutt J. Using uml collaboration diagrams for static checking and test generation. In *Lecture Notes in Computer Science*, pages 383–395, 2000.

[18] Souter A.L. and Pollock L.L. The construction of contextual def-use associations for object-oriented systems. In *IEEE Transactions on Software Engineering*, volume 29-11, pages 1005– 1018, Nov 2003.

[19] Pande H. D. and Landi W. Interprocedural def-use associations in c programs. In *TAV4: Proceedings of the symposium on Testing, analysis, and verification*, pages 139–153, New York, NY, USA, 1991. ACM.

[20] Tok T.B., Guyer S.Z., and Lin C. Efficient flow-sensitive interprocedural data-flow analysis in the presence of pointers. pages 17–31, 2006.

[21] Yuan-Shin H. Interprocedural definition-use chains of dynamic recursive data structures. In *PhD. thesis, Faculty of the Graduate School of the Univ. of Maryland*, 1998.

[22] Cheng B.-C. and Hwu W.-M. W. Modular interprocedural pointer analysis using access paths: design, implementation, and evaluation. In *PLDI '00: Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*, pages 57–69, New York, NY, USA, 2000. ACM.

[23] Cytron R., Ferrante J., Rosen B.K., Wegman M., and Zadeck F.K. Efficiently computing static single assignment form and the control dependence graph. In *ACM Transactions on Programming Languages and Systems (ACM'91)*, pages 13(4): 451–490, October 1991.

[24] Gianfranco Bilardi and Keshav Pingali. Algorithms for computing the static single assignment form. *J. ACM*, 50(3):375–425, 2003.

[25] Tu P. and Padua D. Gated ssa-based demand-driven symbolic analysis for parallelizing compilers. In *ICS '95: Proceedings of the 9th international conference on Supercomputing*, pages 414–423, New York, NY, USA, 1995. ACM.

[26] Lapkowski C. and Hendren L.J. Extended ssa numbering: Introducing ssa properties to languages with multi-level pointers. In *ACAPS Tech. Memo 102, Sch. of Comp. Sci., McGill U*, pages 128–143. Springer-Verlag, 1998.

[27] Menon V. S., Glew N., Murphy B. R., McCreight A., Shpeisman T., Adl-Tabatabai A.-R., and Petersen L. A verifiable ssa program representation for aggressive compiler optimization. *SIGPLAN Not.*, 41(1):397–408, 2006.

[28] Castillo R., Corbera F., Navarro A., Asenjo R., and Zapata E.L. Interprocedural def-use chains for pointer-based codes optimizations. In *Advanced Computer Architecture and Compilation for Embedded Systems (ACACES 2007)*, July 2007.

[29] Castillo R., Corbera F., Navarro A., Asenjo R., and Zapata E.L. Complete defuse analysis in recursive programs with dynamic data structures. In *Workshop on Productivity and Performance (PROPER 2008) Tools for HPC Application Development*, Las Palmas de Gran Canaria(Spain), August 2008.

[30] Hwang Y.S. and Saltz J. Identifying DEF/USE information of statements that construct and traverse dynamic recursive data strucutres. In *Lecture Notes in Computer Science*, volume 1366 of *Languages and Compilers for Parallel Computing (LCPC'97 Issue)*, pages 131–145. 1998.

[31] Castillo R., Corbera F., Navarro A., Asenjo R., and Zapata E.L. Conflict analysis for heap-based data dependence detection. In *Parallel Computing 2009 (ParCo'09)*, Sept 2009.

[32] A. Tineo, F. Corbera, A.Ñavarro, R. Asenjo, and E.L. Zapata. A new strategy for shape analysis based on Coexistent Link Sets. In *Parallel Computing 2005 (ParCo'05)*, Sept 2005.

[33] Asenjo R., Castillo R., Corbera F., Navarro A., Tineo A., and Zapata E.L. Parallelizing irregular C codes assisted by interprocedural shape analysis. In *22nd IEEE International Parallel & Distributed Processing Symposium (IPDPS'08)*, Florida, USA, April 2008.

[34] Rakesh Ghiya and Laurie J. Hendren. Is it a tree, a DAG, or a cyclic graph? A shape analysis for heap-directed pointers in C. In *Conference Record of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, St. Petersburg, Florida, January 1996.

[35] Castillo R., Corbera F., Navarro A., Asenjo R., and Zapata E.L. Parallelization of dynamic data structures in pointer-based programs. In *Transnational Access Meeting, HPC EUROPA EVENT*, Bologna, Italy, June 2007.

[36] Castillo R., Tineo A., Corbera F., Navarro A., Asenjo R., and Zapata E.L. Towards a versatile pointer analysis framework. In *European Conference on Parallel Computing (EURO-PAR)2006*, 29th August - 1st September 2006.

[37] Navarro A., Corbera F., Asenjo R., Tineo A., Plata O., and Zapata E.L. A new dependence test based on shape analysis for pointer-based codes. In *The 17th International Workshop on Languages and Compilers for Parallel Computing (LCPC '04)*, West Lafayette, IN, USA, September 2004.

[38] Ghiya R. and Hendren L.J. Putting pointer analysis to work. In *Proc. 25th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 121–133, San Diego, California, January 1998.

[39] Hwang Y.S. and Saltz J. Identifying parallelism in programs with cyclic graphs. *Journal of Parallel and Distributed Computing*, 63(3):337–355, 2003.

[40] Mark Marron, Darko Stefanovic, Manuel Hermenegildo, and Deepak Kapur. Heap analysis in the presence of collection libraries. In *7th ACM Workshop on Program Analysis for Software Tools and Engineering (PASTE'07)*, San Diego, June 2007.

[41] Hendren L.J. and Nicolau A. Parallelizing programs with recursive data structures. *IEEE Transactions on Parallel and Distributed Systems*, 1:35–47, January 1990.

[42] Brian Hackett and Radu Rugina. Region-based shape analysis with tracked locations. *SIGPLAN Not.*, 40(1):310–323, 2005.

[43] Sang-Ik Lee, Troy A. Johnson, and Rudolf Eigenmann. Cetus - an extensible compiler infrastructure for source-to-source transformation. In *The 16th International Workshop on Languages and Compilers for Parallel Computing (LCPC '03)*, pages 539–553, College Station, Texas, USA, October 2003.

[44] T.A.Johnson, S.I.Lee, L.Fei, A.Basumallik, G.Upadhyaya, R.Eigenmann, and S.P.Midkiff. Experiences in using Cetus for source-to-source transformations. In *The 17th International Workshop on Languages and Compilers for Parallel Computing (LCPC '04)*, September 2004.

[45] Castillo R., Corbera F., Navarro A., Asenjo R., and Zapata E.L. Pointer analysis techniques targeted to accelerate shape analysis. In *XVII Jornadas de Paralelismo*, pages 603–308, Albacete, Spain, 2006.

[46] Reps T., Horwitz S., and Sagiv M. Precise interprocedural dataflow analysis via graph reachability. In *POPL '95: Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 49–61, New York, NY, USA, 1995. ACM.

[47] M.C. Carlisle and A. Rogers. Software caching and computation migration in olden. In *ACM Symposium on Principles and Practice of Parallel Programming (PPoPP)*, July 1995.

[48] J. Plevyak, A. Chien, and V. Karamcheti. Analysis of dynamic structures for efficient parallel execution. In *Int'l Workshop on Languages and Compilers for Parallel Computing (LCPC'93)*, 1993.

[49] M. Sagiv, T. Reps, and R. Wilhelm. Solving shape-analysis problems in languages with destructive updating. *ACM Transactions on Programming Languages and Systems*, 20(1):1–50, January 1998.

[50] Weiser M. Program slicing. In *ICSE '81: Proceedings of the 5th international conference on Software engineering*, pages 439–449, Piscataway, NJ, USA, 1981. IEEE Press.

[51] Bogdan K. and Janusz L. Dynamic slicing of computer programs. *J. Syst. Softw.*, 13(3):187–195, 1990.

[52] Jens K. Context-sensitive slicing of concurrent programs. In *ESEC/FSE-11: Proceedings of the 9th European software engineering conference held jointly with 11th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 178–187, New York, NY, USA, 2003. ACM.

[53] Anna B., Wlodzimierz B., and Pierluigi San P. Extracting coarse-grained parallelism in program loops with the slicing framework. In *ISPDC '07: Proceedings of the Sixth International Symposium on Parallel and Distributed Computing*, page 29, Washington, DC, USA, 2007. IEEE Computer Society.

[54] De Lucia A. Program slicing: Methods and applications. In *Source Code Analysis and Manipulation. Proc. 1st IEEE International Workshop*, 2001.

[55] Binkley D. and Harman M. A survey of empirical results on program slicing. *Advances in Computers*, 62:105–178, 2004.

[56] Mock M., Atkinson D. C., Chambers C., and Eggers S. J. Improving program slicing with dynamic points-to data. *SIGSOFT Softw. Eng. Notes*, 27(6):71–80, 2002.

[57] Mock M., Atkinson D.C., Chambers C., and Eggers S.J. Program slicing with dynamic points-to sets. *IEEE Transactions on Software Engineering*, 31(8):657–678, 2005.

[58] Hortwitz S., Pfeiffer P., and Repps T. Dependence analysis for pointer variables. In *Proc. ACM SIGPLAN'89 Conference on Programming Language Design and Implementation)*, pages 28–40, July 1989.

[59] Larus J.R. and Hilfinger P.N. Detecting conflicts between structure accesses. In *Proc. ACM SIGPLAN'88 Conference on Programming Language Design and Implementation)*, pages 21–34, July 1988.

[60] Hummel J., Hendren L.J., and Nicolau A. A general data dependence test for dynamic, pointer-based data structures. In *Proc. ACM SIGPLAN'94 Conference on Programming Language Design and Implementation)*, pages 218–229, June 1994.

[61] Ghiya R., Hendren L.J., and Zhu Y. Detecting parallelism in C programs with recursive data structures. In *Proc. 1998 International Conference on Compiler Construction*, pages 159–173, March 1998.

[62] Hwang Y.S. and Saltz J. Identifying parallelism in programs with cyclic graphs. In *Proc. 2000 International Conference on Parallel Processing*, pages 201–208, Toronto, Canada, August 2000.

[63] Navarro A., Corbera F., Asenjo R., Tineo A., Plata O., and Zapata E.L. A new dependence test based on shape analysis for pointer-based codes. *LCPC 2004 - LNCS*, 3602:394–408, May 2005.

[64] van Engelen R. A., Birch J., Shou Y., Walsh B., and Gallivan Kyle A. A unified framework for nonlinear dependence testing and symbolic analysis. In *ICS '04: Proceedings of the 18th annual international conference on Supercomputing*, pages 106–115, New York, NY, USA, 2004. ACM.

[65] Luk C.-K. and Mowry T. C. Compiler-based prefetching for recursive data structures. *SIGPLAN Not.*, 31(9):222–233, 1996.

[66] Shou Y., van Engelen R. A., Birch J., and Gallivan K. A. Toward efficient flow-sensitive induction variable analysis and dependence testing for loop optimization. In *ACM-SE 44: Proceedings of the 44th annual Southeast regional conference*, pages 1–6, New York, NY, USA, 2006. ACM.

[67] Gerlek M. P., Stoltz E., and Wolfe M. Beyond induction variables: detecting and classifying sequences using a demand-driven ssa form. *ACM Trans. Program. Lang. Syst.*, 17(1):85–122, 1995.

[68] A. Tineo. Compilation techniques based on shape analysis for pointer-based programs. In *PhD. thesis, E.T.S.I.Informática of the Univ. of Malaga, Spain*, 2009.