Department of Computer Architecture
University of Málaga

PH.D. THESIS

Compilation techniques based on shape analysis for pointer-based programs

Adrian Tineo

Málaga, November 2008

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 *"Compilation techniques based on shape analysis for pointer-based programs"*, ha sido realizada por D. Adrián Tineo Cabello bajo nuestra dirección en el Departamento de Arquitectura de Computadores de la Universidad de Málaga y concluye la Tesis que presenta para optar al grado de Doctor en Ingeniería de Telecomunicación.

Málaga, 1 de Septiembre de 2008

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

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

To the memory of my grandmother Adriana

A la memoria de mi abuela Adriana

Acknowledgements

There is a lot of people that I would like to thank for contributing to the present work. I feel lucky for having enjoyed their support and help.

First of all, I would like to thank my supervisors Dr. Rafael Asenjo and Dr. Angeles Navarro. In equal measure, I must thank Dr. Francisco Corbera. All of them have been indispensable for the fulfillment of this dissertation. About Rafa, I would like to stress his neverending enthusiasm. He was always optimistic about the outcome, even when I feeling down. He is always sure about the right motivation for the work and the direction where it should be heading. About Angeles, I would like to highlight her attention to detail in all aspects of the research. She was usually the one with a sharper view of affairs in our group discussions. About Francisco, I would like to point out his unquenchable capacity for devising solutions. No matter how odd or complicated a problem, I was constantly amazed at his ability to come up with a solution, or if that would not be, for a step in the right direction. Angeles and Francisco have also helped me substantially with the hardest technical issues in this dissertation.

I would also like to thank Emilio L. Zapata for all the management and for accepting me in the Department of Computer Architecture, a place where I have felt at ease during the making of this dissertation. I must also thank Carmen Donoso, always willing to help. All other colleagues in this department also deserve mention for their conversations, support, and camaraderie: Oscar, Eladio, Felipe, Mario, Gerardo, Julián, Javi, Sonia, Manuel, Mari Carmen, Ricardo,... the list goes on and on.

I would like to thank the support of projects TIC2003-06623 and TIN2006-01078 of the Ministry of Education of Spain, as well as the HPC-Europa transnational programme and its partner center EPCC.

I would like to thank professor Marcelo Cintra, for being a valuable host during my stay in Edinburgh in 2006. I value the conversations that we shared, I learned a lot from them. I want to mention professor Mike O'Boyle as well, a charismatic character full of interesting conversations both about work and life in general. I also want to thank Dr. Diego Llanos for his good disposition and for sharing his knowledge.

Also, from my period in Edinburgh, I would like to thank all the nice people that I met there, that helped me or encouraged me in some way or another: Catherine Inglis, Mark Bull, Chris Fench, Carla Delgado, Sergio Pérez, Miguel, Piotr, Marta, Rumi, Cande, Carlos, Ulf, Paco,... the list is way too long. Also, I would like to thank James Connachan for instructing me into a new way of physical education.

During my time in Spain, I have enjoyed the company and interaction with more Ph.D. students of my generation. Countless times we shared ideas or simply chatted for some mental relief. Not all of them continue now, but nevertheless I remember them fondly: Jose Miguel, Antoliano, Pepe, Fede, Ale, Sergio R., Vicky, Maxi, Javi, Fran, Sergio V., Migue, Antonio, Juan, Ricardo, Siham, and Marina.

On a more personal side, there are many people who have helped me throughout. Their company and support has been most soothing when things were not turning out nicely. First and foremost, I want to thank my parents. Your limitless faith and caring support is calming and refreshing. I hope I can make you proud with this dissertation.

My family has also helped me find the required balance of spirit: Ana Mari, Rafael, Javier, Cristina,

Nacho, Dani, Mari Carmen, Jorge, Andrés, Loli, María José, Javi, Yolanda and Alberto. Regarding my grandma, Adriana, recently deceased, I dedicate this dissertation to her.

The Antoni@s group also deserve special mention. It is great that we can keep in touch after the degree, and share experiences about growing older and facing life's multifaceted events: Desi, Juan, Caro T., Álvaro, Caro Co., Irene, Eva, Cristina, Tati, Amabel, and Vanessa. I hope we can keep in touch many years.

I would like to mention two great sources of inspiration in my life. Because of their knowledge and disposition to share it, I am in debt with Dr. Yang, Jwing-Ming and Dan Docherty.

Another source of inspiration for me is my friend Carlos Suarez, and his unconditional love for music and resistance to go through the hardest times.

The YMAA group has become my second family: Paco, Toni, Jose, Juan, Marcela, Jacinto, Rafa, Pilar, Ángel and María José. I am also lucky to have found yet another family in the wudang tradition with Steve from Madrid, Steve from Valencia, Lola, Miguel, Raquel, Raquelita, Rosa y Ramón.

Finally, I would like to close this acknowledgment section mentioning the most important person in my life. Partner in the broadest sense of the word, she is the one that makes me wake up every morning with a smile and a willingness to outperform myself in every way. Rosa, from all the people in the world, you have helped me the most to reach this place in time and space. All I can hope for is to keep on growing beside you.

Agradecimientos

Hay mucha gente a la que me gustaría agradecer su contribución al presente trabajo. Me siento afortunado de haber disfrutado de su apoyo y ayuda.

En primer lugar, quiero agradecer a mis directores de tesis, Rafael Asenjo y Maria Ángeles Navarro. En igual medida, debo agradecer a Francisco Corbera. Todos ellos han sido indispensables para la realización de esta tesis. Sobre Rafa, me gustaría destacar su inagotable entusiasmo. Siempre se ha mostrado optimista sobre el resultado, incluso cuando yo no me sentía muy animado. Siempre está seguro de la motivación adecuada para el trabajo y la dirección en la que debería encaminarse. Sobre María Ángeles, me gustaría resaltar su cuidadosa atención a todos los aspectos de la investigación. En nuestras discusiones de grupo, normalmente era ella la que tenía una visión más concreta del problema. Acerca de Francisco, me gustaría apuntar su infinita capacidad para idear soluciones. No importa lo complicado o extraño del problema, me quedaba constantemente maravillado de su habilidad para encontrar una solución, o si no era posible, un paso en la dirección adecuada. María Ángeles y Francisco también me han ayudado significativamente con la parte más técnica de esta tesis.

También me gustaría agradecer a Emilio L. Zapata, por toda la gestión y por aceptarme en el Departamento de Arquitectura de Computadores, un lugar donde me he sentido a gusto durante la realización de esta tesis. Tengo que agradecer también a Carmen Donoso, siempre de buen talante para ayudarme. Todo los demás compañeros del departamento también merecen mención por sus conversaciones, ayuda y camaradería: Oscar, Eladio, Felipe, Mario, Gerardo, Julián, Javi, Sonia, Manuel, Mari Carmen, Ricardo,... la lista sigue y sigue.

Me gustaría también agradecer el apoyo de los proyectos TIC2003-06623 y TIN2006-01078 del Ministerio de Educación español, así como el programa transnacional HPC-Europa y su centro asociado EPCC.

Asimismo, quiero agradecer a Marcelo Cintra, por ser un valioso anfitrión durante mi estancia en Edimburgo en 2006. Aprendí mucho de las conversaciones con él. Quiero mencionar también a Mike O'Boyle, un personaje carismático lleno de interesantes conversaciones tanto de trabajo como de la vida en general. También quiero agradecer a Diego Llanos su buena disposición y por compartir su conocimiento.

De mi periodo en Edimburgo, me gustaría agradecer a todo el conjunto de personas tan agradables que conocí allí, y que me de un modo u otro me ayudaron o apoyaron: Catherine Inglis, Mark Bull, Chris Fench, Carla Delgado, Sergio Pérez, Miguel, Piotr, Marta, Rumi, Cande, Carlos, Ulf, Paco,... la lista es demasiado larga. También me gustaría agradecer a James Connachan el instruirme en una disciplina nueva para mí.

Durante mi periodo en España, he disfrutado de la compañía y la interacción con otros estudiantes de tesis de mi generación. En incontables ocasiones hemos compartido ideas o simplemente charlado para aliviar un poco la mente. No todos continúan actualmente, pero aún así los recuerdo con cariño: Jose Miguel, Antoliano, Pepe, Fede, Ale, Sergio R., Vicky, Maxi, Javi, Fran, Sergio V., Migue, Antonio, Juan, Ricardo, Siham y Marina.

En un plano más personal, hay mucha gente que me ha ayudado a lo largo del proceso. Su compañía y ayuda ha sido un bálsamo cuando las cosas no salían adecuadamente. En primer lugar, y sobre todo, quiero

agradecer a mis padres. Vuestra fe ilimitada y atento cariño son refrescantes y tranquilizadores. Espero que os sintáis orgullosos con esta tesis.

Mi familia también me ha ayudado a encontrar el adecuado equilibrio mental: Ana Mari, Rafael, Javier, Cristina, Nacho, Dani, Mari Carmen, Jorge, Andrés, Loli, María José, Javi, Yolanda y Alberto. En cuanto a mi abuela Adriana, recientemente fallecida, a ella le dedico esta tesis.

El grupo de las Antoni@s también merece una mención especial. Es fantástico que podamos seguir en contacto después de la carrera, y compartir experiencias sobre hacerse mayor y afrontar los múltiples sucesos de la vida: Desi, Juan, Caro T., Álvaro, Caro Co., Irene, Eva, Cristina, Tati, Amabel y Vanessa. Espero que sigamos en contacto muchos años.

Me gustaría mencionar a dos grandes fuentes de inspiración en mi vida. Por su conocimiento y su disposición a compartirlo, me siento en deuda con el Dr. Yang, Jwing-Ming y Dan Docherty.

Otra fuente de inspiración para mi es mi amigo Carlos Suárez, y su incondicional amor por la música y su resistencia para soportar los momentos más duros.

El grupo de la YMAA se ha convertido en mi segunda familia: Paco, Toni, Jose, Juan, Marcela, Jacinto, Rafa, Pilar, Ángel y María José. También soy afortunado de haber encontrado otra familia más en la tradición wudang con Steve de Madrid, Steve de Valencia, Lola, Miguel, Raquel, Raquelita, Rosa y Ramón.

Finalmente, me gustaría acabar esta sección de agradecimientos mencionando a la persona más importante en mi vida. Compañera en el sentido más amplio del término, ella es la razón por la que me levanto cada día con una sonrisa y con la voluntad de mejorarme a mi mismo. Rosa, de todas las personas del mundo, tú eres la que más me ha ayudado a llegar a este lugar en el tiempo y el espacio. Todo a lo que puedo aspirar es a seguir creciendo junto a ti.

Index

List of Figures

List of Tables

Preface

Parallel computing is nothing new. It has been around for many years now. However, the way we perceive it, as a society, has changed. From its beginning of nearly science fiction promise, to the current state of ubiquitous multiprocessing, the parallel architectures and parallel programming paradigms have been many. As eloquent professor Lawrence Rauchwerger pointed us in the HIPEAC summer school of 2007, most of them have died. There has also been a cycle. In the 80's, everything sounding parallel was fashionable. In the 90's it was something different, as was remembered in a special panel in LCPC'07 entitled *"What have we learned after 20 LCPCs?"*. Funding was cut-off, and the researchers devoting their time to parallel computing were left with a dilemma: change topic to move to the warmer sun of funding, or stick to their guns in the best possible way and wait for a second rise of the parallelism fever.

It seems that those who waited were ahead of their time. Now, multiprocessors are everywhere, from high-end supercomputers to portable devices. There is a great promise for everyone working in parallel computing. At the same time, there is a great pressure on compilers. The time to provide quality parallel code is now. The hardware is ready to take it. The challenge is now for the compiler community to provide it.

Among the most daring challenges in parallel computing, there is the automatic parallelization of irregular applications. There are not so many people working on this. Some of the groups that succeeded in works tailored for regular applications, never turned to extend or adapt their approaches for irregular applications. I often wonder why. I suspect they thought the problem was too hard, or at least that it would not pay off in the short term. They decided to invest their time and resources in some other problem meaningful for the compiler scientific community.

In this Ph.D. dissertation we look toward one of the toughest of problems in parallel computing: the automatic parallelization of pointer-based applications with dynamic recursive data structures. The problem is so tough, that we deal solely with a technique to extract information from the heap to detect data dependences. We will leave for future work the task of generating parallel code based on the information provided by our approach.

It was just a few days after the 11th of March of 2004. Spain was still shocked due to the horrid terrorist attack of the trains in Madrid. A day we will never forget. People were still finding their way into processing the strong feelings of disgust, fright, political incredulity, and anger. My way of doing so was reading through the pages of the Ph.D. thesis of Dr. Francisco Corbera [1], the seed work for the present manuscript. In a weekend, amidst the national turmoil and overall confusion, I had to decide whether I took the position of Ph.D. student to continue work in the field of shape analysis for dynamic data structures or whether I continued my search for a job in the private sector, as was my first intention when I finished the engineering degree. Needless to say, I took up the opportunity. Today, the world is not a safer place. At least, I feel relieved that the goal I set myself that time comes to fruition.

At first, I had to get acquainted with all the nuances of such a complex analysis technique. Gradually, I grasped its meaning and extended the experimental work carried out so far. This produced some worthy publications ([2], [3], [4], [5]) and introduced me in the world of international conferences. This work

was invaluable for me to understand the complexity of shape analysis and its validity for data dependence detection of heap-induced data dependences.

Then, the goal was to redefine the shape analysis strategy from scratch to include the novel idea of storing links existing simultaneously over heap-allocated memory pieces in groups, a notion we baptised as *coexistent links sets* (`cls`). This concept is the single most defining characteristic of our approach and is covered in detail in chapter 2 of this dissertation. This key idea was elaborated and extended in [6], [7] and [8].

I call this period of the Ph.D. research *"the sweet start"*. I was enjoying most of what I was doing and all our efforts were corresponded with worthy publications, often with encouraging reviews. Next in my well-defined schedule, was to add interprocedural support to our new shape analysis technique based on `cls`'s. Chapter 3 of this dissertation details the mechanism involved for this extension. From the first research into related work, to the design, implementation, and tests, this would take me nearly 2 years, being by far the most challenging period in the making of this dissertation.

In between this period, I paid a 3 months visit to the School of Informatics in the University of Edinburgh, funded by the HPC-Europa programme. There, Dr. Diego Llanos, my host Dr. Marcelo Cintra and I designed a scheme for speculative parallelization of pointer-based applications. The resulting work, though preliminary, was featured in [9] and [10]. I call this period *"the happy exile"*. Not only it served me to deepen my understanding of parallel programming and recognize the pitfalls of exploiting parallelism in pointer-based programs, but I was also lucky to share my time in Scotland with a most pleasurable company of other young researchers from Europe. Insiders from the program told us that somehow we had formed one of the most animated groups among all visitors ever.

After coming back from Edinburgh, I resumed work in the interprocedural extensions for our shape analysis technique. I call this period *"the real world"*. Bruce Dickinson, singer and songwriter, put it quite simply: *"the real world can leave you hanging by a thread"*. It surely did that for me, as our efforts were systematically rejected in the ICS'07, PACT'07, SAS'07, and PPoPP'07 conferences. I tried (although sometimes it was hard) to keep my spirit high and continue working at full throttle despite the adversities. By the time we got a poster in ACACES'07 [11], another poster in LCPC'07 [12] and a full article in IPDPS'08 [13], we had already moved into the data dependence analysis, subject of chapter 4.

The present work rounds up with conclusions and ideas for future work in chapter 5. Then, follow some appendixes with some technical details, and the Spanish summary for this dissertation.

When doing research, you often find that your ideas are not really new. Hopefully, you can add a different point of view that leads to an improved result. Sometimes, what you need to solve was done just fine by people before you. As I close this preface, I find that what I feel was already described in 1987 by the drummer and songwriter, Neil Peart, when he wrote the opening verse for the song *"The Mission"*:

"Hold your fire,
keep it burning bright.
Hold the flame
'till the dream ignites.
A spirit with a vision
is a dream with a mission".

1 Introduction

1.1 General background

The scientific community agrees that we have reached the multicore era: dual- and quad-core processors are now common in desktop computers, and manufacturers like Intel target 80-cores processors in their roadmap. Furthermore, multiprocessor architectures abound in middle-size enterprises, research centers, and national organizations, as they are quickly becoming the mainstream in computer architecture.

Single-core architectures cannot cope with Moore's law [14] to increment performance. As we reach the limits of monoprocessor architectures, increasing power consumption (and the associated cooling cost) starts to outweigh the growingly smaller improvements in performance. There seems to be a better way: go multiprocessing. Examples of this trend are the Intel's Centrino Duo architecture for laptops, and IBM's Roadrunner system, the most powerful supercomputer in actuality[1]. The latter is the world's first hybrid supercomputer, connecting 6,562 dual-core AMD Opteron chips as well as 12,240 Cell chips. Not only multiprocessors are becoming widespread, but heterogeneous architectures are emerging too. Rather than a fad of our time, multiprocessors are here to stay.

The key issue now is obtaining good software performance at low cost so that we can exploit all available hardware. Currently, the most common way to obtain programs for these multiprocessor architectures is by explicitly writing parallel algorithms that rely on threading schemes or message passing libraries. There is a major hindrance with this approach, and it is the increasing development cost. Although there is a growing number of languages and libraries trying to popularize parallel programming (e.g., [15], [16], [17]), expert parallel programmers are invariably sought-after, more so due to the variety of available architectures and parallel programming paradigms.

We are witnessing a change of course: from the *High Performance Computing*, mainly concerned with reducing run times and achieving speedup by whatever means necessary, to the *High Productivity Computing*, which seeks to obtain attractive performance improvements, but more importantly, with shorter development costs.

[1] As indicated by the Top 500 Supercomputer list (www.top500.org) in June 2008.

For years, a versatile and powerful parallelizing compiler has been the chimera of the compiler community. The goal is to able to identify and exploit parallelism in sequential programs in an automated basis, a process entirely driven by the compiler. This approach has proven successful for regular applications, mainly in Fortran [18], but irregular applications still pose great challenges.

In particular, data structures based on dynamic memory and accessed through pointers are beyond the scope of current compilers. They are ineffective when it comes to optimizing pointer-based applications for modern multiprocessors. This limitation is mainly caused by their inability to extract the required information from the source code. In general, they are unable to locate the opportunities to exploit parallelism and locality in dynamic data structures. For that, a precise description of what heap-allocated memory locations are accessed and how they are managed is absolutely required. Only then we will be able to advance in the automatic parallelization of irregular programs.

1.2 Motivation

The problem we want to solve is the automatic parallelization of codes based on heap-stored dynamic, recursive data structures. This is a very tough and unsolved problem. Finding its solution would have a big impact since dynamic data structures are widely used in many irregular codes and multiprocessor/multicore architectures are very common nowadays.

Dynamic data structures are those allocated at run time and accessed through heap-directed pointers. More often than not, those structures are also recursive in the sense that each heap element has pointer fields that can point to other heap elements, thus forming common structures such as linked lists, Direct Acyclic Graphs (DAG) or trees. These structures are commonly used in pointer-based, irregular applications, and they pose significant challenges for current compiler analysis passes, due to the alias problem.

The problem of calculating pointer-induced aliases must be solved so that compilers can safely disambiguate memory references. A basic step in the automatic parallelization process is the detection of parallel loops or parallel function calls using a data dependence test. Such a dependence test requires information about the properties of the data structures traversed in the loops or in function bodies. We are convinced that, for the purpose of dependence analysis in the context of applications that deal with dynamic data structures, a very precise description of the heap must be obtained.

There is a body of work based on *points-to analysis* (e.g., [19], or [20]). Its main focus is toward detecting aliases relationships between pointers. For instance, Salcianu [21] builds points-to graphs that represent relationships between heap elements and pointers, even for incomplete parts of a program. His analysis is tested for some simple clients for object oriented languages, such as finding out methods that do not modify global objects (*purity analysis*), and detecting objects that are captured within a method and thus can be allocated in the stack (*stack allocation analysis*).

In our approach, we consider *shape analysis* as the base technique for achieving a characterization of data structures in the heap. Unlike points-to analysis techniques, which are mainly concerned on must- and may-alias sets, shape analysis is concerned about the *shape* of the data structure. This allows for a more precise characterization of data structures in the heap. Such precision is a must for more complex client analysis, such as data dependence analysis. With shape information, it it possible to identify conflicting and non-conflicting accesses in traversals of heap elements, that otherwise are not differentiated in a points-to analysis.

We envision a heap analysis framework, based on shape analysis as its cornerstone, whose purpose is to draw topological and temporal information about recursive data structures. Such a framework would be oriented toward the detection of parallel loops and parallel function calls. The final goal is to generate

threaded, parallel versions of sequential legacy codes. This framework would prove invaluable in the current scenario brimmed with off-the-shelf multiprocessors, which are becoming widely available to the average user in the form of multicore systems.

We present a gross view of such a framework in Fig. 1.1. The heap analysis framework statically derives information from a sequential pointer-based application. This framework should be coupled with a code transformation block that makes proper use of that information to yield an optimized version of the original program. For our purposes, such optimization is related to automatic parallelism for achieving run time speedup.

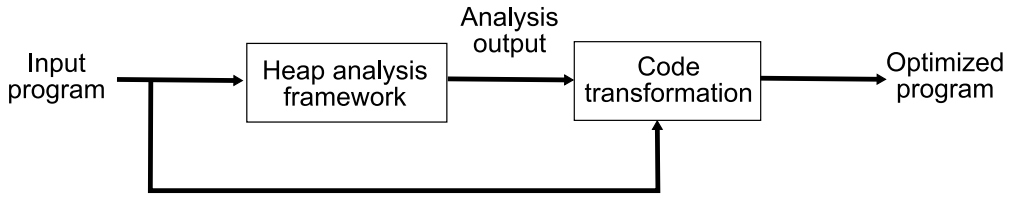

Figure 1.1: Heap analysis framework to report information to a code transformation block.

1.3 Shape analysis for dependence analysis

Shape analysis is a heap analysis technique that considers information available at compile-time to provide detailed information about the heap for pointer-based programs. This is done by extracting information about the *shape* or connectivity of heap elements.

The information derived from the shape analysis of a pointer-based application can be used for several purposes like: (i) data dependence analysis, by determining if two accesses may reach the same memory location; (ii) locality exploitation, by capturing the way memory locations are traversed to determine when they are likely to be contiguous in memory; (iii) program verification, to provide correctness guarantees about heap manipulating programs, and (iv) programmer support, to help detecting incorrect pointer usage or documenting complex data structures. In this dissertation, we will deal exclusively with the use of shape analysis for data dependence analysis, although other uses are possible.

In our approach to shape analysis, we use shape abstractions expressed as graphs to model the heap. Graph-based shape analysis is a very detailed pointer analysis technique that is regarded as context-, flow-, and field-sensitive. As a consequence, it is usually much more costly than other approaches to heap analysis, like points-to analysis.

Let us present now an intuitive idea of how a graph-based shape analysis can be used to find access conflict in a typical pointer-chasing loop. The main idea in our data dependence test scheme is to carry out the abstract interpretation of the statements of the analyzed loop, abstracting the accessed heap locations with nodes of shape graphs and annotating these nodes with read/write information.

The code in Fig. 1.2 creates a singly-linked list and then traverses it, copying the `data` field of the element pointed to by pointer `q`, to the element pointed to by pointer `p`. The overall effect of this algorithm is to shift values in the list one position toward the head. Note how there is a potential loop-carried data dependence, between `S3:val=q->data`, that reads the `data` field, and `S4:p->data=val`, that writes to it.

Our test symbolically executes the code abstracting the data structures in shape graphs. For example, $\texttt{sg}^1$ is the shape graph that abstracts the list created at statement `S1`. Using *abstract interpretation* [22], the *abstract semantics* of each statement update the shape graph resulting from the previous statement. In this

process, memory locations that are read and/or written are annotated accordingly. In this example, the read access of statement `S3` is annotated as `RS3` in the shape graphs, whereas the write access of statement `S4` is annotated as `WS4`. The second symbolic execution of statement `S4:p->data=val` produces shape graph sg^8. Within this shape graph we can detect that a memory location has been read in an iteration and written in the next one, causing a loop-carried dependence due to a write-after-read (WAR) access.

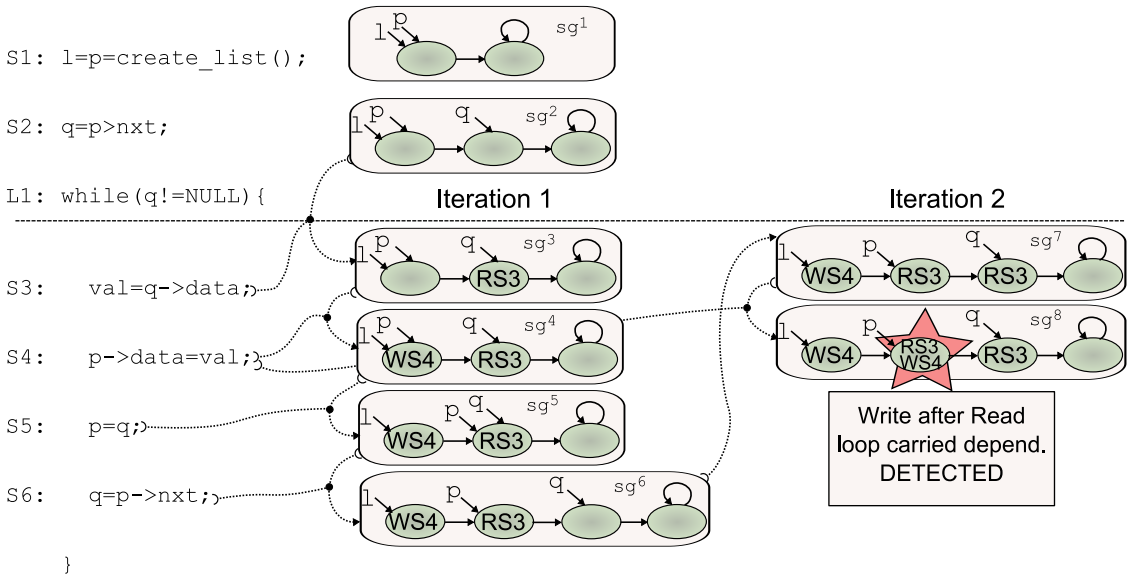

Figure 1.2: The use of shape analysis for heap-induced data dependence detection.

1.3.1 Shape analysis within the heap analysis framework

We expand now on the concept of the heap analysis framework introduced earlier. Fig. 1.3 presents a view of different modules interacting within the heap analysis framework.

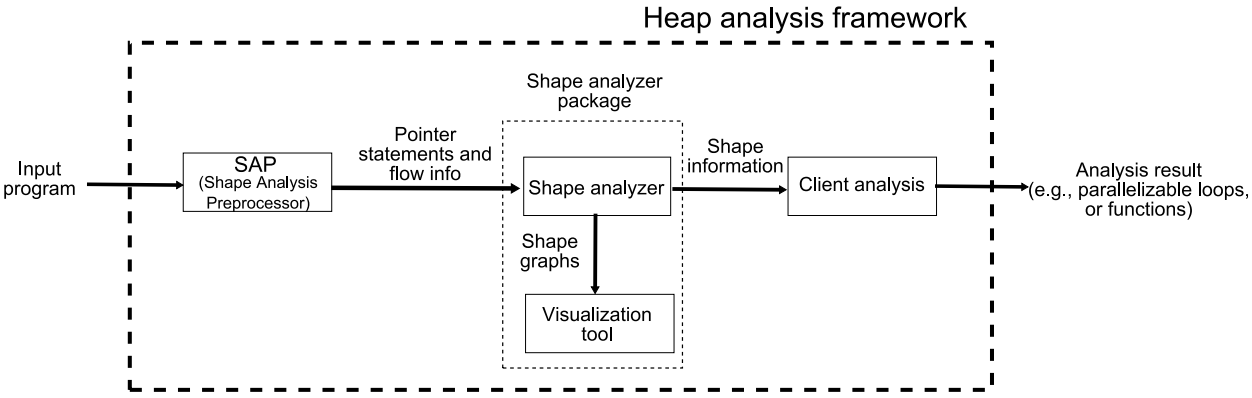

Figure 1.3: Program preprocessing, shape analysis and client analysis within the heap analysis framework.

First, the input program enters the SAP module, which stands for *Shape Analysis Preprocessor*. As it name suggests, it is responsible for performing preprocessing tasks on the program required for its shape analysis. The result of this module is the set of pointer statements that deal with the heap, and the flow information that governs the way those statements are executed in the program.

That information is the input for the *shape analyzer* tool, within the *shape analyzer package*. Also

within this package we find a *visualization tool* [23] that is used to visualize the shape graphs obtained and help debug the technique.

As a result of the execution of the shape analyzer, we will obtain a *shape characterization* of dynamic data structures. That information can be put to use by *client analyses*. For example, a *data dependence test* could be such a client: it can consider shape information combined with heap access information to detect dependences in pointer-based applications. The results of the client are then offered as output of the framework. For instance, a data dependence test client could report parallelizable loops or functions to a parallelization framework, external to the heap analysis framework.

We will deal with the *shape analyzer* in detail in chapters 2 and 3. We present a data dependence test as client analysis for the use of our shape analysis technique in chapter 4. Now, we will focus on the SAP module, and the preprocessing tasks required for proper shape analysis.

1.4 Preprocessing for shape analysis: the Cetus framework

Some preprocessing is required for the shape analysis of a program. We have designed the required preprocessing passes within the extendable Cetus framework [24]. Cetus is a compiler infrastructure designed for source-to-source program transformation. It can parse a program to a well-defined IR, perform some transformations, and emit the result as a new source program. This approach is useful for performing transformations in programs that can be later compiled and run with a production compiler. Cetus can parse C, C++, and Java, although we only target C programs.

Cetus is specially aimed toward the development of compilation passes of high-level nature. This is so because its IR is close to the source code, which is suitable for transformations related to heap analysis. Tackling heap analysis at lower levels in the compilation process is usually more difficult because we lack enough information.

Cetus is written in Java and its source code is publicly available under a non-restrictive license. This has allowed us to design some custom passes over its IR to perform some simple program transformations, annotations and translation. Furthermore, our extended Cetus framework fits seamlessly with our shape analyzer implementation, also designed in Java.

Fig. 1.4 displays the insides of the SAP module presented earlier. It is composed of several modules. First, the input program is parsed with Cetus. The result is the original program in the Cetus IR format, where we can perform some simple passes, structured in three stages: (i) simplification of complex statements, (ii) program instrumentation, and (iii) extraction of pointer statements and flow information. At some points in the process, the resulting state of the program can be emitted optionally as a new source code with the transformations applied so far.

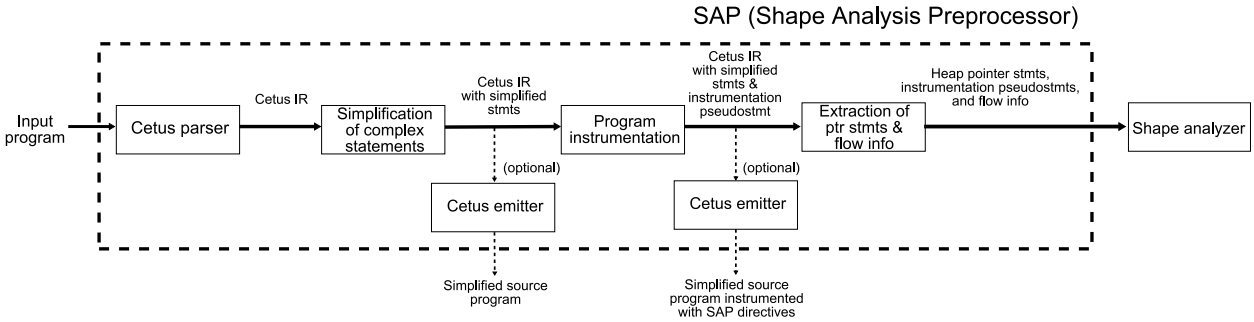

Figure 1.4: Modules of program preprocessing for shape analysis, designed within the Cetus infrastructure.

1.4.1 Simplification of complex statements

The first pass within the shape analysis preprocessing scheme is the simplification of complex statements. The purpose of this pass is obtaining an equivalent version of the input program but with a shorter variety of expressions, so that we can deal with the features of the language more effectively.

We target sequential C programs for our analysis. The variety and complexity of expressions that are legal in ANSI C [25] is significant. We do not support all the features of the language. In particular, we do not support pointer casting, pointer arithmetic, function pointers, or the address-of operator (&). In other words, the type of declared pointers must be known and fixed, and the access through pointers has to follow a pointer chasing path, with no calculation of pointers based on adding values over a base pointer address. Programs using this kind of mechanisms must be rewritten by hand to avoid them or discarded for their analysis.

We focus on a subset of C for our analysis: assigning statements involving pointers to dynamic data types, loops and branching statements, function calls and return statements. Additionally, we need to make sure in the preprocessing stage that statements involving pointers use only *simple access paths*, i.e., they only have one level of indirection. Also, we need to ensure that a pointer is not read and written in the same statement, that conditions in loops are simple pointer checks, and that there are no nested function calls.

These kinds of expressions are substituted for a simpler version by a custom pass designed in the Cetus infrastructure. An example of such expressions and their transformations can be seen in Table 1.1. Note that we employ additional pointer variables when needed. Since the number of assigned pointers in our analysis is relevant, as we shall see in the complexity study of chapter 2, we nullify these temporal pointer variables as soon as they are not needed.

Thanks to the simplification of complex statements performed in our preprocessing pass, the number of different statements that need to be supported by our shape analysis technique, and therefore their associated abstract semantics, is lower. This makes the formulation of our analysis simpler and involves no loss of generality.

Original statements	**Simplified statements**
Turn complex access paths into simple access paths	

`x=y->nxt->dwn;`	`tmp=y->nxt;`
	`x=tmp->dwn;`
	`tmp=NULL;`

Turn updating statements into two separate read and write statements	

`x=x->nxt;`	`tmp=x->nxt;`
	`x=tmp;`
	`tmp=NULL;`

Simplify complex conditions for branches, loops, and function calls	
`if(nav=x->nxt)`	`nav=x->nxt;` `if(nav!=NULL)`
Decompose nested function calls	

`foo1(foo2(x));`	`tmp=foo2(x);`
	`foo1(tmp);`
	`tmp=NULL;`

Table 1.1: Examples of complex expressions and the simplifications performed by the shape analysis preprocessing pass.

1.4.2 Program instrumentation

Shape analysis can benefit from certain information that can be deduced from the statements in the program. In that case, it is worthwhile to communicate that information to the shape analyzer tool for improved precision or functionality. The second pass in our shape analysis preprocessor module can annotate that information in the source code. Such annotations use the common `pragma` nomenclature, followed by the SAP keyword and the name of a directive.

The process is simple: the simplified representation of program statements in Cetus IR format is traversed, looking for opportunities to add SAP directives. The result can then be emitted as a new code thanks to the source-to-source translation capabilities of Cetus. This emitted code can be run with the same result as the original program, as the SAP directives will be ignored by the compiler.

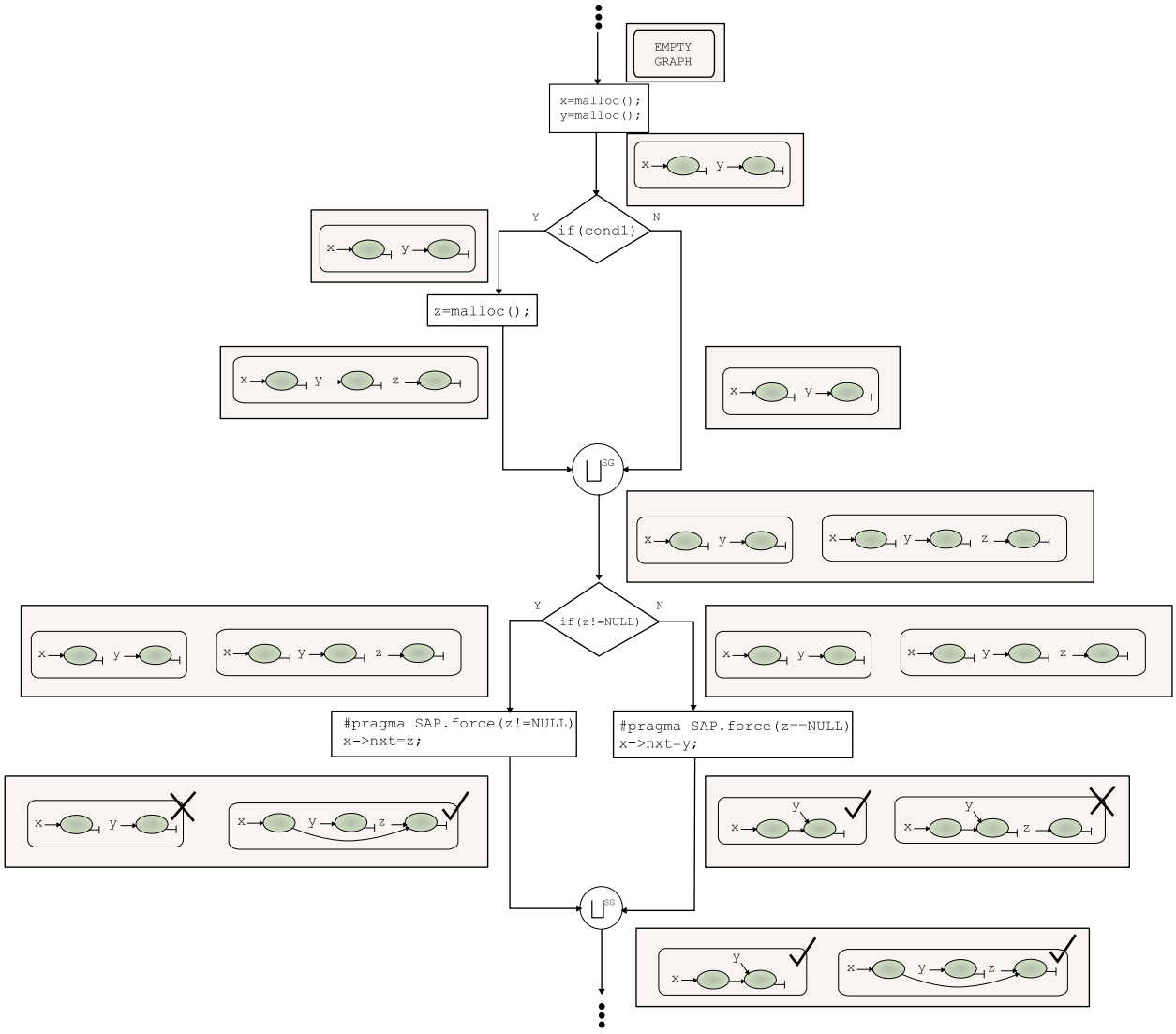

Figure 1.5: Example of the use of force pseudostatements to filter out unrealistic graphs.

Some directives provide information to the next stages of shape analysis preprocessing about the way the statements must be considered for the internal operations, but most directives are translated into *pseudostatements* that will be later abstractly interpreted by the shape analyzer for a specific effect. One of such pseudostatements is the *force pseudostatement*, indicated in a force directive of the form `#pragma`

`SAP.force(condition)`, where `condition` is a boolean condition involving pointers. The possible conditions are `x==NULL`, `x!=NULL`, `x->sel==NULL`, `x->sel!=NULL`, `x->sel==y`, or `x->sel!=y`, where `x` and `y` are pointers to recursive data types `t1` and `t2`, respectively, and `sel` is a pointer field (also called *selector*) to type `t2`, declared in `t1`.

The force pseudostatement is used to filter out unrealistic graphs according to pointer test conditions, mainly in branches and loops. Consider the example of Fig. 1.5. It shows a simple code as a control flow graph with two `if` branches. Let us assume that `x`, `y` and `z` are pointers to a list data type. The first conditional block creates a third element pointed to by `z` if a certain arithmetic condition holds. This condition is not known at compile-time, so both branches must be considered by the analysis. At the join point after the first `if`, there are two possibilities of graph abstractions. These then enter in the next conditional block. However, it is clear from its condition (`z!=NULL`) that only one graph is suitable for each branch. At the join point after the second `if`, we would have four different graphs, according to different flow paths of the analysis. Two of them are correct, the other two are impossible, and thus render the analysis imprecise. To prevent this situation, the `force(z!=NULL)` pseudostatement (inserted by the preprocessing directive `#pragma SAP.force(z!=NULL)`), filters out the graph where `z` is assigned to a node, and the `force(z==NULL)` pseudostatement (expressed in directive `#pragma SAP.force(z==NULL)`) filters out the graph where `z` is *not* assigned to any node. With the use of simple force pseudostatements, inappropriate graphs are eliminated, preserving accuracy in the analysis.

Force pseudostatements are just one of several pseudostatements available. Another kind is the *touch pseudostatement*, which is used to annotate access information in nodes of shape graphs. This is useful for data dependence analysis. For instance, the `RS3` and `WS4` labels from Fig. 1.2 are annotated by touch pseudostatements. The touch pseudostatement and other related pseudostatements will be covered in detail in chapter 4. The key issue here is that all of the pseudostatements needed in our approach are introduced in the same way, i.e., through SAP directives in the source code.

1.4.3 Extraction of pointer statements and flow information

The final stage of our preprocessing scheme for shape analysis involves the translation of the required statements from Cetus internal IR to the format required by the shape analyzer tool. This translation must account for all kinds of statements considered by the shape analysis technique.

The shape analyzer tool operates by modifying shape graphs according to the abstract semantics of the statements dealing with pointers (as well as the pseudostatements introduced in the instrumentation phase). The flow of the analysis is in turn based on the flow imposed by the function calls, loops, and branches in the program.

The statements considered for the shape analysis technique are displayed in Table 1.2. They are sorted as those required for intraprocedural analysis, interprocedural analysis and data dependence test, which is the order in which we will present the shape analysis capabilities in this dissertation, in chapters 2, 3, and 4, respectively.

1.5 Outline of this dissertation

The rest of this dissertation is organized in the following way:

- Chapter 2 explains in more detail our approach to shape analysis based on shape graphs. For this chapter, we focus on the design of the shape analysis for intraprocedural programs, i.e., without

Statements for intraprocedural analysis (chapter 2)	
Pointer nullification stmt.	`x=NULL;`
Heap element creation stmt.	`x=malloc(...);`
Heap element removal stmt.	`free(x);`
Pointer aliasing stmt.	`x=y;`
Selector nullification stmt.	`x->sel=NULL;`
Selector assignment stmt.	`x->sel=y;`
Traversing stmt.	`x=y->sel;`
Loop stmt.	e.g., `while(){...}`
Branch stmt.	e.g., `if(){...}`
Statements for interprocedural analysis (chapter 3)	
Function call stmt.	`x=foo(y,z,...);`
Return stmt.	`return(x);`
Function header	`foo(x,y,...){...}`
Statements for data dependence analysis (chapter 4)	
Heap read access to a data field	`val=x->data;`
Heap write access to a data field	`x->data=val;`

Table 1.2: Statements extracted by the shape analysis preprocessor for shape analysis. `x`, `y`, and `z` are pointers to recursive data types, `sel` is a pointer field (or selector) of recursive data type, `data` is a data field of recursive data type.

support for functions yet. In particular, the abstract semantics and data-flow equations needed for correct shape analysis of programs are explained.

- Chapter 3 deals with the extensions of the technique described in chapter 2 to provide full interprocedural support. In particular, we introduce the mechanisms required for shape analysis in recursive algorithms.

- Chapter 4 puts to use the technique completed in chapter 3 for the detection of heap-induced data dependences. We identify different patterns used to traverse dynamic data structures, and devise mechanisms to identify data dependences due to heap accesses in them.

- Chapter 5 discusses the main contributions of this dissertation and poses ideas for future work.

Related work and experimental results are discussed within the scope of each of these chapters. On top of that, we have deferred some specific content to the appendices for your reference, to avoid detracting from the overall readability. In particular, appendix A contains some algorithms for the shape analysis technique described in chapter2, appendix B contains detailed description of the shape graphs obtained after the analysis of a recursive function that reverses a singly-linked list, and appendix C contains a summary of this dissertation in Spanish, as partial fulfillment of the requirements for the mention of “Doctor Europeus”.

2 Intraprocedural shape analysis

2.1 Our approach to shape analysis

Our approach to shape analysis is based on constructing *shape graphs*. These are graphs made of three base elements: (i) *pointers*, in particular, those that point to recursive data types declared in the program; (ii) *nodes*, which represent dynamically allocated memory pieces; and (iii) *edges*, that connect pointers to nodes or nodes with other nodes. The purpose of a shape graph is to represent the main *shape* features of dynamic, recursive data structures. Such features allow to identify the structures as lists, or trees, for example, including information about the presence or absence of cycles, the kind of locations reachable from a pointer, and so forth.

The analysis works by symbolically executing the pointer statements in the analyzed program. Each pointer statement modifies an input shape graph to produce an output shape graph, in a way that accurately represents the effect of the statement at run time. For instance, Fig. 2.1 sketches how graphs change when analysing the pointer statements that create a singly-linked list. A malloc statement, such as `S1` and `S3`, produces the creation of a new node, which abstracts a memory piece allocated at run time. Statement `S4:p->nxt=a` is used to connect the node pointed to by `p` with the node pointed to by `a`. Aliasing statements, such as `S2` and `S5`, produce a pointer to be updated so that it points to the node pointed to by another pointer. Each kind of pointer statement has its associated behavior for the *shape graph domain*, that imitates or simulates the behavior of the statement execution at run time. The way in which a pointer statement modifies the shape graphs is defined by its *abstract semantics*. The process of actually modifying a shape graph according to those abstract semantics is called *abstract interpretation* [22].

Our shape analysis algorithm is designed as an iterative data-flow analysis. The statements in the program are symbolically executed in an iterative fashion, driven by the branches and loops in the program, for the intraprocedural part of the analysis. In this process the shape graphs are changed according to the abstract semantics of the statements analyzed. This process continues until the shape graphs reach a stationary state, where further abstract interpretation produces no new information. Such state is referred to as the algorithm *fixed point*.

Tightly related to the fixed point in our algorithm, is the notion of *summarization*. Summarization is the process that merges nodes in the shape graph when they are regarded as *similar enough* by the analysis.

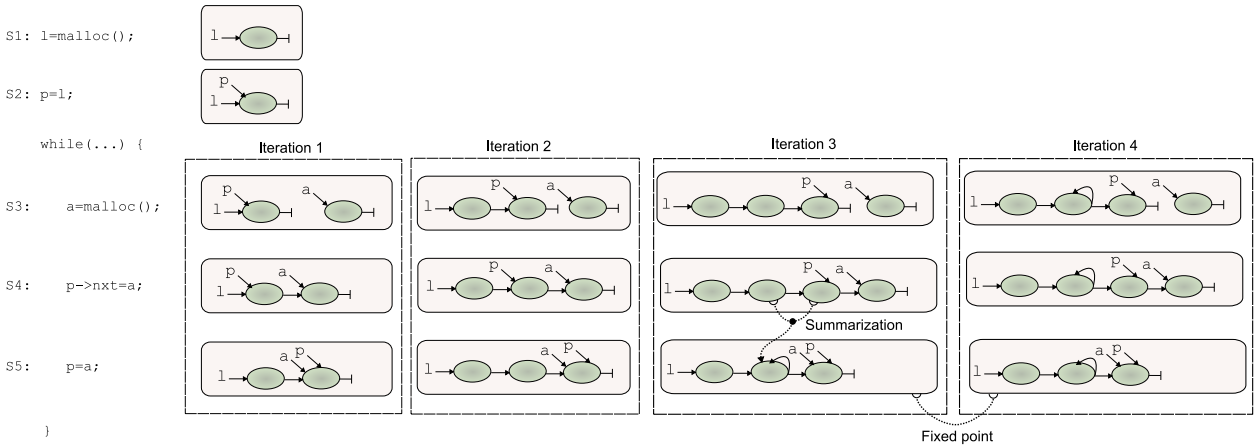

Figure 2.1: Analysing a loop until a fixed-point is reached in the graphs.

Similarity or *compatibility* of nodes is determined by pointer alias relationships and adjustable *properties*. The summarization process binds the shape graphs, by limiting the number of nodes they may have. Additionally, summarization prevents the graphs from changing endlessly in the course of iterative abstract interpretation, thus allowing the fixed point condition to be reached.

For instance, in Fig. 2.1, we find that summarization occurs at the processing of `S5:p=a` in symbolic iteration 3. This makes it possible for the analysis to reach a fixed point in the next symbolic iteration, where we obtain the same shape graph. New symbolic iterations of the loop would not produce new information, so the analysis can terminate.

Summarizing implies losing information in favor of a bounded representation. We provide as well a dual operation to focus over previously summarized nodes: *materialization*. This operation can regain precision where pointer accesses are occurring because it performs *strong update* [26] [27], discarding unnecessary links in most situations. However, highly connected and summarized graphs can make impossible for the materialization operation to recover exactly the original links, leaving some conservative ones.

The whole idea of summarization/materialization, is to fold/unfold the structure in the shape graph, depending on the part of the structure that is being accessed. The part of the structure that is being accessed by pointers becomes *focused* or *unfolded*, while the part of the structure that is not directly accessible by pointers becomes *summarized* or *folded*. Fig. 2.2(a) shows an example of summarization in a singly-linked list: when the `p` pointer is aliased with `a`, two nodes in the middle of the list are no longer directly accessible by pointers and are summarized into a so-called *summary node*. Conversely, Fig. 2.2(b) shows a traversal of a singly-linked list where a summary node is focused by materializing a new node, which represents precisely the memory location pointed to by pointer `p` in that moment of the program execution.

The basic criterion to merge nodes is therefore to summarize all the nodes that are not pointed to by pointers, and thus, are not directly accessible by them. However, we provide a configurable set of properties, which are valuable for fine-tuning summarization decisions in the cases where this basic criterion is insufficient to provide the requested accuracy. Properties are a key instrument to control how precisely shape graphs capture the features of the memory configuration.

At any point during the analysis, there may be several different shape graphs for a statement, to capture all possible memory configurations that can reach that statement from different flow paths. In fact, we associate not just a single shape graph, but a group of them to every statement and for every symbolic iteration. We call such a group a *reduced set of shape graphs* (RSSG). A RSSG can contain just one shape graph, and act as a *wrapper* for it within the analysis, but in general, a RSSG will contain several shape

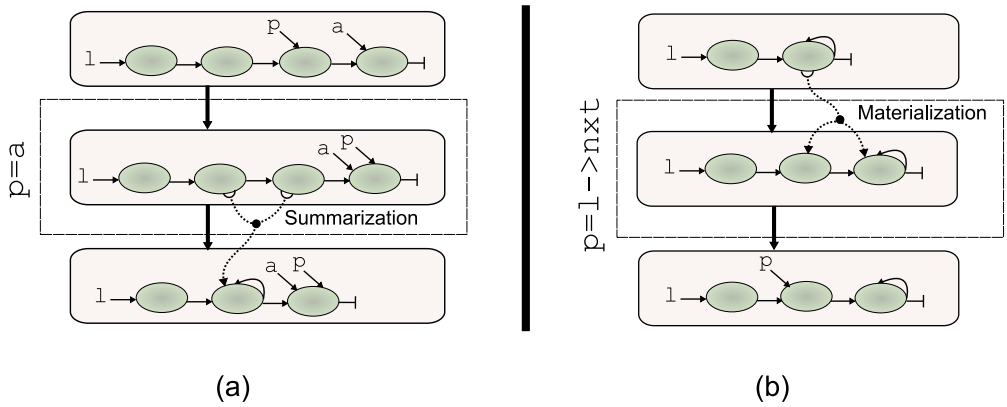

Figure 2.2: (a) Summarization allows to bind the structure; (b) materialization is used to focus on the regions currently accessed.

graphs that are regarded as *different enough*. That is the case of mutually exclusive pointer arrangements in the shape graphs. Conversely, similar or *compatible* graphs in a RSSG are *joined* to bind the number of graphs within a RSSG.

Fig. 2.3 shows the shape graphs generated during the analysis of a piece of code with three branches. Reduced sets of shape graphs from $\mathtt{RSSG}^0$ to $\mathtt{RSSG}^8$, just serve as wrappers for shape graphs $\mathtt{sg}^0$ to $\mathtt{sg}^8$. At the join point in the CFG though, the temporal $\mathtt{RSSG}^{9'}$ gathers the shape graphs resulting from the three different branches. Shape graphs $\mathtt{sg}^{10}$ and $\mathtt{sg}^{11}$ are compatible because they have the same pointer arrangement (x, y and z are all pointing to nodes). Accordingly, they are joined to form $\mathtt{sg}^{13}$ in $\mathtt{RSSG}^9$, the resulting reduced set of shape graphs for this example. On the contrary, the shape graph resulting from the first branch is stored as a separate graph, $\mathtt{sg}^{12}$, because pointer z is not assigned in it.

The final result of our analysis is the set of shape graphs that describe the state of the heap for every statement and by following any possible flow path in the program. These results are always conservative, meaning that a super-set for all possible shape graphs that represent the program heap, is constructed.

Termination of the analysis is guaranteed by the existence of the summarization of nodes and the joining of graphs: (i) similar nodes are *summarized* to bind each shape graph; and (ii) shape graphs with the same alias relationships between pointers are *joined* to bind each RSSG. Since the number of pointer variables to recursive data types declared in the program is fixed and known at compile time, the number of graphs per statement is limited by the different and mutually exclusive combinations of pointer over nodes. The theoretical maximum number of nodes per graph and graphs per RSSG, and its impact on the analysis complexity is further discussed in section 2.8.

2.2 Registering possible combinations of links: coexistent links sets

A program dealing with dynamic data structures performs run time allocation of memory pieces, that we call *memory locations*. Those locations are accessed and connected through pointers. More precisely, *stack-based heap-directed pointers*, or simply *pointers*, are used to access the structure, and *heap-based heap-directed pointers*, which we call *selectors*, are used to interconnect the heap-allocated elements. Such interconnected elements create *recursive data structures*, such as linked lists or trees.

We call *memory configuration* to the memory arrangement of the heap at a given point during a program execution. As we have mentioned, our approach to shape analysis is based on building shape graphs. These shape graphs abstract memory configurations arising in the analyzed program.

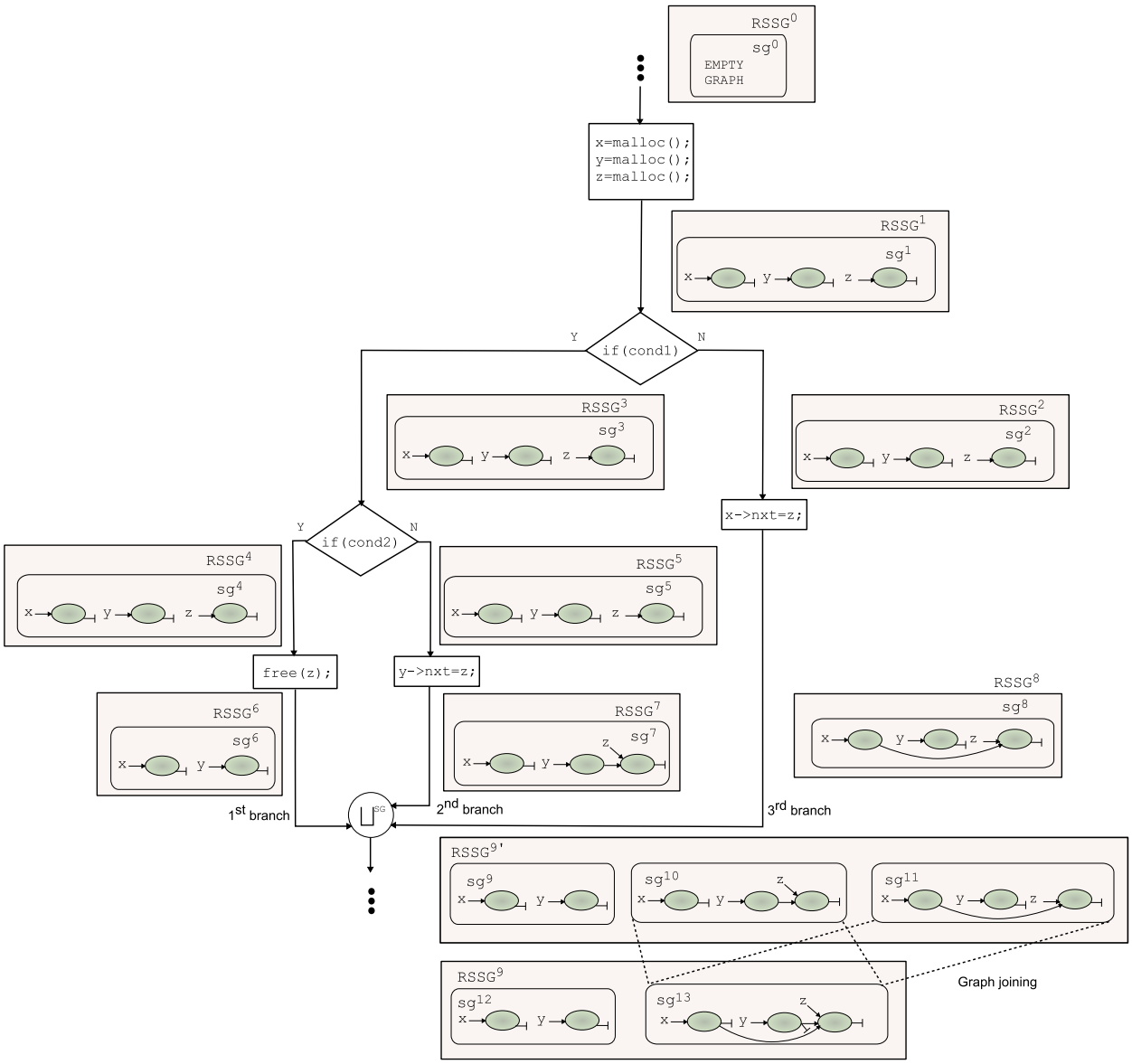

Figure 2.3: Different shape graphs for a statement are grouped into a RSSG.

The approach to shape analysis described so far is not substantially different from others found in the literature, such as [28], [26], [27], or [29]. Here, we introduce the main aspect that sets our technique apart from related work: the codification of possible connectivity patterns between nodes with the use of coexistent links sets.

Let us first introduce what are the elements that constitute the shape graphs. Fig. 2.4 shows a hierarchical view of those elements. At the lowest level we have: (i) *pointers*, used as access points to the structures; (ii) *nodes*, used to represent heap-allocated pieces of memory; and (iii) *selectors*, used to link nodes. Combining these base elements together, we can create two kind of relations: *pointer links* (`pl`'s), which are links between pointers and nodes; and *selector links* (`sl`'s), which are links between nodes through a selector. Finally, `pl`'s and `sl`'s can be combined together to form *coexistent links sets* (`cls`'s), that describe combinations of `pl`'s and `sl`'s that may exist *simultaneously* over a node.

The following example will help us introduce the concept of coexistent links sets. Fig. 2.5 shows $\mathtt{sg}^{13}$, the final graph that joins the effects of the second and third branch in the example in Fig. 2.3. This time

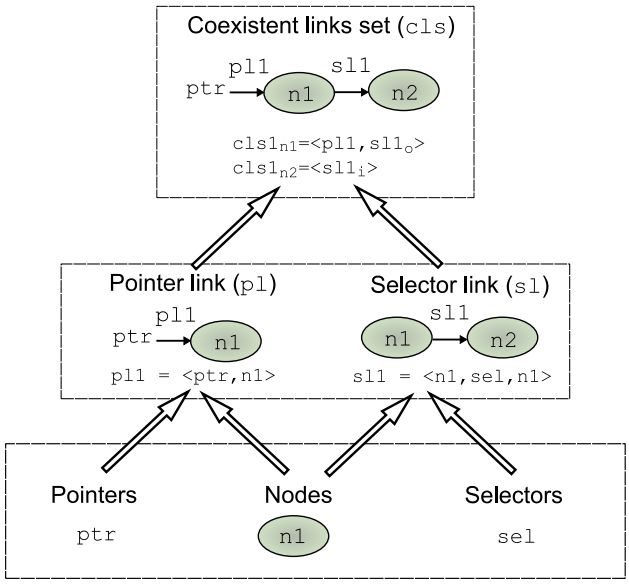

Figure 2.4: Hierarchical view of the elements in a shape graph.

the shape graph also displays the pointer links, selector links and coexistent links sets within it. We can see there are three nodes, labeled `n1`, `n2` and `n3`. Each one is pointed to by a different pointer: `x`, `y`, and `z`, respectively. A `cls` describes the links that may reach and leave a node in the memory abstraction. In the example, $\texttt{cls1}_{\texttt{n1}}$ is telling us that `n1` supports `pl1=<x,n1>` and `sl1=<n1,nxt,NULL>`. Alternatively, $\texttt{cls2}_{\texttt{n1}}$ also features `pl1` but this time `sl4=<n1,nxt,n3>` tells us that from `n1`, by following `nxt`, we could also reach another memory location, abstracted by `n3`. No other combination of links is possible for this node, because there are no more $\texttt{clsj}_{\texttt{n1}}$.

Similarly, $\texttt{cls1}_{\texttt{n2}}\texttt{=<pl2,sl2}_{\texttt{o}}\texttt{>}$ indicates that `n2` may abstract a memory location that is reached through pointer `y` and that connects to other location abstracted by `n3` through the `nxt` selector of `sl2=<n2,nxt,n3>`. There is another chance, and it is that the location pointed to by `y` does not point to any other location, and that is captured with $\texttt{cls2}_{\texttt{n2}}\texttt{=<pl2,sl5}_{\texttt{o}}\texttt{>}$. The `cls`'s for `n3` in turn indicate that `n3` can be reached in two ways: in $\texttt{cls1}_{\texttt{n3}}\texttt{=<pl3,sl2}_{\texttt{i}}\texttt{,sl3}_{\texttt{o}}\texttt{>}$, node `n3` is reached from `n2`, while in $\texttt{cls2}_{\texttt{n3}}\texttt{=<pl3,sl4}_{\texttt{i}}\texttt{,sl3}_{\texttt{o}}\texttt{>}$, node `n3` is reached from `n1`.

Selector links (`sl`'s) have a meaning on their own: they represent links between nodes through selectors. However, when used in the context of a coexistent links sets, they are complemented with *attributes* to correctly describe the connectivity pattern between nodes. In Fig. 2.5 two attributes are considered: (i) *incoming* (`i`), as $\texttt{sl2}_{\texttt{i}}$ in $\texttt{cls1}_{\texttt{n3}}$, which indicates an incoming link to the memory location represented by `n3`; and (ii) *outgoing* (`o`), as $\texttt{sl1}_{\texttt{o}}$ in $\texttt{cls1}_{\texttt{n1}}$, which indicates an outgoing link from the memory location represented by `n1`.

Coexistent links sets register possible connectivity patterns between heap-allocated elements. In general, they provide *may* information, i.e., they record connectivity patterns that may exist in the heap. In the case where there is more than one `cls` for a node, then one of them (and only one) will hold for any of the memory locations abstracted. When there is only one `cls` for a node, the information it provides is certain (*must* information). Coexistent links sets also provide definite or *must* information about what cannot hold. In other words, connectivity patterns that have *not* been registered in any `cls` for a node are impossible for the memory locations abstracted in the node.

For example, `n3` cannot be reached both from `n1` *and* `n2` because that possibility is not contemplated either by $\texttt{cls1}_{\texttt{n3}}$ or $\texttt{cls2}_{\texttt{n3}}$. Likewise, there is no chance that `n3` is *not* pointed to by some `nxt` selector:

either $\texttt{cls1}_{\texttt{n3}}$ or $\texttt{cls2}_{\texttt{n3}}$ must hold, although at compile time, we do not know which one.

Another aspect related to `cls`'s is that their information can be used to check *connectivity coherence* in the graph. This means that `cls`'s in a node must find a *match* in the `cls`'s for the nodes it connects to. For example, consider the case that $\texttt{cls2}_{\texttt{n1}}$ from $\texttt{sg}^{13}$ would be dropped. Such a graph would not be coherent regarding its `cls`'s because $\texttt{cls2}_{\texttt{n3}}$ is expecting some $\texttt{clsj}_{\texttt{n1}}$ to connect to it via `sl4`.

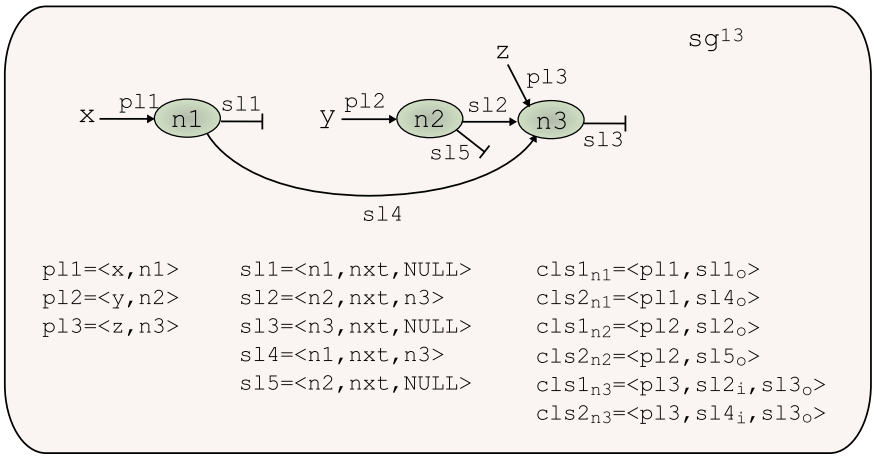

Figure 2.5: Coexistent links sets (`cls`'s) describe possible connections that may exist between nodes in a shape graph.

Coexistent links sets are also a neat way to capture different combination of links over memory locations in a single graph. Such a compact representation of the heap interconnection is key to building a precise yet affordable shape analysis technique.

2.3 A formal description of shape analysis

We have presented an informal view of our shape analysis strategy and its main motivation in the previous sections. Now, let us delve deeper into the description of our technique, through more formal definitions.

For this chapter we will only cover the formulation of an intraprocedural version of our shape analysis strategy, i.e., we assume the program has no function calls and therefore no context changes. This strategy is capable of analyzing single-function programs, or programs with inlined procedures. Recursive functions cannot be inlined and therefore are not supported by the technique described in this chapter. Extensions for interprocedural analysis, including recursive functions, will be covered in chapter 3.

To formalize the description of our model, we use the simple statements and definitions shown in Fig. 2.6. We only consider statements dealing with pointers as the ones shown in the figure (they are C-like imperative statements with dynamic allocation), because other complex pointer statements can be transformed into several of these simple pointer statements in a preprocessing stage, as seen in section 1.4.1.

We assume that the data types of all pointer variables are explicitly declared. A data type is comprised by some data fields, and some pointer fields, which we call *selectors*: `t=<field1,field2,...,fieldn,sel1,sel2,...,selm>`. $\texttt{SEL}^{\texttt{t}}$ is the set of selectors for type `t`, being $\texttt{SEL}^{\texttt{t}} \neq \emptyset$ for recursive data types. `SEL` is the set of all the selectors defined in the program.

As an example, let us consider the program excerpt of Fig. 2.7, in C syntax. In it, we find the declaration of a recursive data type, `struct node`, which is comprised by a data field named `data`, and a selector (or pointer field), named `nxt`. Next, a piece of the `main()` function follows, where a singly-

programs:	prog ∈ P, P=<STMT,PTR,TYPE,SEL>
statements:	s ∈ STMT, s::= x=NULL|x=malloc()|free(x)|x=y |x->sel=NULL|x->sel=y|x=y->sel
pointer variables:	x,y ∈ PTR
data types:	t ∈ TYPE
selectors fields:	sel ∈ SEL

Figure 2.6: Simple statements and definitions.

linked list is created. We numerate the pointer statements that have abstract semantics associated to them. The information provided by flow statements, such as the `while` loop is embedded in the data-flow equations that drive the iterative analysis (more information in section 2.4). For this example program we have STMT={st.1-st.8}, PTR={list,p,q,r}, TYPE={struct node} and SEL={nxt}.

```
   // Declare recursive type "node"
   struct node{
       int data;
       struct node *nxt;
   } *list,*p,*q,*r;
   int main(){
       int cont=0;
       // Create a singly-linked list
1:     list=(struct node *)malloc(...);
2:     p=list;
   L1: while(cont++<NUM_ELEM){
3:        q=(struct node *)malloc(...);
          q->data=cont;
4:        p->nxt=q;
5:        p=q;
       }
6:     p->nxt=NULL;
7:     p=NULL;
8:     q=NULL;
       [...]
   }
```

Figure 2.7: Excerpt of a program where a recursive data type is declared and later used to build a singly-linked list.

Next, we present how memory states that occur in the program are abstracted by our model. For that, we differentiate between the *concrete domain*, for run time memory configurations, and the *abstract domain*, for the shape graph abstractions.

2.3.1 Concrete heap

The heap information present at run time belongs to the *concrete domain* and is described as *memory configurations*, `mc`. That information is abstracted for its analysis in the *abstract domain*, in the form of *shape graphs*, `sg`.

In the concrete domain, memory locations represent single heap-allocated memory pieces. We use *pointer links in the concrete domain* (`plc`) and *selector links in the concrete domain* (`slc`) to describe

the relations between pointers and memory locations, and between memory locations through selectors, respectively. Based on the example program of Fig. 2.7, we now present a linked list of four elements in our representation of the concrete domain, in Fig. 2.8. Here, we explicitly display the information for the concrete pointer links and concrete selector links.

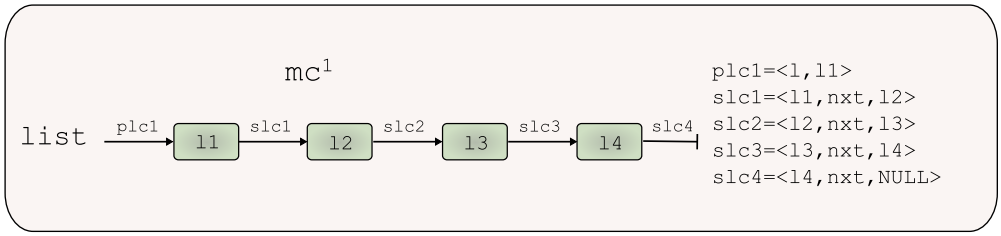

Figure 2.8: A singly-linked list of four elements in the concrete domain.

We model the concrete heap as a set of memory locations $\mathtt{l} \in \mathtt{L}$. We incorporate some instrumental functions in the concrete domain. For instance, we define the total function $\mathcal{T}$: $(\mathtt{PTR} \cup \mathtt{SEL}) \longrightarrow \mathtt{TYPE}$ to compute the type for each pointer or selector as:

$\forall\, \mathtt{x} \in \mathtt{PTR} \vee \mathtt{sel} \in \mathtt{SEL}, \exists\, \mathtt{t} \in \mathtt{TYPE} | \mathcal{T}(\mathtt{x})=\mathtt{t} \vee \mathcal{T}(\mathtt{sel})=\mathtt{t}.$

Initially, we define two mapping functions $\mathcal{PM}^c$ and $\mathcal{SM}^c$ to model the relations of pointers variables and selector fields to memory locations. $\mathcal{PM}^c$ and $\mathcal{SM}^c$ are partial functions that can be defined as follows:

Pointer Map (in the concrete domain): $\mathcal{PM}^c$: $\mathtt{PTR} \longrightarrow \mathtt{L}$

Selector Map (in the concrete domain): $\mathcal{SM}^c$: $\mathtt{L} \times \mathtt{SEL} \longrightarrow (\mathtt{L} \cup \mathtt{NULL})$

- $\mathcal{PM}^c$ maps a pointer variable x to the location l pointed to by x:

  $\forall\, \mathtt{x} \in \mathtt{PTR}, \exists\, \mathtt{l} \in \mathtt{L} | \mathcal{PM}^c(\mathtt{x})=\mathtt{l}.$

  We use the tuple plc=<x,n>, which we name *pointer link in the concrete domain*, to represent this binary relation. The set of all pointer links in the concrete domain is named PLc.

- $\mathcal{SM}^c$ models points-to relations between locations l1 and l2, through selector sel:

  $\forall\, \mathtt{l1} \in \mathtt{L}$ s.t. $\mathcal{T}(\mathtt{l1})=\mathtt{t} \wedge \forall\, \mathtt{sel} \in \mathtt{SEL}^{\mathtt{t}}, \exists\, \mathtt{l2} \in (\mathtt{L} \cup \mathtt{NULL}) | \mathcal{SM}^c(\mathtt{l1},\mathtt{sel})=\mathtt{l2}.$

  We use a tuple slc=<l1,sel,l2>, which we name *selector link in the concrete domain*, to represent this relation. The set of all concrete selector links in the concrete domain is called SLc.

Our concrete heap is modeled as a directed multi-graph. The domain for a concrete heap graph is the set $\mathtt{MC} \subset \mathcal{P}(\mathtt{L}) \times \mathcal{P}(\mathtt{PLc}) \times \mathcal{P}(\mathtt{SLc})$[1]. Each graph of our concrete domain is what we call a memory configuration $\mathtt{mc}^{\mathtt{i}} \in \mathtt{MC}$ and it is represented as a tuple $\mathtt{mc}^{\mathtt{i}}=<\mathtt{L}^{\mathtt{i}},\mathtt{PLc}^{\mathtt{i}},\mathtt{SLc}^{\mathtt{i}}>$ with $\mathtt{L}^{\mathtt{i}} \subset \mathtt{L}$, $\mathtt{PLc}^{\mathtt{i}} \subset \mathtt{PLc}$ and $\mathtt{SLc}^{\mathtt{i}} \subset \mathtt{SLc}$. At a given program statement s, we can represent our concrete heap as: $\mathtt{MC}^{\mathtt{s}}=\{\mathtt{mc}^{\mathtt{i}}$ $\forall$ path from entry to s$\}$.

2.3.2 Abstract heap

Our abstract domain is based on shape graphs. The base element for our representation of the abstract heap is the node, n. In a shape graph, each node may represent a set of memory locations from the concrete domain, whereas each edge may represent a pointer variable or a set of selectors with the same name.

[1] In this work we will use the notation $\mathcal{P}(\mathtt{A})$ to represent the power set of set A.

The set of all the nodes in the graph is N, and includes a special node named NULL, designating "no location". In a graph, the number of nodes is bounded by the *summarization* policy. The base policy states that nodes are distinguishable by the set of pointer variables that point to them. Two nodes are said to be *compatible*, if they are indistinguishable in the representation. More precisely, they will be compatible if they are pointed to by the same set of pointers. In particular, all memory locations that are not pointed to by pointers are represented by a single summary node. This policy can be refined for a greater differentiation of nodes with the use of *properties*, but to simplify the presentation and until further notice, we will use this simple summarization policy. Therefore, the domain for the nodes is $\mathtt{N}=\{\mathcal{P}(\mathtt{PTR}) \cup \{\mathtt{NULL}\}\}$.

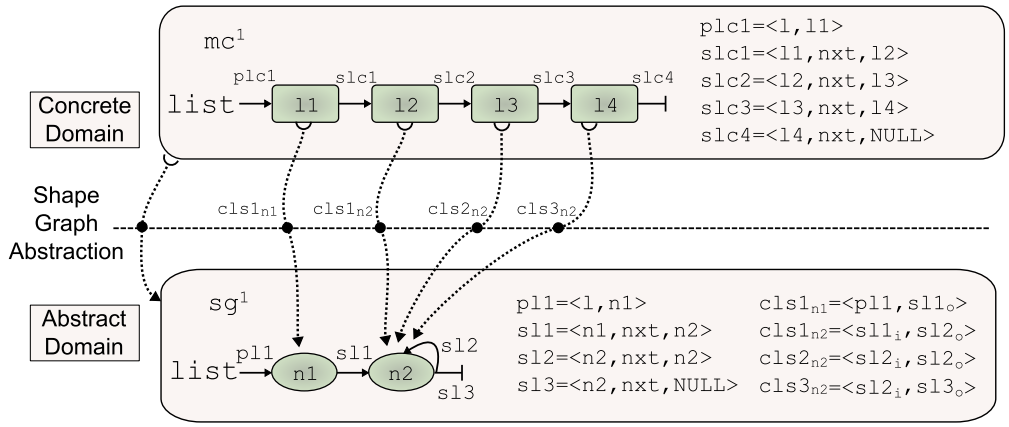

Figure 2.9: The singly-linked list used as example in the concrete and abstract domain representations.

A shape graph is displayed as a set of nodes, pointer links, selector links, and coexistent links sets, the latter grouping pointer links and selector links in the available combinations for every node. Let us present now an example of how these elements in the abstract domain are put together to capture a recursive data structure. For that, we consider the singly-linked list that we presented in the concrete domain in Fig. 2.8. The abstraction of such a list in the abstract domain is presented in Fig. 2.9. All three memory locations l2, l3 and l4 translate into n2, because they are not pointed to by pointers. Accordingly, l1 translates into n1, pointed to by pointer list. Note that a selector link in the abstract domain can represent several instances of selector links in the concrete domain. Such is the case of sl2=<n2,nxt,n2>, which accounts for slc2 and slc3.

Despite this reduction in the number of matching elements, shape graph $\mathtt{sg}^1$ contains, within the coexistent links sets, *all* the information present in the memory configuration, $\mathtt{mc}^1$ (although, as a result of being a conservative abstraction, $\mathtt{sg}^1$ represents a list of *4 or more* list elements). Note that there may be more than one coexistent links set for a node. Since a node can represent several memory locations, its coexistent links sets must contain all the possibilities of links existing in those memory locations. For example, coexistent links sets for node n2 ($\mathtt{cls1}_{\mathtt{n2}}$, $\mathtt{cls2}_{\mathtt{n2}}$, $\mathtt{cls3}_{\mathtt{n3}}$), represent the three different connectivity patterns for the memory locations abstracted by node n2. Attributes *incoming* (i) and *outgoing* (o) are used within the coexistent link sets to accurately express the possible connections in the structure.

It must be stressed that sl2 in Fig. 2.9 does not involve a cycle in the structure. For example, $\mathtt{cls2}_{\mathtt{n2}}$=<$\mathtt{sl2}_{\mathtt{i}}$,$\mathtt{sl2}_{\mathtt{o}}$> indicates that n2 can represent a memory location that is reached from another location abstracted by n2, and leaves for another destination also abstracted by n2, being the origin and destination *different* memory locations (albeit represented in the same node). In the case of a direct cycle (i.e., a memory location that points to itself) in the structure, a new attribute is introduced, as will be seen shortly.

Now we define three mapping functions $\mathcal{LM}$, $\mathcal{PM}^a$, $\mathcal{SM}^a$ to model the relationship between memory

locations and nodes in the concrete and abstract domain, as well as the connections of pointers variables and selectors to nodes in the abstract heap. The mapping functions $\mathcal{LM}$ and $\mathcal{PM}^a$ are total functions, while $\mathcal{SM}^a$ is a multivalued function. They can be defined as follows:

Location Map :	$\mathcal{LM}$: L $\longrightarrow$ N
Pointer Map (in the abstract domain) :	$\mathcal{PM}^a$: PTR $\longrightarrow$ N
Selector Map (in the abstract domain):	$\mathcal{SM}^a$: N $\times$ SEL $\longrightarrow$ N

- $\mathcal{LM}$ assigns a node n to a concrete memory location l: $\forall$ l $\in$ L, $\exists$ n $\in$ N | $\mathcal{LM}$(l)=n.
- $\mathcal{PM}^a$ maps a pointer variable x which points to a location l in the concrete domain, to a node n in the abstract domain:

  $\forall\, \mathcal{PM}^c$(x)=l $\subset$ MC, $\exists$ n $\in$ N s.t. $\mathcal{LM}$(l)=n | $\mathcal{PM}^a$(x)=n.

  We use the tuple pl=<x,n>, which we name *pointer link*, to represent this binary relation. The set of all pointer links in the abstract domain is named PL.
- $\mathcal{SM}^a$ models points to relations between locations li and lj through selector field sel in the concrete domain, as relations between nodes n1 and n2 in the abstract domain:

  $\forall$ $\mathcal{SM}^c$(li,sel)=lj $\subset$ MC, $\exists$ n1 $\in$ (N-{NULL}) $\wedge$ $\exists$ n2 $\in$ N s.t. $\mathcal{LM}$(li)=n1 $\wedge$ $\mathcal{LM}$(lj)=n2 | $\mathcal{SM}^a$(n1,sel)=n2.

  Again, we use a tuple sl=<n1,sel,n2>, which now we name *selector link* to represent this relation. The set of all selector links in the abstract domain is called SL.

The novelty of our approach is that we keep the information about connectivity and aliasing in a node-oriented fashion. For it, we build new instrumentation domains, that when added to the nodes in the abstract heap will improve the accuracy of the connectivity and aliasing information.

2.3.2.1 Selector links with attributes

We have already shown that selector links are complemented with attributes in the context of coexistent links sets. Attributes are used to define how a particular selector link relates to the nodes that are linked through it. So far, we have introduced the incoming and outgoing attributes in an intuitive fashion. Now, we define them more formally, and complete the description with two new attributes that are used to capture *cyclic links* and *sharing*.

Fig. 2.10 is meant as an example to present the role of the different kinds of attributes within coexistent links sets to capture information in the heap accurately. Again, we present the information both in the concrete and abstract domains. In particular, we will look at the newly introduced cyclic (c) and shared (s) attributes. Note that locations l2 and l3 are summarized in the node n2. Concrete selector links slc1 and slc2 translate to sl1 and sl2 respectively, since they refer to different selectors (nxt and prv). Note that sl1 and sl2 appear in different $\mathtt{clsj}_{\mathtt{n2}}$ so they can *not* coexist, which precisely captures the fact that following nxt or prv from n1 leads to different locations. However, slc3 and slc4 (both using nxt) are mapped into sl3. That way, $\mathtt{sl3}_{\mathtt{s}}$ in $\mathtt{cls1}_{\mathtt{n3}}$ indicates that you can point to a location represented by node n3 from more than one different locations represented in node n2 by following the same selector (nxt). On the other hand, $\mathtt{sl4}_{\mathtt{c}}$ in $\mathtt{cls1}_{\mathtt{n3}}$ expresses that the location l4 represented in n3 is pointing to itself. Note the difference with $\mathtt{cls2}_{\mathtt{n2}}$=<$\mathtt{sl2}_{\mathtt{i}}$,$\mathtt{sl2}_{\mathtt{o}}$> in Fig. 2.9, which indicates that one location represented in node n2 is pointing to a *different* location represented in the same node.

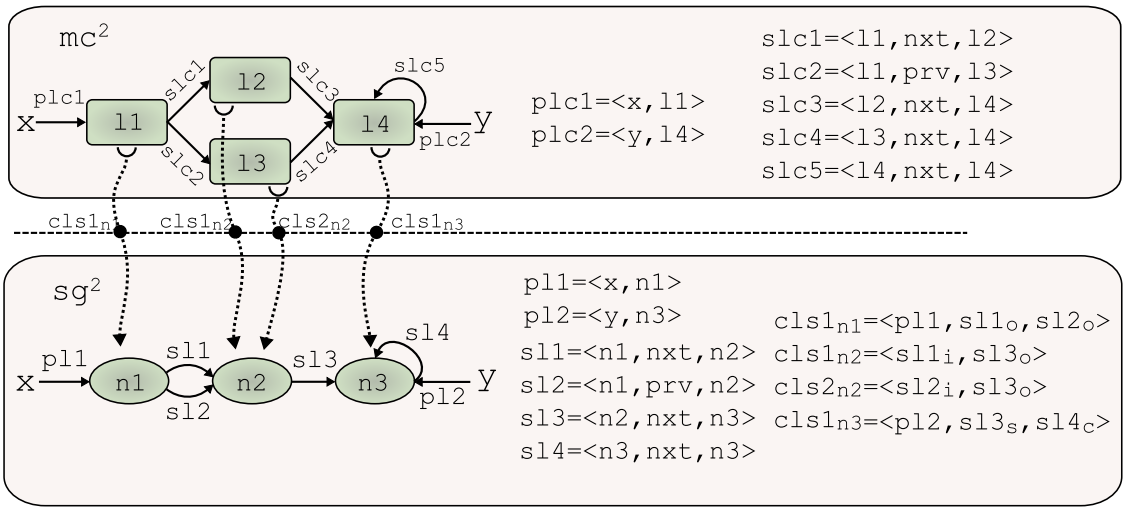

Figure 2.10: Different attributes and their role for precise heap abstraction.

We define a set of attributes, ATT={i,o,c,s}, where each element $\texttt{att} \in \texttt{ATT}$ codifies information about the direction of a selector link when it is related to a node. From the set ATT we define a new domain ATTSL=$\mathcal{P}$(ATT), where each element of this new domain $\texttt{attsl} \in \texttt{ATTSL}$ represents a possible combination of attributes that describe the characteristics of a selector link when it is associated to a node. Operator $\uplus$ stands for the join operation in the ATTSL domain.

In particular, from the set of all selector links, SL and from ATTSL we define the domain $\texttt{SL}_{\texttt{att}} = \texttt{SL} \times \texttt{ATTSL}$. An element $\texttt{sl}_{\texttt{att}}$ in this domain, which we call a *selector link with attributes*, is represented as a tuple $\texttt{sl}_{\texttt{att}}$=<sl,attsl>, where $\texttt{sl} \in \texttt{SL}$ and $\texttt{attsl} \in \texttt{ATTSL}$. Note that we choose to represent a selector link with several attributes as several selector links with just one attribute each. For example, we write $\texttt{cls2}_{\texttt{n2}}$=<$\texttt{sl2}_{\texttt{i}}$, $\texttt{sl2}_{\texttt{o}}$>, rather than $\texttt{cls2}_{\texttt{n2}}$=<$\texttt{sl2}_{\texttt{io}}$>, to improve readability of shape graphs.

2.3.2.2 Coexistent links sets

The key feature of our model is the ability to maintain the connectivity and aliasing information that can coexist in an abstract node, even when the node represents different memory locations with different connection patterns. This is achieved through the *coexistent links set* abstraction. The domain of our coexistent links set abstraction is defined in terms of a mapping function $\mathcal{CLM}$ as follows:

Coexistent Links Map : $\mathcal{CLM}$: $\texttt{N} \longrightarrow \mathcal{P}(\texttt{PL}) \times \mathcal{P}(\texttt{SL}_{\texttt{att}})$

$\mathcal{CLM}$ is a multivalued function which maps, for a node n, one or more components, each one called a *coexistent links set*, $\texttt{cls}_{\texttt{n}}$: $\forall\, \texttt{n} \in \texttt{N}$, $\mathcal{CLM}$(n)={$\texttt{cls}_{\texttt{n}}$}. A coexistent links set, $\texttt{cls}_{\texttt{n}}$, codifies an aliasing and connectivity pattern for that node, and it is defined as follows:

$\texttt{cls}_{\texttt{n}}$={$\texttt{PL}_{\texttt{n}}$, $\texttt{SL}_{\texttt{n}}$}

where:

$\texttt{PL}_{\texttt{n}}$ = {pl $\in$ PL s.t. pl=<x,n>}
$\texttt{SL}_{\texttt{n}}$ = {$\texttt{sl}_{\texttt{att}} \in \texttt{SL}_{\texttt{att}}$ s.t. $\texttt{sl}_{\texttt{att}}$ =<<n1,sel,n2>,attsl>, being (n1=n $\vee$ n2=n)}

Regarding the attributes codified at attsl, they are obtained from the concrete domain, in particular from L and the set of selector links in the concrete domain, SLc. These attributes have meaning when they are interpreted in a $\texttt{cls}_{\texttt{n}}$ context (i.e. associated with a node), as we expose next.

Let $\texttt{cls}_\texttt{n}$=$\{\texttt{PL}_\texttt{n}, \texttt{SL}_\texttt{n}\}$ be. For each $\texttt{sl}_\texttt{att}$=<<n1,sel,n2>,attsl> $\in$ $\texttt{SL}_\texttt{n}$ we can find one of the following cases:

If l1 $\neq$ l2 $\wedge$ $\exists$ $\texttt{slc}_1$(l1,sel,l) $\wedge$ $\exists$ $\texttt{slc}_2$(l2,sel,l) s.t. ($\mathcal{LM}$(l1)=$\mathcal{LM}$(l2)=n1 $\wedge$ $\mathcal{LM}$(l)=n2=n) $\Longrightarrow$ s $\in$ attsl

else

If l1 $\neq$ l2 $\wedge$ $\exists$ slc=<l1,sel,l2> s.t. ($\mathcal{LM}$(l1)=n1 $\wedge$ $\mathcal{LM}$(l2)=n2=n) $\Longrightarrow$ i $\in$ attsl.

If l1 $\neq$ l2 $\wedge$ $\exists$ slc=<l1,sel,l2> s.t. ($\mathcal{LM}$(l1)=n1=n $\wedge$ $\mathcal{LM}$(l2)=n2) $\Longrightarrow$ o $\in$ attsl.

If l1=l2=l $\wedge$ $\exists$ slc=<l,sel,l> s.t. ($\mathcal{LM}$(l)=n1=n2=n) $\Longrightarrow$ c $\in$ attsl.

The set of all the $\texttt{cls}_\texttt{n}$ associated to a node n is called $\texttt{CLS}_\texttt{n}$. In addition, for every nodes n defined in our abstract heap, we can create the set CLS=$\{\texttt{CLS}_\texttt{n}, \forall \texttt{n} \in \texttt{N}\}$.

2.3.2.3 Shape graphs

Our abstract heap is modeled as a directed multi-graph. The domain for an abstract heap graph is the set SG $\subset \mathcal{P}$(N) $\times$ $\mathcal{P}$(CLS). Each element of this domain, $\texttt{sg}^\texttt{i} \in \texttt{SG}$ is what we call a *shape graph*, which we represent as a tuple $\texttt{sg}^\texttt{i}$=<$\texttt{N}^\texttt{i}$,$\texttt{CLS}^\texttt{i}$>, with $\texttt{N}^\texttt{i} \subset$ N and $\texttt{CLS}^\texttt{i}$=$\{\texttt{CLS}_\texttt{n}, \forall \texttt{n} \in \texttt{N}^\texttt{i}\} \subset$ CLS.

We restrict this abstract domain by defining a *normal form* of the shape graphs. To help us describe the normal form of a graph we use the Compatible_Node() and Unreachable() functions, shown in Fig. 2.11. The Compatible_Node(n1,n2,$\texttt{CLS}_\texttt{n1}$,$\texttt{CLS}_\texttt{n2}$) function returns TRUE if n1 and n2 are compatible, and thus, can be summarized into a single node. The Unreachable(n1,$\texttt{sg}^\texttt{i}$) function returns TRUE if n1 cannot be reached either directly by a pointer or indirectly through a path formed by a pointer link and some selector links in graph $\texttt{sg}^\texttt{i}$. Otherwise, it returns FALSE.

We say that a shape graph $\texttt{sg}^\texttt{i}$=<$\texttt{N}^\texttt{i}$,$\texttt{CLS}^\texttt{i}$> is in normal form if:

1. It has no compatible nodes: $\nexists$ n1, n2 $\in$ $\texttt{N}^\texttt{i}$ s.t. Compatible_Node(n1,n2,$\texttt{CLS}_\texttt{n1}$,$\texttt{CLS}_\texttt{n2}$)=TRUE
2. It has no unreachable nodes: $\nexists$ n1 $\in$ $\texttt{N}^\texttt{i}$ s.t. Unreachable(n1,$\texttt{sg}^\texttt{i}$)=TRUE
3. Each pointer variable unambiguously points to *just one* node: $\forall$ n1, n2 $\in$ $\texttt{N}^\texttt{i}$ s.t. n1 $\neq$ n2, If $\exists$ pl1=<x,n1> $\subset$ $\texttt{CLS}_\texttt{n1}$ $\Longrightarrow$ $\nexists$ pl2=<x,n2> $\subset$ $\texttt{CLS}_\texttt{n2}$
4. The selector links of connected nodes, are coherent, i.e., for a given $\texttt{cls}_\texttt{n1} \subset \texttt{CLS}_\texttt{n1}$, every incoming or shared (i|s) selector link with attributes, $\texttt{sl}_\texttt{att}$, is *matched* by an outgoing (o) $\texttt{sl}_\texttt{att}$ in another node, n2 and viceversa : $\forall$ n1, n2 $\in$ $\texttt{N}^\texttt{i}$ s.t. n1 $\neq$ n2, If $\exists$ $\texttt{sl}_\texttt{att}$=<<n1,selk,n2>,attsl> $\subset$ $\texttt{CLS}_\texttt{n1}$ $\Longrightarrow$ $\exists$ $\texttt{sl}_\texttt{att}$=<<n1,selk,n2>,attsl'> $\subset$ $\texttt{CLS}_\texttt{n2}$

Fig. 2.12 shows the aliasing of pointers in the traversal of a singly-linked list. It is the example introduced in Fig. 2.2(a), only expanded to show all the information within the shape graph, featuring the pointer links, the selector links and the coexistent links sets. This example shows the process of abstract interpretation for an incoming graph and aliasing statement p=a. The input and output shape graphs, $\texttt{sg}^1$ and $\texttt{sg}^2$, are in normal form because (i) they have no compatible nodes, (ii) they have no unreachable nodes, (iii) each pointer points to just one node, and (iv) selector links are coherent throughout. On the other hand, the intermediate graph, $\texttt{sg}^\texttt{A}$, is not in normal form, as there are compatible nodes that are not merged, namely n2 and n3.

```
Compatible_Node()
 Input: n1, n2, CLS_n1, CLS_n2
     # 2 nodes and their cls's
 Output: TRUE/FALSE

 If ((∀ pl1=<x,n1> ⊂ CLS_n1,
     ∃ pl2=<x,n2> ⊂ CLS_n2) ∧
     (∀ pl2=<y,n2> ⊂ CLS_n2,
     ∃ pl1=<y,n1> ⊂ CLS_n1 )),
         return(TRUE)
 else
         return(FALSE)
end
```

(a)

```
Unreachable()
 Input: n1, sg^i=<N^i, CLS^i>   # A node and a shape graph
 Output: TRUE/FALSE

 If ( (∃ pl1=<x,n1> ⊂ CLS_n1) ∨
     (∃ n2 ∈ N^i s.t. ∃ pl2=<x,n2> ⊂ CLS_n2 ∧
     ∃ sl_atti=<<n2,seli,na>,attsli={o}> ⊂ CLS_n1 ∧
     ∃ sl_attj=<<na,selj,nb>,attslj={o}> ⊂ CLS_na ∧ ...
     ... ∧ ∃ sl_attk=<<nk,selk,n1>,attslk={o}>> ⊂ CLS_nk) ),
             return(FALSE)
 else
             return(TRUE)
end
```

(b)

Figure 2.11: (a) Check whether two nodes are compatible; (b) Check whether a node is unreachable in the current graph.

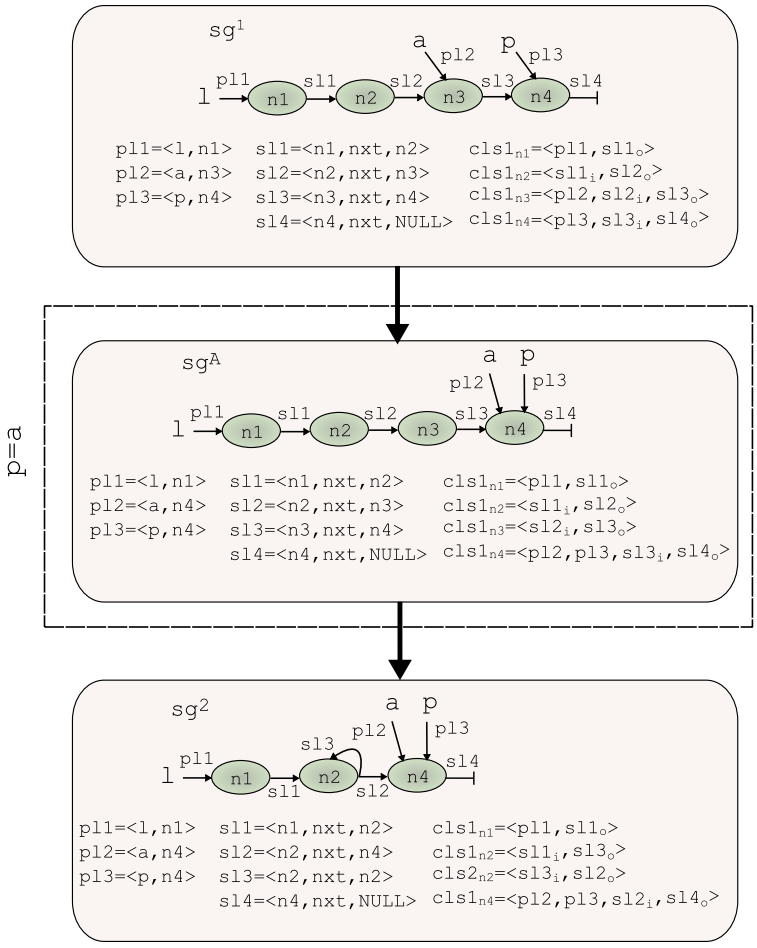

Figure 2.12: Graphs in normal form around a pointer aliasing operation.

2.3.2.4 Reduced set of shape graphs

We call a *reduced set of shape graphs* to the set of shape graphs that represents the state of the heap at a given program statement s: $\texttt{RSSG}^s=\{\texttt{sg}^i \in \texttt{SG}$ s.t. $\texttt{sg}^i$ is in normal form$\}$.

Again, we impose a restriction in this set of graphs, and it is that the set is in *normal form*. The constraint that a reduced set of shape graphs $\texttt{RSSG}^s$ is in normal form ensures that each graph $\texttt{sg}^i \in \texttt{RSSG}^s$ represents a different alias configuration.

More formally, we say that a reduced set of shape graphs, $\texttt{RSSG}^s=\{\texttt{sg}^i\}$ is in normal form if it has no compatible shape graphs: $\nexists\ \texttt{sg}^1, \texttt{sg}^2 \in \texttt{RSSG}^s$ s.t. $\texttt{Compatible\_SG(sg}^1\texttt{,sg}^2\texttt{)=TRUE}$.

```
Compatible_SG()
 Input: sg1=<N1,CLS1>,sg2=<N2,CLS2>        # Two shape graphs
 Output: TRUE/FALSE

 If ( (∀ ni ∈ N1, ∃ pl=<x,ni> ⊂ CLSni ∧ ∃ nj ∈ N2 s.t. Compatible_Node(ni,nj,CLSni,CLSnj)=TRUE) ∧
    (∀ nj ∈ N2, ∃ pl=<y,nj> ⊂ CLSnj ∧ ∃ ni ∈ N1 s.t. Compatible_Node(nj,ni,CLSnj,CLSni)=TRUE) ),
        return(TRUE)
 else
        return(FALSE)
end
```

Figure 2.13: Check whether two shape graphs are compatible.

The auxiliary function $\texttt{Compatible\_SG(sg}^1\texttt{,sg}^2\texttt{)}$ is described now in Fig. 2.13. The function checks that for each node of graph $\texttt{sg}^1$ pointed to by a pointer (or set of pointers), there is another node of graph $\texttt{sg}^2$ pointed to by the same pointer (or set of pointers). The same check is done for all the nodes in graph $\texttt{sg}^2$. In other words, the function checks that all the nodes pointed to by pointer variables in graphs $\texttt{sg}^1$ and $\texttt{sg}^2$ are compatible. In this case, we would say that the two graphs are compatible, and they could be *joined* in a new *summary graph*. Clearly, only the graphs with the same alias relationships can be joined.

Let us revisit the example of Fig. 2.3. We expand the information of the final RSSG's in the new Fig. 2.14, with cls information. For readability, we omit the explicit description of pointer links and selector links, which can be easily guessed by looking at the shape graphs.

When a join point in the CFG is found, the RSSG's resulting from the different flow paths are joined. This is done by calling the Join_RSSG() function, shown in Fig. 2.15(a). This function join graphs from different RSSG's by adding them to a working shape graph set $\texttt{RSSG}^{k'}$ ($\texttt{RSSG}^{9'}$ in our example), and then summarizing it with the Summarize_RSSG() function (Fig. 2.15(b)). The Summarize_RSSG() function checks whether its input $\texttt{RSSG}^1$ is in normal form. This is done by checking for compatible shape graphs with the Compatible_SG() function presented earlier (Fig. 2.13). In this example, $\texttt{Compatible\_SG(sg}^{10}\texttt{,sg}^{11}\texttt{)=}$ TRUE, i.e., $\texttt{sg}^{10}$ and $\texttt{sg}^{11}$ are compatible. Therefore $\texttt{RSSG}^{9'}$ is not in normal form.

Compatible shape graphs must be joined. This is done with the Join_SG() function. This latter function can be found for your reference in Appendix A. Here, we will just state its overall behavior: to pair matching nodes between graphs, and to add cls information. In this example, n1, n2, and n3, the nodes pointed by the x, y, and z pointers respectively, are compatible on a one-to-one basis between $\texttt{sg}^{10}$ and $\texttt{sg}^{11}$. Therefore $\texttt{sg}^{13}$ in $\texttt{RSSG}^9$ contains those same three nodes pointed to by the same pointers.

Additionally, all the information from the `cls`'s in sg^{10} and sg^{11} is added to capture the fact that `n3` can be reached from `n2` *or* `n1`. This is reflected upon the `cls`'s in sg^{13}.

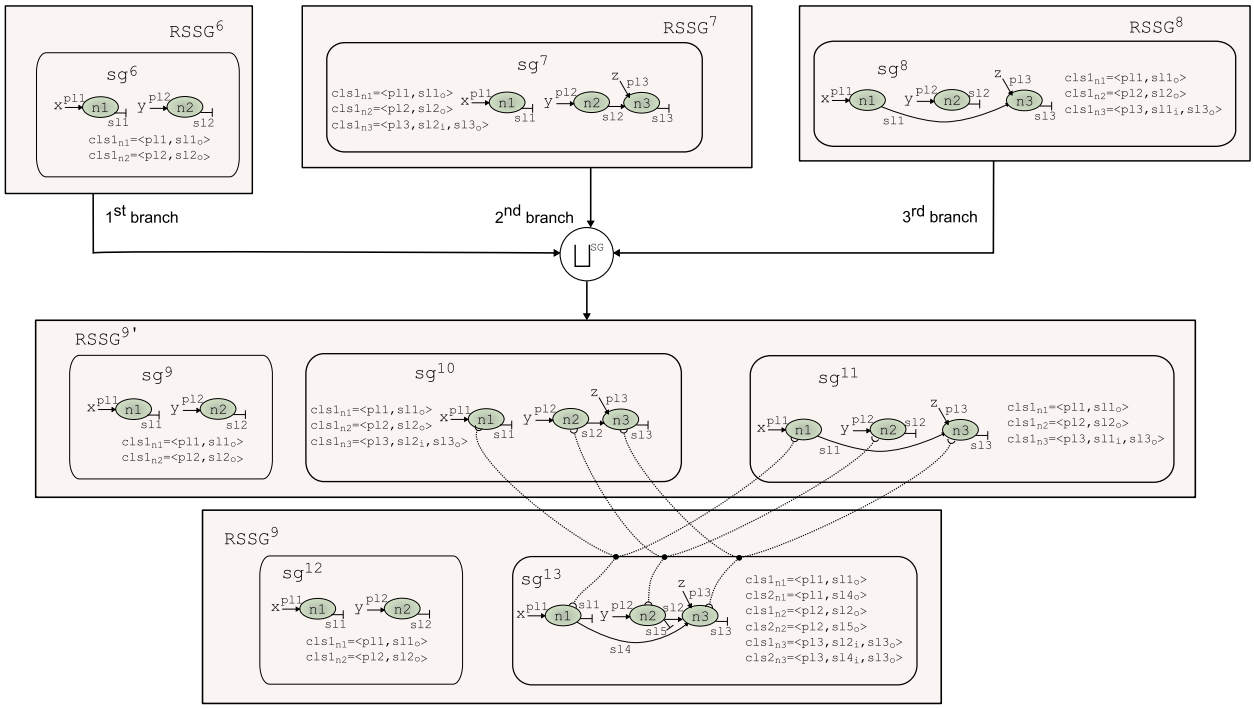

Figure 2.14: Joining compatible shape graphs in a RSSG.

```
Join_RSSG() ( ⊔^RSSG )
 Input: RSSG^1, RSSG^2
     # Two reduced sets of shape graphs
 Output: RSSG^k
     # A reduced set of shape graphs in normal form

 RSSG^k=∅
 Create RSSG^k'=RSSG^1 ∪ RSSG^2
 RSSG^k=Summarize_RSSG(RSSG^k')
 return(RSSG^k)
end
```

(a)

```
Summarize_RSSG()
 Input: RSSG^1
     # A reduced set of shape graphs
 Output: RSSG^k
     # A reduced set of shape graphs in normal form

 RSSG^k=∅
 forall sg^i ∈ RSSG^1
     If (∃ sg^j ∈ RSSG^k
         s.t. Compatible_SG(sg^i,sg^j)=TRUE),
                 RSSG^k=RSSG^k-sg^i ∪ Join_SG(sg^i,sg^j)
     else
                 RSSG^k=RSSG^k ∪ sg^i
 endfor
 return(RSSG^k)
end
```

(b)

Figure 2.15: (a) The operator $\sqcup^{RSSG}$ as the `Join_RSSG()` function; (b) `Summarize_RSSG()` function.

2.4 Data-flow equations and worklist algorithm

We consider two kinds of statements for our analysis: (i) those that allocate, free, traverse, or connect memory locations through pointers, and (ii) those that determine the control flow of the program, such as `while` loops or `if` branches. Both kinds of statements must be modeled within our technique for effective analysis. We associate some abstract semantics to every statement in the first category. Shape graphs are modified according to those abstract semantics. The effect of the second kind of statements is reflected upon the data-flow equations. The information they provide drives the iterative analysis.

We formulate our analysis as a data-flow analysis that computes a reduced set of shape graphs at each program point. For each statement in the program, $s \in STMT$, we define two program points: $\bullet s$ is the program point before s, and $s\bullet$ is the program point after s. The result of the analysis for s is a reduced set of shape graphs, $RSSG^{\bullet s}$ before s, and $RSSG^{s\bullet}$ after that. Let $pred()$ map statements to their predecessor statements in the control flow (these can be easily computed from the syntactic structure of control statements). Fig. 2.16 shows the data-flow equations for our intraprocedural shape analysis.

$$\text{[JOIN]:} \quad RSSG^{\bullet s} = \bigsqcup^{RSSG}_{s' \in pred(s)} RSSG^{s'\bullet}$$

$$\text{[TRANSF]:} \quad RSSG^{s\bullet} = AS_s(RSSG^{\bullet s}), \text{ where}$$

$$AS_{s::=x=NULL}(RSSG^{\bullet s}) = \bigsqcup^{RSSG}_{sg^i \in RSSG^{\bullet s}} XNULL(sg^i, x)$$

$$AS_{s::=x=malloc()}(RSSG^{\bullet s}) = \bigsqcup^{RSSG}_{sg^i \in RSSG^{\bullet s}} XNew(sg^i, x)$$

$$AS_{s::=free(x)}(RSSG^{\bullet s}) = \bigsqcup^{RSSG}_{sg^i \in RSSG^{\bullet s}} FreeX(sg^i, x)$$

$$AS_{s::=x=y}(RSSG^{\bullet s}) = \bigsqcup^{RSSG}_{sg^i \in RSSG^{\bullet s}} XY(sg^i, x, y)$$

$$AS_{s::=x->sel=NULL}(RSSG^{\bullet s}) = \bigsqcup^{RSSG}_{sg^i \in RSSG^{\bullet s}} XSelNULL(sg^i, x, sel)$$

$$AS_{s::=x->sel=y}(RSSG^{\bullet s}) = \bigsqcup^{RSSG}_{sg^i \in RSSG^{\bullet s}} XselY(sg^i, x, sel, y)$$

$$AS_{s::=x=y->sel}(RSSG^{\bullet s}) = \bigsqcup^{RSSG}_{sg^i \in RSSG^{\bullet s}} XYSel(sg^i, x, y, sel)$$

Figure 2.16: Data-flow equations for intraprocedural analysis.

We model the analysis of statements which have some associated abstract semantics by computing a transfer function for each one. To simplify the formal definitions of the transfer functions, we use the functions `XNULL()`, `XNew()`, `FreeX()`, `XY()`, `XSelNULL()`, `XselY()` and `XYSel()` to describe the transformations that take place in the abstract heap when a simple pointer statement `s` is interpreted. These functions are detailed in the following sections. The operator $\bigsqcup^{RSSG}$ represents the join operation in the `RSSG` domain, and is described as function `Join_RSSG()` (Fig. 2.15(a)).

We present in Fig. 2.17 a worklist algorithm to solve the data-flow equations presented in Fig. 2.16. The input of our worklist algorithm is a program `P` and an initial $RSSG^{in}=\emptyset$, whereas the output is the $RSSG^{out}$ resultant at the exit program point, assuming that the exit point is statement $sr \in STMT$. This algorithm also computes the resultant $RSSG^{s\bullet}$ at each program point. Lines 1–3 perform the initialization, where the `RSSG` at the input of the program entry point (in our case statement $se \in STMT$) is initialized with $RSSG^{in}$. Next, the algorithm processes the worklist using the loop defined in lines 4–12. At each iteration, it removes, in program lexicographic order, a statement for the worklist, computes the join of the `RSSG`'s from the predecessors as the statement input (`pred(s)`), and then it applies the corresponding transfer function.

The worklist algorithm is responsible for achieving a fixed point for the analysis, and therefore it guarantees its termination. The analysis will continue iteratively while the shape graphs obtained keep changing.

Line 8 in the worklist algorithm checks whether the $\mathtt{RSSG}^{s\bullet}$ obtained after the analysis of a statement has changed. If it has changed, then new information has been added to or subtracted from the shape graphs and the analysis must continue. The analysis continues by adding the successors of statements `s` in the CFG (`succ(s)`) to the worklist. If there is no change, it means that further interpretation of pointer statements will not change the graphs either. This is guaranteed by the node summarization and graph joining mechanisms. In such a case we have achieved a fixed point, and the analysis terminates.

```
Worklist()
  Input: P=<STMT,PTR,TYPE,SEL>, RSSG^in   # A program and an input RSSG
  Output: RSSG^out                        # The RSSG at the exit program point

1:  Create W=STMT
2:  RSSG•se=RSSG^in
3:  ∀ s ∈ STMT → RSSG^s•=∅
4:  repeat
5:      Remove s from W in lexicographic order
6:      RSSG•s=⊔^RSSG_{s'∈pred(s)} RSSG^s'•
7:      RSSG^s•=AS_s(RSSG•s)
8:      If (RSSG^s• has changed),
9:          forall s' ∈ succ(s),
10:             W=W ∪ s'
11:         endfor
12: until (W=∅)
13: RSSG^out=RSSG^sr•
14: return(RSSG^out)
end
```

Figure 2.17: The worklist algorithm. It computes the $\mathtt{RSSG}^{s\bullet}$ at each program point.

2.5 Abstract semantics and operations

Our analysis works by symbolically executing the abstract semantics of pointer statements. In this section we describe, at high level, the abstract semantics associated to each statement. For a full reference of the operations involved in this section, please refer to Appendix A.

2.5.1 Running example

To ease the presentation of the shape analysis abstract semantics, we will use the example code in Fig. 2.18. It expands on the code excerpt presented in Fig. 2.7. Besides creating a singly-linked list, it then reverses it and finally frees its space. Once again, pointer statements that have abstract semantics associated to them, and thus are modeled through transfer functions, are numbered. This simple example will help us explain the details that are involved in the shape analysis operations. We will present such information by considering common tasks performed in programs that make use of recursive data structures. More precisely, we will cover: (i) the creation of new elements; (ii) linking already created elements to create a recursive data structure; (iii) traversing a recursive data structure; and (iv) freeing memory.

```
    // Declare recursive type "node"
    struct node{
        int data;
        struct node *nxt;
    } *list,*p,*q,*r;
    int main(){
        int cont=0;
        // Create a singly-linked list
 1:     list=(struct node *)malloc(...);
 2:     p=list;
   L1: while(cont++<NUM_ELEM){
 3:        q=(struct node *)malloc(...);
           q->data=cont;
 4:        p->nxt=q;
 5:        p=q;
       }
 6:    p->nxt=NULL;
 7:    p=NULL;
 8:    q=NULL;
       // Iteratively reverse the list
 9:    p=list;
10:    list=NULL;
11:    r=NULL;
   L2: while(p!=NULL){
12:       #pragma SAP.force(p!=NULL)
13:       q=p->nxt;
14:       p->nxt=r;
15:       r=p;
16:       p=q;
       }
17:    #pragma SAP.force(p==NULL)
18:    q=NULL;
19:    list=r;
20:    r=NULL;
       // Delete the list
21:    p=list;
22:    list=NULL;
   L3: while(p!=NULL){
23:       #pragma SAP.force(p!=NULL)
24:       q=p->nxt;
25:       free(p);
26:       p=q;
27:       q=NULL;
       }
28:    #pragma SAP.force(p==NULL)
       return 1;
   }
```

Figure 2.18: Running example to introduce shape analysis operations: iteratively create, reverse and delete a singly-linked list.

2.5.2 Creating new elements

Statement	Function	Brief description
`x=malloc()`	`XNew()`	Nullify pointer `x`; and create a new node pointed to by `x`, initializing the selectors in `x`'s type to `NULL`
`x=NULL`	`XNULL()`	Nullify pointer `x`; and summarize resulting graph.

Pointer programs that use dynamic data structures need to create elements during program execution. These elements can then be interconnected to form recursive structures that are commonly traversed in pointer-chasing loops or recursive function calls. The first step is therefore to be able to create new elements in the heap. In our approach, a new node is created in the graph representation of the heap each time a malloc statement is encountered in the analysis.

Let us consider malloc statement `st.3:q=malloc()` in Fig. 2.18. It creates a new node in our graph-based heap representation. This new node represents the memory location that would have been created at run time by the program in this statement. To make things interesting, let us present this operation at the fourth symbolic iteration of the loop, so four elements have already been created. The new graph that reaches the fourth symbolic iteration of `st.3:q=malloc()` comes from the third iteration of `st.5:p=q`. It is shown as sg^1 in Fig. 2.19. It shows the list created so far, and the `p` and `q` pointers are aliased as a result of `st. 5`. The `malloc` statement calls the `XNew()` function. As a first step, the pointer used to allocate the new memory piece is nullified, by calling `XNULL()`. This function removes the pointer link `pl3=<q,n3>` from the coexistent links sets for `n3`, i.e., $CLS_{n3}=\{cls1_{n3}\}$. At the end, it checks for compatible nodes, summarizing the graph if necessary (there are no compatible nodes in this case). The

result is sg^A in Fig. 2.19, an intermediate step of the abstract semantics for `st.3:q=malloc()`.

Coming back to the `XNew()` function, a new node `n4` is created. The selectors for its type (just `nxt` in this case) are initialized, and a single $cls1_{n4}$ is created, with the pointer link `pl3=<q,n4>` and selector link with attributes $sl5_{att}$=<<n4,nxt,NULL>,o>. The final result is shown as sg^2 in Fig. 2.19.

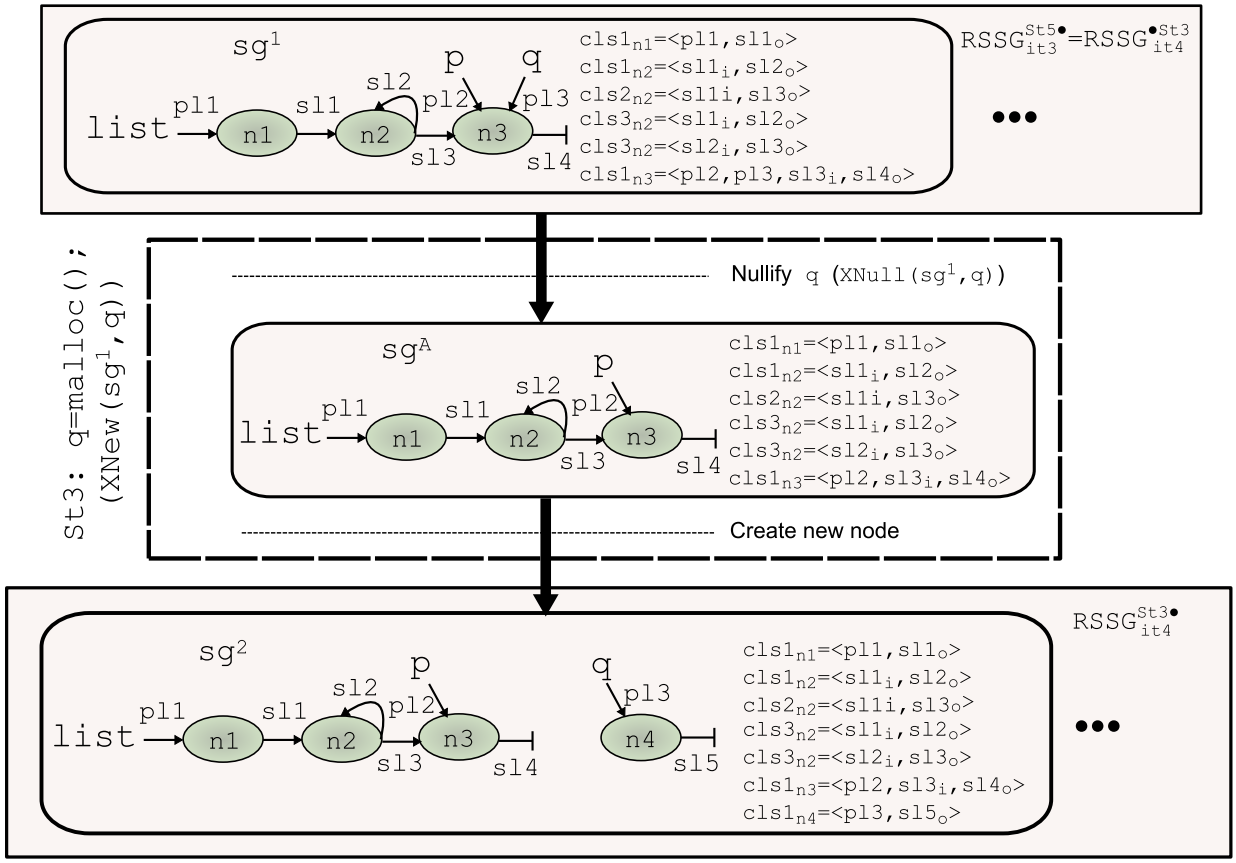

Figure 2.19: Creating a new element through the malloc statement and its associated `XNew()` function.

2.5.3 Creating a recursive data structure

Statement	Function	Brief description
`x->sel=y`	`XSelY()`	Split graph by `x->sel`; nullify `x->sel` link; establish link between node pointed to by `x` to node pointed to by `y` through selector `sel`; and summarize all resulting graphs to form output `RSSG`.
`x=y`	`XY()`	Nullify pointer `x`; and point `x` to the node pointed to by `y`.

Once elements are created in the heap, they can be linked through selectors to form recursive data structures. Let us continue the example after the creation of the fifth element in the list (sg^1 in Fig. 2.20). `St.4:p->nxt=q` links the node pointed by `p` to the newly created node pointed to by `q`, through selector `nxt`. This is done by calling the `XselY()` function. This function starts by splitting the graph by the `p->nxt` path and then nullifying it. Since `p->nxt` leads to `NULL`, this has no effect here. Then, it creates the selector link `sl4=<n3,nxt,n4>`, which is updated as selector link with attributes $sl4_{att}$=<<n3,nxt,n4>,o> in CLS_{n3}, and added as $sl4_{att}$=<<n3,nxt,n4>,i> in CLS_{n4}. The result is sg^2 in Fig. 2.20.

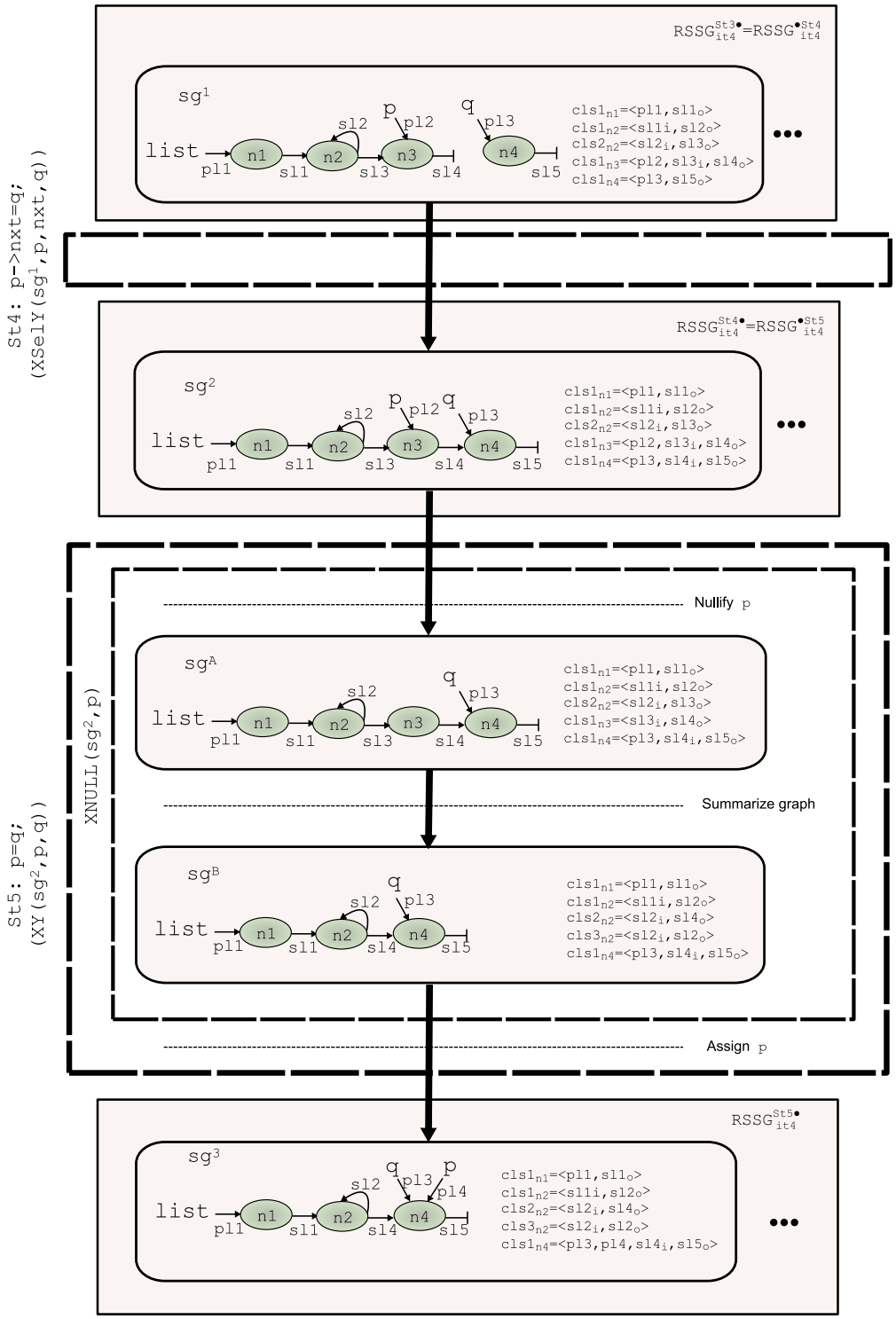

Figure 2.20: Use of the `XSelY()` and `XY()` functions to create a recursive data structure.

Next, pointer `p` is made to point to `q`, advancing in the list, with `st.5:p=q`. This is done by calling the `XY()` function. As a first step, the assigned pointer, `p`, is nullified. For that, the analysis calls `XNULL(sg2,p)`. The result is sg^A (Fig. 2.20), where the pointer link `pl2=<p,n3>` has been removed. sg^A is not in normal form, as `n2` and `n3` are now compatible (not pointed to by any pointers). As part of the `p` nullification process, this circumstance is checked and these nodes are summarized, producing sg^B.

Finally, the abstract semantics of the alias statement `st.5:p=q` is completed by creating the pointer link `pl4=<p,n4>`, and adding it to every coexistent links set in $\texttt{CLS}_{\texttt{n4}}$. The result can be observed as $\texttt{sg}^3$ in Fig. 2.20.

2.5.4 Traversing a recursive data structure

Statement	Function	Brief description
`x=y->sel`	`XYSel()`	Nullify pointer `x`; split graph by `y->sel`; materialize new node from node pointed to by `y->sel`; assign `x` to the materialized node; and summarize all resulting graphs to form output `RSSG`.
`x->sel=NULL`	`XSelNULL()`	Split graph by `x->sel`; materialize new node from node pointed to by `x->sel`; nullify selector `sel` from node pointed to by `x`; normalize graphs (i.e., remove unreachable elements); and summarize all resulting graphs to form output `RSSG`.

A crucial part in pointer-based programs is the traversal of recursive data structures. It is common that pointer applications have sections where recursive data structures are created, and sections where they are traversed while computing some result. Sometimes, traversals also include structure modification such as creation of new elements, deletion of elements or rearranging of links. It is in the traversal of a structure where the information provided by a shape analysis technique is put to the test. In our running example for this section, lines 1–8 involve the creation of the list, while lines 9–20 involve the traversal of the list, where the structure is modified.

Let us skip forward in the analysis of our running example (Fig. 2.18), while we consider the loop `L2`, where the list created in `L1` is reversed. Fig. 2.21 shows the graph that reaches the second iteration of `st.13:q=p->nxt`. The first element of the list is pointed to by `r`, and `p` and `q` are aliased over the second element in the list. `St.12:force(p!=NULL)` is a pseudostatement that prevents graphs where `p` is `NULL` to be analyzed within the loop. As the loop condition in `L2` is `p!=NULL`, it is clear that such graphs do not correspond to realistic memory configurations inside the loop.

`St.13:q=p->nxt`, calls the `XYSel()` function. As a first step, the pointer which is to be assigned, `q`, is nullified by calling $\texttt{XNULL(sg}^{\texttt{1}}\texttt{,q)}$. The result is then split by `p->nxt`. This means that a new working graph will be generated for each possible $\texttt{clsj}_{\texttt{n2}}$ with a selector link with attributes of the kind $\texttt{sl}_{\texttt{att}}\texttt{=<<n2,nxt,na>,attsl=\{o|c\}>}$. Each split graph deletes the information that does not correspond to the link followed. The results are $\texttt{sg}^{\texttt{B}}$ (no more elements after `n2`) and $\texttt{sg}^{\texttt{C}}$ (1 or more elements after `n2`).

After splitting, a new node is materialized from the node pointed to by `p->nxt`. Since $\texttt{sg}^{\texttt{B}}$ points to `NULL` through `p->nxt`, this has no effect for $\texttt{sg}^{\texttt{B}}$. However, in $\texttt{sg}^{\texttt{D}}$ a new node, `n4`, is materialized from `n3`. Node `n3` is a *summary node* that represents different kinds of memory locations (as many as coexistent links sets describe its connectivity). The materialization operation uses all available information within the coexistent links sets in `n3` (and that of properties if there were any) to yield an accurate materialization for `n4`. For instance, we can determine that there is no self-link over `n4` or that a link between `n2` and `n3` must not exist, as `n2` was pointing to *just one* location represented by `n3`. After the materialization of `n4`, the pointer `q` can be made to point to it, creating the pointer link `pl2=<q,n4>`. Finally, the graphs are summarized by calling function `Summarize_RSSG()` (which has no effect in this case), and the resulting $\texttt{RSSG}^{\texttt{St13}\bullet}_{\texttt{it2}}$ is generated.

The example continues by performing a *destructive update* through `st.14:p->nxt=r` (Fig. 2.22). This kind of statement is typical from pointer applications. It involves breaking a link and establishing a

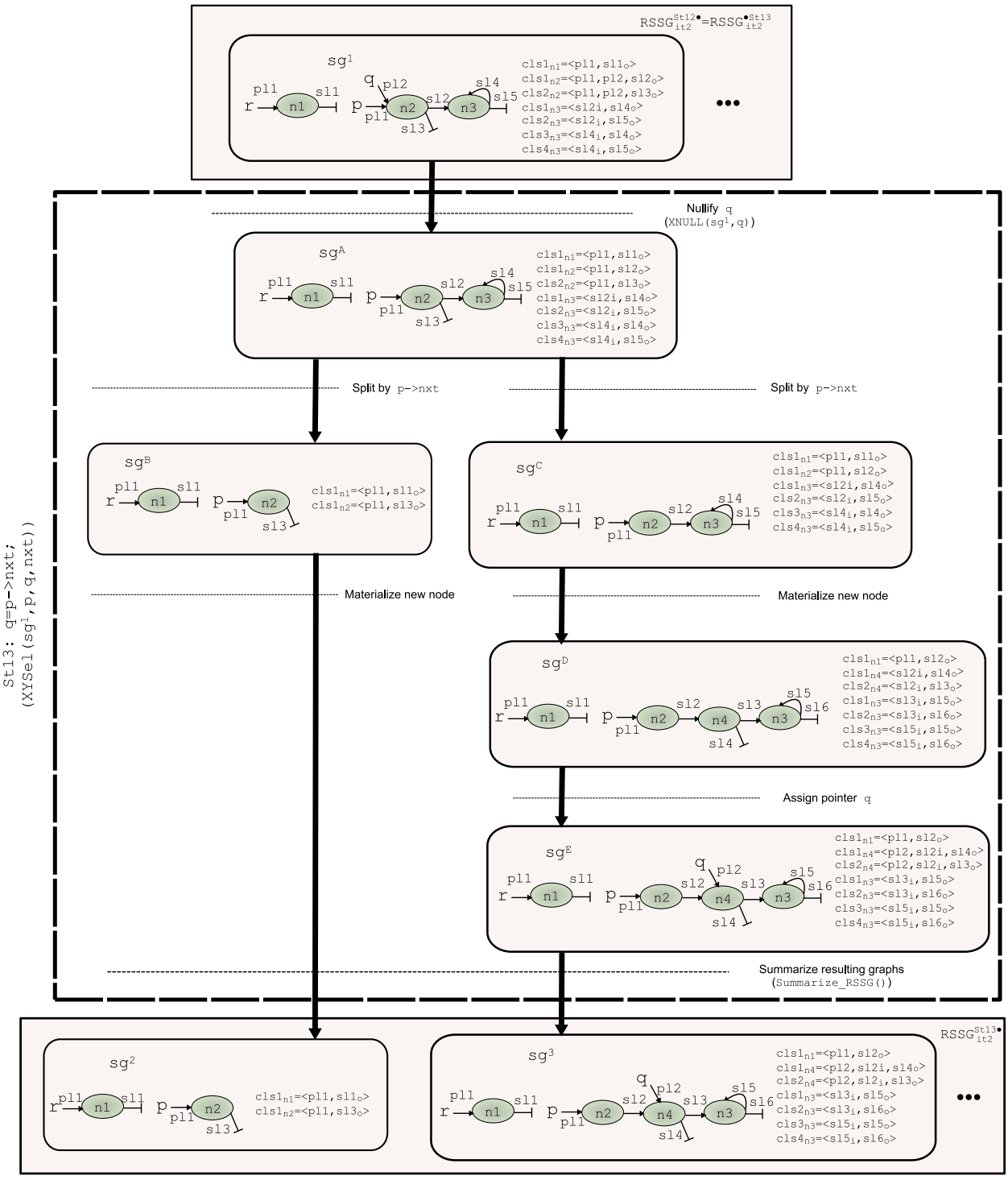

Figure 2.21: Traversing a recursive data structure with the `XYSel()` function.

new one. It is performed by calling the `XselY()` function. First, the graph is split by `p->nxt` so that the nullification of the link can be performed accurately, which is done by the `XSelNULL()` function. Continuing with the $\texttt{RSSG}_{\texttt{it2}}^{\texttt{St13}\bullet}$, we see that the splitting of graph and node materialization have no effect here, as the node pointed to by `p->nxt` is already *focused* and pointed to by pointer `q`. Then, the link between `n2` and `n3` can be nullified. For that, the selector link `sl2=<n2,nxt,n3>` is turned to `sl2=<n2,nxt,NULL>`, updating the set $\texttt{CLS}_{\texttt{n3}}$ in the process. The subsequent summarization of graphs

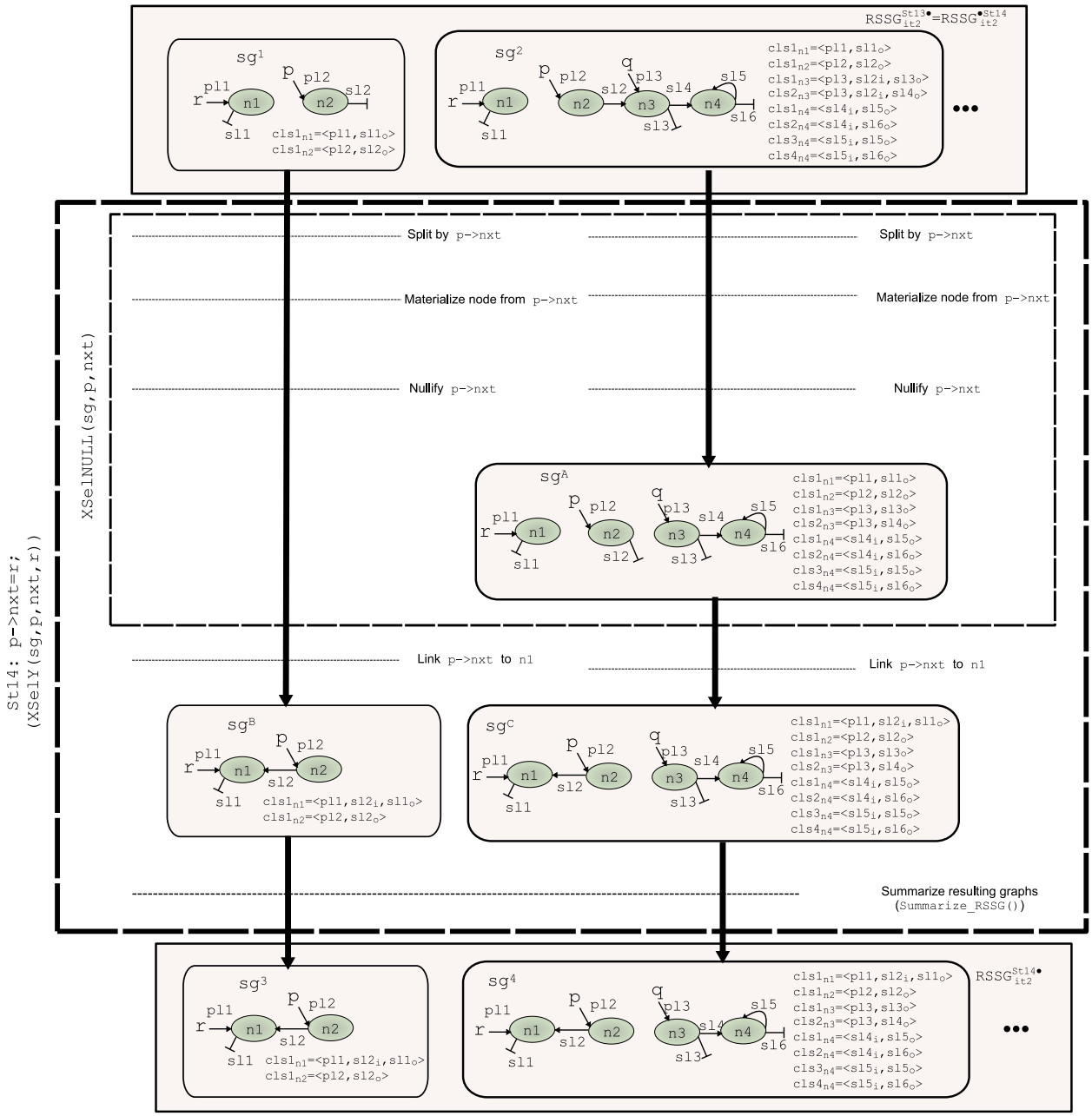

Figure 2.22: Destructive update in a recursive data structure, using the `XselY()` function, and its implicit `XSelNULL()` function.

within `XSelNULL()` has no effect here. After that, the new selector link `sl2=<n2,nxt,n1>` is created and $\mathtt{CLS}_{n1}$ and $\mathtt{CLS}_{n2}$ are updated accordingly. The result does not need to be summarized and $\mathtt{RSSG}^{St14\bullet}_{it2}$ is generated as output for this statement.

2.5.5 Freeing memory

Statement	Function	Brief description
`free(x)`	`FreeX()`	Remove node pointed to by `x`, removing as well inconsistent selector links that point to the freed node, if any.

Once a recursive structure is no longer needed, a program can deallocate the unneeded elements to release resources. Typically a traversal of the structure is performed freeing its elements. Such a traversal is performed in loop `L3` for our running example (Fig. 2.18).

Let us consider the graphs for $\mathtt{RSSG}_{\mathtt{it1}}^{\mathtt{St25}\bullet}$, shown in Fig. 2.23. As results of analysing the first iteration for `st.24:q=p->nxt`, there are two possibilities: $\mathtt{sg}^1$, with no more elements after that pointed to by `p`, and $\mathtt{sg}^2$, where more elements still exist in the list. By calling the `FreeX()` function, the node pointed to by `p` is freed, i.e., removed from the graph, along with its associated coexistent links sets.

It is worth to notice that the `FreeX()` function also removes inconsistent selector links that are pointing to the freed node, if there are any. This is done to preserve coherence of the resulting graphs, and is safe to do as long as we assume *code correctness*. Since our approach is not toward verification, we assume that the code has no memory related bugs and therefore there are no *memory leaks* or *NULL-pointer dereferencing*.

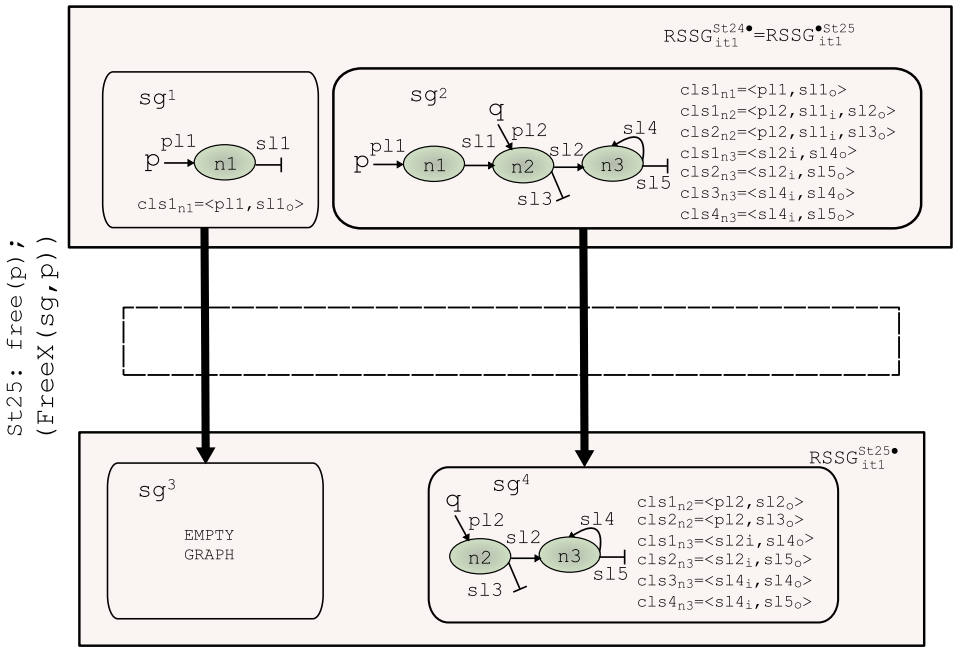

Figure 2.23: Freeing memory using the `FreeX()` function.

2.6 Modeling pointer arrays: multiselectors

Within pointer-based programs, pointer arrays are commonly used. They can be either of a fixed size, as specified by the array declaration in the program, or dynamically allocated at run time with size according to the program input. Whatever the case, we model a pointer array as a node with a *multiselector*. A multiselector is a special kind of selector that may point to several memory locations *simultaneously*, whereas a regular selector can only point to one *single* memory location at a time.

Essentially, a pointer array can be thought of as a summary node. It represents several memory locations that are allocated together in an array. Each of these memory locations has a selector to point to other memory locations. For the node abstracting the pointer array, we just have one selector that can point to an indefinite number of other memory locations abstracted in the graph. In this manner, we are not restricted by the size of the array, which anyway is not known in the case of dynamically allocated arrays.

In the context of coexistent links sets, we may need to complement the meaning of attributes incoming (`i`) and shared (`s`) for a selector link, if it is based on a multiselector. That will be the case when more

than one selector link in the concrete domain is represented by a selector link in the abstract domain. The modified attributes are (i) `im` (*incoming from multiple locations*), which means that, within a summary node, the selector link is incoming on a *one-to-one* basis from the node abstracting the pointer array, and (ii) `sm` (*shared from multiple locations*), which means that the selector link is incoming in a *many-to-one* basis.

The use of the modified `im` and `sm` attributes, along with the idea of using a node and a multiselector to abstract a pointer array are exemplified in Fig. 2.24. This figure shows three variants of a sparse matrix data structure based on pointer array `M`, which points to `struct node` memory locations. Each variant is shown both in the concrete and abstract domains.

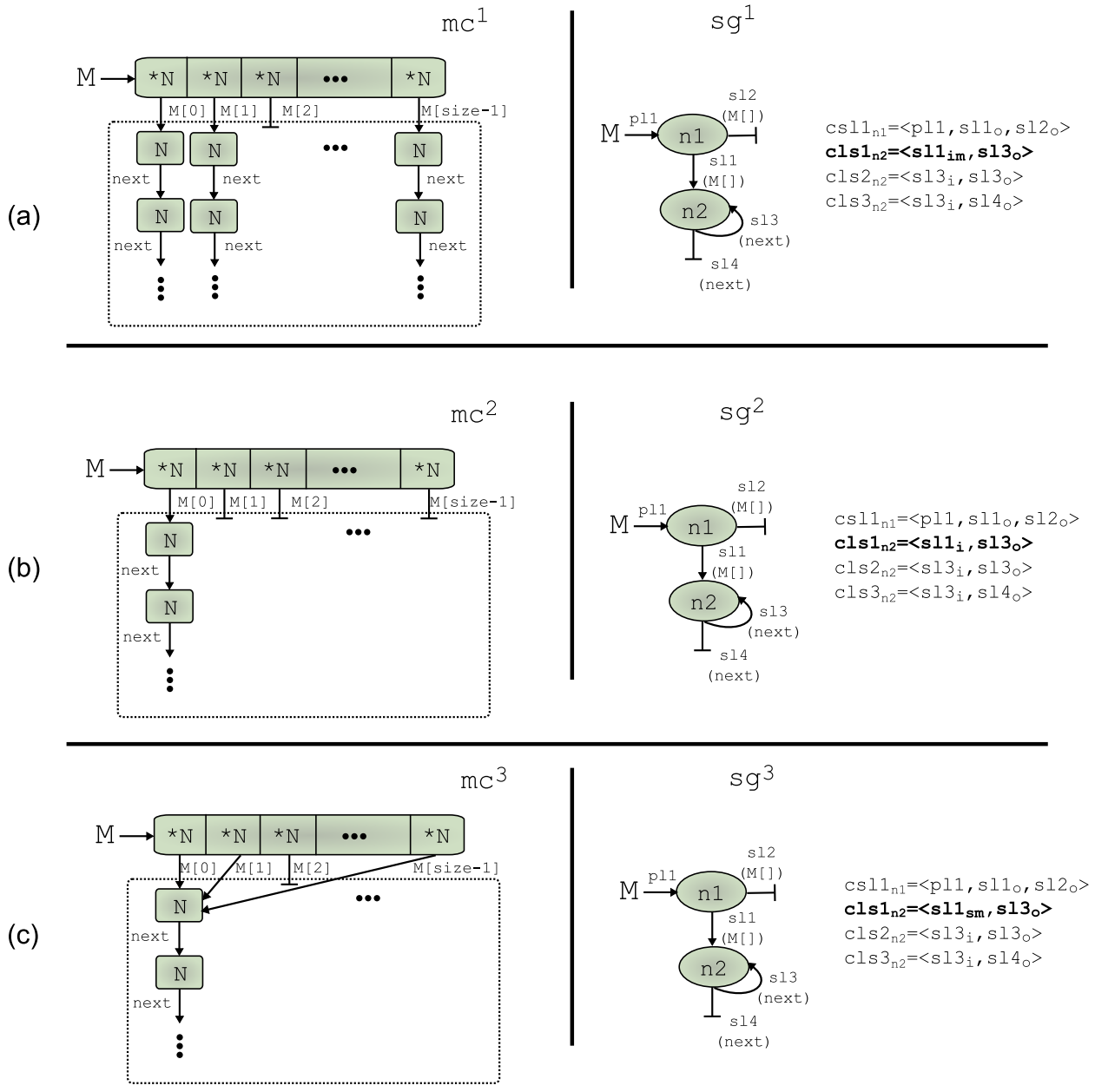

Figure 2.24: Three variants of a sparse matrix data structure based on pointer-array in both the concrete and abstract domains: (a) *one-to-one* relationship for several lists of elements of type `N`; (b) *one-to-one* relationship for just one list; and (c) *many-to-one* relationship for one list.

In `mc1`, each memory location within the array (labeled as `*N`) can point to the head of a list of `struct`

node locations (labeled as N) or to NULL. Shape graph sg^1 shows our abstraction for this first structure: n1 abstracts the M pointer array, and n2 abstract the struct node elements, whereas selector links sl1=<n1,M[],n2> and sl2=<n1,M[],NULL> abstract the two kinds of connections from the array. Coexistent links set $cls1_{n1}$=<pl1,$sl1_o$,$sl2_o$> indicates that, from n1, the multiselector M[] points both to NULL *and* to memory locations abstracted by node n2 (remember that a multiselector can point to several locations simultaneously). However, the key factor here is $cls1_{n2}$=<$sl1_{im}$,$sl3_o$>, which indicates that, from the pointer array abstracted by n1, multiselector M[] may point to several memory locations represented by summary node n2, but it does so on a *one-to-one* basis. This is tantamount to saying that an indetermined number of locations within the array point to the same number of struct node elements, and no struct node element is pointed to by more than one location in the array.

Fig. 2.24(b) displays mc^2, where only the first location within the array points to struct node elements. The rest of locations within the array point to NULL. The abstract representation of this structure is sg^2, which is almost the same as sg^1, except for $cls1_{n2}$=<$sl1_i$,$sl3_o$>, shown in bold. This cls indicates that only one location from the array is pointing to a struct node element.

Finally, Fig. 2.24(c) with mc^3, shows a case where a list of struct node elements is pointed to from more than one memory location in the pointer array. This is reflected in $cls1_{n2}$=<$sl1_{sm}$,$sl3_o$>, which indicates that there is at least one struct node element which is pointed to by multiselector M[] from several locations in the array, effectively creating a *many-to-one* relationship.

The concept of multiselectors to model pointer arrays was introduced in [1]. Here, we adapt its main ideas in the context of coexistent links sets. New statements need to be introduced in our abstract semantics. These are x->msel[i]=NULL, x->msel[i]=y and y=x->msel[i]. They roughly work like the statements x->sel=NULL, x->sel=y and y=x->sel presented before, but with minor extensions to support multiselectors. The x=malloc() statement is sensitive to the kind of location allocated, whether a struct element or a pointer array, initializing the selector/multiselectors accordingly.

2.7 Analysis refinement: properties

During the analysis, memory locations in the heap are merged into *summary nodes* to avoid unbounded recursive data structures, being the summarization criterion to join compatible nodes. Obviously, the node summarization operation may suppose some loss of accuracy. By default, our analysis finds two compatible nodes when the set of pointer links associated with them (i.e., the pointer variables pointing to a node) is the same in both nodes. Let us recall that in our initial abstract heap representation, the abstract domain for the nodes is defined as N=$\{\mathcal{P}$(PTR) $\cup$ $\{$NULL$\}\}$, making the nodes distinguishable only by the set of pointer variables which point to them.

This way of summarizing effectively groups together every node not pointed to by pointers. In certain situations, and depending on the client analysis, this may result in over-conservative shape graphs. In order to avoid aggressive summarizations, we can use *properties*. Properties annotate information in the nodes that is considered by the compatibility check. Nodes whose properties values are not compatible, will not be compatible either with regards to summarization, even if they are pointed to by the same set of pointers.

To introduce the role of properties, we will consider an example structure, drawn from the Em3d benchmark in the Olden suite [30]. It is formed by two singly-linked lists for the electric (E) and magnetic fields (H), respectively. Each element in a list points to the next element in the same list through the next selector, and to other elements in the other list through the to_nodes[] and from_nodes[] pointer arrays. A simplified version of Em3d's structure, showing only the next and to_nodes[] links, is shown in Fig. 2.25(a) for the concrete domain. The to_nodes[] pointer array is used to point to a variable number

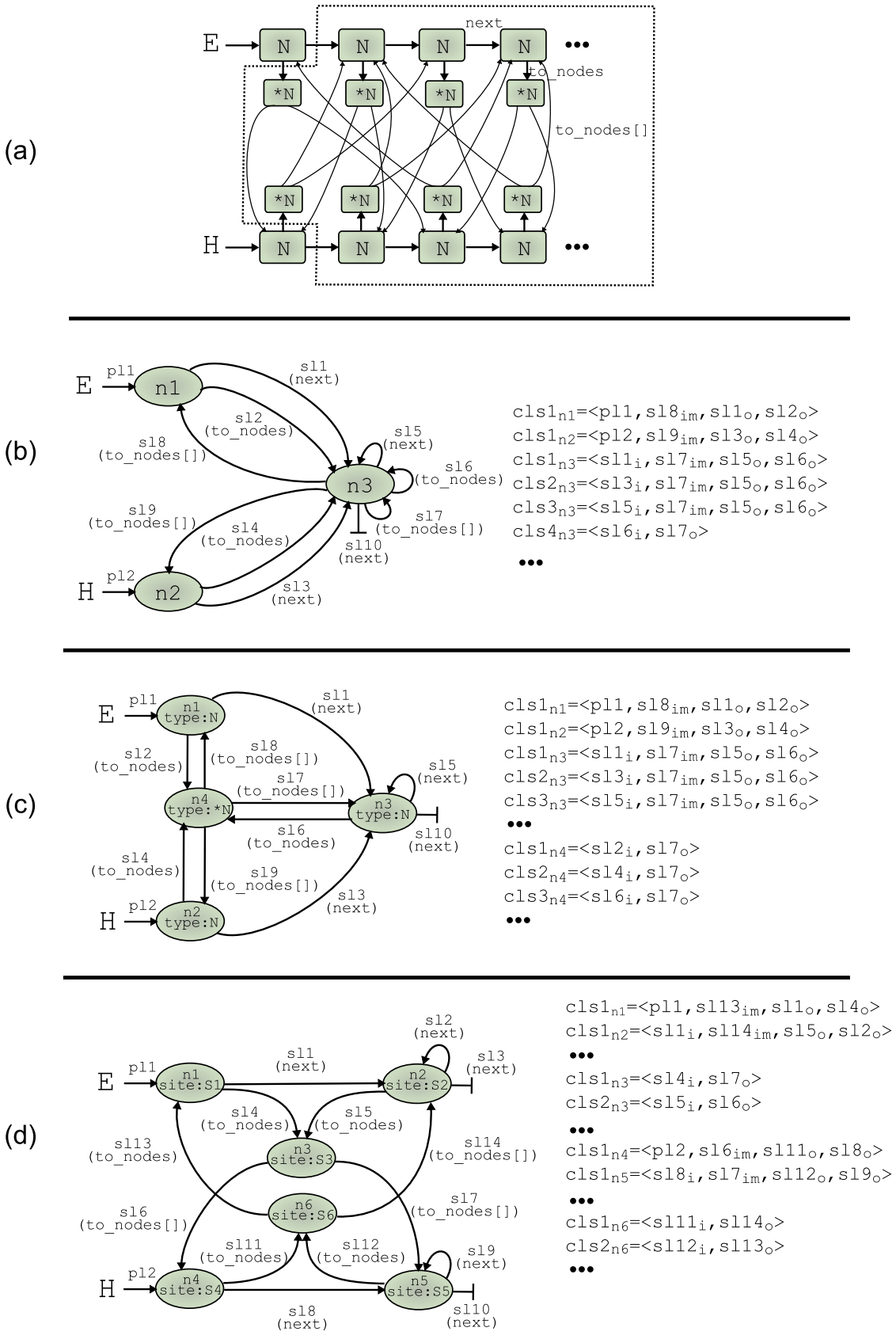

Figure 2.25: `Em3d`'s data structure in the concrete domain (a), and the abstract domain without properties (b), with *type* property (c), and with *site* property (d).

of elements in the opposite list, although for simplicity, we have shown just two connections per array.

It is not determined at compile time what neighboring elements each element will link to, but one important shape characteristic is preserved: the elements in a list only connect through its `to_nodes[]` pointer array to elements in the *other* list, effectively forming a *bipartite* structure. This is the key shape characteristic that allows `Em3d` to be parallelized. Therefore, if we aim to provide some shape abstraction for this structure that is useful for a subsequent dependence analysis, we must be able to capture its bipartite feature.

Fig. 2.25(b) shows the shape graph abstraction of the structure, without properties. All memory locations within the dotted lines in (a) are abstracted by node `n3`. Some `cls`'s are shown for this shape graph. They reflect some interesting characteristics in the structure, namely: (i) the lists are not cyclic through `next`, as there is no `cls` with two incoming `sl`'s for selector `next`; (ii) there are pointer array elements that are reached through `to_nodes` from *one location only* and point to other nodes through multiselector `to_nodes[]` (as described by $\texttt{cls4}_{\texttt{n3}}$=<$\texttt{sl6}_{\texttt{i}}$,$\texttt{sl7}_{\texttt{o}}$>); and (iii) elements in the lists can be reached from various locations within one or several pointer array elements (as described by $\texttt{cls3}_{\texttt{n3}}$=<$\texttt{sl5}_{\texttt{i}}$,$\texttt{sl7}_{\texttt{im}}$,$\texttt{sl5}_{\texttt{o}}$,$\texttt{sl6}_{\texttt{o}}$> for example).

However, we do not have information about the origin of the links to a list element *beyond* the pointer array that points to it. That pointer array could belong to a list element in the other list, which respects the bipartite feature of the structure, or it could belong to the same list, which breaks the bipartite feature. Since we must be conservative according to the information that we have collected in the analysis, it is clear that the shape abstraction in Fig. 2.25(b) is not suitable for a subsequent dependence analysis for `Em3d`.

We would like to refine our abstraction of `Em3d`'s data structure so that we can preserve its bipartite feature. First, let us consider the *type property*, which annotates information about the data type abstracted in the node. The result of abstracting the `Em3d` data structure with the use of the type property is depicted in Fig. 2.25(c). Here the node `n4` now abstracts all the pointer arrays separately from the list elements. Although this graph gives a clearer visual representation of the structure, it does not provide new information. All the characteristics that we discovered in Fig. 2.25(a) still hold, but we cannot yet guarantee that the elements in a list do not link to other elements within the same list through its `to_nodes` pointer array.

Next, we will consider the *site property*, which annotates the nodes with information about the allocation site, specifically its statement number. The shape graph representation of `Em3d`'s data structure with the use of the site property is shown in Fig. 2.25(d). Here, we assume that there is a different allocation site for the elements in the two lists. Within this chapter we focus on intraprocedural shape analysis, therefore to analyze `Em3d` we implicitly assume inlining of functions. In such scenario, the previous assumption clearly holds. Here, the elements in the two lists and the pointer arrays from the elements in the two lists are separated, as they were allocated in different statements of the program (`S1` to `S6` in the figure). This abstraction provides enough separation as to guarantee that the elements in a list only connect to the elements in the other list through its `to_nodes` pointer array (for example, `n3` does not connect back to `n1` or `n2`). Using the site property allows us to preserve the bipartite feature in `Em3d`.

There is another important property that we can introduce here: the *touch property*. It is used to label nodes with access information along the process of abstract interpretation. During the analysis of the traversal of a recursive data structure, the touch property can be used to separate in the abstraction the elements that have been accessed from those not yet accessed. This is crucial for dependence analysis. The use of the touch property for dependence analysis will be further explored in chapter 4.

Depending on the data structure abstracted and the client analyses, different properties might be required for the requested precision. For example, we require the use of the site property for the data dependence analysis of `Em3d`'s data structure. With properties, we can adjust the ability of the technique to accurately

represent complex data structures. However, this comes at a cost: there will be more nodes per graph, as can be seen in Fig. 2.25. Properties can be used isolated or in combination (e.g., type *and* touch).

From a formulation point of view, we define a set of properties PROP={type,site,touch}, where each element prop $\in$ PROP will identify one property that can be incorporated to our analysis through specific compilation flags. Here, we describe the general framework to incorporate these (or even new) properties. For each property, we define new instrumentation domains:

- $\mathtt{P_{type}}$=TYPE is the domain for the property prop=type, and it is defined as a set that contains the type objects declared in the program:

  $\mathtt{P_{type}}$={$\mathtt{p_{type}}$ s.t. $\mathtt{p_{type}} \in$ TYPE}

- $\mathtt{P_{site}}$ is the domain for the property prop=site and it is defined as a set that contains the malloc statements defined in the program:

  $\mathtt{P_{site}}$ = {$\mathtt{p_{site}}$ s.t. $\mathtt{p_{site}}$=s $\in$ STMT $\wedge$ s::=x=malloc()}

  Note that separation of nodes induced by the type property is implicitly included in the site property, as different data types of memory locations are necessarily allocated at different allocation sites. Therefore, we could think of the type property as a more relaxed version of the site property.

- Let ID be the set of identifiers declared during the preprocessing pass of the analysis. These identifiers are defined in touch pseudostatements. $\mathtt{P_{touch}}$ is the domain for the property prop=touch and is defined as a set that contains a set of identifiers:

  $\mathtt{P_{touch}}$ = $\mathcal{P}$(ID) = {$\mathtt{p_{touch}}$ s.t. $\mathtt{p_{touch}} \subset$ ID}

Now, we can extend the definition of the abstract domain for the nodes as N = ($\mathcal{P}$(PTR) $\cup$ {NULL}) $\times$ $\mathtt{P_{type}} \times \mathtt{P_{site}} \times \mathtt{P_{touch}}$. This makes the nodes distinguishable through the set of pointer variables which point to them and the values of the properties annotated in each node. For each property, we can define a mapping function $\mathcal{PPM}_{\mathtt{prop}}$(n) as follows:

$$\text{Property Map}: \quad \mathcal{PPM}_{\mathtt{prop}}: \mathtt{N} \longrightarrow \mathtt{P_{prop}}$$

where, $\forall$ prop $\in$ PROP, $\mathtt{P_{prop}}$ represents the domain for the corresponding property.

The introduction of the node properties, will affect some of the main operations of our analysis, specially those that deal with nodes. In particular, Compatible_Node() is now redefined as shown in Fig. 2.26(a). It checks that two nodes are compatible (and can be summarized) when the set of pointer links is the same in both nodes *and* their properties are compatible. Precisely, this is done by the auxiliary function Compatible_Property() (Fig. 2.26(b)) which checks if property prop $\in$ PROP is compatible for the two nodes n1 and n2. For the *type*, *site* and *touch* properties introduced here, the compatibility criterion is simple: two nodes will be compatible with regards to any of these properties if they have the same value for the property.

2.8 Complexity

In this section, we will focus firstly on the computation of the main parameters which will help us to find the complexity of our method. Let us keep in mind that we are going to compute the worst case behavior. One

```
Compatible_Node()
 Input: n1, n2, CLS_n1, CLS_n2
     # 2 nodes and their CLS's
 Output: TRUE/FALSE

 If ((∀ pl1=<x,n1> ⊂ CLS_n1,
     ∃ pl2=<x,n2> ⊂ CLS_n2) ∧
     (∀ pl2=<y,n2> ⊂ CLS_n2,
     ∃ pl1=<y,n1> ⊂ CLS_n1) ∧
     (∀ prop ∈ PROP,
     Compatible_Property(n1,n2,prop)==TRUE )),
         return (TRUE)
 else
     return (FALSE)
end
```

(a)

```
Compatible_Property()
 Input: n1, n2, prop ∈ PROP
     # Two nodes and a property
 Output: TRUE/FALSE

 If (prop==type ∨ prop==site ∨ prop==touch)
     return PPM_prop(n1) == PPM_prop(n2)
end
```

(b)

Figure 2.26: (a) Check whether two nodes are compatible, incorporating the properties check; (b) Check whether two nodes are compatible with regards to a certain property.

of the parameters of interest, is the maximum number of shape graphs generated by our approach. After a given program statement $s\bullet$, such number of graphs are included in a $RSSG^{s\bullet}$, and it depends on the number of ways of partitioning the live pointer variables at that point. For instance, if the set of live pointer variables is $\{p1, p2, p3\}$, i.e. three live pointer variables, we could find the following shape graphs:

- One graph with one node n1 pointed to by $\{p1, p2, p3\}$.
- Three graphs with two nodes: n1 & n2, pointed to by:
  - $\{p1, p2\}$ & $\{p3\}$
  - $\{p1, p3\}$ & $\{p2\}$
  - $\{p2, p3\}$ & $\{p1\}$
- One graph with three nodes n1 & n2 & n3, pointed to by $\{p1\}$ & $\{p2\}$ & $\{p3\}$, respectively.

We firstly have to compute the number of ways of partitioning a set of j elements (in our case, j live pointer variables) into k blocks (in this case, nodes). Such a number is named the j-th number of Bell, $B(j)$, and can be computed from $B(j) = \sum_{k=1}^{j} S(j,k)$, where $S(j,k)$ is the Stirling number of the second kind [31],

$$S(j,k) = \frac{1}{k!} \cdot \sum_{l=0}^{k} (-1)^l \cdot \binom{k}{l} \cdot (k-l)^j$$

As we are interested in computing the maximum number of shape graphs generated by our approach, we should consider all the possibilities due to different control flow paths, because different paths can establish different alias relationships between pointer variables and let us recall that each shape graph in a RSSG represents a different alias configuration. For instance, a path could generate graphs with just one live pointer variable, another path could generate graphs with two live pointer variables, etc. Assuming that nv represents the maximum number of live pointer variables at any program point, the maximum number of

graphs generated at a point should be the sum of all the ways of partitioning j live pointer variables, from $j = 1$ till $j = nv$, i.e., $\sum_{j=1}^{nv} B(j)$.

We will assume that dead pointer variables are nullified, i.e., as soon as a pointer value is not going to be read before it is re-assigned, it is made to point to `NULL`, so that it does not point to any node. This way it will not contribute to a larger number of shape graphs.

In addition, we should consider the number of properties evaluated in the shape analysis, np, as well as the range of the values for each property p_j, range that we define as $0 : rp_j$. In this case, each value for each property can contribute with a new graph, therefore the number of graphs should be multiplied by $\left[2^{\sum_{j=1}^{np} rp_j}\right]$. In the case that no properties are considered in the analysis, then $np = 1$ and $rp = 0$.

Let us not forget that we are computing the maximum number of shape graphs for a `RSSG` at a program point `s`$\bullet$, i.e. for each statement. With all of this, the **maximum number of graphs per statement**, which we name Ng_s, could be estimated as we indicate in Eq. 2.1. An obvious way to compute the **maximum number of graphs** generated for the analyzed code, which we will name Ng, would be obtained multiplying Ng_s by the number of statements analyzed in the program, $nstmt$, as we see in Eq. 2.2.

$$Ng_s = \left[2^{\sum_{j=1}^{np} rp_j}\right] \cdot \sum_{j=1}^{nv} B(j) \tag{2.1}$$

$$Ng = nstmt \cdot Ng_s = nstmt \cdot \left[2^{\sum_{j=1}^{np} rp_j}\right] \cdot \sum_{j=1}^{nv} B(j) \tag{2.2}$$

There are other interesting parameters that give us more detailed information about how complex the shape graphs are and that are measurable: for instance how many nodes does a graph have and how interconnected these nodes are. About the number of nodes, we are interested in computing an upper bound, i.e. the maximum size of a shape graph. In other words, the **maximum number of nodes per graph**, which we will name Nn. It depends on the maximum number of live pointer variables, nv, because, in a worst case, when none of the pointers are aliased, then each one could point to a different node. Nn depends too on the number of properties considered, np and the range of the values for each property p_j, i.e. $0 : rp_j$, because each value for each property can contribute as a new node. With all of this, Nn can be estimated as we show in Eq. 2.3.

$$Nn = nv + 2^{\sum_{j=1}^{np} rp_j} \tag{2.3}$$

About how interconnected the nodes are, we should compute the maximum number of `sl`'s -selector links- and the maximum number of `cls`'s (coexistent links sets), which are precisely the parameters that encode this information in our approach. We will name the **maximum number of `sl`'s per node**, as Nsl_{node} and the **maximum number of `sl`'s per graph**, as Nsl. The former depends on the maximum number of selector or pointer fields declared in the most complex data structure, nl. It depends too on the maximum number of nodes, to which any node can be connected through a selector link, i.e. $Nn - 1$. As the links that can coexist in a given node can be incoming from any other node, outgoing to any other node, or a link to/from itself, then the maximum number of selector links of a given type could be $2 \cdot Nn - 1$. Therefore, Nsl_{node} can be computed as we see in Eq. 2.4. $Nsl_{node}(Nn)$ denotes the maximum number of selector links when we consider that the number of nodes is Nn. The maximum number of `sl`'s per graph should be the sum of all the selector links per node when we iteratively incorporate $Nsl_{node}(j)$ for each new node, from $j = 1$ till Nn, as we see in Eq. 2.6.

$$Nsl_{node} = Nsl_{node}(Nn) = nl \cdot (2 \cdot Nn - 1) \quad (2.4)$$

$$Nsl = \sum_{j=1}^{Nn} Nsl_{node}(j) = \sum_{j=1}^{Nn} nl \cdot (2 \cdot j - 1) = \quad (2.5)$$

$$= nl \cdot (2 \cdot Nn - 1) \cdot (Nn - 1) \quad (2.6)$$

However, the most important parameter is the maximum number of `cls`'s. A `cls` contains pointer links and selector links with attributes. As a shape graph represents a particular alias configuration, the number of pointer links is fixed. The variations come from the selector links with attributes. For instance, for a node, the maximum number of selector links with attributes depends on the combination of the maximum number of selector links that can coexist in the node (excluding the links from/to itself, i.e. $2^{Nsl_{node}-nl}$, see Eq. 2.4), as well as the number of variations due to the attributes, 5^{nl}. Let us see this last factor in detail: in a `cls` there could be five different states representing the attributes for each selector link from/to the same node: (i) the selector link does not appear, (ii) it is just incoming (`attsl={i} or attsl={s}`), (iii) it is just outgoing (`attsl={o}`), (iv) it is just cyclic (`attsl={c}`) and (v) it is a summary node with the same incoming and outgoing link (`attsl={i,o}`, `attsl={i,c}`, or `attsl={s,o}`, `attsl={s,c}` for a shared summary node). With all of this, we could compute the **maximum number of `cls`'s for a node**, named $Ncls_{node}$, by Eq. 2.7. Clearly, the **maximum number of `cls`'s per graph** named $Ncls$, can be computed from Eq. 2.7 and Nn (the maximum number of nodes) as we see in Eq. 2.8.

$$Ncls_{node} = \left(2^{Nsl_{node}-nl}\right) \cdot 5^{nl} = \left(2^{2 \cdot nl \cdot (Nn-1)}\right) \cdot 5^{nl} \quad (2.7)$$

$$Ncls = Ncls_{node} \cdot Nn = \left[\left(2^{2 \cdot nl \cdot (Nn-1)}\right) \cdot 5^{nl}\right] \cdot Nn \quad (2.8)$$

Eq. 2.7 is a first approximation that gives us a worst case upper bound for the estimation of the maximum number of `cls`'s for a node when there is not available information about the data structures. However, such a number can be greatly reduced when we have some information about the data structures. Till now, we have assumed that all the selector links can be incoming to and outgoing from a node. But, in a `cls` that represents a real data structure, there is as most, a maximum number of "real" incoming selector links. We will call nli to this important piece of information. For instance, in a singly-linked list $nli = 1$, in a doubly-linked list $nli = 2$, or in a binary tree $nli = 1$. With this information we have to compute all the `cls`'s that are combinations due to the selector links with attributes that are incoming in a node, multiplied by combinations due to the selector links with attributes that can be outgoing from the node. In a node, we know that there could be at most: (a) $nl \cdot (Nn - 1)$ selector links from other (different) nodes (cases in which attribute is {`i`} or {`o`}), plus (b) nl selector links from the same node with attribute `c`, plus (c) nl selector links from the same node that represent incoming and outgoing in a summary node (cases in which attributes are {`i,o`} or {`s,o`} or {`i,c`} or {`s,c`}). Thus, there could be $nl \cdot (Nn + 1)$ selector links with attributes in a node. From them, at most, only nli would appear as incoming selector links in a `cls`. Therefore, the computation of the combination of the selector links with attributes that are incoming in a node yields the following:

$$\sum_{j=1}^{nli} \binom{nl \cdot (Nn + 1)}{j}$$

From the $nl \cdot (Nn+1)$ selector links with attributes that there could be in a node, we know that in a `cls` there could be from 0 till nl outgoing links. Thus, for the computation of the combination of the selector links with attributes that are outgoing from a node we consider the expression:

$$\sum_{k=0}^{nl} \binom{nl \cdot (Nn+1)}{k}$$

In other words, a more accurate estimation for the computation of the maximum number of `cls`'s, $Ncls_{node}$, is given by Eq. 2.9. Again, the maximum number of `cls`'s per graph, named $Ncls$, can be computed from Eq. 2.9 and the maximum number of nodes, Nn, as we see in Eq. 2.10.

$$Ncls_{node} = \sum_{j=1}^{nli} \binom{nl \cdot (Nn+1)}{j} \cdot \sum_{k=0}^{nl} \binom{nl \cdot (Nn+1)}{k} \quad (2.9)$$

$$Ncls = Ncls_{node} \cdot Nn \quad (2.10)$$

For instance, working with a singly-linked lists, we know that $nl = 1$ and $nli = 1$, so applying Eq. 2.10 we could get $O(Nn^3)$ as the maximum number of different `cls`'s per graph. With a doubly linked list, where $nl = 2$ and $nli = 2$, for Eq. 2.10 we could get $O(Nn^5)$, whereas for a binary tree we should get $O(Nn^4)$.

To take into account the effect of multiselectors in the maximum number of `cls`'s in a node, we would need to consider the $ninst$ factor for the term describing the maximum number of selector links with attributes in a node, i.e., $ninst \cdot nl(Nn+1)$. The factor $ninst$ stands for the maximum number of *instantiations of multiselectors* that may occur in the program [32], being $ninst = 1$ for no instantiations.

Other parameter of our abstraction, that could be interesting to compute is the **maximum number of `pl`'s per node**, and we will name it as Npl_{node}. It depends on the number of live pointer variables, nv, and it can be easily computed as we can see in Eq. 2.11. The **maximum number of `pl`'s per graph**, named Npl, is represented in Eq. 2.12. As we assume that any `RSSG` will be in normal form, then each pointer variable can appear only once on each graph, therefore $Npl = Npl_{node}$.

$$Npl_{node} = nv \quad (2.11)$$

$$Npl = Npl_{node} = nv \quad (2.12)$$

Table 2.1 summarizes the main parameters used in our complexity study, as well as their definitions and their values.

Now, our goal is to estimate the worst theoretical performance of our shape analysis framework. Roughly, the cost of analyzing a pointer statement will depend on the cost of the corresponding transfer function, and more concretely it will depend on the operations that the transfer function invokes. We would like to start summarizing the dominant costs for the main operations that our transfer functions call. These costs can safely be deduced from the algorithms presented in Appendix A. For the estimation of these dominant costs, we assume a worst case scenario: each shape graph contains the maximum number of nodes (Nn), the maximum number of `sl`'s (Nsl) and the maximum number of `cls`'s ($Ncls$). Let us see then the costs for the main operations:

- The `Summarize_SG()` operation has a computational cost given by $O(Nn + Nn \cdot Ncls_{node})$, due to the fist and second forall, respectively . We can easily deduce, that the dominant cost for this operation can be estimated as $O(Nn \cdot Ncls_{node}) = O(Ncls)$.

Parameter	Definition	Value
$nstmt$	number of statements to be analyzed	
nv	maximum number of live pointer variables at any program point	
nl	maximum number of selectors - or pointer fields- declared in the data structures	
nli	maximum number of "real" incoming links in the data structures	
np	number of properties considered in the shape analysis	by default 1
rp_j	upper value in the range of the values for property j, $0 : rp_j$	by default 0
Ng_s	maximum number of graphs per statement `s`	Eq. 2.1
Ng	maximum number of graphs	Eq. 2.2
Nn	maximum number of nodes per graph	Eq. 2.3
Nsl_{node}	maximum number of `sl`'s per node	Eq. 2.4
Nsl	maximum number of `sl`'s per graph	Eq. 2.6
$Ncls_{node}$	maximum number of `cls`'s per node	Eq. 2.9
$Ncls$	maximum number of `cls`'s per graph	Eq. 2.10
Npl_{node}	maximum number of `pl`'s per node	Eq. 2.11
Npl	maximum number of `pl`'s per graph	Eq. 2.12

Table 2.1: Parameters of our complexity study.

- The `Normalize_SG()` operation depends basically on two findings: (i) find unreachable nodes, which has a cost of $O(Nn \cdot log(Nn))$ and (ii) find `cls`'s with incoherent selector links, which has a cost of $O(Ncls \cdot log(Ncls))$. In other words, the computational cost is dominated by $O(Nn \cdot log(Nn) + Ncls \cdot log(Ncls))$. As we know from Eqs. 2.3 and 2.10, $Ncls >> Nn$, therefore, the cost of this operation is dominated by $O(Ncls \cdot log(Ncls))$.

- The `Split()` operation depends on finding a node and then creating a new graph for each `cls` of that node. When creating the new graphs, the `Normalize_SG()` function is called. Clearly, it presents a cost given by $O(Nn + Ncls_{node} \cdot (Ncls \cdot log(Ncls)))$. Simplifying, The dominant cost of this operation can be expressed as $O(Ncls_{node} \cdot (Ncls \cdot log(Ncls)))$.

- The `Materialize_Node()` operation has a cost of $O(2 \cdot Nn + 2 \cdot Ncls_{node})$ for the two first nodes finding and the creation of the `cls`'s of the new materialized node (the `Create CLS'`$_{\texttt{nm}}$ forall). Next, the `Create CLS'`$_{\texttt{nj}}$ forall has a cost given by $O(Ncls_{node} \cdot Nsl_{node})$, whereas the `Create CLS'`$_{\texttt{nk}}$ forall presents a cost given by $O(Nn \cdot Ncls_{node} \cdot Nsl_{node})$. Finally, a call to the `Normalize_SG()` function will have a cost of $O(Ncls \cdot log(Ncls))$. In summary, the cost of the materialization is given by $O(2 \cdot Nn + 2 \cdot Nncls_{node} + Ncls_{node} \cdot Nsl_{node} + Nn \cdot Ncls_{node} \cdot Nsl_{node} + Ncls \cdot log(Ncls))$. As $Nn \cdot Ncls_{node} = Ncls$, and from Eqs. 2.4 and 2.10 we deduce that $Nsl_{node} < log(Ncls)$, we can approximate the dominant cost for this operation as $O(Ncls \cdot log(Ncls))$.

Now that we know the dominant costs of the main operations, we could estimate the costs for the transfer functions. In Appendix A we present the formulation of our operations as functions. The goal of such formulation is to present in a clear and formal context, the operations involved in our analysis. However, we should remark that those functions are different from our real implementations. In other words, the dominant cost of each transfer function depends on the algorithm implemented. We present here a short indication of these costs. For the estimation of these dominant costs, we have assumed again a worst case scenario: the maximum number of shape graphs included in a `RSSG`$^{\bullet\texttt{s}}$ is Ng_s (see Eq. 2.1). In the computation

of the dominant costs of our real implementations of the transfer functions we have included the operator $\bigsqcup^{\mathtt{RSSG}}$ which roughly has a cost given by $O(Ng_s)$. For instance, the statements `x=NULL`, `x=new` and `x=y` call to the `Summarize_SG()` operation. In our implementation, the cost for these statements is given by $O(Ng_s \cdot Ncls)$. However, the statements `x->sel=NULL`, `x->sel=y` and `x=y->sel` call to the `Split()`, `Materialize_Node()` and `Normalize_SG()` operations and, roughly, they present a cost given by $O(Ng_s \cdot Ncls \cdot log(Ncls))$. Clearly, the complexity is dominated by the transfer function of these last statements, so our method has a complexity of $O(Ng_s \cdot Ncls \cdot log(Ncls))$.

The fixed point requires that the transfer functions be applied until the graphs in $\mathtt{RSSG}^{\mathtt{s}\bullet}$ do not change any more. However, we have considered the maximum number of possible graphs, nodes, `sl`'s and `cls`'s so the complexity to reach the fixed point is included in the previous discussion.

Summarizing, we find that the complexity of our approach depends on the upper bounds of Ng_s and $Ncls$. From Eq. 2.10 we know that $Ncls$ has a polynomial behavior: $O(Nn^3)$ for a singly-linked list, $O(Nn^5)$ for a doubly linked list, ... Ignoring the properties, from Eq. 2.3 we know that $Nn = nv + 1$. Therefore, roughly we can approximate an upper bound for the $Ncls$ parameter as $O((nv)^k)$, where k is a constant that depends on the maximum number of links in the structures analyzed, and nv is the maximum number of live pointer variables. On the other hand, from Eq. 2.1 which represent the theoretical maximum value for Ng_s, again ignoring the properties, we can notice that depends on the sum of the numbers of Bell, $\sum_{j=1}^{nv} B(j) < nv \cdot B(nv)$. From [33], we know that the asymptotic limit of numbers of Bell is,

$$B(nv) < \frac{1}{\sqrt{(nv)}} \cdot (\lambda(nv))^{nv+1/2} \cdot e^{\lambda(nv)-nv-1}$$

being $\lambda(nv) = \frac{nv}{W(nv)}$, with $W(nv)$ as the Lambert W-function. That limit, very roughly is much lower than nv^{nv}, so we can approximate an upper bound of Ng_s as $O(nv \cdot nv^{nv})$. In other words, taking into account the upper bounds for $Ncls$ and the Ng_s parameters, our approach would have a exponential behavior given by $O(nv^{nv+k})$, as a worst case. However, we think that the important issues are: is the worst case reached in practice, and how often? We will address these questions in section 2.10.

2.9 Related work in heap analysis

In the past few years pointer analysis has attracted a great deal of attention. A lot of studies have focused on stack-pointer analysis, like [34] and [35], while others, more related to our work, have focused on heap-pointer analysis. Both fields require different techniques of analysis.

We can find methods based on deriving information from the program to describe the heap as predefined structures. Among them, we acknowledge the work of Ghiya and Hendren [36], and Hwang and Saltz [37]. This kind of approach is called *storeless*, as it does not keep a representation for the heap in every statement in the program.

Ghiya and Hendren [36] use access paths in path matrices to give a coarse characterization of structures in the heap. They use matrices to encode information about paths from one pointer to another, and to record possible interferences of heap objects through different pointers. The information derived from the matrices is used to estimate the shape of the data structure accessible from a pointer, sorting it as *Tree*, *DAG* or *Cycle*. The case for *Cycle* stands for any structure containing cycles, with no further distinction.

Hwang and Saltz [37] use the base classification from Ghiya and Hendren for a primary characterization of the heap. They also build def-use chains for the pointers in the program to derive information from the "shape" of the traversal of the structure. This allows them to identify non-cyclic traversals of cyclic structures, for example. However, the analysis does not work if the structure is modified in the loop of

study.

There is another body of work ([38], [39], [40], [41]) that uses separation logic to describe the shape of data structures through recursively defined predicates. Some of these works rely on pre-defined recursive predicates [38], [41], whereas [40],[39] resort to inductive synthesis to infer recursive shape invariants. [40] is suitable for list-processing programs limiting the class of analyzable programs, whereas [39] can handle more general data structures, specially data types with tree-like backbone.

More related to our approach though, there are a number of techniques that use some variety of graph to maintain information from the heap. These approaches are usually referred to as *stored-based*, because some representation is kept for the objects allocated in the heap for every analyzed statement.

Some early works progressed in the understanding of the requirements for precise heap abstractions. For example, Jones and Muchnick [42] use a set of graphs for a heap abstraction in a simple language. There are labels for sharing and cycles. Graphs are bounded by *k-limiting*, i.e., elements beyond *k* indirections are summarized. Larus and Hilfinger [43] extend on k-limited graphs by introducing path expressions to build so-called *alias graphs*. Horwitz, Pfeiffer, and Reps [44] work with storage graphs, another variation of k-limited graphs used to determine conflicts between heap statements. Chase, Wegman, and Zadeck [28] do not rely on k-limiting to bind their *Storage Shape Graphs*, but rather, create a node for every pointer and for every allocation site in the program. They are able to perform *strong update* in certain situations, a process that may exchange links in a node in a definite way. Later, Plevyak, Chien, and Karamcheti [26] extend the model of Chase et. al to manage cycles with the *Abstract Storage Graphs*. In general, all these techniques suffer from serious limitations in the kind of analyzable structures, and perform very complex operations. Besides, the definite actions like *strong update* are rare, more so as information becomes summarized during the analysis.

Sagiv, Reps, and Wilhelm [27] improve on these previous efforts. They also make use of graphs to capture the heap. However, their *Static Shape Graphs* differentiate nodes by the pointers that point to them. They are able to perform *strong nullification* in all situations thanks to a very precise materialization operation. Overall, they provide significant improvement over the previous works but they still are unable to correctly analyze complex structures, and the analysis complexity hinders scalability.

Later, Sagiv, Reps, and Wilhelm [29] devised a method to instantiate different shape analyzers according to a set of specified *predicates*. Such predicates can take one of three values: *true*, *false*, or *unknown*, hence the 3-valued logic that is used as the core of the method. This work produced an implementation known as TVLA (Tri-Valued Logic Analyzer) [45], which spawned several shape analysis strategies ([46], [47], [48], [49], [50]), each using a specific set of predicates for specific purposes. The main works stemmed from TVLA will be discussed in the next chapter, as they use the base 3-valued analyzer approach with extensions to deal with interprocedural analysis.

Marron et. al [51], [52] combine some of the previous ideas to come up with another graph based shape analysis technique. From Ghiya and Hendren [36], they borrow the idea of specifying predefined shapes, albeit with a greater range of shapes: *singleton*, *list*, *tree*, *multipath* and *cycle*. From Sagiv et. al [27] they use the ideas of abstract interpretation, abstract semantics for pointer statements, graph operations, and materialization (now called *refinement*). They are able to obtain fast analysis times for relatively complex structures in Java programs modified to use collection libraries that have been previously characterized for the analysis.

Hackett and Rugina [53] describe a novel shape graph abstraction built on top of a region points-to analysis. Their analysis is based on local reasoning over individual heap locations, called *tracked* locations. By avoiding to analyze the heap as a whole, they can achieve a fast analysis (less than a minute for tens of thousands of C code for a bug detection extension). However, their analysis relies heavily on the underlying points-to analysis. If it cannot determine enough disjoint regions, the analysis is worthless.

The work by Sagiv et al. [27] was the first to successfully combine abstract interpretation based on abstract semantics for pointer statements in a framework that incorporated the materialization operation for *strong nullification*. These concepts were used as the base for the work of Corbera, Asenjo, and Zapata [54]. They augmented the analysis precision by adding several graphs per statement and introduced the use of properties in nodes to be able to separate summary nodes. Besides, this work provided explicit support for pointer arrays, modeled as multiselectors, an aspect neglected by previous works.

The shape analysis technique described in this chapter is inspired in the work by Corbera et al. [54], [1]. It borrows the main ideas of abstract interpretation, worklist algorithm, summarization/materialization, several graphs per statement, use of properties to fine tune the analysis precision,and modeling of pointer arrays as multiselectors. The main contribution here regarding this previous work and the intraprocedural aspect of shape analysis is the addition of the coexistent links set abstraction for a more compact representation, and a new Java-based implementation ready for further development. It should be noted that our approach does not suppose any characteristic of the data structure nor does it assume the behavior of operations over it.

2.10 Experimental results

We have implemented the algorithms presented in this chapter within our heap analysis framework, written in Java. We focus on the analysis of C sequential programs. All the needed preprocessing passes are performed with custom-made passes built upon Cetus [55], as described in chapter 1.

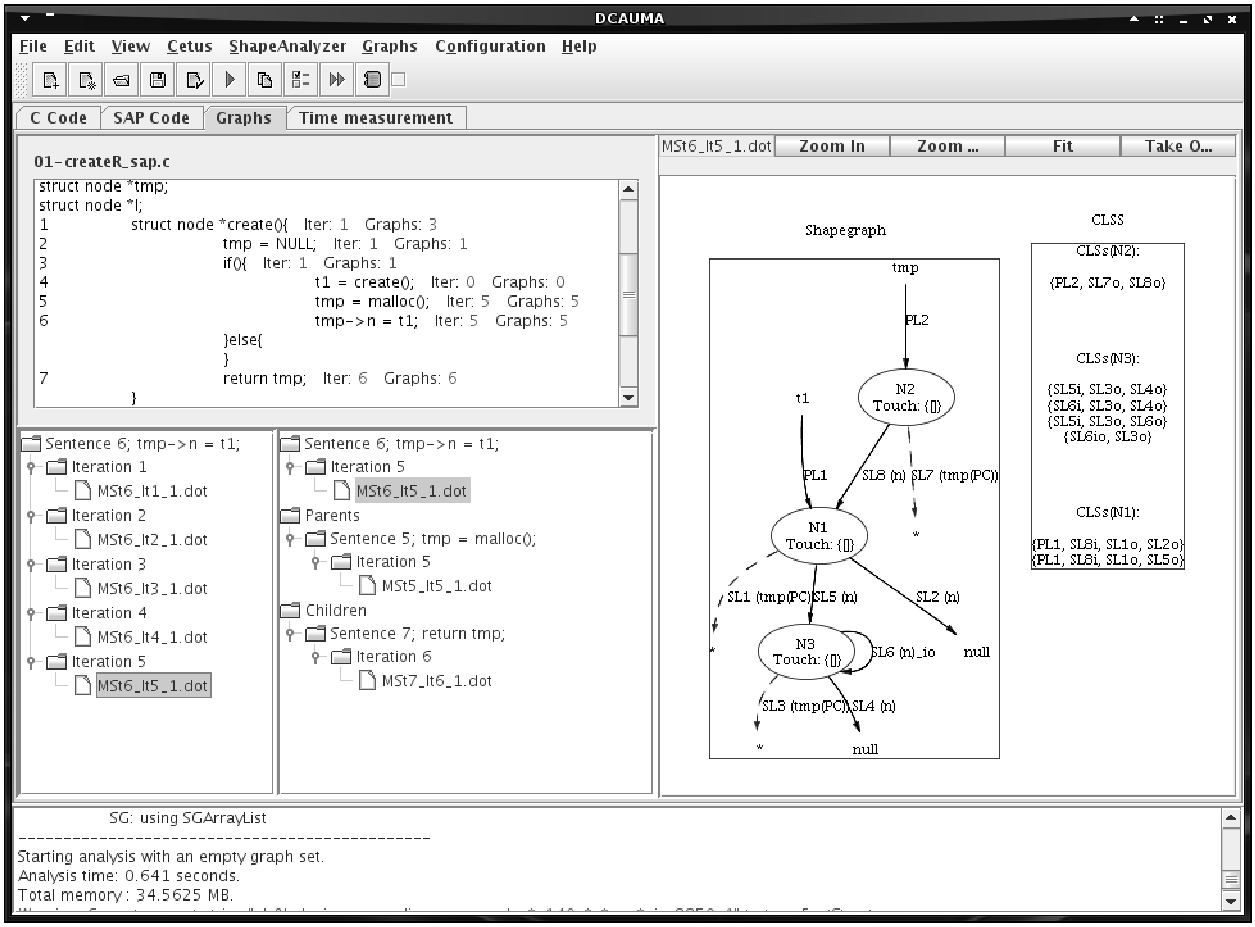

Figure 2.27: Graphical User Interface for shape analysis.

We have also implemented a GUI to enable a friendly use of our shape analyzer tool. In Fig. 2.27 we can see one of the available windows in which the "Graphs" tab is selected. In that tab we have the analyzed code with each statement annotated with information regarding the number of times that statement has been symbolically executed and the number of `sg`'s associated with it. We also provide the links to each graph and information about the parents and children of each one of them, as well as the graphical view of the graphs and its `cls`'s. There is also a "SAP code" tab (SAP stands for Shape Analyzer Preprocessing). This tab is very useful to compare the original C code and the preprocessed version resulting from the Cetus compiler pass. This preprocessing takes care of the insertion of force pseudostatements and other transformations and simplifications that have to be performed to optimize the shape analysis.

2.10.1 Benchmarks and tests

We have considered six programs for the tests in this section. The structures they use include singly-linked lists, binary trees, n-ary trees, sparse matrices, sparse vectors, and highly interconnected bipartite graphs. We outline them next:

- `1-Running example`. This benchmark analyzes the running example used for this chapter, i.e., the program in Fig. 2.18. It creates, traverses, and then deletes a singly-linked list. The singly-linked list is depicted in Fig. 2.28(a). The purpose of this benchmark is to provide a baseline for the allocation, traversal and disposal of a simple recursive data structure.
- `2-Binary tree`. This benchmark creates and traverses a binary tree, a very common data structure used in pointer-based applications, shown in Fig. 2.28(b). The tree is created as it is traversed from the root within a loop. The resulting tree needs not be balanced, i.e., in general not all leaves will be found at the same depth of the tree.
- `3-N-ary tree`. Creation and traversal of a tree based on pointer arrays for its children. The number of children for each tree element is not known at compile time. Fig. 2.28(c) depicts this data structure. Like the previous benchmark, the tree is created as it is traversed from its root within a loop.
- `4-Matrix x Vector`. Creation of a sparse matrix `M` and a sparse vector `V`, plus the generation of the output vector `R=MxV`. The output vector is created along the traversal of `M` and `V`. Sparse structures are those where the majority of its elements are zero. This kind of structures are usually represented as pointer-based data structures, where only the non-zero elements are stored, with some additional information for the location of the element within the structure. For example, for the sparse matrix, each element has its value within the matrix, but also additional information for its row and column. Note that this is different from a matrix representation as a bidimensional array, where all zero elements would consume storage in memory. An example of a sparse matrix and a sparse vector is shown in Fig. 2.28(d). This benchmark is a kernel of common programs manipulating sparse matrices. The product itself features three nested loops.
- `5-Matrix x Matrix`. Creation of sparse matrices `M1` and `M2`, plus the generation of the output matrix `M3=M1xM2`. The output matrix is created along the traversal of `M1` and `M2`. The sparse matrices used are of the same data type as the matrix used for the `4-Matrix x Vector` benchmark. This benchmark is used to help understand the effect of adding more complexity to the data structure *and* the control flow of the program, with regards to the `4-Matrix x Vector` benchmark, as the product now features three matrices and a nest of four loops.
- `6-Em3d`. Program from the Olden suite [30], that creates two singly-linked lists for the electrical and magnetic fields, and then links each element in a list to several elements in the other list, creating a

bipartite graph. This example has been used in the literature [37], [51], as a key example for shape analysis, due to its heavily interconnected structure. The main data structure is shown in Fig. 2.28(e).

These codes do not include functions. In the case of `6-Em3d`, functions were inlined by a preprocessing pass into a single body.

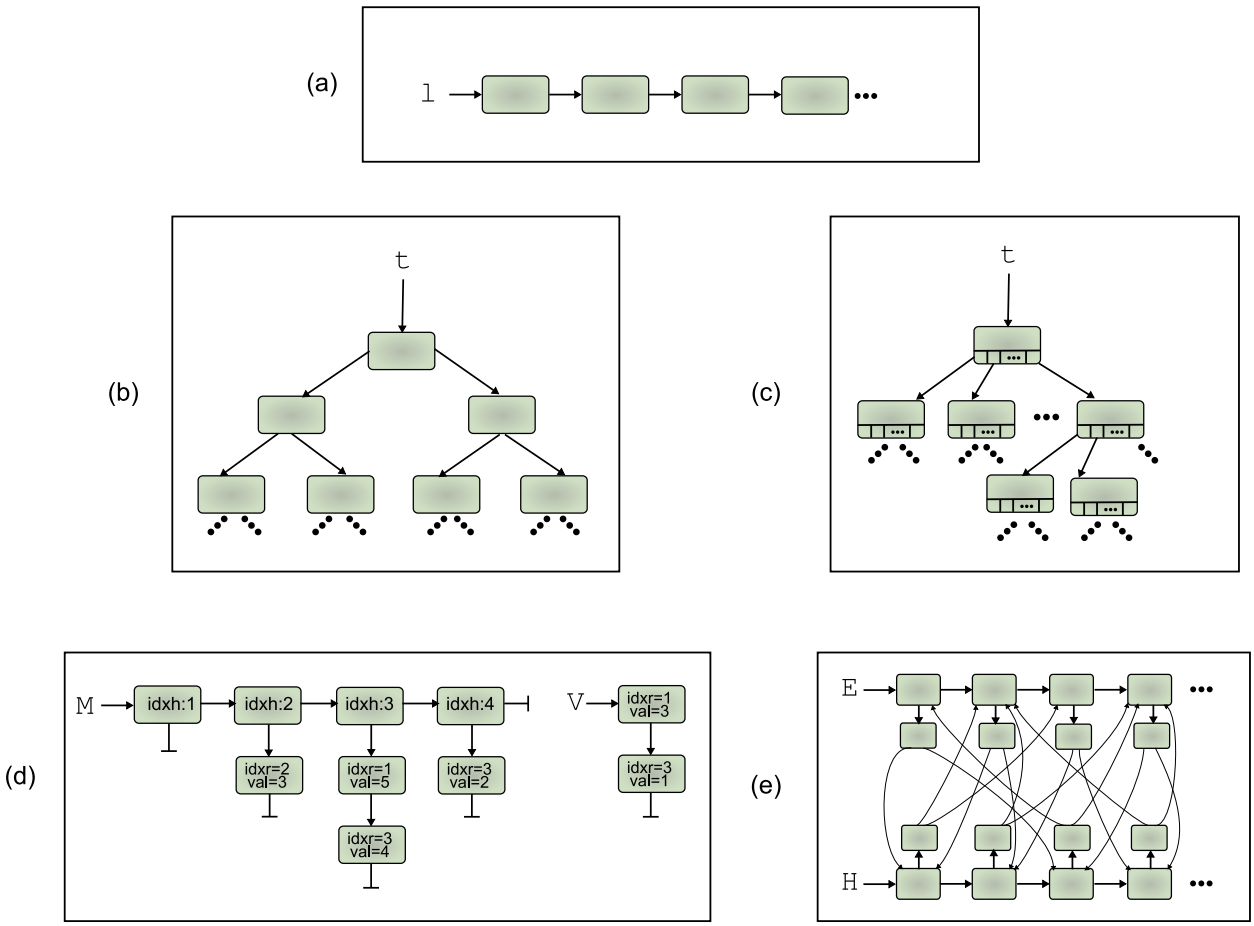

Figure 2.28: Data structures for the benchmarks considered for intraprocedural shape analysis.

The purpose of the experimental results for the intraprocedural version of our shape analysis tool is twofold: first, to test its ability to capture different kinds of dynamic data structures that are common in pointer-based applications; second, to ponder over the data gathered in the experiments so that we can understand better the strong and weak points in the technique.

Our technique was able to capture accurately all the data structures in these benchmarks. This means that the lists do not contain cycles, a tree child does not point to its parent, the columns in a matrix are independent, `Em3d`'s graph is bipartite, etc. It should be noted that we are able to capture the structures even when they are created during the traversal of the same or another structure, like in the `2-Binary tree`, `3-N-ary tree`, `4-Matrix x Vector` and `5-Matrix x Matrix` benchmarks. This is not possible for some related approaches, like [37].

All data structures where accurately represented in our shape graphs without the use of any property, except for the `6-Em3d` benchmark that uses the *site* property. Refer to section 2.7 for a discussion about the way we capture the structure in `6-Em3d`.

Once we have checked the ability of our shape analysis strategy to capture the data structures tested, our next concern involves the performance of our implementation. In Table 2.2 we show some metrics regarding

performance of the analysis and size of the problem. The first column shows analysis time in seconds. The testing platform is a 3GHz Pentium 4 with 1GB RAM. The second column shows the memory used by the analysis. Then, we show the number of statements in the program (`Code stmts.` column), the number of analyzed statements until a fixed point is reached (`Analyzed stmts.` column) and the number of shape graphs generated during the analysis (`Shape graphs` column).

Benchmark	Time	Space	Code stmts.	Analyzed stmts.	Shape graphs
1-Running example	0.78 s	1.9 MB	28	84	108
2-Binary tree	0.49 s	1.9 MB	27	254	331
3-N-ary tree	0.13 s	1.9 MB	20	135	225
4-Matrix x Vector	3.44 s	2.8 MB	95	1,071	3,398
5-Matrix x Matrix	47.75 s	5.9 MB	128	3,682	14,611
6-Em3d	18.45 s	5.5 MB	175	1,267	2,014

Table 2.2: The codes tested for intraprocedural analysis, with metrics about performance, and size of problem. The testing platform is a 3GHz Pentium 4 with 1GB RAM.

We see that the analysis times range from less than a second to a few seconds, clocking under a minute in any case. The `5-Matrix x Matrix` benchmark takes the longest to be analyzed, with 47.75 seconds. The memory used by the analyzer fits in less than 2 MB for the smaller benchmarks, peaking at 5.9 MB for `5-Matrix x Matrix`. The programs used are relatively small in size, in the same range as works in the literature, with only `6-Em3d` having near 200 statements to analyze. It should be noted though, that the size compiled at the `Code Stmts.` column makes reference to the statements that are analyzed by the shape analyzer, as affecting heap structures. In general, the program may have many more statements that do not affect the heap and thus are not considered for this metric. The value of the `Analyzed stmts.` column indicates the number of statements that have been analyzed in total by the analysis. Some statements would have been analyzed several times until a fixed point is reached. For example, the `5-Matrix x Matrix` benchmark repeatedly iterates over its 128 analyzable statements, reaching a total of 3,682 analyzed statements. Regarding the total number of graphs for the analysis, we reach into the thousands for the moderately complex programs, which hints about the complexity of the technique and the accuracy that goes with it.

We will consider more information about these experiments to shed some light into the limiting factors of the analysis. For that, we have compiled Table 2.3, which presents information about the complexity of the shape graphs obtained. Next to each benchmark, we display the shape graph per code statement, the average number of nodes per shape graph (with maximum in parentheses), and the average number of coexistent links sets per shape graph (with maximum in parentheses).

Benchmark	Sg's per code stmt.	Avg. nodes per sg (max)	Avg. cls's per sg (max)
1-Running example	3.86	2.41 (4)	4.89 (10)
2-Binary tree	12.26	2.78 (4)	24.41 (87)
3-N-ary tree	11.25	2.50 (4)	6.16 (15)
4-Matrix x Vector	35.77	5.75 (9)	30.89 (67)
5-Matrix x Matrix	114.15	8.34 (12)	52.36 (109)
6-Em3d	7.24	8.75(12)	74.40 (267)

Table 2.3: The codes tested for intraprocedural analysis, with parameters that relate to shape graph complexity.

The metric of shape graphs per code statement gives a measure of the average number of shape graphs that are used to represent the different heap states for an analyzable statement in the program. This value ranges from less than 4 shape graphs per code statement for `1-Running example` to more than a hundred for `5-Matrix x Matrix`. The average number of nodes or `cls`'s per shape graph indicate the

complexity of the shape graphs for each benchmark, and they are directly related to the abstracted data structure. Since the data structure for `6-Em3d` is the most complex (see Fig. 2.28(e)), its values for these metrics are the highest. Note also the peak in shape graph complexity achieved with 12 nodes and 267 `cls`'s.

We can draw some conclusions from the measures in Table 2.2 and Table 2.3:

- The analysis time is largely dependent on the number of shape graphs generated. That is why `5-Matrix x Matrix`, with over 14,000 generated graphs, takes the longest to analyze. Additionally, the number of shape graphs is directly dependent upon the number of analyzed statements. More iterations in the analysis provoke more shape graphs to register heap states at the different symbolic iterations during the analysis. A deep loop nest like the 4-loop nest found for `5-Matrix x Matrix` can make the number of shape graphs skyrocket very easily, as it takes a lot of symbolic iterations to find a fixed point. Note how `4-Matrix x Vector`, with a similar algorithm and similar data structures but a 3-loop nest, produces below 1/4th of the shape graphs with about 1/3rd of the analyzed statements in a much shorter time.

- More complex data structures, create shape graphs with more nodes and more `cls`'s, as can be seen for `6-Em3d`. Shape graph complexity, which is directly related to the complexity of the abstracted data structure, also affects the analysis time. For example, while there is not much difference between the number of analyzed statements for `4-Matrix x Vector` and `6-Em3d`, with about a thousand each, the analysis times differ from 3.44 to 18.45 seconds. This difference is caused by the more complex structure in `6-Em3d`, with 8.75 nodes and 74.40 `cls`'s per graph in average, against 5.75 nodes and 30.89 `cls`'s in average for `4-Matrix x Vector`. In both cases, a similar number of statements is analyzed by the tool, but the shape graphs for `6-Em3d` are more complex. This complexity taxes the abstract semantics operations, which take longer to complete.

Overall, we believe that the experimental results presented here provide evidence that the shape analysis based on the coexistent links set abstraction is precise, yields correct abstractions, and does so at reasonable cost. Next, we consider whether the measures obtained are similar to the worst case predicted by the complexity study in section 2.8.

2.10.2 Comparison with predictions of the complexity study

Let us recall that the two dominant factors for the analysis complexity are the maximum number of shape graphs, Ng, and the maximum number of `cls`'s per graph, $Ncls$. In Table 2.4 we display the maximum values measured against the maximum values predicted by the complexity analysis. The maximum number of graphs, Ng, is calculated according to Eq.2.2 in section 2.8, where nv, the number of live pointer variables is calculated for *each* statement to give a more adjusted value, rather than considering its maximum value for all statements in the program. The number of `cls`'s is calculated according to Eq.2.9. In this formula, nli, the maximum number of incoming links to the nodes in the structure is $nli = 1$ for all tests, except `6-Em3d` with $nli = 3$. The maximum number of selectors declared for a type, nl, ranges from $nl = 1$ for `1-Running example` to $nl = 3$ for `6-Em3d`.

For the larger benchmarks, `4-Matrix x Vector`, `5-Matrix x Matrix`, and `6-Em3d`, the measured value is a negligible percentage of the theoretical worst case, both in the number of shape graphs for the whole analysis and the number of `cls`'s per graph. Even for the simpler benchmarks, `1-Running example`, `2-Binary tree`, and `3-N-ary tree`, the values measured are just a small percentage of the values predicted.

Benchmark	Ng **meas.**	Ng **pred.**	Ng **meas./pred.**	$Ncls$ **meas.**	$Ncls$ **pred.**	$Ncls$ **meas./pred.**
1-Running example	108	409	26.4%	10	120	8.3%
2-Binary tree	331	1,977	16.7%	87	4740	1.8%
3-N-ary tree	225	605	37.2%	15	110	13.63%
4-Matrix x Vector	3,398	$> 1.0 \cdot 10^6$	0.3%	67	4,220	1.6%
5-Matrix x Matrix	14,611	$> 4.8 \cdot 10^8$	$< 0.0\%$	109	9,152	1.2%
6-Em3d	2,014	$> 7.8 \cdot 10^6$	$< 0.0\%$	267	$> 1.8 \cdot 10^{15}$	$< 0.0\%$

Table 2.4: Comparisons of maximum number of graphs and number of `cls`'s measured versus predicted by the complexity study.

These results provide evidence that the analysis behaves with far less complexity than the theoretical worst case. Even so, we are aware that we deal with a complex technique that is likely to be too expensive for medium or large applications. In this regard, we think it may be a valuable tool to analyze fragments of programs. Next, we will look at ways to improve the analysis performance.

2.10.3 Improving the analysis performance

Both the complexity study and the experiments conducted so far reveal that the number of generated graphs is crucial for the analysis performance. In this regard, *dead pointer nullification* helps to reduce the number of shape graphs, as the different possibilities for pointer arrangements are also reduced. This produces an improvement in performance over a version of the program that leaves dead pointers assigned. In our tests, we have manually performed dead pointer nullification as part of the program preprocessing prior to the analysis. However, more mechanisms to improve the analysis performance would be desirable.

Another aspect that should be considered is the separation of nodes in the abstraction due to the effect of properties. Adding properties produces more nodes to be kept separate, i.e. not summarized. This increases the number of nodes in shape graphs, which has a burdening effect on the shape analysis operations. However, it is sometimes the case that node separation can result in "cleaner" analyses, where nodes that stand for clearly different memory locations are not merged. This can lead to a quicker way to the fixed point and/or fewer graphs per shape graph set.

However, the chief issue is in trying to reduce the number of statements to analyze. Sometimes, there are heap statements that are analyzed by the technique that do not provoke any change in the data structure, and therefore their net effect amounts to nothing for the abstraction obtained. Such is the case in traversals that do not modify the data structure. It would be relevant for our technique to avoid analyzing such parts of the program, running instead over a *pruned* version of the program.

To measure the impact of these two ideas, namely (i) using properties for cleaner analysis, and (ii) pruning to reduce analyzable statements, we have conducted more experiments over the base benchmarks `4-Matrix x Vector` and `5-Matrix x Matrix`. We have gathered some information for four version of each: `(full)`, for the program as tested previously; `(site)`, for the base program analyzed with the site property; `(pruned)`, for the pruned version of the program; and `(pruned & site)` where the site property is used in the analysis of the pruned version of the program. The results are shown in Table 2.5.

It should be noted that the use of the site property for the `4-Matrix x Vector` and `5-Matrix x Matrix` benchmarks is not required to accurately capture the structures created and traversed, but it is considered solely for its impact on performance. Among the available properties, the site property was chosen because it provides the greater separation of nodes. For the pruned versions we have manually discarded for its analysis the statements in the product that are not related to the construction of the output vector/matrix.

Benchmark	Time	Space	Code stmts.	Analyzed stmts.	Shape graphs
4-Matrix x Vector (full)	3.44 s	2.8 MB	95	1,071	3,398
4-Matrix x Vector (site)	2.56 s	2.8 MB	95	989	2,413
4-Matrix x Vector (pruned)	0.25 s	1.9 MB	75	531	594
4-Matrix x Vector (pruned & site)	0.28 s	1.9 MB	75	551	612
5-Matrix x Matrix (full)	47.75 s	5.9 MB	128	3,682	14,611
5-Matrix x Matrix (site)	30.58 s	3.7 MB	128	3,819	9,619
5-Matrix x Matrix (pruned)	3.72 s	3.3 MB	114	1,300	2,405
5-Matrix x Matrix (pruned & site)	3.97 s	2.7 MB	114	1,463	2,299

Table 2.5: Measures for the `4-Matrix x Vector` and `5-Matrix x Matrix` benchmarks in four versions each: `full`, `site`, `pruned` and `pruned & site`.

Regarding the use of properties for better performance, we observe in Table 2.5 a slight improvement for `4-Matrix x Vector (site)` in the analysis time, the number of analyzed statements and the number of generated graphs, as a result of achieving the fixed point slightly faster. For `5-Matrix x Matrix (site)`, there is improvement in analysis time, memory consumed and generated graphs. Here the fixed point takes slightly longer to achieve with a few more statements analyzed, but the number of generated graphs has dropped nearly 5,000 graphs. This is a consequence of not mixing nodes from different structures in the deep loop nests, which creates fewer shape graphs in a shape graph set.

Regarding the use of pruning for reducing the number of analyzable statements, we see how the pruned versions of the benchmarks have improved in all measures over the full version. The graphs obtained as abstractions for the data structures are the same for both the full and pruned versions of each benchmark. Analysis times have improved dramatically as we have changed from a 3-loop nest to a single loop in `4-Matrix x Vector (pruned)` and from a 4-loop nest to a 2-loop nest in `5-Matrix x Matrix (pruned)`. Memory consumption has also decreased, the analyzed statements to the fixed point have halved and the number of generated graphs has decreased around 6 times. It is remarkable the impact on performance measures due to removing just a few lines of code for these benchmarks.

Finally, the combination of both the pruning and use of the site property introduce a slight overhead over the pruned version, which indicates that the pruning has already achieved a version that is very tolerable by the technique and adding properties only adds more complexity with no payback in performance. Even so, `5-Matrix x Matrix (pruned & site)` improves on memory consumption and number of generated graphs with regards to the pruned version.

The results in Table 2.5 prove that it is possible to obtain significant improvement in performance for the same shape abstraction, specially if pruning of the program if possible. We think that an automatic compiler pass to prune programs in this way would be an interesting subject for future work.

2.11 Summary

In this chapter we have presented the following content:

- In the first place, we have provided a general outline of our approach to heap analysis abstracting heap states as shape graphs (section 2.1).
- We have described a high-level view of the key concept of coexistent links sets (section 2.2).
- Then, we entered into a formal description of our shape graph abstraction, first defining a concrete heap model, then its matching abstract heap model (section 2.3).

- Next, we describe our data-flow equations, and the worklist algorithm that implements them (section 2.4).
- The abstract semantics and operations for the analysis are also described (section 2.5), featuring examples for common operations carried over dynamic data structures.
- We also describe our approach for dealing with pointer arrays in shape graphs (section 2.6).
- We describe the extendable mechanism of properties for refining heap abstractions (section 2.7).
- A complexity study for the technique is undertaken as well (section 2.8). It identifies the leading parameters for worst-case behavior of the analysis.
- We have identified and described meaningful related work in heap analysis (section 2.9).
- Finally, we have conducted some tests that provide experimental evidence that our technique yields correct abstractions for a variety of common recursive data structures featured in some selected benchmarks (section 2.10). We have also collected information about performance, problem size, shape graph complexity and have extracted conclusions from those results, including comparison with the theoretical worst case for complexity. We conclude by hinting on how to improve performance by *pruning* certain parts in the analyzed programs that are irrelevant to obtain the shape abstractions.

At this point, we feel encouraged to continue our work to provide full interprocedural support, with aims to complete a precise shape analysis tool suitable for dependence detection in pointer-based applications.

3 Interprocedural shape analysis

3.1 Introduction

Support for interprocedural programs in shape analysis is still a challenge, especially in the presence of recursive functions. Yet traversing recursive data structures with recursive algorithms is very common, as some structures, such as trees, are expressed in a way that makes it natural to traverse them in a recursive fashion. The main issue that we face when analyzing recursive functions is the problem of tracking the state of pointer parameters in context changes. For non-recursive context changes, it is enough to know the relationship between actual and formal pointer parameters. In such a case, the context change can be easily translated to the shape graph domain.

However, when dealing with pointer formal parameters in recursive functions, it is not so simple: the same pointer variable must be tracked along a sequence of indefinite recursive calls. The name of the pointer is the same, but depending on the call, it can point to different locations. Those locations must be tracked so that we know where the pointer was pointing to when returning from a recursive call.

Therefore, to keep track of a pointer formal parameter we need to change its *naming scheme*: not only we need to know its name, but also some information that relates it to the particular call where it belongs to. The same can be said for pointers defined in the recursive function body, the *local pointers*. Such pointers are redefined for every call, i.e., their scope belongs only to a certain recursive function call, yet they must be correctly assigned upon return of the recursive calls that ensue.

At run time, this is done by keeping different registers in the *Activation Record Stack* (ARS). Among other information, the ARS keeps the state of pointer actual parameters and local pointers before a call, so that when returning from the call, they can be properly reassigned. Keep in mind that a compile-time pointer analysis technique cannot know the number of times a recursive function will be called, yet a fixed point must be reached for the analysis, even in the presence of pointer parameters and local pointers. This makes it tricky to reach a fixed point shape graph abstraction for recursive functions and, at the same time, keep a precise abstraction.

To help us explain the concepts in this chapter, let us introduce now an example program that creates and reverses a singly-linked list. Actually, this creation and reversal provide the same result than the running

example in chapter 2. For this chapter though, we will focus on the recursive function `reverse()` and the extensions introduced for its analysis within our framework. Fig. 3.1 shows the code considered here.

```
   // Declare recursive type "node"
   struct node{
       int data;
       struct node *nxt;
   }
   int main(int argc, char argv[]){
       struct node *list,*r;
1:     list=create_list(SIZE);
2:     r=reverse(list);
3:     return 1;
   }
```

```
   struct node *reverse(struct node *x){
       struct node *y,*z;
4:     z=x->nxt;
       if(z!=NULL){
5:        #pragma SAP.force(z!=NULL)
6:        y=reverse(z);
7:        #pragma SAP.force(x!=NULL)
8:        x->nxt=NULL;
9:        z->nxt=x;
       }else{
10:       #pragma SAP.force(z==NULL)
11:       y=x;
       }
12:    return y;
   }
```

Figure 3.1: Running example for presentation of interprocedural analysis.

Fig. 3.2 exemplifies the different kinds of context changes that we may encounter, and how the ARS is used to keep record of pointer state so they can be recovered when returning from the calls. In (a) we show a non-recursive context change in the concrete domain: we invoke `reverse()` from `main()`, with a singly-linked list of four elements. A new register for `reverse()` is added on top of the ARS for the context change. Note how each register in the ARS keeps the information about the locations that the local pointers are pointing to. In (b), there is a recursive context change from the first call to `reverse()` to the second call. Unlike the first call, which was invoked from `main()`, this is a recursive call, as it is called from `reverse()`. A new register for `reverse()` is added to the ARS. In (c), we return from the second call to `reverse()` to the first call, after having reversed memory locations `l2`, `l3` and `l4`. As the uppermost register of the ARS is removed, the record below is used to reestablish local pointers `x` and `z` (`y` is assigned at the call site, `st.6:y=reverse(z)`). Finally, in Fig. 3.2 (d), the return to the `main()` function is performed, with the list completely reversed, and the last record for `reverse()` is removed from the ARS.

For simplicity in formal-actual parameter matching, we do not allow pointer formal parameters to be modified within the function body. This involves no loss of generality as it is always possible to comply with this condition with additional pointer variables. Also, we consider pointers passed by-value. Passing a reference to a pointer (pass by-reference) actually involves a double indirection and shall not be considered here.

3.2 Extensions for interprocedural analysis

The way we have constructed interprocedural support within our shape analysis framework is by extending on the base, intraprocedural technique, as described in chapter 2. This allows us to take advantage of the already existing abstraction, abstract semantics operations and data-flow equations.

To support interprocedural programs, including recursive functions, we extend on four different axes: (i) new statements to include function calls and return sites, (ii) new elements within the graph to capture

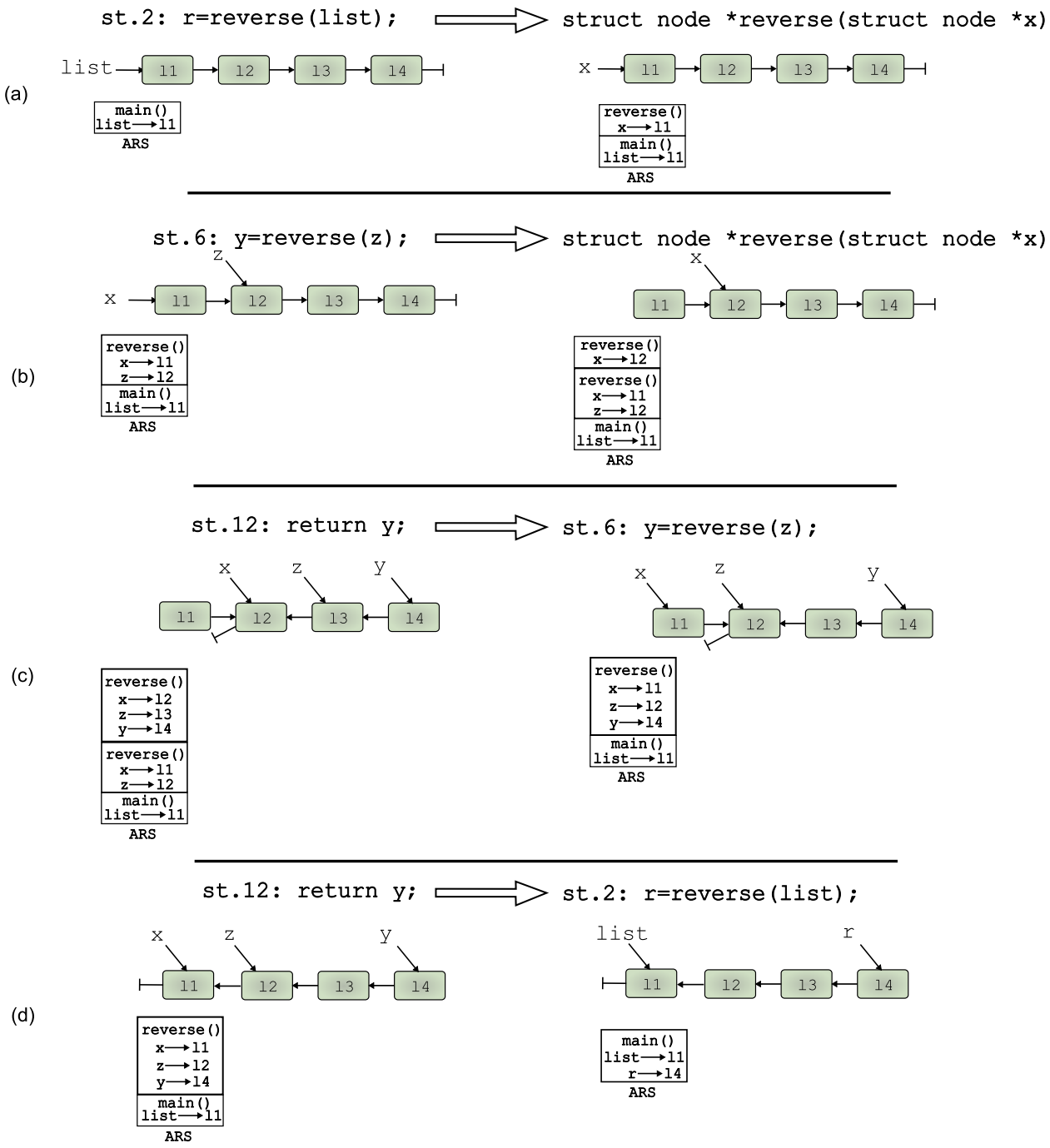

Figure 3.2: The use of the Activation Record Stack (ARS) for recursive function analysis.

information from the *Activation Record Stack* (ARS), (iii) context change rules, that determine graph transformation when entering to or exiting from a function, and (iv) extended data-flow equations and associated worklist algorithm.

3.2.1 New statements

We extend the type of analyzable statements to include the `call()` and `return()` of these functions (see Fig. 3.3). Accordingly, the definition of a program `P` is now extended to include the set of functions that

use pointers to recursive data structures. Such functions are contained in FUN. Function pointers are not supported. We designate $\mathtt{FUN}_{\mathtt{fun}}$ to the set of functions that are directly called in the body of function fun, and $\mathtt{STMT}_{\mathtt{fun}}$ to the statements in the body of fun, including function calls.

An important detail is that we distinguish between non-recursive and recursive call sites. The set of call statements defined in non-recursive call sites is called $\mathtt{S}_{\mathtt{call\_nrec}}$, whereas the set of call statements defined in recursive call sites is called $\mathtt{S}_{\mathtt{call\_rec}}$. Return statements can, therefore, return to recursive or non-recursive call sites. The set of return statements defined for the functions in the program is called $\mathtt{S}_{\mathtt{return}}$.

programs:	prog ∈ P, P=<FUN,STMT,PTR,TYPE,SEL>
functions:	fun ∈ FUN, FUN=<$\mathtt{FUN}_{\mathtt{fun}}$,$\mathtt{STMT}_{\mathtt{fun}}$,PTR,TYPE,SEL>
statements:	s ∈ STMT, s::= x=NULL|x=malloc()|free(x)|x=y |x->sel=NULL|x->sel=y|x=y->sel |x=call()|return(y)

Figure 3.3: New statements for interprocedural support.

It is straightforward to see that, for the example in Fig. 3.1, we have the following sets available: FUN={create_list,reverse,main}, $\mathtt{FUN}_{\mathtt{reverse}}$={reverse}, $\mathtt{FUN}_{\mathtt{main}}$={create_list,reverse}, $\mathtt{STMT}_{\mathtt{reverse}}$={st.4-st.12}, $\mathtt{STMT}_{\mathtt{main}}$={st.1-st.3}, where only the statements that have abstract semantics operations associated to them are numbered. The information provided by the statements related to the program flow, such as loops, branches, and function headers, is considered in the data-flow equations.

The introduction of functions in our technique gives rise to new instrumentation mapping functions that account for (i) the relationship of pointers to the functions where they are declared, (ii) actual and formal pointer parameters correspondence, and (iii) the matching of the pointer returned by a return statement and the one assigned at a call site.

Local Pointers Map:	$\mathcal{LPM}$: FUN $\longrightarrow$ PTR
Actual-Formal Ptrs Map:	$\mathcal{AFPM}$: ($\mathtt{S}_{\mathtt{call\_nrec}}$ ∪ $\mathtt{S}_{\mathtt{call\_rec}}$)×FUN $\longrightarrow$ PTR×$\mathtt{PTR}_{\mathtt{fun}}$
Ret to Assigned Ptr Map:	$\mathcal{RAPM}$: ($\mathtt{S}_{\mathtt{call\_nrec}}$ ∪ $\mathtt{S}_{\mathtt{call\_rec}}$)×FUN $\longrightarrow$ ($\mathtt{PTR}_{\mathtt{fun}}$×PTR) ∪ ∅

- $\mathcal{LPM}$ is a multivalued function that maps for a function fun ∈ FUN, the set of local pointers associated with it, i.e. the formal and local pointer variables declared within the body of the function:

  ∀ fun ∈ FUN, $\mathcal{LPM}$(fun)={lptr ∈ PTR, being lptr a pointer formal parameter or local pointer variable defined for fun}.

- $\mathcal{AFPM}$ is a multivalued partial function that maps for a call statement s (being s a non-recursive or a recursive call, i.e. s ∈ $\mathtt{S}_{\mathtt{call\_nrec}}$ ∪ $\mathtt{S}_{\mathtt{call\_rec}}$) and the function fun ∈ FUN called by s, the set of matching pointer actual (aptr) and formal (fptr) parameter pairs:

  ∀ s ∈ ($\mathtt{S}_{\mathtt{call\_nrec}}$ ∪ $\mathtt{S}_{\mathtt{call\_rec}}$), being fun ∈ FUN called by s, $\mathcal{AFPM}$(s,fun)={<aptr,fptr>, where aptr ∈ PTR is an actual parameter in statement s, and fptr ∈ $\mathtt{PTR}_{\mathtt{fun}}$ is a formal parameter in fun}.

  For example, $\mathcal{AFPM}$(st.6,reverse)=<z,x>. Sometimes, we just need the set of actual

pointer parameters (aptr) for a call statement s. We will name $\texttt{APTR}_\texttt{s}$ to that set. It can easily be deduced from $\mathcal{AFPM}$(s,fun).

- $\mathcal{RAPM}$ is a partial map that computes, for a call statement s (being s a non-recursive or a recursive call, i.e. $\texttt{s} \in \texttt{S}_{\texttt{call\_nrec}} \cup \texttt{S}_{\texttt{call\_rec}}$) and the function fun $\in$ FUN called by s, the corresponding pointer returned at the exit point (retptr) vs. the pointer assigned at the call site (assptr):

  $\forall$ s $\in$ ($\texttt{S}_{\texttt{call\_nrec}} \cup \texttt{S}_{\texttt{call\_rec}}$), being fun $\in$ FUN called by s, $\mathcal{RAPM}$(s,fun)=<retptr,assptr>, where retptr $\in$ $\texttt{PTR}_{\texttt{fun}}$ is the pointer returned at the exit point of fun and assptr $\in$ PTR is the pointer assigned at statement s. In the case that the function does not return a pointer, then this function gives $\emptyset$. For our example, $\mathcal{RAPM}$(st.2,reverse)=<y,r>.

3.2.2 Recursive Flow Links

We have stated the problem of tracking the state of formal pointer parameters and local pointers in a sequence of recursive calls. A proper naming scheme is required for these pointers in the analysis. In our approach, we abstract the information of the ARS by using a new kind of link over the base shape graph representation, that we call *recursive flow links*. Recursive flow links do not represent actual links existing in the program data structure but rather *trace* the arrangement of pointer formal parameters and local pointers along the recursive, interprocedural control flow. This is done with two kinds of recursive flow links: *recursive flow pointer links* (rfpl) and *recursive flow selector links* (rfsl).

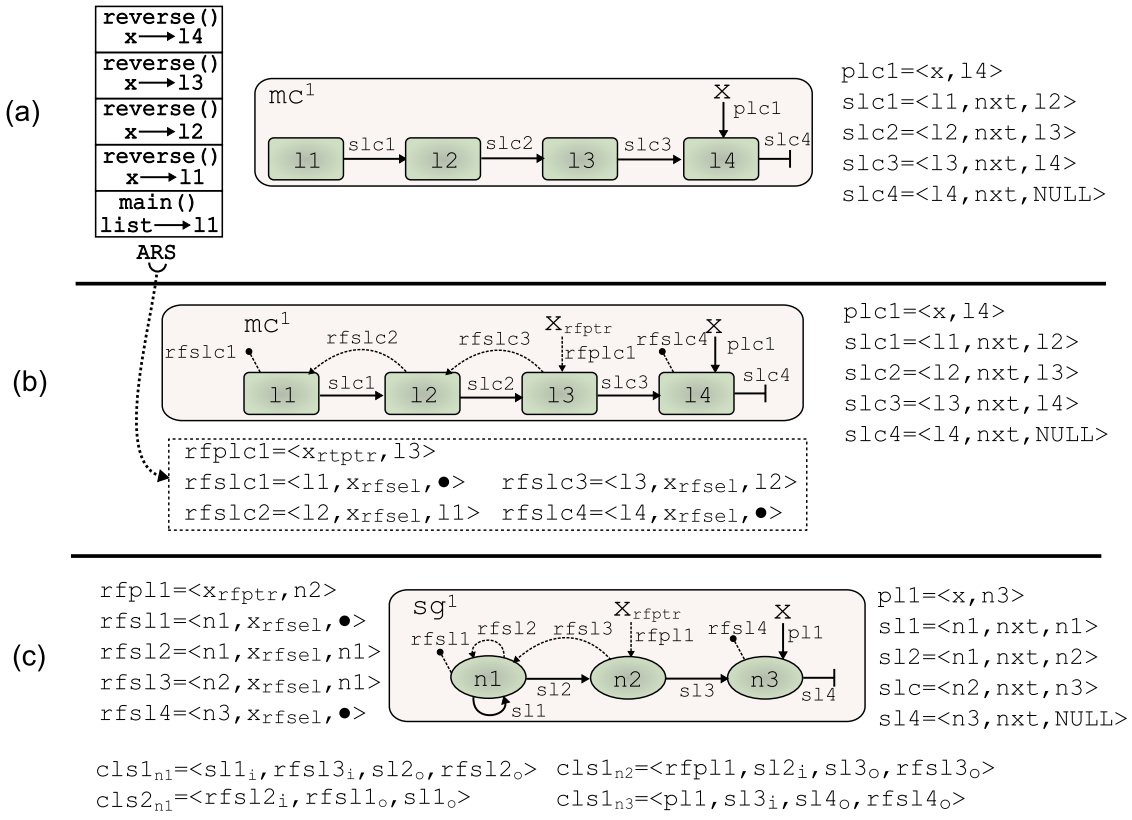

Figure 3.4: A 4-element list after the 4th invocation to reverse(): (a) with ARS, (b) with recursive flow links, and (c) its shape graph.

Fig. 3.4(a) shows the concrete domain version of our singly-linked list at the fourth invocation of reverse(). The ARS keeps information about the state of pointers in previous calls. In (b) we dis-

play how the same list can be represented with the aid of *recursive flow links in the concrete domain*, which are shown in dashed edges. x_{rfptr} marks the location that x was pointing to in the previous call, i.e., at the third invocation of reverse(). x_{rfptr} is a *recursive flow pointer*. It is not defined in the original program, but has been introduced by our analysis to track the variations of pointer formal parameter x along the interprocedural control flow. rfplc1=<x_{rfptr}, l3> is a *recursive flow pointer link in the concrete domain* (rfplc), which is defined in the same way as a regular *pointer link in the concrete domain* but based on a *recursive flow pointer* rather than a regular pointer.

We also need to keep track of the location of the formal parameter x in the invocations of reverse() prior to the previous one, i.e., tracking x beyond the immediately previous call. For that, we use *recursive flow selector links in the concrete domain* rfslc3, rfslc2 and rfslc1. They are, not surprisingly, based on *recursive flow selectors* rather than regular selectors. For example, rfslc3=<l3, x_{rfsel}, l2>, based on recursive flow selector x_{rfsel}, indicates that two calls back in the ARS, x was pointing to l2.

The location denoted by • is representing the NULL location for the recursive flow path. Following the trace through a recursive flow selector link with • as destination would not correspond to any activation record in the succession of recursive calls, and therefore would not render any realistic memory configuration. For example, rfslc1=<l1, x_{rfsel}, •> indicates that x is not defined beyond three previous calls to reverse().

In Fig. 3.4(c) we have abstracted the memory location from (b) into the abstract domain. Here, we must provide a bound representation. We build now the recursive flow information into *recursive flow pointer links* (rfpl) and *recursive flow selector links* (rfsl). Memory locations l1 and l2 are now abstracted as node n1, and this is reflected in rfsl2=<n1, x_{rfsel}, n1>. Coexistent links sets include now rfpl's and rfsl's with attributes as a natural addition.

To sum up, we have introduced recursive flow pointer x_{rfptr} and recursive flow selector x_{rfsel} to track the locations x has pointed to in previous calls of the recursive function reverse(). They are used to build *recursive flow pointer links* (rfpl) and *recursive flow selector links* (rfsl). For the time being, let us assume that we only need this new kind of links for formal parameter x. More information about this aspect can be found in section 3.2.3.5.

The main advantage of this approach is that it allows us to reuse all the existing operations that deal with graphs, because *recursive flow links* are treated just as *pointer links* or *selector links*, with regards to node summarization, materialization, graph joining, etc.

Let us see how these new elements are considered in our representation. Along with the existing set of pointers, PTR, we now define a new set of recursive flow pointers, RFPTR. Besides SEL for selectors, we include the set RFSEL for recursive flow selectors. Fig. 3.5 presents the extended sets for pointers and selectors in interprocedural analysis. The type objects remain as they were presented in chapter 2.

pointer variables:	$x \in PTR$, $x_{rfptr} \in RFPTR$
type objects:	$t \in TYPE$
selectors fields:	$sel \in SEL$, $sel_{rfsel} \in RFSEL$

Figure 3.5: Extended sets for pointers and selectors in interprocedural analysis.

We name PTR_{fun} to the set of pointer formal parameters *and* local pointer variables associated with function fun. GLB is the set of global pointers, GLB ⊂ PTR. For the example in Fig. 3.1, PTR_{main}={list,r}, $PTR_{reverse}$={x,y,z}, and GLB=∅.

$RFPTR_{fun}$ and $RFSEL_{fun}$ contain the set of recursive flow pointers and recursive flow selectors respectively, for function fun. We only need to use recursive flow links to track those pointers that can-

not be recovered by the *pointer matching* induced by the context changes. In the worst case, we need a `rfptr-rfsel` pair for every pointer formal parameter or local pointer variable in a recursive function. However, it is usually the case that only one pointer needs to be traced. For the example discussed here, $\mathtt{RFPTR}_{\mathtt{reverse}}=\{\mathtt{x}_{\mathtt{rfptr}}\}$ and $\mathtt{RFSEL}_{\mathtt{reverse}}=\{\mathtt{x}_{\mathtt{rfsel}}\}$.

The following mapping functions are introduced to identify relations between a recursive flow pointer and (i) the pointer it tracks along the interprocedural control flow, and (ii) its associated recursive flow selector.

Recursive Flow Pointer to Pointer Map:	$\mathcal{RFPPM}$: $\mathtt{RFPTR}_{\mathtt{fun}} \longrightarrow \mathtt{PTR}_{\mathtt{fun}}$
Recursive Flow Pointer to recursive flow Selector Map:	$\mathcal{RFPSM}$: $\mathtt{RFPTR}_{\mathtt{fun}} \longrightarrow \mathtt{RFSEL}_{\mathtt{fun}}$

- $\mathcal{RFPPM}$ maps a recursive flow pointer in function `fun` to the pointer it tracks recursively in the same function.

  $\forall$ `rfptr` $\in \mathtt{RFPTR}_{\mathtt{fun}}$, $\exists$ `ptr` $\in \mathtt{PTR}_{\mathtt{fun}}$ | `rfptr` is the recursive flow pointer that tracks the location of `ptr` in the previous recursive call to `fun`.

  For example, $\mathcal{RFPPM}_{\mathtt{reverse}}(\mathtt{x}_{\mathtt{rfptr}})=\mathtt{x}$.

- $\mathcal{RFPSM}$ maps a recursive flow pointer in function `fun` to its matching recursive flow selector.

  $\forall$ `rfptr` $\in \mathtt{RFPTR}_{\mathtt{fun}}$, $\exists$ `rfsel` $\in \mathtt{RFSEL}_{\mathtt{fun}}$ | `rfsel` is the recursive flow selector that tracks pointer `ptr`=$\mathcal{RFPPM}_{\mathtt{fun}}$`(rfptr)` beyond the previous recursive call to `fun`.

  For example, $\mathcal{RFPSM}_{\mathtt{reverse}}(\mathtt{x}_{\mathtt{rfptr}})=\mathtt{x}_{\mathtt{rfsel}}$.

3.2.2.1 Recursive flow links in the concrete domain

The process of instrumenting the information from the ARS into the concrete domain makes use of two new partial functions, $\mathcal{RFPM}^c$ and $\mathcal{RFSM}^c$.

Recursive Flow Pointer Map (in the concrete domain):	$\mathcal{RFPM}^c$: $\mathtt{RFPTR}_{\mathtt{fun}} \longrightarrow \mathtt{L}$
Recursive Flow Selector Map (in the concrete domain):	$\mathcal{RFSM}^c$: $\mathtt{L}\times\mathtt{RFSEL}_{\mathtt{fun}} \longrightarrow (\mathtt{L} \cup \mathtt{NULL})$

- $\mathcal{RFPM}^c$ maps a recursive flow pointer `rfptr` $\in \mathtt{RFPTR}_{\mathtt{fun}}$ to the location `l` pointed to by the tracked pointer `ptr` in the immediately previous pending call (previous context):

  $\forall$ `rfptr` $\in \mathtt{RFPTR}_{\mathtt{fun}}$, $\exists$ `ptr` $\in \mathtt{PTR}_{\mathtt{fun}}$ s.t. $\mathcal{RFPPM}$`(rfptr)=ptr` $\wedge$ $\exists$ `l` $\in$ `L` | $\mathcal{RFPM}^c$`(rfptr)=l` $\wedge$ $\mathcal{PM}^c$`(ptr)=l` in the immediately previous pending call.

  The $\mathcal{PM}^c$ mapping was defined in chapter 2. As a reminder, it maps a pointer variable `ptr` to the location `l` it points to. We use the tuple `rfplc=<rfptr,l>`, which we name *recursive flow pointer link in the concrete domain*, to represent this binary relation. The set of all recursive flow pointer links in the concrete domain is named `RFPLc`.

- $\mathcal{RFSM}^c$ models the path (between locations `l1` and `l2`) tracked for a formal or local pointer `ptr` $\in \mathtt{PTR}_{\mathtt{fun}}$ through two consecutive previous pending calls. Let us assume that we name $\mathtt{pc}_{\mathtt{t}}$ to a pending call and $\mathtt{pc}_{\mathtt{t-1}}$ to the immediately previous pending call:

$\forall$ l2 $\in$ L s.t. $\mathcal{PM}^c$(ptr)=l2 in a previous pending call pc_t, $\exists$ rfptr s.t. $\mathcal{RFPPM}$(rfptr)=ptr $\wedge$ $\exists$ l1 $\in$ (L $\cup$ NULL) s.t. $\mathcal{PM}^c$(x)=l1 in the immediately previous pending call pc_{t-1}| $\mathcal{RFSM}^c$(l2,rfsel)=l1 $\wedge$ $\mathcal{RFPSM}$(rfptr)=rfsel.

We use a tuple rfslc=<l2,rfsel,l1>, which we name *recursive flow selector link in the concrete domain*, to represent this relation. The set of all recursive flow selector links in the concrete domain is called RFSLc.

The domain for a graph in our concrete heap is the set MC $\subset \mathcal{P}$(L)$\times\mathcal{P}$(PLc $\cup$ RFPLc)$\times\mathcal{P}$(SLc $\cup$ RFSLc). Each memory configuration of our concrete domain $mc^i \in$ MC, is now represented as a tuple mc^i=<L^i,PLc^i $\cup$ $RFPLc^i$,SLc $\cup$ $RFSLc^i$> with $L^i \subset$ L, $PLc^i \subset$ PLc, $SLc^i \subset$ SLc and the new sets $RFPLc^i \subset$ RFPLc and $RFSLc^i \subset$ RFSLc. Fig. 3.4(b) shows the memory configuration for (a) with the information of the ARS reflected as the appropriate recursive flow links.

3.2.2.2 Recursive flow links in the abstract domain

Similarly, to model the information provided by the ARS in our abstract domain, we include two new partial functions, $\mathcal{RFPM}^a$ and $\mathcal{RFSM}^a$ which model, on each function call, a trace of the nodes where each formal and local pointer was pointing to in the previous pending calls in a stack of recursive calls. They are defined as follows:

Recursive Flow Pointer Map (in the abstract domain): $\mathcal{RFPM}^a$: $RFPTR_{fun} \longrightarrow$ N

Recursive Flow Selector Map (in the abstract domain): $\mathcal{RFSM}^a$: N$\times RFSEL_{fun} \longrightarrow$ N

- $\mathcal{RFPM}^a$ maps a recursive flow pointer rfptr $\in RFPTR_{fun}$ to the node n pointed to by the tracked pointer ptr in the immediately previous pending call (previous context):

  $\forall$ rfptr $\in RFPTR_{fun}$, $\exists$ ptr $\in PTR_{fun}$ s.t. $\mathcal{RFPPM}$(rfptr)=ptr $\wedge$ $\exists$ n $\in$ N | $\mathcal{RFPM}^a$(rfptr)=n $\wedge$ $\mathcal{PM}^a$(ptr)=n in the immediately previous pending call.

  We use the tuple rfpl=<rfptr,n>, which we name *recursive flow pointer link*, to represent this binary relation. The set of all recursive flow pointer links is named RFPL.

- $\mathcal{RFSM}^a$ models the path (between nodes n1 and n2) tracked for a formal or local pointer ptr $\in$ PTR_{fun} through two or more consecutive previous pending calls. Let us assume that we name pc_t to a pending call and pc_{t-1} to the immediately previous pending call:

  $\forall$ n2 $\in$ (N-NULL) s.t. $\mathcal{PM}^a$(ptr)=n2 in a previous pending call pc_t, $\exists$ rfptr s.t. $\mathcal{RFPPM}$(rfptr)=ptr $\wedge$ $\exists$ n1 $\in$ N s.t. $\mathcal{PM}^a$(ptr)=n1 in the immediately previous pending call pc_{t-1} | $\mathcal{RFSM}^a$(n2,rfsel)=n1 $\wedge$ $\mathcal{RFPSM}$(rfptr)=rfsel.

  We use a tuple rfsl=<n2,rfsel,n1>, which we name *recursive flow selector link*, to represent this relation. The set of all recursive flow selector links is called RFSL.

Coexistent links sets are now naturally expanded to include the new elements. First, we group pointer links and recursive flow pointer links, PL $\cup$ RFPL. Then, selector links are grouped with recursive flow selector links, SL $\cup$ RFSL, to augment the domain of the selector links with attributes: SL_{att}=(SL $\cup$ RFSL)$\times$ATTSL. The new domain for coexistent links sets is $\mathcal{CLM}$: N $\longrightarrow \mathcal{P}$(PL $\cup$ RFPL)$\times\mathcal{P}$(SL_{att}). Therefore, a coexistent links set cls_n for node n is redefined as:

$\mathtt{cls_n}=\{\mathtt{PL_n},\mathtt{SL_n}\}$

where:

$\mathtt{PL_n} = \{\mathtt{pl} \in \mathtt{PL}\text{ s.t. }\mathtt{pl}=\langle \mathtt{x},\mathtt{n}\rangle\} \cup \{\mathtt{rfpl} \in \mathtt{RFPL}\text{ s.t. }\mathtt{rfpl}=\langle \mathtt{x_{rfptr}},\mathtt{n}\rangle\}$

$\mathtt{SL_n} = \{\mathtt{sl_{att}} \in \mathtt{SL_{att}}\text{ s.t. }\mathtt{sl_{att}}=\langle\langle \mathtt{n1},\mathtt{sel},\mathtt{n2}\rangle,\mathtt{attsl}\rangle \vee$
$\mathtt{sl_{att}}=\langle\langle \mathtt{n1},\mathtt{x_{rfsel}},\mathtt{n2}\rangle,\mathtt{attsl}\rangle, \text{being } (\mathtt{n1}=\mathtt{n} \vee \mathtt{n2}=\mathtt{n})\}$

Obviously, the domain for an abstract graph is the set $\mathtt{SG} \subset \mathcal{P}(\mathtt{N}) \times \mathcal{P}(\mathtt{CLS})$, and each element of this domain, a shape graph $\mathtt{sg^i} \in \mathtt{SG}$, is a tuple $\mathtt{sg^i}=\langle \mathtt{N^i},\mathtt{CLS^i}\rangle$, as previously defined.

Fig. 3.4(c) shows the abstract domain representation of (b). We should note that in the case that $\mathtt{n2}=\mathtt{n1}$ in the `rfsl`, then more than two consecutive pending calls are represented by this relation: in this case, all the pending calls for which $\mathcal{PM}^a(\mathtt{x})=\mathtt{n1}=\mathtt{n2}$ are represented by just one recursive flow selector link. For example, $\mathtt{rfsl2}=\langle \mathtt{n1},\mathtt{x_{rfsel}},\mathtt{n1}\rangle$ in Fig. 3.4(c) stands for the tracing of pointer `x` along *two* previous recursive calls, as indicated by $\mathtt{cls1_{n1}}=\langle \ldots,\mathtt{rfsl2_o}\rangle$ and $\mathtt{cls2_{n1}}=\langle \mathtt{rfsl2_i},\mathtt{rfsl1_o},\ldots\rangle$.

3.2.3 Context change rules

The analysis at function calls must account for the assignment of actual to formal parameters and for the change of analysis domain between the caller and the callee. For it, shape graphs are transformed into the appropriate context while flowing in and out of functions by the *context change rules*, namely the *call-to-start* (`CTS`) rule, and the *return-to-call* (`RTC`) rule.

The call-to-start rule determines how the recursive flow links in the shape graphs are transformed from a function call to the context inside the function. On the other hand, the return-to-call rule transforms the heap abstraction returned by a function to the appropriate context at the calling site. Each of these rules has a recursive ($\mathtt{CTS_{rec}}$ and $\mathtt{RTC_{rec}}$) and non-recursive ($\mathtt{CTS_{nrec}}$ and $\mathtt{RTC_{nrec}}$) version. Next, we will cover each of these rules showing their algorithms and illustrating them with examples.

3.2.3.1 Non-recursive call-to-start rule

The non-recursive call-to-start rule ($\mathtt{CTS_{nrec}}$) determines how a shape graph is changed when entering a new function context. Such a case occurs when `reverse()` is called from the `main()` function. The list passed as argument is then transformed to the callee context. This case is illustrated by Fig. 3.6.

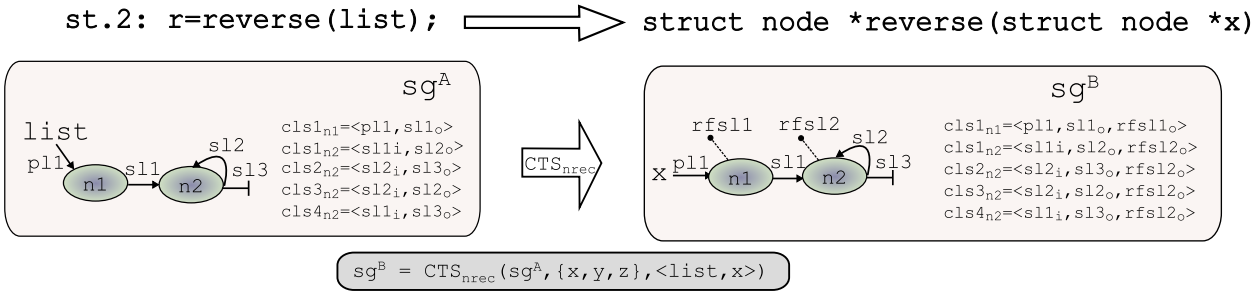

Figure 3.6: Example of shape graph transformation by the $\mathtt{CTS_{nrec}}$ rule.

The algorithm that explains this context change is shown in Fig. 3.7. First, pointer formal parameters are assigned to the pointer actual parameters, which are then nullified as they fall out of scope (unless they are global pointer variables). In this example (Fig. 3.6), actual parameter `list` is exchanged for pointer formal parameter `x`.

```
CTS_nrec()
  Input: sg^1=<N^1,CLS^1>, PTR_fun, AFPM(s,fun)
  # A shape graph, formal and local ptrs for fun, and set of pairs <aptr,fptr> of call site s
  Output: RSSG^k
  # A reduced set of shape graphs

  RSSG^2=sg^1
  forall x ∈ APTR_s                          # APTR_s is the set of actual pointers in the call stmt. s
     Find the pair <aptr,fptr> ∈ AFPM(s,fun) s.t. x=aptr
     RSSG^3=⨆_{∀sg'∈RSSG^2}^{RSSG} XY(sg',fptr,aptr)      # fptr=aptr
     If (aptr ∉ GLB),
        RSSG^4=⨆_{∀sg''∈RSSG^3}^{RSSG} XNULL(sg'',aptr)   # aptr=NULL
     else
        RSSG^4=RSSG^3
     RSSG^2=RSSG^4
  endfor
  If (∃ s' ∈ STM_fun s.t. s' ∈ S_call_rec),        # The case when fun will include a recursive call site
     forall rfsel ∈ RFSEL_fun,
        forall sg^i=<N^i,CLS^i> ∈ RSSG^2,
           forall nj ∈ N^i,                      # Initialize x_rfsel for all nodes in all graphs
              Create sl'_att=<<nj,rfsel,•>,attsl'={o}>
              forall cls_nj={PL_nj,SL_nj} ∈ CLS_nj (being CLS_nj ⊂ CLS^i)
                 SL_nj=SL_nj ∪ sl'_att
              endfor
           endfor
        endfor
     endfor
  RSSG^k=RSSG^2
  return(RSSG^k)
end
```

Figure 3.7: The $\texttt{CTS}_{\texttt{nrec}}\texttt{()}$ function.

Then, if the callee is a recursive function, a recursive flow selector link initialization phase takes place: we create a `rfsl` from every node to •, which is added with the output attribute (`o`) to every `cls` in the node, i.e., $\texttt{sl'}_{\texttt{att}}\texttt{=<<nj,rfsel,•>,attsl'=\{o\}>}$ is added for every $\texttt{cls}_{\texttt{nj}} \in \texttt{CLS}_{\texttt{nj}}$. This is done for every recursive flow selector contained in $\texttt{RFSEL}_{\texttt{fun}}$. For our example, $\texttt{RFSEL}_{\texttt{reverse}}=\{\texttt{x}_{\texttt{rfsel}}\}$, and `rfsl1` and `rfsl2` are added to the `cls`'s for `n1` and `n2` respectively. The shape graph obtained, $\texttt{sg}^{\texttt{B}}$, is now ready to be analyzed within the context of `reverse()`.

3.2.3.2 Recursive call-to-start rule

The recursive version of the call-to-start rule ($\texttt{CTS}_{\texttt{rec}}$) determines the context change in a graph that encounters a recursive function call, $\texttt{s} \in \texttt{S}_{\texttt{call\_rec}}$. It works differently from the non-recursive version as the change of context does not involve a change in pointer names but a change in the state of the same pointers within a new recursive context. Fig. 3.8 shows such a context change when calling `reverse()` for the third time in our running example. The algorithm that describes the changes in the shape graph is found in Fig. 3.9

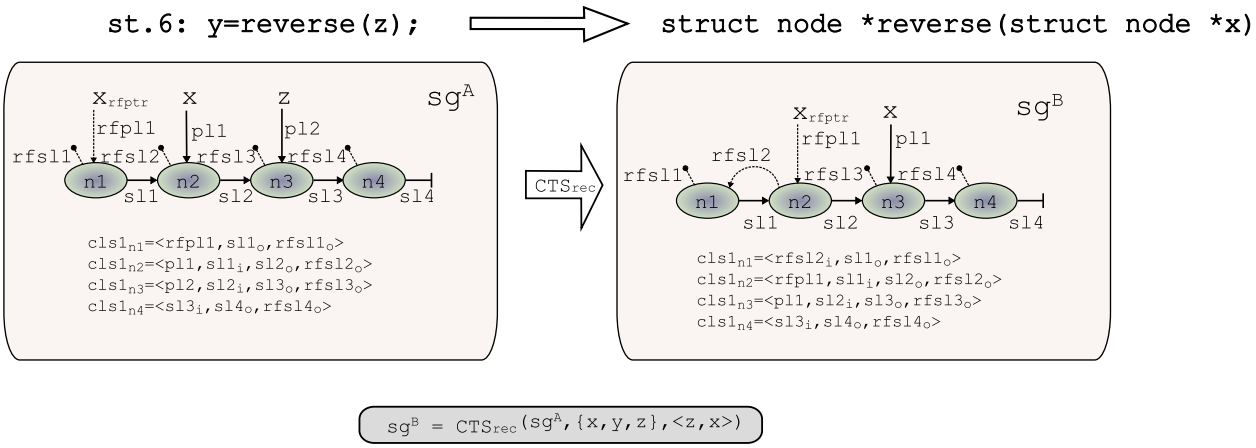

Figure 3.8: Example of graph transformation by the $\mathtt{CTS_{rec}}$ rule.

$\mathtt{CTS_{rec}()}$
Input: $\mathtt{sg^1}$=<$\mathtt{N^1}$,$\mathtt{CLS^1}$>, $\mathtt{PTR_{fun}}$, $\mathcal{AFPM}$(s,fun)
A shape graph, formal and local ptrs for fun and set of pairs <aptr,fptr> of call site s
Output: $\mathtt{RSSG^k}$
A reduced set of shape graphs

$\mathtt{RSSG^2}$=$\mathtt{sg^1}$
forall rfptr $\in$ $\mathtt{RFPTR_{fun}}$
Find ptr and rfsel s.t. $\mathcal{RFPPM}$(rfptr)=ptr and $\mathcal{RFPSM}$(rfptr)=rfsel
$\mathtt{RSSG^3}$=$\bigsqcup^{\mathtt{RSSG}}_{\forall \mathtt{sg'} \in \mathtt{RSSG^2}}$XSelY(sg',ptr,rfsel,rfptr) # ptr->rfsel=rfptr
$\mathtt{RSSG^4}$=$\bigsqcup^{\mathtt{RSSG}}_{\forall \mathtt{sg''} \in \mathtt{RSSG^3}}$XY(sg'',rfptr,ptr) # rfptr=ptr
$\mathtt{RSSG^5}$=$\bigsqcup^{\mathtt{RSSG}}_{\forall \mathtt{sg'''} \in \mathtt{RSSG^4}}$XNULL(sg''',ptr) # ptr=NULL
endfor
forall x $\in$ $\mathtt{APTR_s}$
Find the pair <aptr,fptr> $\in$ $\mathcal{AFPM}$(s,fun) s.t. x=aptr
$\mathtt{RSSG^3}$=$\bigsqcup^{\mathtt{RSSG}}_{\forall \mathtt{sg'} \in \mathtt{RSSG^2}}$XY(sg',fptr,aptr) # fptr=aptr
If (aptr $\notin$ GLB),
$\mathtt{RSSG^4}$=$\bigsqcup^{\mathtt{RSSG}}_{\forall \mathtt{sg''} \in \mathtt{RSSG^3}}$XNULL(sg'',aptr) # aptr=NULL
else
$\mathtt{RSSG^4}$=$\mathtt{RSSG^3}$
$\mathtt{RSSG^2}$=$\mathtt{RSSG^4}$
endfor
$\mathtt{RSSG^k}$=$\mathtt{RSSG^2}$
return($\mathtt{RSSG^k}$)
end

Figure 3.9: The $\mathtt{CTS_{rec}()}$ function.

First, for every recursive flow pointers rfptr $\in$ $\mathtt{RFPTR_{fun}}$ considered for the function, we assign it to the node pointed to by the tracked pointer ptr, prior to the change of context. Also the associated recursive flow selector rfsel is used to leave a trace the the previous node for rfptr. When the trace is established, the tracked pointer ptr can be nullified. Its value for the new context will be set in the next phase of this algorithm.

In this example, only x is traced along the interprocedural control flow. In sg^B from Fig. 3.8, rfsl2 is set between n2 and n1, to keep track of the location where x was pointing to two calls back in the ARS. Then, x_{rfptr} is made to point to n2 in sg^B. This step is completed by nullifying x. Next in the algorithm, actual and formal parameters are matched. Pointer formal parameter x is assigned to n3 in sg^B, which was pointed to by actual parameter z in sg^A. Since z is not a global pointer, it is nullified for the new context.

3.2.3.3 Recursive return-to-call

The recursive return-to-call rule (RTC_{rec}) describes the context change when returning to a recursive call. Fig. 3.10 shows such a context change when returning from the third call to reverse(). The algorithm that describes these changes is depicted in Fig. 3.11.

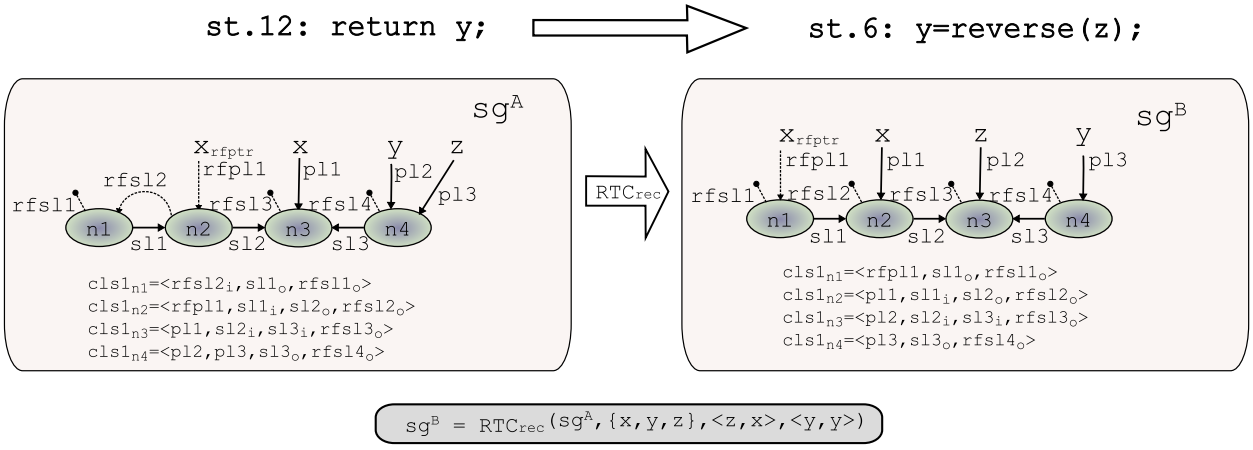

Figure 3.10: Example of graph transformation by the RTC_{rec} rule.

First, the pointer assigned at the recursive call statement, assptr, is made to point to the node pointed to by the pointer returned by the function return statement, retptr. In the example we showcase here, this involves no change as assptr=retptr=y. Then, actual parameters from the previous context are recovered by matching with formal parameters with the $\mathcal{AFPM}$ mapping: z now points to n3 in sg^B (Fig. 3.10), the node that its matching formal parameter x was pointing to in sg^A. Remember that we do not allow pointer formal parameters to be modified (see section 3.1).

Finally, the previous state of recursive flow links is restored: x in sg^B points to n2, which was pointed to by x_{rfptr} in sg^A and, from the new state of x, recursive flow pointer x_{rfptr} follows through the $x\text{->}x_{rfsel}$ path to n1. The recursive flow selector link rfsl2 from sg^A is nullified now, once its path has been followed.

3.2.3.4 Non-recursive return-to-call

The non-recursive version of the return-to-call rule (RTC_{nrec}) is the last context change rule. It performs the appropriate context change when returning to a non-recursive call site, $s \in S_{call\_nrec}$. Let us consider now the sg^A shape graph in Fig. 3.12. It is one of the *function summaries* obtained in the analysis of reverse(). It is achieved at the fixed point for the analysis of the function. On top of that, it is a shape graph that is eligible for the non-recursive return, as it represents a heap state for the first call to reverse(). This can be easily deduced by the fact that recursive flow pointer x_{rfptr} is not assigned. This means that x could not have been assigned at a previous recursive call. For a full reference of all the summary shape graphs resulting from the analysis of reverse(), please refer to Appendix B.

$\texttt{RTC}_{\texttt{rec}}\texttt{()}$
Input: $\texttt{sg}^1$=<$\texttt{N}^1$,$\texttt{CLS}^1$>, $\texttt{PTR}_{\texttt{fun}}$, $\mathcal{AFPM}$(s,fun), $\mathcal{RAPM}$(s,fun)
A shape graph, formal and local ptrs for fun, set of pairs <aptr,fptr> of call site s,
and the corresponding <retprt,assptr> pair
Output: $\texttt{RSSG}^k$
A reduced set of shape graphs

$\texttt{RSSG}^1$=XY($\texttt{sg}^1$,assptr,retptr) # assptr=retptr
$\texttt{RSSG}^2$=$\texttt{RSSG}^1$
forall x ∈ $\texttt{APTR}_s$ # $\texttt{APTR}_s$ is the set of actual pointers in the call stmt. s
Find the pair <aptr,fptr> ∈ $\mathcal{AFPM}$(s,fun) s.t. x=aptr
$\texttt{RSSG}^3$=$\bigsqcup^{\texttt{RSSG}}_{\forall \texttt{sg'} \in \texttt{RSSG}^2}$XY(sg',aptr,fptr) # aptr=fptr
$\texttt{RSSG}^2$=$\texttt{RSSG}^3$
endfor
forall rfptr ∈ $\texttt{RFPTR}_{\texttt{fun}}$
Find ptr and rfsel s.t. $\mathcal{RFPPM}$(rfptr)=ptr and $\mathcal{RFPSM}$(rfptr)=rfsel
$\texttt{RSSG}^4$=$\bigsqcup^{\texttt{RSSG}}_{\forall \texttt{sg''} \in \texttt{RSSG}^2}$XY(sg'',ptr,rfptr) # ptr=rfptr
$\texttt{RSSG}^5$=$\bigsqcup^{\texttt{RSSG}}_{\forall \texttt{sg'''} \in \texttt{RSSG}^4}$XYSel(sg''',rfptr,ptr,rfsel) # rfptr=ptr->rfsel
$\texttt{RSSG}^6$=$\bigsqcup^{\texttt{RSSG}}_{\forall \texttt{sg''''} \in \texttt{RSSG}^5}$XSelNULL(sg'''',ptr,rfsel) # ptr->rfsel=NULL
$\texttt{RSSG}^2$=$\texttt{RSSG}^6$
endfor
$\texttt{RSSG}^k$=$\texttt{RSSG}^2$
return($\texttt{RSSG}^k$)
end

Figure 3.11: The $\texttt{RTC}_{\texttt{rec}}\texttt{()}$ function.

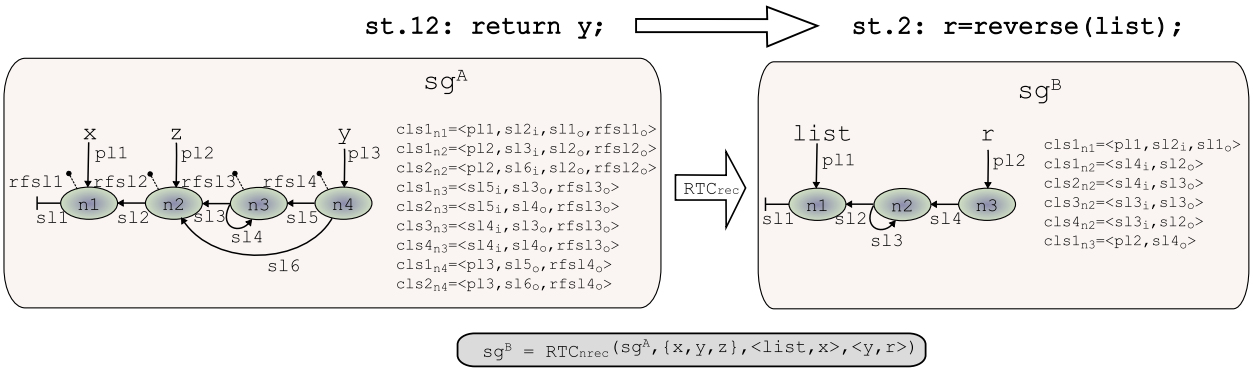

Figure 3.12: Example of graph transformation by the $\texttt{RTC}_{\texttt{nrec}}$ rule.

Fig. 3.13 presents the algorithm that explains the steps involved for $\texttt{RTC}_{\texttt{nrec}}$. First, the shape graph is checked for validity. If the shape graph contains any rfptr for the function which is still assigned, it represents heap states beyond the non-recursive call to the recursive function, and does not represent heap states that must exit the function. On the other hand, if all rfptr's in the function are nullified, then we can proceed with the context change to exit the function. In that case, the next step is to alias the assigned pointer, assptr, with the pointer returned by the function return statement, retptr. In our example, r is made to point to n3 in $\texttt{sg}^B$ (Fig. 3.12), the node that the returned pointer y was pointing to in $\texttt{sg}^A$. Then, the $\mathcal{AFPM}$ mapping is used to determine the matching of actual and formal parameters.

```
RTC_nrec()
  Input: sg^1=<N^1,CLS^1>, PTR_fun, AFPM(s,fun), RAPM(s,fun)
  # A shape graph, formal and local ptrs for fun, set of pairs <aptr,fptr> of call site s,
  # and the corresponding <retprt,assptr> pair
  Output: RSSG^k
  # A reduced set of shape graphs

  If (∃ rfpl ⊂ cls_nj s.t. cls_nj ⊂ CLS^1, being rfpl=<rfptr,nj> with rfptr ∈ RFPTR_fun)
     RSSG^k=∅
  else
     RSSG^1=XY(sg^1,assptr,retptr)          # assptr=retptr
     RSSG^2=RSSG^1
     forall x ∈ APTR_s                      # APTR_s is the set of actual pointers in the call stmt. s
        Find the pair <aptr,fptr> ∈ AFPM(s,fun) s.t. x=aptr
        RSSG^3=⨆_{∀sg'∈RSSG^2}^{RSSG} XY(sg',aptr,fptr)    # aptr=fptr
        RSSG^2=RSSG^3
     endfor
     forall x ∈ PTR_fun,
        RSSG^3=⨆_{∀sg''∈RSSG^2}^{RSSG} XNULL(sg'',x)       # x=NULL
        RSSG^2=RSSG^3
     endfor
     forall rfsel ∈ RFSEL_fun,
        RSSG^3=∅
        forall sg^i=<N^i,CLS^i> ∈ RSSG^2
           forall nj ∈ N^i                  # Remove x_rfsel for all nodes in all graphs
              forall cls_nj={PL_nj,SL_nj} ∈ CLS_nj (being CLS_nj ⊂ CLS^i),
                 Find sl_att1 ⊂ cls_nj being sl_att1=<<nk,rfsel,np>,attsl1>
                 SL_nj=SL_nj-sl_att1
              endfor
           endfor
           RSSG^3=RSSG^3 ∪ sg^i
        endfor
        RSSG^2=RSSG^3
     endfor
     RSSG^k=RSSG^2
  return(RSSG^k)
end
```

Figure 3.13: The $\texttt{RTC}_{\texttt{nrec}}\texttt{()}$ function.

$\mathcal{AFPM}$(st.2,reverse)={<list,x>} makes us point list to the node x was pointing to, at the head of the list.

When returning from a non-recursive call, all pointers in the function fall out of scope so they must be nullified. That is why we remove x, y and z from the graph. As a consequence of nullifying this pointers, some nodes become compatible and the graph is summarized to preserve the normal form. Once we leave the recursive flow of the analysis, the recursive flow selector links that were introduced in $\texttt{CTS}_{\texttt{nrec}}$ are no longer needed, and they are removed.

3.2.3.5 Keeping track of a reduced number of recursive flow links

So far we have overlooked the issue of selecting the proper recursive flow links for a recursive function. This was done for the sake of simplicity when presenting recursive flow links and the context change rules. Here, we address the subject in detail.

In our approach, we have chosen to abstract the information that we require from the ARS as an extension to the links that exist in our shape graph domain. This makes it easy for us to accommodate recursive analysis by introducing new links that are treated just as existing links, so we do not need to change our abstract semantics operations. However, the drawback is that we introduce more complexity in the graph, by adding elements that do not correspond to actual heap elements. Instead they abstract information that is related to the recursive flow of the analysis.

The pointers that are tracked along the interprocedural analysis provide key information for the context change algorithms to work properly. As the reader may have noticed, the content of the $\texttt{RFPTR}_{\texttt{fun}}$ and $\texttt{RFSEL}_{\texttt{fun}}$ sets is checked in these algorithms, to determine the recursive flow links that are generated in the call-to-start rules and followed in the return-to-call rules.

As previously mentioned, in the worst case, for a recursive function `fun`, we need to trace *every* pointer in $\texttt{PTR}_{\texttt{fun}}$, i.e., its locally defined pointers and pointer formal parameters. In fact, a naive approach might do just that. However, the shape graphs would be unnecessarily complicated, because only *some* pointers need to be traced. The rest can be safely deduced by the actual vs. formal parameters matching and the assigned pointer vs. returned pointer matching implicitly occurring at context changes. In particular, only the pointers within a recursive function `fun` that are not actual parameters of a recursive call to `fun` neither they are assigned at any recursive call site, are traced. This is more formally expressed as:

- $\forall\, \texttt{ptr} \in \texttt{PTR}_{\texttt{fun}} \wedge \forall\, \texttt{s} \in \texttt{S}_{\texttt{call\_rec}}$ calling to `fun`, s.t. $\texttt{ptr} \notin \texttt{APTR}_{\texttt{s}} \wedge \texttt{ptr} \neq \texttt{assptr}$, the pointer assigned at `s`, $\exists\, \texttt{rfptr} \in \texttt{RFPTR}_{\texttt{fun}}$ s.t. $\mathcal{RFPPM}(\texttt{rfptr})=\texttt{ptr} \wedge \exists\, \texttt{rfsel} \in \texttt{RFSEL}_{\texttt{fun}}$ s.t. $\mathcal{RFPSM}(\texttt{rfptr})=\texttt{rfsel}$.

For example, let us apply this criterion to all the local pointers for `reverse()`, $\texttt{PTR}_{\texttt{reverse}}=\{\texttt{x},\texttt{y},\texttt{z}\}$. There is only one recursive call site for this function, `st.6:y=reverse(z)`, where $\texttt{APTR}_{\texttt{st.6}}=\{\texttt{z}\}$ and `assptr=y`. Pointer `x` does not belong to $\texttt{APTR}_{\texttt{st.6}}$, nor it is `assptr`, therefore a recursive flow pointer and recursive flow selector must exist for `x`. Conversely, pointer `y` is the assigned pointer, `assptr`, and pointer `z` belongs to $\texttt{APTR}_{\texttt{st.6}}$, so no recursive flow pointers or recursive flow selectors are needed for them.

Usually, just one pointer needs to be traced: the formal parameter used to navigate the structure. In the `reverse()` example, `x` is such a pointer. Carrying just one recursive flow pointer - recursive flow selector pair, such as $\texttt{x}_{\texttt{rfptr}}$-$\texttt{x}_{\texttt{rfsel}}$, for a recursive function is not a heavy burden for our analysis.

On a practical note, the sets $\texttt{RFPTR}_{\texttt{fun}}$ and $\texttt{RFSEL}_{\texttt{fun}}$ are built according to information provided externally to the analysis. This is done through a special preprocessing directive called `excludeRFPTR`. The original code is instrumented to feature this directive, yielding the code for `reverse()` shown in Fig. 3.14. Within the function body the directive `#pragma SAP.excludeRFPTR(y,z)` indicates to the preprocessing pass built onto Cetus that pointers `y` and `z` do not need to be traced along the interprocedural flow of the analysis.

3.2.3.6 Limitations in the use of recursive flow links

The mechanism to analyze recursive functions based on recursive flow links and context change rules has a limitation on the kind of functions that it can analyze. In the presence of more than one recursive call to the

```
  struct node * reverse(struct node *x){
      struct node *y,*z;
      #pragma SAP.excludeRFPTR(y,z)
1:    z=x->nxt;
2:    x->nxt=NULL;
      if(z!=NULL){
3:       #pragma SAP.force(z!=NULL)
4:       y=reverse(z);
5:       #pragma SAP.force(x!=NULL)
6:       z->nxt=x;
      }else{
7:       #pragma SAP.force(z==NULL)
8:       y=x;
      }
9:    return y;
  }
```

Figure 3.14: The `reverse()` recursive function instrumented with the `excludeRFPTR` directive in bold typeface.

same function, pointers that are matched as actual parameters or assigned pointers at a recursive call site, may be untraceable in a subsequent recursive call where they are not used as actual parameters. In such a case, it will not be possible to recover those pointers when returning from a recursive call, and the analysis cannot proceed.

More specifically, we characterize the cases that are unsupported by the the following check:

- Let `s1` and `s2` $\in \mathtt{S}_{\mathtt{call\_rec}}$ be two recursive call sites for `fun`, and let `s1` precede `s2` in the function lexicographic order. If ($\exists$ `ptr` $\in \mathtt{PTR}_{\mathtt{fun}}$ s.t. `ptr` $\in \mathtt{APTRS}_{\mathtt{s1}} \vee$ `ptr=assptr`, pointer assigned at `s1`) $\wedge$ `ptr` $\notin \mathtt{APTRS}_{\mathtt{s2}} \wedge$ `ptr` is *live* after `s2`) $\Longrightarrow$ the analysis cannot trace `ptr` and it must abort.

This check is performed at the Cetus preprocessing pass, prior to the shape analysis. In the case where we encounter a recursive function that cannot be analyzed, the analysis cannot be performed.

Consider the example of Fig. 3.15(a). It shows a recursive function to create a binary tree, `create_tree()`, where pointer `n` is traced with recursive flow links. This function features two recursive function calls, one to create the left subtree and another to create the right subtree. When returning from the call to the left side, pointer `l` points to the root of a subtree that will be used as the left child for the current location pointed to by `n`, therefore it is a *live* value (it will be read before it is reassigned or *killed*). The call for the right side is performed immediately after the call to the left side, though. Since `l` is a local pointer, it is nullified on the context change ($\mathtt{CTS}_{\mathtt{rec}}$). Therefore, it will not be possible to recover it when returning from the right side call. Note that not even a `rfptr-rfsel` pair for `l`, or even `r`, would fix this, as we would not know how to leave the trace for a pointer that has not yet been assigned when encountering a new left side call in the next recursive call.

Fortunately and according to our experience, these cases can be rewritten to an equivalent version that avoids the problem, just by reordering a few statements in a way that preserves the program behavior. This has been done in the code shown in Fig. 3.15(b), where the statements in bold typeface have been rearranged. Here, the subtree reached through `l` is linked as left subtree of `n`, just after returning from the left side call. Once this is done, `l` is no longer needed (it is *dead*), and can be nullified. Then, the right side call can be

```
struct tree_node *create_tree(int depth){
    struct tree_node *n,*l,*r;
    #pragma SAP.excludeRFPTR(l,r)
    if(depth>0){
       n=(struct tree_node *)malloc(...);
       l=create_tree(depth-1);
       r=create_tree(depth-1);
       n->lft=l;
       l=NULL;
       n->rgh=r;
       r=NULL;
    }else{
       n=NULL;
    }
    return n;
}
```

(a)

```
struct tree_node *create_tree(int depth){
    struct tree_node *n,*l,*r;
    #pragma SAP.excludeRFPTR(l,r)
    if(depth>0){
       n=(struct tree_node *)malloc(...);
       l=create_tree(depth-1);
       n->lft=l;
       l=NULL;
       r=create_tree(depth-1);
       n->rgh=r;
       r=NULL;
    }else{
       n=NULL;
    }
    return n;
}
```

(b)

Figure 3.15: (a) A function to create a binary tree whose pointer `l` cannot be traced by our technique. (b) A rearranged version of the same function that works in the same way and that is adequately supported. Rearranged statements appear in bold.

performed with no trouble. In the context of current compilers, this transformation is very simple and can be performed by a previous step in the compilation process.

3.2.4 Data-flow equations and worklist algorithm

The data-flow equations presented in chapter 2, for intraprocedural analysis, accounted for the behavior of loops and branches, e.g., `while` and `if` statements. These statements do not have associated abstract semantics operations, and therefore do not modify the shape graphs, but rather drive the analysis toward its fixed point. Branch statements contemplate the incoming shape graph for all branches and later join the results. Loop statements iterate the analysis of the statements within their body until the shape graphs change no more.

For interprocedural analysis we need to model the effect of function calls and return statements, as we have mentioned earlier. Upon finding a call statement, the context appropriate for the function body is adopted, and upon finding a return statement, the context of the caller is recovered. The analysis is now entitled to find a fixed point for recursive functions as well.

$$
\begin{aligned}
&[\mathtt{ENTRY}_{\mathtt{nrec}}]: \mathtt{RSSG}^{\bullet \mathtt{se}_{\mathtt{fun}}} = \mathtt{IN}_{\mathtt{s} \in \mathtt{S}_{\mathtt{call\_nrec}}}(\mathtt{RSSG}^{\bullet \mathtt{s}}) = \\
&\qquad = \bigsqcup_{\mathtt{sg}^{\mathtt{i}} \in \mathtt{RSSG}^{\bullet \mathtt{s}}}^{\mathtt{RSSG}} \mathtt{CTS}_{\mathtt{nrec}}(\mathtt{sg}^{\mathtt{i}}, \mathtt{PTR}_{\mathtt{fun}}, \mathcal{AFPM}(\mathtt{s},\mathtt{fun})) \\
&[\mathtt{ENTRY}_{\mathtt{rec}}]: \mathtt{RSSG}^{\bullet \mathtt{se}_{\mathtt{fun}}} = \mathtt{IN}_{\mathtt{s} \in \mathtt{S}_{\mathtt{call\_rec}}}(\mathtt{RSSG}^{\bullet \mathtt{s}}) = \\
&\qquad = \bigsqcup_{\mathtt{sg}^{\mathtt{i}} \in \mathtt{RSSG}^{\bullet \mathtt{s}}}^{\mathtt{RSSG}} \mathtt{CTS}_{\mathtt{rec}}(\mathtt{sg}^{\mathtt{i}}, \mathtt{PTR}_{\mathtt{fun}}, \mathcal{AFPM}(\mathtt{s},\mathtt{fun})) \\
&[\mathtt{EXIT}_{\mathtt{nrec}}]: \mathtt{RSSG}^{\mathtt{s}\bullet} = \mathtt{OUT}_{\mathtt{s} \in \mathtt{S}_{\mathtt{call\_nrec}}}(\mathtt{RSSG}^{\bullet \mathtt{sr}_{\mathtt{fun}}}) = \\
&\qquad = \bigsqcup_{\mathtt{sg}^{\mathtt{i}} \in \mathtt{RSSG}^{\bullet \mathtt{sr}_{\mathtt{fun}}}}^{\mathtt{RSSG}} \mathtt{RTC}_{\mathtt{nrec}}(\mathtt{sg}^{\mathtt{i}}, \mathtt{PTR}_{\mathtt{fun}}, \mathcal{AFPM}(\mathtt{s},\mathtt{fun}), \mathcal{RAPM}(\mathtt{s},\mathtt{fun})) \\
&[\mathtt{EXIT}_{\mathtt{rec}}]: \mathtt{RSSG}^{\mathtt{s}\bullet} = \mathtt{OUT}_{\mathtt{s} \in \mathtt{S}_{\mathtt{call\_rec}}}(\mathtt{RSSG}^{\bullet \mathtt{sr}_{\mathtt{fun}}}) = \\
&\qquad = \bigsqcup_{\mathtt{sg}^{\mathtt{i}} \in \mathtt{RSSG}^{\bullet \mathtt{sr}_{\mathtt{fun}}}}^{\mathtt{RSSG}} \mathtt{RTC}_{\mathtt{rec}}(\mathtt{sg}^{\mathtt{i}}, \mathtt{PTR}_{\mathtt{fun}}, \mathcal{AFPM}(\mathtt{s},\mathtt{fun}), \mathcal{RAPM}(\mathtt{s},\mathtt{fun}))
\end{aligned}
$$

Figure 3.16: Data-flow equations for interprocedural support.

```
Worklist()
  Input: P=<FUN,STMT,PTR,TYPE,SEL>, FUN=<FUN_fun,STMT_fun,PTR,TYPE,SEL>, RSSG^in
  # A program or a non-recursive function and an input RSSG
  Output: RSSG^out
  # The RSSG at the exit program point

1:  Create W=STMT
2:  RSSG•se=RSSG^in
3:  ∀ s ∈ STMT → RSSG^s•=∅
4:  repeat
5:      Remove s from W in lexicographic order
6:      RSSG•s=⊔^RSSG_{s'∈pred(s)} RSSG^s'•
7:      Case (s),
8:          s ∈ S_call_nrec
9:              Let fun ∈ FUN, be the function called by s
10:             RSSG^s•=Tabulate(s,<FUN_fun,STMT_fun,PTR,TYPE,SEL>,RSSG•se_fun)
11:             break
12:         s ∈ S_call_rec
13:             Let fun ∈ FUN, be the function called by s
14:             RSSG•se_fun=IN_{s∈S_call_rec}(RSSG•s)
15:             RSSG^s•=Worklist_rec(<FUN_fun,STMT_fun,PTR,TYPE,SEL>,RSSG•se_fun)
16:             break
17:         s ∈ S_return
18:             TAB_fun(CUR_TAB^in)=RSSG•s
19:             Let s' ∈ S_call_nrec, be the non-recursive call site that called fun
20:             RSSG^out=RSSG^s•=OUT_{s'∈S_call_nrec}(RSSG•s)
21:             succ(s)=∅
22:             break
23:         default
24:             RSSG^s•=AS_s(RSSG•s)
25:             break
26:     If (RSSG^s• has changed),
27:         forall s' ∈ succ(s),
28:             W=W ∪ s'
29:         endfor
30: until (W=∅)
31: return(RSSG^out)
end
```

Figure 3.17: The extended worklist algorithm for interprocedural support. It computes the $\mathtt{RSSG}^{s\bullet}$ at each program point.

The process of finding a fixed point in the analysis of interprocedural programs is controlled by the interprocedural data-flow equations. We present them now in Fig. 3.16. They augment the intraprocedural data-flow equations presented in chapter 2. Basically, we present two different equations for the ENTRY/EXIT data-flow transfers from the caller to the callee and from the callee to the caller. We distinguish between non-recursive and recursive calls and returns. In these new equations, we assume that fun is the function called by statement s, $\mathtt{se}_{\mathtt{fun}}$ the entry point at fun and $\mathtt{sr}_{\mathtt{fun}}$ the return point of fun. Equations $[\mathtt{ENTRY}_{\mathtt{nrec}}]$ and $[\mathtt{ENTRY}_{\mathtt{rec}}]$ perform the transfer from the caller to the callee in the case of a non-recursive or a recursive call, respectively; Equations $[\mathtt{EXIT}_{\mathtt{nrec}}]$ and $[\mathtt{EXIT}_{\mathtt{rec}}]$ transfer the analysis back to the caller.

Context change rules are used according to the ENTRY/EXIT and recursive/non-recursive character of each equation.

We present in Fig. 3.17 the extended worklist algorithm for solving the data-flow equations presented. The input of our worklist algorithm is a program P with functions, or a function FUN with its corresponding functions, and an input *reduced set of shape graphs*, $\mathtt{RSSG}^{\mathtt{in}}$. The initial set is empty, i.e., $\mathtt{RSSG}^{\mathtt{in}}=\emptyset$. The output of the algorithm is the $\mathtt{RSSG}^{\mathtt{out}}$ resultant at the exit program or function point. Without loss of generality we assume that there is only one return point on each function. The algorithm also computes the resultant $\mathtt{RSSG}^{\mathtt{s}\bullet}$ at each program point.

Our algorithm processes the worklist using the main loop defined in lines 4–30. We can see that the algorithm is sensitive to the type of statement being processed (line 7). If $\mathtt{s} \in \mathtt{S}_{\mathtt{call\_nrec}}$, i.e., it is a non-recursive call (lines 8–11) the Tabulate() algorithm is called with the call site statement, the function to enter and the RSSG available at that point. The Tabulate() function transforms the incoming shape graphs to the context inside the function called, checking if similar contexts have been analyzed before. If they have, then a previously *tabulated output* is returned, so that the same graphs are not reanalyzed. If the incoming graphs have not been analyzed before, a new instance of the Worklist() algorithm is called for the current function. The Tabulate() function will be covered in detail in section 3.3.

In the case of a recursive call statement, i.e., $\mathtt{s} \in \mathtt{S}_{\mathtt{call\_rec}}$ (lines 12–16), the current shape graphs are first put into the new context, then a new worklist algorithm for the recursive function is called, Worklist_rec() (Fig. 3.18). We shall discuss this algorithm shortly.

In the case of a return statement, i.e., $\mathtt{s} \in \mathtt{S}_{\mathtt{return}}$ (lines 17–22), found within a Worklist() algorithm, it means that we have found the end of the body of the function being analyzed by the algorithm. The shape graphs obtained are tabulated for future reference, and then adapted for the context suitable when returning to the calling statement. To mark the end of the analysis for the current function, the successor set for the return statement is set to empty (succ(s)=$\emptyset$).

If the analyzed statement is another statement (not a function call or return statement, lines 23–25), then the appropriate abstract semantics is applied, as presented in chapter 2. If the graphs change, the successors of the analyzed statement are added to the worklist. If the graphs do not change, this means we have reached a fixed point and the analysis of the function has finished.

Let us now continue with the process of interprocedural analysis by contemplating the Worklist_rec() algorithm (Fig. 3.18). It operates similarly to Worklist(), again considering different cases according to the currently analyzed statement. In the case of a non-recursive call statement or a *regular* statement (not a function call nor a return statement), it works in the same way than the Worklist() algorithm.

The specific behavior comes when encountering a recursive call site or a return statement. For the recursive call site (lines 12–15), the incoming shape graphs are first transformed to the appropriate context with the $\mathtt{CTS}_{\mathtt{rec}}$ rule (called within $\mathtt{IN}_{\mathtt{s}\in\mathtt{S}_{\mathtt{call\_rec}}}()$). Then, the entry point of the function is set as successor for the current statement, so that the function body can be analyzed again for the new context.

On the other hand, when encountering a return statement (lines 16–20), we must return to the recursive call site that called the current function. The graphs are put into the correct context with $\mathtt{RTC}_{\mathtt{rec}}$ (called within $\mathtt{OUT}_{\mathtt{s'}\in\mathtt{S}_{\mathtt{call\_rec}}}()$), and *all* the recursive call sites for the function are set as successors in the analysis. Thus, we propagate the result obtained at the return statement to every possible recursive call that may have called the function. Keep in mind that the context change has been performed according to the actual call statement considered, $\mathtt{s'} \in \mathtt{S}_{\mathtt{call\_rec}}$.

```
Worklist_rec()
  Input: FUN=<FUN_fun, STMT_fun, PTR, TYPE, SEL>, RSSG^in   # A rec. fun ∈ FUN and an input RSSG
  Output: RSSG^out                                          # The RSSG at the exit program point

1:  Create W=STMT_fun
2:  RSSG^•se_fun=RSSG^in
3:  ∀ s ∈ STMT_fun → RSSG^s•=∅
4:  repeat
5:      Remove s from W in lexicographic order
6:      RSSG^•s=⊔^RSSG_{s'∈pred(s)} RSSG^s'•
7:      Case (s),
8:          s ∈ S_call_nrec
9:              Let foo ∈ FUN_fun, be the function called by s
10:             RSSG^s•=Tabulate(s,<FUN_foo, STMT_foo, PTR, TYPE, SEL>, RSSG^•se_foo)
11:             break
12:         s ∈ S_call_rec
13:             RSSG^•se_fun=IN_{s∈S_call_rec}(RSSG^•s)
14:             succ(s)=se_fun
15:             break
16:         s ∈ S_return
17:             Let {s' ∈ S_call_rec ⊂ STMT_fun}      # The recursive call sites at fun
18:             RSSG^out=RSSG^s•=⊔_{s'∈S_call_rec} OUT_s'(RSSG^•s)
19:             succ(s)={succ(s') ∀ s' ∈ S_call_rec ⊂ STMT_fun}
20:             break
21:         default
22:             RSSG^s•=AS_s(RSSG^•s)
23:             break
24:     If (RSSG^s• has changed),
25:         forall s' ∈ succ(s),
26:             W=W ∪ s'
27:         endfor
28: until (W=∅)
29: return(RSSG^out)
end
```

Figure 3.18: The `Worklist_rec` algorithm for recursive support. It computes the $RSSG^{s\bullet}$ at each statement function point.

3.3 Reuse of function summaries

In the context of interprocedural analysis, it is important to be able to reuse the computed effect of functions that have already been analyzed, a technique sometimes referred to as *memoization*. If we store the input-output abstractions obtained for the analysis of a function, then we can reuse the computed result for an equivalent input. For a complex technique like shape analysis, reuse of function summaries is particularly useful, because it may save repetition of costly analyses.

Our implementation employs a tabulation algorithm similar to [48], recording function summaries for reuse under equivalent calling contexts. This task is performed by the `Tabulate()` algorithm. Every time a non-recursive call statement is encountered, the `Tabulate()` algorithm is invoked. It starts by dividing or *splitting* the heap representation according to the *reachability* of actual parameters in `s` and global pointers. This way, for each incoming shape graph, we obtain two graphs: (i) the *reachable graph*,

which abstracts the part of the heap accessible through the function call actual parameters ($\texttt{APTR}_\texttt{s}$) and global pointers (GLB), by following any pointer-chasing path from them; and (ii) the *unreachable graph*, which abstracts the part of the heap that is *not* accessible inside the function called by s, or equivalently, the part of the heap accessible through the rest of pointers in the program ($\texttt{PTR-}\{\texttt{APTR}_\texttt{s} \cup \texttt{GLB}\}$).

Let us present the basic procedure for reusing function summaries. Fig. 3.19 and Fig. 3.20 present a scenario where two singly-linked lists are reversed, with the recursive `reverse()` function presented in this chapter. We start with $\texttt{sg}^1$, the shape graph that captures both lists, pointed to by pointers a and b. For simplicity, only some graphs in the figures feature the `cls`'s associated with them, the rest of the graphs will be just shown as nodes and link-edges. The `cls`'s in $\texttt{sg}^1$ (Fig. 3.19) show that the two lists are independent and that there are no cycles in them, because there is not any `cls` with two incoming `sl`'s.

When encountering the non-recursive call to `reverse()`, the shape graph is split by reachability: $\texttt{sg}^1$ is split starting from a, the pointer actual parameter used for the call. The result is the singly-linked list reached from a, in $\texttt{sg}^2$. The same shape graph, $\texttt{sg}^1$ is split considering the reachability from the rest of pointers, i.e., b, r1 and r2, yielding $\texttt{sg}^3$, which stands for the list reached from b (r1 and r2 are not yet assigned at this point). This way, only the part of the heap abstraction that is reachable inside the `reverse()` function, and thus can be affected by it, is actually passed on to be analyzed.

Once we have obtained the reachable graph, $\texttt{sg}^2$, it is put into context with the $\texttt{CTS}_\texttt{nrec}$ rule, yielding $\texttt{sg}^4$. Next, the `reverse()` function is analyzed using the `Worklist_rec()` algorithm as previously explained. We already presented $\texttt{sg}^5$ as one possible result for the analysis of `reverse()` for the input of a singly-linked list. There are more possible shape graphs for this analysis (see Appendix B), but for simplicity let us continue the example with just $\texttt{sg}^5$. At this point, we now store the input-output pair registered for this analysis run of `reverse()`, $\texttt{TAB}_\texttt{reverse}(\texttt{RSSG}^3)=\texttt{RSSG}^4$. The shape graphs obtained are then put into context with the $\texttt{RTC}_\texttt{nrec}$ rule. Each of the graphs obtained is joined with the unreachable graph split before the call, $\texttt{sg}^3$. As a consequence, we obtain $\texttt{sg}^7$, which is the heap abstraction of the reversed list whose head is now pointed to by r1 and the list pointed to by b. The box in dashed line contains the overall effect of the analysis of `r1=reverse(a)`, where $\texttt{RSSG}^1$ is received as input and $\texttt{RSSG}^6$ is the resulting output. Pointer a is now longer required (it now points to the tail of the first list) and can be nullified, to obtain $\texttt{sg}^8$.

The example continues with a second non-recursive call to `reverse()` in Fig. 3.20. This time we invoke `reverse()` for the list pointed to by b, `r2=reverse(b)`. The reachable graph, $\texttt{sg}^9$, contains only the list pointed to by b, while the unreachable graph contains the list pointed to by r1 (a and r2 are not assigned). The input graph for the analysis of `reverse()`, $\texttt{sg}^{11}$, is the same that was registered when reversing the first list, so we can use the previously computed result, $\texttt{RSSG}^{10}=\texttt{TAB}_\texttt{reverse}(\texttt{RSSG}^9)$, thus saving the analysis time of `reverse()`. Again, the result is put into context by the $\texttt{RTC}_\texttt{nrec}$ rule, and then joined with the unreachable graph to produce the final result $\texttt{sg}^{14}$ in $\texttt{RSSG}^{12}$. Again, the dashed-line box captures the effect of `r2=reverse(b)`, with input $\texttt{RSSG}^7$ and output $\texttt{RSSG}^{12}$. This analysis runs faster than that of `r1=reverse(a)`, because it does not need to analyze `reverse()` again. Finally, pointer b can be nullified to produce the final result of $\texttt{RSSG}^{13}$, with the two lists reversed.

The `Tabulate()` algorithm is depicted in Fig. 3.21. This algorithm is invoked any time the analysis encounters a non-recursive call statement, $\texttt{s} \in \texttt{S}_\texttt{call\_nrec}$. Each incoming shape graph in the input $\texttt{RSSG}^\texttt{in}$ is split by its reaching ($\texttt{APTR}_\texttt{s} \cup \texttt{GLB}$) and non-reaching ($\texttt{PTR-}\{\texttt{APTR}_\texttt{s} \cup \texttt{GLB}\}$) pointers. In this way, we obtain the reachable graph $\texttt{sg}^\texttt{r}$ and unreachable graph $\texttt{sg}^\texttt{u}$. The reachable shape graph is wrapped into a reduced set of shape graphs $\texttt{RSSG}^\texttt{r}$ for the context change by $\texttt{CTS}_\texttt{nrec}$ due in $\texttt{IN}_{\texttt{s}\in\texttt{S}_\texttt{call\_nrec}}()$. The input $\texttt{RSSG}^\texttt{i}$ is kept in $\texttt{CUR\_TAB}^\texttt{in}$ for future tabulation.

If the input $\texttt{RSSG}^\texttt{i}$ was already used for a previous analysis, the stored $\texttt{RSSG}^\texttt{t}$ is obtained without reanalyzing the function. If the input $\texttt{RSSG}^\texttt{i}$ has not yet been analyzed, then we proceed normally by

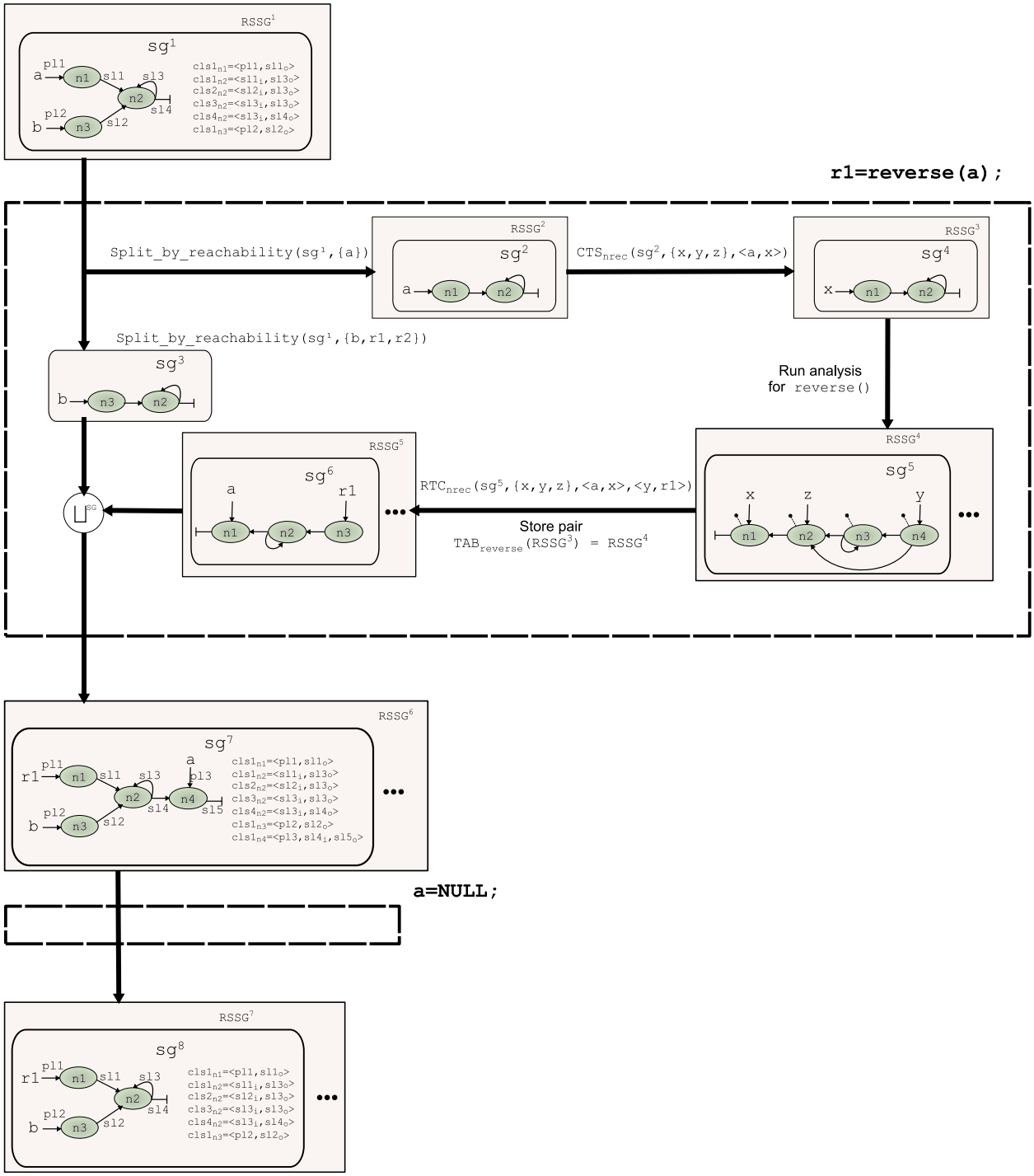

Figure 3.19: Storing pair of input-output RSSG for the analysis of `reverse()`, after splitting incoming shape graph by reachability of reaching and non-reaching pointers.

invoking the `Worklist()` algorithm for the function called by $s \in S_{call\_nrec}$. When that invocation of the `Worklist()` algorithm finds its return statement, then the tabulation pair $\texttt{CUR\_TAB}^{in}\texttt{-RSSG}^{\bullet s}$ is established (line 18 in Fig. 3.17). Note that we store this relation for *each* incoming shape graph in `Tabulate()` (wrapped in $\texttt{CUR\_TAB}^{in}$), which maximizes the possibilities of reuse. Whatever the case the resulting $\texttt{RSSG}^{k}$ is obtained, `Tabulate()` continues by joining each of the shape graphs that capture the behavior of the function with the unreachable shape graph saved previously, $\texttt{sg}^{u}$. The results accumulate

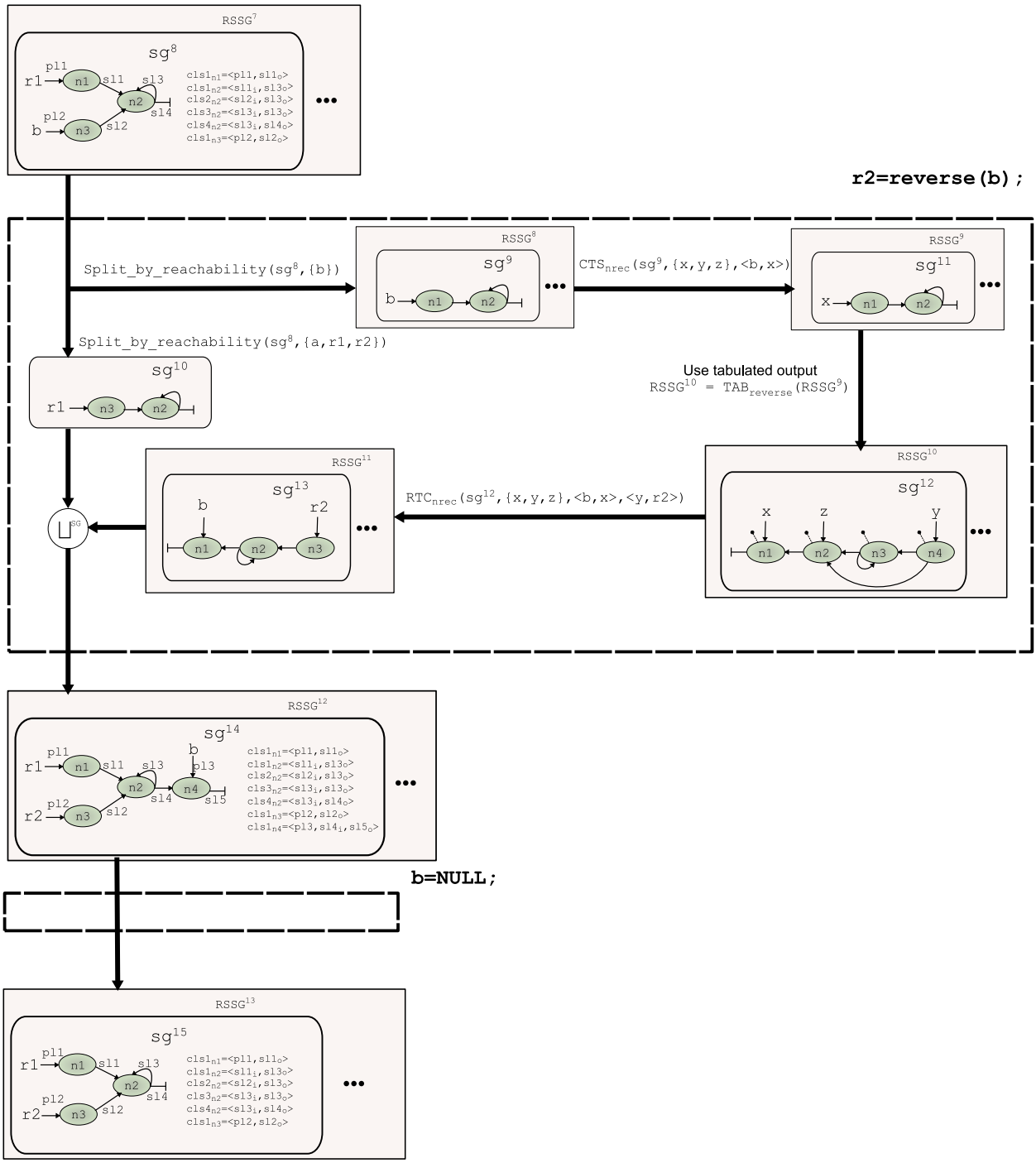

Figure 3.20: Example of function summary reuse when calling `reverse()` with a new list.

in the final RSSG^{out}.

The splitting process is performed by the `Split_by_reachability()` algorithm, featured in Fig. 3.22. It takes as input a shape graph and a set of *reaching pointers*. It performs a cleanup of the input shape graph so that only the portion of the abstract heap that is reachable through these reaching pointers is left. The approach for this algorithm is incremental, i.e. we start with an empty graph, and we add elements as we follow the possible paths from the reaching pointers. For every reaching pointer `ptri`, we first add the node `n1` it directly points to, and its coexistent links sets,

```
Tabulate()
  Input: s, FUN=<FUN_fun, STMT_fun, PTR, TYPE, SEL>, RSSG^in
  # A non-recursive function call statement, the called function and an input RSSG
  Output: RSSG^out                    # The RSSG after the function analysis

1:  RSSG^out=∅
2:  forall sg^i ∈ RSSG^in
3:     sg^r=<N^r,CLS^r>=Split_by_reachability(sg^i,APTR_s ∪ GLB)
4:     If (N^r ≠ ∅ ∧ CLS^r ≠ ∅)
5:        sg^u=Split_by_reachability(sg^i,PTR-{APTR_s ∪ GLB})
6:     else
7:        sg^r=sg^i; N^u=∅; CLS^u=∅; sg^u=<N^u,CLS^u>
8:     RSSG^r=sg^r
9:     RSSG^i=IN_{s∈S_call_nrec}(RSSG^r)
10:    CUR_TAB^in=RSSG^i                # Keep RSSG^i for future tabulation
11:    If (RSSG^i ∈ TAB_fun^keys)
12:       RSSG^t=TAB_fun(RSSG^i)        # Get tabulated output
13:       RSSG^k=OUT_s(RSSG^t)
14:    else
15:       RSSG^k=Worklist(<FUN_fun,STMT_fun,PTR,TYPE,SEL>,RSSG^i)
16:    forall sg^k ∈ RSSG^k
17:       sg^1=Join_SG(sg^u,sg^k)
18:       RSSG^out=RSSG^out ∪ sg^1
19:    endfor
20: endfor
21: return(RSSG^out)
end
```

Figure 3.21: The `Tabulate()` algorithm to calculate and reuse function summaries.

CLS_{ni} (lines 3–5). Then, we find all the selector links with attributes that stem from it, i.e., every sl_{att}=<<ni,sel,n2>,attsl={o}>. We add the node n2 reached through each sl_{att}, and their cls_{n2}'s that feature a reaching path from n1, i.e., the added cls_{n2} must have a selector links with attributes of the form sl_{att}=<<n1,sel,n2>,attsl={i|c|s}>. Now, taking as a starting point the newly added nodes n2's, we repeat the process considering new paths reaching to other nodes in the graph. This iterative process continues until the whole input graph, sg^1, has been scanned (lines 10–20).

For example, consider the splitting process for sg^1 in Fig. 3.19. By calling Split_by_reachability(sg^1,{a}), we obtain in sg^2 the portion of the heap that is accessible inside of the r1=reverse(a) call, since a is the only pointer used as actual parameter, and there are no global pointers. The rest of the heap abstraction is collected by calling the same algorithm, with a different set of pointers: Split_by_reachability(sg^1,{b,r1,r2}). In sg^3 we obtain the part of the heap abstraction that is effectively unreachable inside the r1=reverse(a) call.

It should be noted though that not all input shape graphs can be split. Whenever a shape graph represents memory locations that are found both in the reachable and unreachable graphs, then the graph cannot be safely split, because it could not be reconstructed by a simple join graph operation (Join_SG()). This is similar to the concept of *cutpoint* presented in [48], but more restrictive. In the case of such a *cutpoint*, the analysis must proceed with the whole shape graph and will be less likely to reuse function summaries.

This situation is checked in the last part of the Split_by_reachability() algorithm (lines 22–23,

```
Split_by_reachability()
  Input: sg^1=<N^1,CLS^1>, PTR^1 # A shape graph and a pointer set
  Output: sg^k=<N^k,CLS^k> # Output shape graph as split by the function

1:  N^k=∅; CLS^k=∅
2:  forall ptri ∈ PTR^1
3:      Find ni ∈ N^1 s.t. ∃ pl1=<ptri,ni> ∈ CLS_ni
4:      N^k=N^k ∪ ni
5:      CLS^k=CLS^k ∪ CLS_ni
6:      SL*=∅
7:      forall cls_ni={PL_ni,SL_ni} ∈ CLS_ni s.t. sl_att=<<ni,sel,n2>,attsl={o}> ∈ SL_ni
8:          SL*=SL* ∪ sl_att
9:      endfor
10:     repeat
11:         Remove sl_att=<<n1,sel,n2>,attsl={o}> from SL*
12:         N^k=N^k ∪ n2
13:         forall cls_n2={PL_n2,SL_n2} ∈ CLS_n2 s.t. sl_att=<<n1,sel,n2>,attsl={i|c|s}> ∈ SL_n2
14:             CLS^k=CLS^k ∪ cls_n2
15:         endfor
16:         forall cls_n2={PL_n2,SL_n2} ∈ CLS_n2 s.t. sl_att=<<n2,sel,n3>,attsl={o}> ∈ SL_n2
17:             CLS^k=CLS^k ∪ cls_n2
18:             SL*=SL* ∪ sl_att
19:         endfor
20:     until (SL*=∅)
21: endfor
    # Now check that all reaching pointer-chasing paths are self-contained in the graph
22: If ((∃ ptr1 ∈ PTR s.t.
        (ptr1 ∉ PTR^1 ∧ ∃ pl=<ptr1,nk> ∈ PL_nk s.t. cls_nk={PL_nk,SL_nk} ∈ CLS^k, nk ∈ N^k))
    ∨ (∃ n1 ∈ N^1 s.t.
        (n1 ∉ N^k ∧ ∃ sl_att=<<n1,sel,n2>,att={i|s}> ∈ SL_n2 s.t. cls_n2={PL_n2,SL_n2} ∈ CLS^k, n2 ∈ N^k)))
23:         N^k=∅; CLS^k=∅          # Return empty graph, as it could not be safely split
24: return(sg^k=<N^k,CLS^k>)
end
```

Figure 3.22: The `Split_by_reachability()` algorithm that gets the reachable part of a graph for the given accessing pointers.

Fig. 3.22). If a pointer other than the pointers used as input for the algorithm points to a node contained in the output sg^k, or if a `cls` has reaching paths from both the reaching pointers and the non-reaching pointers, then the graph cannot be split and an empty graph is returned. This case is considered in `Tabulate()` (lines 4–7 in Fig. 3.21), to continue the analysis with an empty unreachable graph and the whole graph as the reachable graph.

Consider the example displayed in Fig. 3.19 if we tried to split sg^7 by the reachability of pointer `b` as is due in call statement `r2=reverse(b)`. In such a case, the graph could not be safely split because non-reaching pointer `a` is pointing to a node reachable from reaching pointer `b`. The reader could argue that the node pointed to by `a` belongs to the first list, while `b` points to a second, independent list. However, due to the summarization involved in the graph, such information is not guaranteed, and sg^7 could also stand for a situation where `a` points to the last element of the second list. Anyhow, the information that the lists are not shared and that there are no cycles is still preserved in the `cls`'s for sg^7. By nullifying pointer `a`,

which is *dead* (no longer used), we obtain $\texttt{sg}^8$, which can be safely split and allows us to reuse the effect of the previous analysis of `reverse()`. Incidentally, this example hints us of the importance of nullifying dead pointer variables in our approach.

3.4 Refining interprocedural analysis

The extensions described so far allow us to extend the intraprocedural shape analysis technique presented in chapter 2 to support interprocedural programs with recursive functions. However, the analysis of recursive functions may be faced with situations where very conservative shape graph abstractions are obtained, yielding the analysis too imprecise for our purposes. In this section, we present some mechanisms to alleviate such problems.

3.4.1 Previous call property to separate traversed and non-traversed nodes

In our approach, the natural method of refining a heap abstraction is by using properties. For interprocedural analysis, we introduce a new property called *previous call property* or simply *PC property*. It takes the value of a set of pointer variables tracked along the recursive analysis. For example, a node annotated with the PC={x} property value is known to have been pointed to by pointer x in a previous recursive call. The basic function of the PC property is to separate the nodes that have been left behind along a recursive traversal from the nodes that are yet to be traversed.

Let us consider the example of Fig. 3.23(a). It shows an arbitrary long singly-linked list in the concrete domain, being traversed in a recursive function `rec_fun()`. In particular, the state presented here is the state found at the start of an arbitrary deep recursive call, after the context change. Pointer x points to the currently accessed element, $\texttt{x}_{\texttt{rfptr}}$ points to the element in the list pointed to by x in the previous recursive call, and the recursive flow selector links in the concrete domain `rfslc`'s, in dotted lines, maintain the trace of x along former recursive calls.

In (b) in the same figure, we show the shape graph abstraction without the PC property. All the memory locations contained within the boxes in dashed lines in (a) are abstracted by n3, because they are not pointed to by pointers and we do not consider any property at this point. Node n3 abstracts the elements in the list that have been traversed through the recursive calls *as well as* the elements that are yet to be traversed. We display the `cls`'s for n3 that capture some particular locations from (a). This representation introduces a shape feature that we would like to avoid: there is the possibility that further down the list, by following through the nxt selector, we find an element that points to another one through a `rfsl`, which is indicative of an element that was traversed in a previous call. This is the case if we follow from $\texttt{cls2}_{\texttt{n3}}$=<...,$\texttt{sl2}_{\texttt{o}}$> to $\texttt{cls4}_{\texttt{n3}}$=<$\texttt{sl2}_{\texttt{i}}$,...,$\texttt{rfsl2}_{\texttt{o}}$>, for example.

In Fig. 3.23(c) we show the graph abstraction for (a) considering the PC property. Now, the nodes that have been involved in the recursive traversal, more precisely, those that have been previously pointed to by x, are annotated with the PC={x} property. For improved presentation, the nodes with their PC property value to an empty set of pointers are shown as nodes without annotation. In this example, the PC property unables the nodes that represent the traversed elements to be summarized with the nodes that represent the rest of elements. This graph is more precise since it does not introduce inconsistent information related to interprocedural flow.

Hence, the previous call property is now added to the initial set defined in chapter 2, now updated as PROP={type,site,touch,PC}. The new instrumentation domain for the PC property is defined as follows:

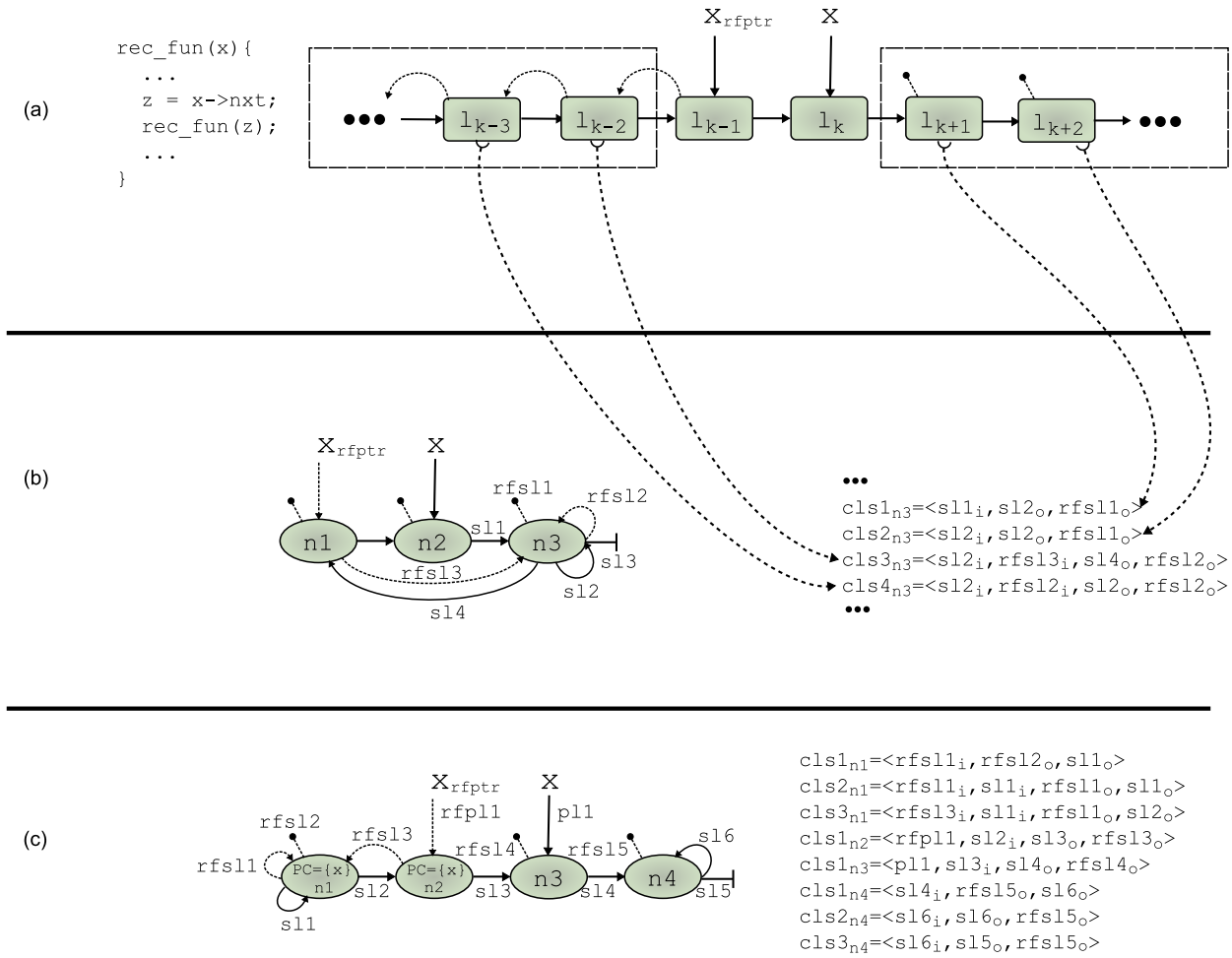

Figure 3.23: An arbitrary long singly-linked list being traversed in a recursive function in (a) the concrete domain, (b) the abstract domain *without* the PC property, and (c) the abstract domain *with* the PC property.

- $\mathsf{P_{PC}}$ is the domain for the property `prop=PC` and it is defined as a set that contains the tracked pointers defined for the program:

  $\mathsf{P_{PC}}=\{\mathsf{p_{PC}}$ s.t. $\mathsf{p_{PC}} \in \mathtt{PTR} \wedge \exists\, \mathtt{rfptr} \in \mathtt{RFPTR} \mid \mathcal{RFPPM}(\mathtt{rfptr})=\mathsf{p_{PC}}\}$

For the PC property the compatibility is defined as equality of the set of pointers contained in the property, i.e., two nodes `n1` and `n2` are compatible with regards to the PC property *iff* $\mathcal{PPM}_{\mathsf{PC}}(\mathtt{n1})=\mathcal{PPM}_{\mathsf{PC}}(\mathtt{n2})$.

The information pertaining to the PC property is added in the $\mathtt{CTS}_{\mathtt{rec}}$ rule. When creating a node in a malloc statement, its set of properties is initialized. In the case of the PC property, it is initialized to $\mathsf{p_{PC}}=\emptyset$. Each time a recursive flow pointer advances in a structure traversal, pointing to a node `n`, the PC property value is updated for `n`. In particular, the algorithm for $\mathtt{CTS}_{\mathtt{rec}}$ presented in Fig. 3.9 is modified as shown in Fig. 3.24 to support the changes required by the PC property.

On the other hand, the PC property annotations are removed in the $\mathtt{RTC}_{\mathtt{rec}}$ rule. In the process of returning from a recursive call, recursive flow pointers go back in the opposite direction of the traversal. The nodes they were pointing to are then cleared of their reference in their values of the PC property. The algorithm presented in Fig. 3.11 for $\mathtt{RTC}_{\mathtt{rec}}$ is modified for this purpose in Fig. 3.25.

```
CTS_rec()
  Input: sg^1=<N^1,CLS^1>, PTR_fun, AFPM(s,fun)
  # A shape graph, formal and local ptrs for fun and set of pairs <aptr,fptr> of call site s
  Output: RSSG^k
  # A reduced set of shape graphs

  RSSG^2=sg^1
  forall rfptr ∈ RFPTR_fun
     Find ptr and rfsel s.t. RFPPM(rfptr)=ptr and RFPSM(rfptr)=rfsel
     RSSG^3=⨆_{∀sg'∈RSSG^2}^{RSSG} XSelY(sg',ptr,rfsel,rfptr)    # ptr->rfsel=rfptr
     RSSG^4=⨆_{∀sg''∈RSSG^3}^{RSSG} XY(sg'',rfptr,ptr)           # rfptr=ptr
     If (p_PC ∈ PROP)
        forall sg^i=<N^i,CLS^i> ∈ RSSG^4
           Find nk ∈ N^i s.t. ∃ rfpl1=<rfptr,nk> ⊂ CLS_nk (being CLS_nk ⊂ CLS^i)
           PPM_PC(nk)=PPM_PC(nk) ∪ rfptr
        endfor
     RSSG^5=⨆_{∀sg'''∈RSSG^4}^{RSSG} XNULL(sg''',ptr)            # ptr=NULL
  endfor
  forall x ∈ APTR_s
     Find the pair <aptr,fptr> ∈ AFPM(s,fun) s.t. x=aptr
     RSSG^3=⨆_{∀sg'∈RSSG^2}^{RSSG} XY(sg',fptr,aptr)    # fptr=aptr
     If (aptr ∉ GLB),
        RSSG^4=⨆_{∀sg''∈RSSG^3}^{RSSG} XNULL(sg'',aptr)  # aptr=NULL
     else
        RSSG^4=RSSG^3
     RSSG^2=RSSG^4
  endfor
  RSSG^k=RSSG^2
  return(RSSG^k)
end
```

Figure 3.24: The recursive version of the call-to-start rule extended to support the previous call (PC) property, with the statements in bold.

3.4.2 Force pseudostatements to filter out improper contexts

In our analysis, the results obtained in return statements for recursive functions are first transformed according to the $\mathtt{RTC}_{\mathtt{rec}}$ rule of context change, and then considered for the analysis at the successors of every recursive call site in the function (lines 16–20 in Fig. 3.18).

Therefore, our technique offers limited context-sensitivity: it is fully context-sensitive as long as there are no more than one recursive call site for a function. If there are more, then the contexts from the different recursive call sites are merged for the recursive analysis. Our technique accumulates shape graphs that result from different flow paths in the `return` statement. For correct analysis, all these shape graphs must be considered for all possible flow paths of the program, as we do not have information about what call site performed the call. This can yield the analysis too imprecise for the purposes of dependence detection. We can use certain *force pseudostatements* to palliate this problem.

Let us consider now a binary tree dynamic data structure. Fig. 3.26 shows recursive function `TreeAdd()` that performs a depth-first traverse of such a binary tree, adding the values from left and right children and

```
RTC_rec()
  Input: sg^1=<N^1,CLS^1>, PTR_fun, AFPM(s,fun), RAPM(s,fun)
  # A shape graph, formal and local ptrs for fun, set of pairs <aptr,fptr> of call site s,
  # and the corresponding <retprt,assptr> pair
  Output: RSSG^k
  # A reduced set of shape graphs

  RSSG^1=XY(sg^1,assptr,retptr)          # assptr=retptr
  RSSG^2=RSSG^1
  forall x ∈ APTR_s                      # APTR_s is the set of actual pointers in the call stmt. s
     Find the pair <aptr,fptr> ∈ AFPM(s,fun) s.t. x=aptr
     RSSG^3=⊔^RSSG_{∀sg'∈RSSG^2} XY(sg',aptr,fptr)    # aptr=fptr
     RSSG^2=RSSG^3
  endfor
  forall rfptr ∈ RFPTR_fun
     Find ptr and rfsel s.t. RFPPM(rfptr)=ptr and RFPSM(rfptr)=rfsel
     RSSG^4=⊔^RSSG_{∀sg''∈RSSG^2} XY(sg'',ptr,rfptr)          # ptr=rfptr
     If( p_PC ∈ PROP)
        forall sg^i=<N^i,CLS^i> ∈ RSSG^4
           Find nk ∈ N^i s.t. ∃ rfpl1=<rfptr,nk> ⊂ CLS_nk (being CLS_nk ⊂ CLS^i)
           PPM_PC(nk)=PPM_PC(nk)-rfptr
        endfor
     RSSG^5=⊔^RSSG_{∀sg'''∈RSSG^4} XYSel(sg''',rfptr,ptr,rfsel)    # rfptr=ptr->rfsel
     RSSG^6=⊔^RSSG_{∀sg''''∈RSSG^5} XSelNULL(sg'''',ptr,rfsel)     # ptr->rfsel=NULL
     RSSG^2=RSSG^6
  endfor
  RSSG^k=RSSG^2
  return(RSSG^k)
end
```

Figure 3.25: The recursive version of the return-to-call rule extended to support the previous call (PC) property, with the statements in bold.

storing them in the current tree element. As a result of the execution of `TreeAdd()`, the root of the tree contains the sum of all the values in the tree. This function is drawn from the `TreeAdd` Olden benchmark suite [30], but modified to write in every tree node the value obtained as the sum of the values of the left and right children, like in [56].

Statements 1 and 2 are typical force pseudostatements that ensure proper abstractions in each branch of the `if(t==NULL)` statement. Pseudostatements in bold though are added to filter out improper contexts in recursive analysis.

In Fig. 3.27, mc^1 shows the memory configuration for a binary tree pointed to by pointer `t`. Shape graph sg^1 shows the shape graph abstraction for the binary tree. Note how all elements within the dashed-line box in mc^1 are abstracted in `n2` in sg^1, as we use no properties for this abstraction. Also note that, within the `cls`'s in sg^1, there is the possibility that the tree is not balanced, i.e. a tree node may have one left child and no right child, and viceversa, so we are actually abstracting not just a typical balanced binary tree but also other variations of trees with whatever number of children following different paths from the root.

The shape graph sg^1 is found at the function entry at the beginning of the `TreeAdd()` analysis. After

```
   int TreeAdd (struct tree *t){
       int total_val,value,leftval,rightval;
       struct tree *tleft,*tright;
       #pragma SAP.excludeRFPTR(tleft,tright)
       if (t==NULL) {
1:        #pragma SAP.force(t==NULL)
          total_val=0;
       }else{
2:        #pragma SAP.force(t!=NULL)
3:        tleft=t->left;
4:        leftval=TreeAdd(tleft);
5:        #pragma SAP.force(t!=NULL)
6:        #pragma SAP.force(t->left==tleft)
7:        tleft=NULL;
8:        tright=t->right;
9:        rightval=TreeAdd(tright);
10:       #pragma SAP.force(t!=NULL)
11:       #pragma SAP.force(t->right==tright)
12:       tright=NULL;
          value=t->val;
          total_val=value+leftval+rightval;
          t->val=total_val;
       }
13:    return total_val;
   }
```

Figure 3.26: The `TreeAdd()` recursive function instrumented with the `force` pseudostatements that allow proper context filtering displayed in bold typeface.

traversing the whole tree and adding the values, we obtain this same shape graph as the final result of the analysis, at the return statement of the first, non-recursive call of `TreeAdd()`. During the process of achieving the fixed point for all statements in the function body, sg^1 is transformed by the RTC_{rec} rule and used for the analysis of the successors of the recursive call sites at statements 4 and 9, like every other shape graph that reaches the function return statement.

For example, if we apply the RTC_{rec} rule over sg^1 returning to the left side call site, we obtain sg^2. Note that since sg^1 has no pointer t_{rfptr} assigned, then the tracked pointer `t`, would not be assigned either in sg^2. However, such a graph would be inappropriate as returned by a *recursive* call to `TreeAdd()`, where `t` should be assigned to the current tree node. We can filter this situation with the force pseudostatements `st.5=st.10=#pragma SAP.force(t!=NULL)`. These force pseudostatements prevent the graphs where `t` is not assigned from being analyzed by the successors of both recursive call sites (st. 4 and st. 9).

Considering the left side call, we can safely use the condition `t!=NULL` for the pseudostatements, because (i) we read successfully `t` in `st.3:tleft=t->left` (under assumption of code correctness) and (ii) we know that `t` could not be modified by another recursive call because it is a local variable for the current context and it was not modified between `st.3` and `st.5`. Similarly, we can use the same condition for the right side call.

Let us elaborate further, by considering a possible memory configuration at the return statement of the `TreeAdd()` function. In particular, mc^2 in Fig. 3.28. This state is found when traversing through the `left` selector twice from the root element. Reaching the return statement for this configuration means that we

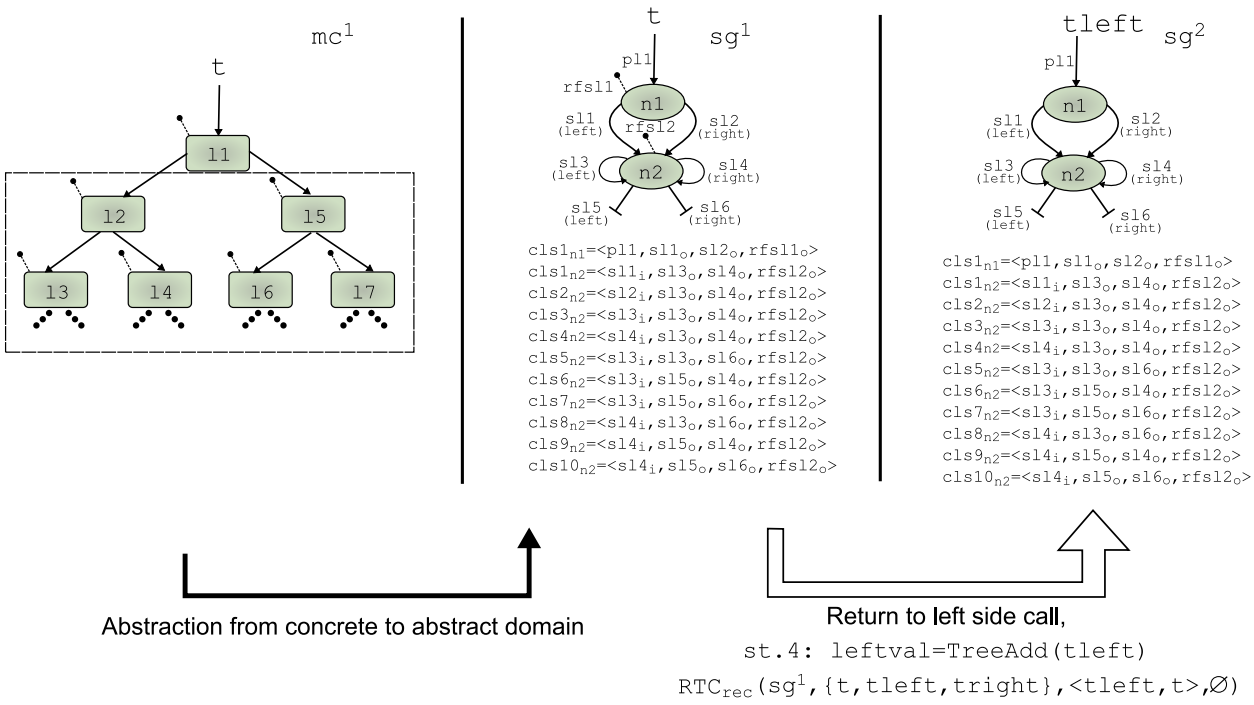

Figure 3.27: A binary tree abstracted to the abstract domain, and then used for returning to the left side call in `TreeAdd()`.

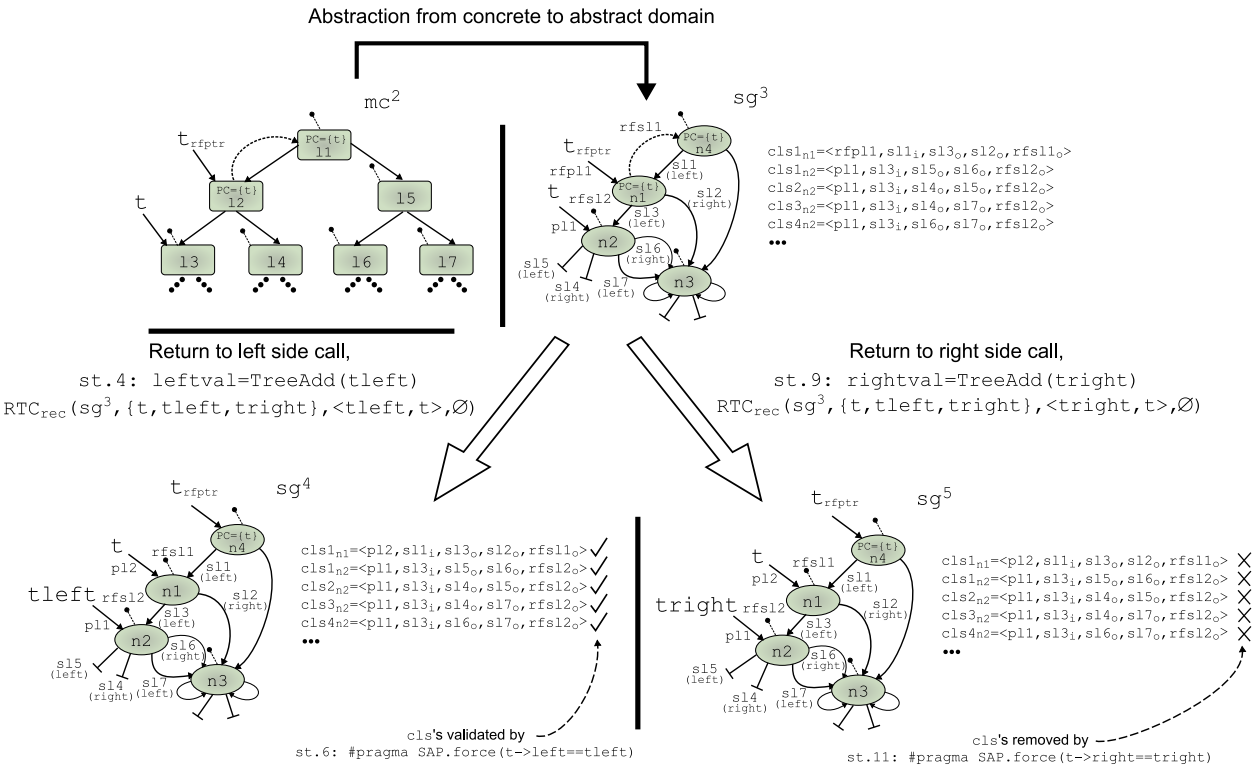

Figure 3.28: The use of force pseudostatements to filter out improper contexts when returning to different call sites.

have already traversed the left and right children from `l3`, and stored in it the sum of the reachable elements from `l3`. The shape graph that corresponds to this memory configuration is sg^3.

As discussed previously, in our scheme every shape graph obtained at the return statement of a recursive function must be passed to the recursive call sites as a possible result of a previous recursive call. If we consider sg^3 for the change of context to the left side recursive call site (st.4), we obtain sg^4, where tleft now points to the node pointed to by t in the previous context (sg^3). On the other hand, if we consider sg^3 for the change of context to the right side recursive call (st.9), we obtain sg^5, where tright now points to the node pointed to by t in the previous context.

However, sg^5 does not make sense as a valid shape graph abstraction for the successors of the right side call, because tright now points to the node reached through t->left, not through t->right. A mirror case would happen if we had traversed through the left, then right path from the root in mc^2, and we were trying to return to the left side call. In other words, we may mix contexts resulting from the left side call as returning to the right side call and viceversa.

We can prevent sg^5 from progressing through the successors of the right side call by using the force pseudostatement st.11:#pragma SAP.force(t->right==tright). Every cls from CLS_{n1} and CLS_{n2} features selector link sl3, which connects n1 to n2 through left. As a consequence, these sets are left empty, because no cls in them satisfies the condition expressed by the force pseudostatement. The subsequent normalization process yields an empty graph, therefore no new graph progresses from sg^5 to be analyzed by the next successors of the right side call.

By the same mechanism, pseudostatement st.6:#pragma SAP.force(t->left==tleft) filters out improper shape graphs for the successors of the left side call site. Note though that sg^4 in Fig. 3.28 is a valid shape graph to return to the left side call. This shape graph is not filtered out by st.6, as it verifies that through t->left we find the node pointed to by tleft. Therefore, we are achieving a proper filtering of contexts between the left and right side calls. For a full reference on the behavior of the force pseudostatements, you can refer to Appendix A.

```
fun(fp1, fp2, ...){
        ...
  tr:   x=y->sel;
        ...
  rcs:  fun(x,ap2, ...);
  f1:   #pragma SAP.force(y!=NULL)
  f2:   #pragma SAP.force(y->sel==x)
        ...
        return;
}
```

Figure 3.29: General scenario of applying force pseudostatements to filter out improper contexts for recursive analysis.

Next, we will establish the conditions that we need to safely incorporate these force pseudostatements to filter out improper contexts for recursive analysis. For this purpose, we consider the generalized presentation of a recursive function fun() in Fig. 3.29. It features a traversing statement labeled tr, and a recursive call site labeled rcs, that recursively calls fun() with the acquired x as actual parameter. This scenario is typical of recursive traversals of dynamic data structures. The two kinds of force pseudostatements discussed in this section are added as the first successors of the call site, with labels f1 and f2, to filter out improper contexts for recursive analysis.

For the scheme presented in Fig. 3.29, force pseudostatements labeled f1 is correct if the following conditions hold:

- Pointer y is not written (*killed*) between tr and rcs.
- Pointer y is a local pointer or formal parameter for fun(), i.e., $y \in PTR_{fun}$.

Firstly, consider that the access through y in tr ensures that y is not NULL at that statement, since we assume code correctness and therefore no NULL dereferences are expected. Then, we need to guarantee that y is not modified from the point where it is used to assign x, to the moment the force pseudostatement f1 is analyzed. Note that we consider actual parameters passed by value, so even if ap2 in rcs is y, it will not be modified as it is a local or formal parameter in fun().

Force pseudostatement f2 is safe to apply if the following conditions hold:

- Pointers x and y are not written (*killed*) between tr and rcs.
- Pointers x and y are local or formal parameter for fun(), i.e., $x, y \in PTR_{fun}$.
- There is no write through selector sel either in the body of fun() or in any function called from fun().

This way we ensure that x and y are not modified between tr and rcs. Furthermore, no write through the sel may be performed so that the condition y->sel==x can hold safely. Of course, it might be the case that some value through sel is written that does not affect the condition we are considering here. However, since the insertion of the force pseudostatements is performed in the Cetus-based preprocessing phase, we do not have access to shape information at this point. These conditions could be overly restrictive, but they work just fine for typical recursive traversals of dynamic data structures.

3.4.3 Paired selectors property

There is yet another undesired effect that occurs in recursive analysis, particularly with tree-like data structures. This effect is found for summary nodes that abstract several jumps back of a tracked pointer along recursive flow path and some other selectors related to it.

Consider shape graph sg^1 in Fig. 3.30. It shows a possible abstraction found at the return statement of the TreeAdd() function (Fig. 3.26), where we have followed an uncertain number of calls through the left child from the root, reaching n1 as the node pointed to by t in the immediately previous call. From there, we may have taken the left or right path and traversed all the children below n1 in the tree, being n2 the node pointed to by the current value of pointer t. Note how in n4 we have accumulated the memory locations that have been traversed through the left child path. The previous locations of pointer t are traced back by the recursive flow selector t_{rfsel}.

We return now to the previous context, by calling RTC_{rec}, obtaining sg^2 (still Fig. 3.30). In the process of returning to the previous context at the left call site, tleft now points to the node previously pointed to by t, t now points to the node previously pointed to by t_{rfptr}, and t_{rfptr} points to a new node materialized from the summary node n4 in sg^1. To sum up, we have traced back to the shape graph abstraction in the context of the previous left side recursive call.

Consider now $cls2_{n4}$ in sg^1. Recursive flow selector link rfsl1 and selector link sl1 are both registered with the incoming *and* outgoing attributes. Take $rfsl1_o$ and $sl1_i$, for instance. All this cls is saying, regarding these links, is that from the current location abstracted by n4 we may go through t_{rfsel} to another location abstracted by n4, and that from another location abstracted by n4 the current location can be reached through left. That "other location" that t_{rfsel} is pointing to from the current location

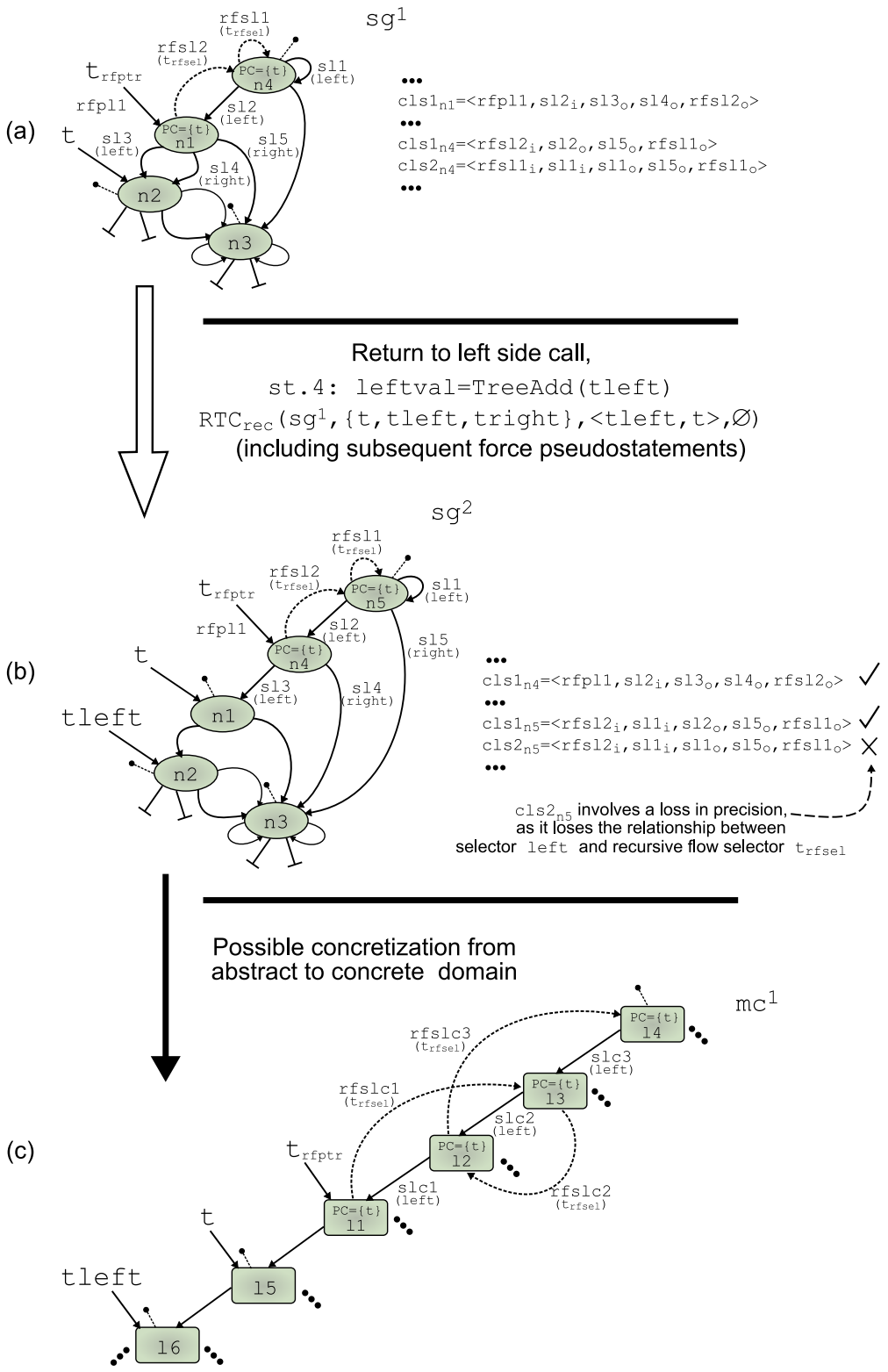

Figure 3.30: (a) A shape graph found at the return statement in `TreeAdd()`. (b) The shape graph obtained after applying $\mathtt{RTC}_{\mathtt{rec}}$ for the left side call and subsequent force pseudostatements over the graph in (a). (c) A possible concretizacion of the graph in (b) for the concrete domain. Note how the relation between `left` and $\mathtt{t}_{\mathtt{rfsel}}$ may be lost.

does not need to be the same "other location" that is pointing back through left. Thus, the materialization operation must assume that any combination of the t_{rfsel} and left links is possible. This means we may obtain $cls1_{n4}$ and $cls1_{n5}$ in sg^2, which keep the relation that from n4 we point to n5 through t_{rfsel}, and from n5 we point to n4 through left. This matches correctly with the binary tree data structure. But we may also get $cls2_{n5}$, where n5 is *not* pointing back to n4 through lft, but to other location also abstracted in n4.

Memory configuration mc^1 in Fig. 3.30 shows a *concretization* for this case in the concrete domain, where l1 is concretized based on $cls1_{n4}$ in sg^2 and l3 is concretized based on $cls2_{n5}$. It is easier to see in this concretization the effect that we want to avoid.

Furthermore, this inaccuracy propagates in subsequent context changes finally producing cycles in the tree, where children can point to their parents. This supposes a loss of accuracy in the shape of the data structure that we need to avoid for correct shape abstraction and subsequent client analysis. The originating factor is that the left and t_{rfsel} links are not interrelated in the summary node. Therefore, the analysis does not have enough information to rule out the materialization possibilities that induce a lack of precision.

To solve this shortcoming we introduce a new property which keeps information about how pair of links relate to each other in a recursive data structure. We name it the *paired selectors property*, or just PS property. The value annotated by the property is the set of interrelated pairs of selectors with their attributes.

Consider Fig. 3.31, which mirrors the case just presented but with PS property information. In sg^3 we find the same shape graph as sg^1 in Fig. 3.30, but n1 and n4 are annotated with the PS property. The property map establishes that for n4 we have $\mathcal{PPM}_{PS}$(n4)={<t_{rfsel_i}, $left_o$>,<$left_i$, t_{rfsel_o}>}. This means that the recursive flow selector t_{rfsel} is *paired* with the selector left in two possible ways: when one of them is incoming from other node, the other is then outgoing to that *same* node, and *not* to any other. The property establishes a *contract* with regards to these two links in two neighboring nodes: for a cls that relates two neighboring nodes through any of the links (selector or recursive flow selector) recorded in the PS property, *at least one* of the pairings recorded *must* hold.

Each cls for n5 in sg^4 must verify <t_{rfsel_i}, $left_o$> or <$left_i$, t_{rfsel_o}> to be valid. Non conforming cls's were removed at the last stage of materialization of n4 from n5. For example, n4 and n5 are connected, as directly neighboring nodes, by rfsl2=<n4, t_{rfsel}, n5> and sl2=<n5, left, n4>. Coexistent links set $cls1_{n5}$=<$rfsl2_i$, $sl1_i$, $sl2_o$, $sl5_o$, $rfsl1_o$> in sg^4 in Fig. 3.30 verifies the <t_{rfsel_i}, $left_o$> relation with $rfsl2_i$ and $sl2_o$. Therefore, $cls1_{n5}$ passes the filtering imposed by the PS property and it is present both in sg^2(Fig. 3.30) and sg^4(Fig. 3.31).

However, $cls2_{n5}$=<$rfsl2_i$, $sl1_i$, $sl1_o$, $sl5_o$, $rfsl1_o$> in sg^2 does not verify <t_{rfsel_i}, $left_o$> ($sl2_o$ does not appear in $cls2_{n5}$) nor <$left_i$, t_{rfsel_o}> (there are no links to support that relation between n4 and n5). The materialization operation takes into account these considerations, and $cls2_{n5}$ is not allowed to exist in sg^4. One possible concretization of shape graph sg^4 is shown as memory configuration mc^2 (Fig. 3.31(c)), where the appropriate relationship between t_{rfsel} and left is preserved.

As we progress in the tree analysis, the summary node n_{summ} for the locations that have been pointed to by instances of the tracked pointer t in previous recursive calls (like n5 in sg^4), will hold the following value of the PS property: $\mathcal{PPM}_{PS}$(n_{summ})={<t_{rfsel_i}, $left_o$>, <$left_i$, t_{rfsel_o}>, <t_{rfsel_i}, $right_o$>, <$right_i$, t_{rfsel_o}>}. In other words, we acknowledge the fact that the recursive flow selector t_{rfsel} is related either with the left or the right selectors.

Finally, the set of properties defined for our shape analysis strategy is configured as PROP={type, site, touch, PC, PS}. The new instrumentation domain for the PS property is defined as follows:

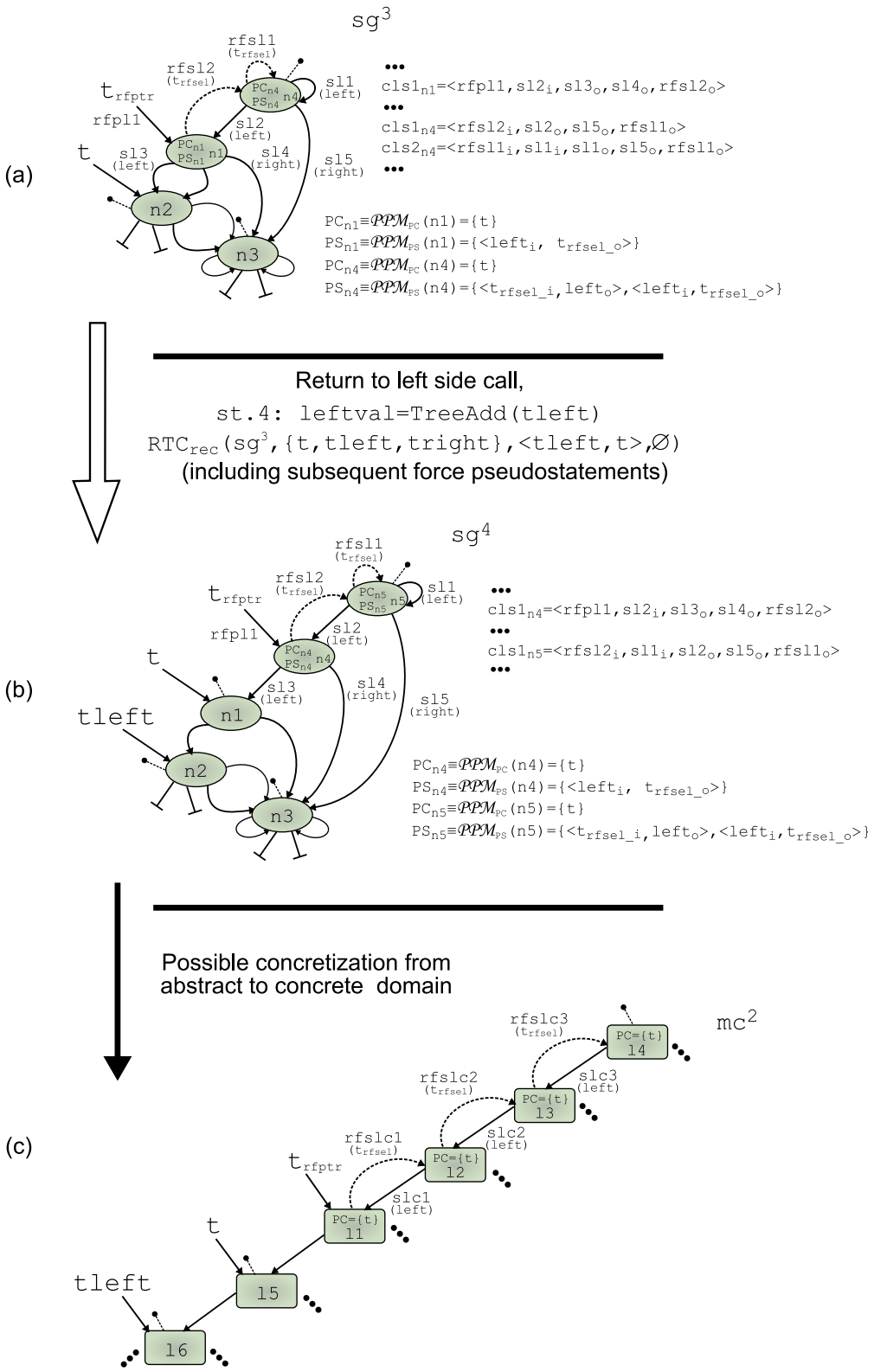

Figure 3.31: (a) A shape graph found at the return statement in `TreeAdd()`, with PS info. (b) The shape graph obtained after applying $\mathtt{RTC}_{\mathtt{rec}}$ for the left side call and subsequent force pseudostatements over the graph in (a). (c) A possible concretizacion of the graph in (b) for the concrete domain. Note how the relation between `left` and $\mathtt{t}_{\mathtt{rfsel}}$ is preserved.

- P_{PS} is the domain for the property prop=PS and it is defined as a set that contains the pairs of selectors with attributes that establish input-output relationships over nodes:

  P_{PS}={<$sel1_{att1}$, $sel2_{att2}$> s.t. sel1, sel2 $\in$ SEL $\cup$ RFSEL, att1, att2 $\in$ {i|o} $\wedge$ att1 $\neq$ att2 }

The compatibility criterion for the PS property is different though than the equality criterion presented so far. Two nodes n1 and n2 are compatible regarding the PS property if they have some value for the property, or if they do not have any value for it. In the case of compatibility, the values of the properties are simply added to the resulting summary node. This way, we summarize information of paired selectors at the same time that we summarize nodes. This controls the growth of the number of nodes with different values for the PS property. The Compatible_Property() function presented in chapter 2 is now completed as shown in Fig. 3.32.

When a new node is created by a malloc() statement, the PS value for the node is initialized to empty, provided the PS property was activated for the analysis. New values may be added to the property when performing linking statements, x->sel=y. Paired selectors are removed when they no longer hold when performing x=y->sel or x->sel=NULL statements. For example, node n4 in sg^4 (Fig. 3.31) is materialized from summary node n5. While the PS value for n5 is $\mathcal{PPM}_{PS}$(n5)={<t_{rfsel_i}, $left_o$>, <$left_i$, t_{rfsel_o} >}, for n4 we have just $\mathcal{PPM}_{PS}$(n4)={<$left_i$, t_{rfsel_o}>}. There is no incoming rfsl with t_{rfsel} for n4, so the pair <t_{rfsel_i}, $left_o$> is clearly no longer applicable to n4, and it is removed from its PS property during the materialization involved in the change of context.

To sum up, we introduce the *paired selectors property* to provide further finesse for node materialization in the presence of interrelated pairs of links in summary nodes. However, the PS property is not only useful for recursive analysis, but also for capturing the shape of any data structure whose elements are linked by two or more selectors with input-output relationships, such as a doubly-linked list. Doubly-linked structures pose a challenge for shape analysis techniques. We will provide examples for these structures in the experimental section.

```
Compatible_Property()
 Input: n1, n2, prop ∈ PROP        # two nodes and a property
 Output: TRUE/FALSE

 If (prop==type ∨ prop==site ∨ prop==touch ∨ prop==PC)
     return(PPM_prop(n1)==PPM_prop(n2))
 If (prop==PS)
     If ((PPM_PS(n1) ≠ ∅ ∧ PPM_PS(n2)=∅) ∨ (PPM_PS(n1)=∅ ∧ PPM_PS(n2) ≠ ∅))
         return(FALSE)
     else
         return(TRUE)
end
```

Figure 3.32: The Compatible_Property() featuring properties: type, site, touch, PC, and PS.

A similar mechanism to our PS property was introduced in [1], called *cyclelinks property* to preserve relationships of paired selectors, tailored for doubly-linked lists. Our PS property is less restrictive when driving the materialization operation and that makes it suitable for the recursive analysis of trees, as well.

3.5 Related work in interprocedural shape analysis

We discuss now some related work regarding interprocedural shape analysis. The first approaches to shape analysis only supported programs without functions or procedures, such as [42], [43], [44], [28], [26], [27] or [29].

There are three main works ([46], [47], [48]) built as extensions to the 3-valued logic analyzer (TVLA) [45], that provided interprocedural support for shape analysis. Each of these works presents its own characteristics. We discuss them next.

As pioneers within graph-based shape analysis, Rinetzky and Sagiv [46] explored the idea of abstracting the *Activation Record stack* as a new entity in the graphs to track the locations of pointers in a sequence of recursive calls. This simple idea and a few new predicates yield an interesting interprocedural shape analysis technique, whose use is only reported for singly-linked lists and that suffers from scalability problems.

Jeannet et. al [47] rely on the computation of *summary transformers* for functions, by solving data-flow equations with modified operators. This is done by using a double vocabulary of predicates to encode the relationship between input and output states. Their analysis is very costly, to the point that it is unable to complete for some simple programs that manipulate binary trees, due to combinatorial explosion of possibilities in the analysis.

Rinetzky, Sagiv, and Yahav [48] improve the previous efforts for interprocedural support in 3-valued shape analyzers with a system that relies on context change rules upon entering to and exiting from functions, and some specific predicates. The key aspect of their analysis comes from its ability to reuse computed function summaries by means of a powerful tabulation algorithm. However, there is a whole range of programs that present so-called *cutpoints*, which involve certain patterns of node linking that are unsupported by their technique.

All the works derived from the TVLA system require the design of specific *predicates* that encode the characteristics of the analyzed data structure. Although predicates for the analysis of doubly-linked lists were identified in [29], they are not used for the implementations of [46], [47], or [48]. Consequently, no tests with doubly-linked lists are reported for these works. Each of the works in TVLA requires different predicates, albeit they share some common ones. The appropriate predicates are found by the analysis designer and require expert knowledge, although [57] introduces machine learning mechanisms to automatically find recursive predicates.

Independently to these works based on TVLA, Hackett and Rugina [53] devised interprocedural data-flow equations and a worklist algorithm for their modular shape abstraction based on *tracked* locations. They are able to analyze programs for a memory leak detection client with significant speed. Their results are based on the accuracy of the underlying points-to analysis required to build the graphs. When it fails to detect enough disjoint regions, the shape of the data structure is not accurately detected, the number of heap configurations rises uncontrollably and the analysis fails. They present limitations in the kind of analyzable structures, not supporting doubly-linked lists. Later, Cherem and Rugina [58] design a specific approach to handle doubly-linked lists, this time with no concern for interprocedural support.

Gotsman, Berdine, and Cook [59] create their own interprocedural shape analyzer based on separation logic. It exploits spatial locality in the abstraction, and is tailored for use solely with linked lists (singly- or doubly-linked) and trees. The support of another kind of structures would require redefining the inductive predicates used as the base for the design. It supports a bounded number of cutpoints in the analysis of recursive functions, treating them just as another parameter for the call. The number of supported cutpoints must be specified as parameter for the analysis. The presence of more cutpoints than the number set would result in imprecise abstractions.

In the design of our extensions for interprocedural analysis, we have been inspired mainly by the works of Rinetzky et. al. From [46] we borrowed the idea of abstracting the information needed for recursive analysis from the *Activation Record Stack*. In our approach, this is done by adding new links to the abstraction. Based on [48], we developed our own variation of the interprocedural worklist algorithm, context change rules, and tabulation mechanism. Finally, we also developed our own set of properties to solve some shortcomings found in the analysis. We have achieved a technique supporting a greater range of structures and with better performance than any of the TVLA approaches known, as we shall see in the next section. It must be stressed that although our work shares ideas with the TVLA framework we provide a new, unrelated shape analysis technique.

Regarding *expressiveness*, some of these works can provide more information about the data structures than our approach. For example, [48] recognizes that a sorting function returns a permutation of elements in the input, and [47] is able to find out that reversing a list twice yields the same list. However, we think that kind of information is tailored for *verification* clients and is not so important for data dependence detection, which is our main concern.

In our approach, we do not impose a restriction on the kind of structure to analyze, being able to obtain correct abstractions for structures that cannot be pigeonholed as pure trees or lists. In the case of cutpoints, our analysis is not able to reuse computed summaries for a function, but it is able to continue the analysis with the whole heap. This results in slower analysis but supports a greater variety of structures and algorithms.

3.6 Experimental results

We have expanded the shape analyzer implementation of chapter 2 with the elements introduced in this chapter, to fully support interprocedural, recursive programs based on dynamic data structures. We have conducted some experiments with the augmented shape analyzer and we describe the results and reflect upon them in the present section.

3.6.1 Interprocedural suite for comparison with related work

We first conducted some experiments over a simple interprocedural suite. The purposes for these tests were: (i) to prove that the algorithm obtains precise memory abstractions in a variety of recursive algorithms that manipulate common dynamic data structures, and (ii) to measure its performance in terms of analysis time and required memory compared to [48], a landmark related work that improves upon previous efforts in [46] and [47]. We have considered some small programs that deal with singly-linked lists or binary trees. These programs are also tested in [48], which allows us to establish a common ground for comparison. All programs are complete, in the sense that they include the allocation of the structures used. For this analysis, the properties *previous call* (PC) and *paired selectors* (PS) were used.

Both [48] and our method are able to determine that the invariants of the structure are preserved after the call to the recursive functions. This means that for the list tests, if the function is called with an acyclic singly-linked list, then the output is also an acyclic singly-linked list, i.e., no cycles have been introduced in the list. For the tree tests, if the input is an unshared binary tree (no child has 2 parents), we verify that the shape is preserved at the output.

We show the results for these experiments in Table 3.1. Next to each program we display a short description of it. The two last columns in the table present the comparison in analysis time (measured in seconds) and memory consumed (measured in MB) between the results presented in [48] and our own

results. The testing platform for [48] is a Pentium M 1.5 GHz with 1 GB. Our platform is very similar: a Pentium M 1.6 GHz with 224 MB. In all cases our method runs in significantly shorter times, while obtaining shape graphs that model precisely the effect of the recursive functions. The memory consumed fits in a block of 1.9 MB for all the list tests. For the tree tests is never goes above 4 MB. In every case, our need for memory is clearly less than that of [48].

Benchmark	Description	Time[48]/this	Space[48]/this
Programs that create and manipulate singly-linked lists			
1-createL	Creates a singly-linked list	9.3 / 0.09	2.3 / 1.9
2-findL	Finds an element in a list	37.1 / 0.49	3.6 / 1.9
3-insertL	Inserts an element in a list	46.8 / 0.38	5.4 / 1.9
4-deleteL	Deletes an element in a list	35.8 / 0.37	3.9 / 1.9
5-appendL	Appends a list at the end of another	22.5 / 0.56	3.9 /1.9
6-reverseL	Reverses a list (example in Fig. 3.1)	21.0 / 0.36	3.4 /1.9
7-revAppL	Reverses a list appending reversed part	41.7 / 0.45	4.3 /1.9
8-spliceL	Splices a list into another	33.6 / 0.48	4.8 /1.9
9-splicex2L	Splices 2 lists into a 3rd	36.5 / 1.25	5.0 /1.9
Programs that create and manipulate binary trees			
10-createT	Creates a binary tree	14.3 / 5.02	2.6 /1.9
11-insertT	Inserts an element in tree	49.6 / 19.4	5.6 / 3.2
12-findT	Finds an element in tree	105.7 / 31.45	6.5 / 4.0
13-heightT	Finds out tree height	76.1 / 15.90	5.4 / 2.9
14-spliceLeftT	Add tree as leftmost child	35.7 / 6.28	5.3 / 2.1
15-rotateT	Exchange children in every node	57.1 /6.12	4.9 /2.2

Table 3.1: Comparison of analysis times and required memory between the approach of Rinetzky et. al. and our method, for a small suite of recursive algorithms that manipulate singly-linked lists and binary trees. Time is measured in seconds, space in MB.

Another difference between the proposal in [48] and ours, is in the asymptotic complexity of the compilation algorithms. For programs without global variables, being nv the maximum number of pointer formal parameters and local pointer variables, the worst case time complexity for [48] is $O(2^{(2^{nv})})$, whereas in our case we have found it is lower than $O(nv^{nv})$. Although our technique is still very complex, these experimental results indicate that the worst case behavior is not reached in practice.

3.6.2 More realistic benchmarks

Once we have checked that our approach yields correct shape abstractions in recursive algorithms dealing with dynamic data structures and that it compares favorably to related work, our next concern is for the analysis of more realistic benchmarks. For that, we have considered the following recursive programs from the Olden suite [30]:

1. `TreeAdd`. Already presented in this chapter (section 3.4.2), this program creates and then traverses a binary tree (Fig.3.33(a)) summing values along the traversal. The partial sums of the subtrees are written in every node in the tree, as in [56].

2. `Power`. This program creates and then traverses a multilevel structure, depicted in Fig.3.33(b). The `struct root` (`R`) element points to an undetermined number of `struct lateral` (`L`) elements through a pointer array. Each of the `struct lateral` elements connect recursively to other elements of the same type, forming singly-linked lists. Besides, each `L` element points to a singly-linked

list of `struct branch` (`B`) elements, which in turn point to several `struct leave` (`Lv`) elements through a pointer array. This programs features a complex structure in a complex control flow of nested recursion (recursive functions that call to other recursive functions), as each level in the structure is traversed.

3. `Bisort`. This is another benchmark from Olden that creates and manipulates a binary tree (Fig.3.33(a)). This program features a recursive function that traverses the tree once and calls to another recursive function that repeatedly traverses the subtrees from the current level. This programs presents another case of nested recursion, with the additional complexity of two recursive calls per recursive function (one for each branch of the tree).

The results gathered for the `16-TreeAdd`, `17-Power`, and `18-Bisort` benchmarks are displayed in Tables 3.2 and 3.3. In every case the dynamic data structures are correctly captured, preserving the defining shape characteristics. The testing platform for these tests is a 3GHz Pentium 4 with 1GB RAM. These three programs are analyzed with the *previous call* and *paired selectors* properties enabled, for adequate recursive analysis. Additionally, `17-Power` uses the *type property* to differentiate between types of elements in the structure (Fig.3.33(b)).

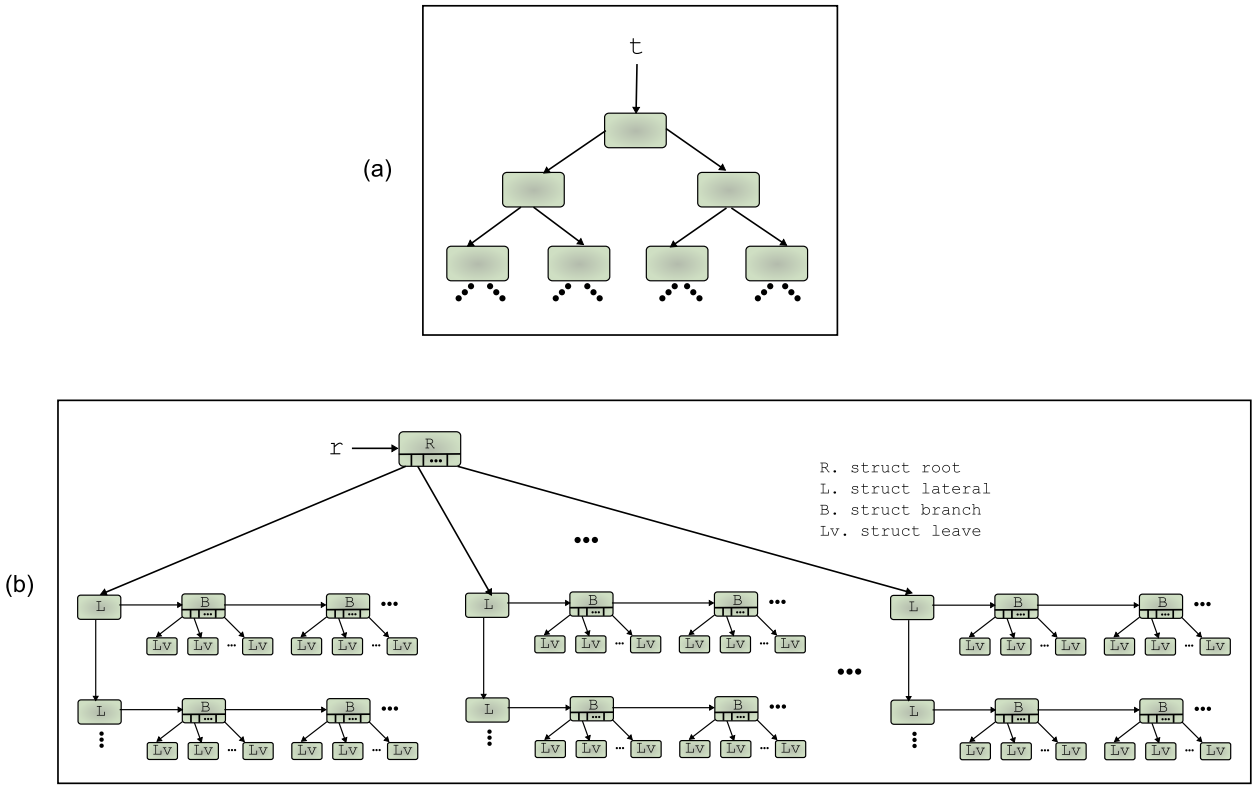

Figure 3.33: The data structures used for the recursive benchmarks from Olden: (a) `16-TreeAdd` and `18-Bisort`; (b) `17-Power`.

Benchmark	Time	Space	Code stmts.	Analyzed stmts.	Shape graphs
16-TreeAdd	6.33 s	3.9 MB	31	2,163	5,763
17-Power	8.13 s	6.4 MB	68	3,541	6,172
18-Bisort	1 m 39.47 s	9.2 MB	50	11,716	39,811

Table 3.2: Metrics of performance and problem size for recursive benchmarks from Olden. The testing platform is a 3GHz Pentium 4 with 1GB RAM.

Benchmark	Sg's per code stmt.	Avg. nodes per sg (max)	Avg. cls's per sg (max)
16-TreeAdd	185.90	3.38 (5)	28.03 (92)
17-Power	90.76	7.68 (11)	28.56 (49)
18-Bisort	1,426.71	5.50 (7)	62.51 (128)

Table 3.3: Shape graph complexity measures for recursive programs.

We observe analysis times in the range of seconds, with `18-Bisort` clocking above one minute. The memory consumed takes a few megabytes, under 10 in the worst case. The number of generated graphs is well into the thousands, peaking at nearly 40,000 for `18-Bisort`. The ratio of shape graphs per code statement (`Sg's per code stmt.` in Table 3.3) is higher than the intraprocedural tests from chapter 2. The complexity of the graphs is in the range of the number of nodes and `cls`'s already registered for the intraprocedural tests in chapter 2.

The results from Tables 3.2 and 3.3, and the comparison with the results of the intraprocedural programs in chapter 2, allow us to draw the following observations:

- As stated previously, the analysis time depends on the number of generated graphs, which in turn is affected mainly by the amount of analyzed statements. The number of shape graphs per code statement is higher than in the examples for intraprocedural analysis, explored in chapter 2. This is specially so for the programs featuring more than one recursive call per recursive function, as it is the case for the benchmarks manipulating trees (`16-TreeAdd` and `18-Bisort`). Naturally, the fixed point takes longer to achieve when there are more possible return points for the shape graphs obtained at a return statement in a recursive function.

- The complexity of the shape graphs is mainly determined by the data structure abstracted. For example, the data structure in `17-Power` is more complex than that of `16-TreeAdd`, and that explains why the first presents more nodes and `cls`'s per graph than the latter. In these tests though, we also find that the size of shape graphs is affected by the way the structure is traversed. Consider the binary tree data structure used by `16-TreeAdd` and `18-Bisort`. Both programs make use of the same data structure. However, the double traversal performed in `18-Bisort` uses more pointers to traverse the tree and therefore the graphs are more complex: they have more nodes and more `cls`'s per node than those of `16-TreeAdd`. The maximum number of nodes is reached for `17-Power`, the benchmark with the most complex structure, but still the maximum in `cls`'s is found for `18-Bisort` due to its double traversal with two nested recursive functions.

As conclusion, the tests presented so far provide evidence that our extensions for recursive analysis over the base technique based on coexistent links set yields correct abstractions for common dynamic data structures in a variety of recursive algorithms. This still holds in the case of structures several levels deep like that of `17-Power` and nested recursive calls featuring two recursive calls per function, like in `18-Bisort`. Besides, the analysis cost is lower than that of comparable approaches [48], [46] and [47].

3.6.3 Doubly-linked structures

As the last experiments for this section, we wanted to test the ability of the *paired selectors property* to deal with doubly-linked structures. For that, we have run the *sparse matrix by sparse vector* and *sparse matrix by sparse matrix* benchmarks presented in chapter 2, in a new version based on doubly-linked lists, rather than singly-linked lists. The results for these tests are shown in Table 3.4. The singly-linked list version of these benchmarks (`Matrix x Vector(s)` and `Matrix x Matrix(s)`) are compiled here next

to the doubly-linked versions (`19-Matrix x Vector(d)` and `20-Matrix x Matrix(d)`) of the same benchmarks for easier comparison.

The base version used for these programs is the pruned version, which avoids analyzing the statements that are not involved in the structure creation. The doubly-linked versions use the *paired selectors property*, to correctly manage the doubly-linked character of the structure, and the *site property* for separating nodes from different structures. To isolate better the effect of the *paired selectors property*, the singly-linked versions of the sparse matrix benchmarks also use the *site property*, although it is not necessary for correct shape abstractions.

Note that in the previous tests in this section, the use of the *paired selectors property* was necessary for correct recursive analysis, as the data structures become "doubly linked" when we add recursive flow links. For `19-Matrix x Vector(d)` and `20-Matrix x Matrix(d)` however, it is used to correctly capture the *structure*, not the *control flow*.

Benchmark	Time	Space	Sgs	Avg. nodes / sg (max)	Avg. cls's / sg (max)
Matrix x Vector(s)	0.28 s	1.9 MB	612	5.19 (9)	16.22 (33)
19-Matrix x Vector(d)	0.66 s	2.4 MB	1,096	6.25 (10)	17.76 (35)
Matrix x Matrix(s)	3.97 s	2.7 MB	2,299	9.43 (15)	43.49 (83)
20-Matrix x Matrix(d)	8.97 s	4.4 MB	3,804	10.89 (18)	46.65 (108)

Table 3.4: The sparse matrix benchmarks compared in their singly-linked(s) and doubly-linked(d) versions.

With the use of the *paired selectors property*, doubly-linked structures are correctly captured in these tests. The results in Table 3.4 indicate an increment in every measure of the analysis cost and complexity. The doubly-linked structures present a small increase in shape graph size, with about one more node per graph in average, and a slightly bigger number of `cls`'s per graph. Likewise, more shape graphs need to be generated to achieve the fixed point for the analysis. With just a small overhead in performance, now we are able to correctly capture doubly-linked data structures by adjusting the properties in the analysis. Let us recall that doubly-linked structures are not dealt with in [48] or [53].

3.7 Summary

In this chapter we have covered the following issues:

- We have extended a working shape analysis strategy based on coexistent links set to add interprocedural support (section 3.2). This has been done by adding:
  - new analyzable statements for function calls and function return statements (section 3.2.1)
  - *recursive flow links* to simulate recursive control flow with links in the shape graphs (section 3.2.2)
  - *context change rules* to determine the shape graph transformations upon entering to and exiting from functions (section 3.2.3)
  - updated data-flow equations and new worklist algorithm (section 3.2.4)
- We have provided a tabulation mechanism that allows the analysis to reuse the computed effect for functions in the case of similar input (section 3.3).
- We have identified common cases where the analysis loses precision and have devised refining mechanisms to solve them (section 3.4). Namely we have proposed the following:

  - *previous call property* to separate in the abstraction portions of the heap that have been traversed from parts that have not yet been traversed (section 3.4.1)
  - *force pseudostatements* to filter out improper shape graphs abstractions in the presence of more than one recursive call in a recursive function (section 3.4.2)
  - *paired selectors property* to achieve precision when several connections exist between memory locations abstracted in a summary node (section 3.4.3)

- We have presented related work in the field of interprocedural heap analysis, highlighting strong and weak points (section 3.5).
- We have collected experimental evidence that our technique yields correct shape abstractions for common dynamic data structures in a variety of recursive benchmarks (section 3.6). Furthermore, the cost of the analysis compares favorably to related work.

Next, we will use the shape analysis technique described so far as a key tool for data dependence analysis in pointer-based applications.

4 Data dependence analysis

4.1 Introduction

Optimizing and parallelizing compilers rely upon accurate static disambiguation of memory references, i.e. determining at compile time if two given memory references always access disjoint memory locations. This problem, known as data dependences detection is crucial to various compiler optimizations such as instruction scheduling, data-cache optimizations, loop transformations, automatic vectorization and parallelization. Unfortunately the presence of alias in pointer-based codes makes memory disambiguation a non-trivial issue.

Knowledge about the shape of the data structure accessible from heap-directed pointers, provides critical information for disambiguating heap accesses originating from them, and hence to determine that there are no data dependences between iterations of a loop or between different function calls. In that regard, shape analysis can provide such knowledge.

We have conducted our research to develop an accurate and versatile shape analysis technique that can be used as the base tool for a dependence test. We focus on the detection of data dependences due to heap-directed pointers in dynamic data structures in two common scenarios: (i) loops, by identifying dependences that may arise between two iterations of a loop (*loop-carried* dependences), and (ii) function calls, by identifying conflicting accesses in different calls.

In our approach, we annotate information about read/write access during the abstract interpretation of possibly conflicting pointer statements. This is done with the *touch property*. This property records the history of accesses over nodes. This history is created during abstract interpretation, as the analysis progresses toward the fixed point. The heap accesses thus gathered are then used to detect data dependences.

4.1.1 Traversal patterns

We identify two different patterns for the purpose of heap-induced data dependence detection, according to the way the data structure is traversed: the *1-way* and *n-ways* traversal patterns. For that we consider how many selectors or pointer fields are traversed in a single *traversal step*. A traversal step indicates the

traversal of the data structure involved by one iteration of a loop, or by one run of the body of a recursive function.

- The *1-way* traversal pattern is found when the structure is traversed by following *only one* selector (or pointer field) for a single traversal step. The *1-way* traversal pattern is usually found in loops and recursive functions with just one recursive call in its body.
- The *n-ways* traversal pattern is found when the structure is traversed by following *more than one* (namely *n*) selectors (or pointer fields) for a single traversal step. The *n-ways* traversal pattern is typically found in recursive functions with more than one recursive call through different selectors within its body.

Fig. 4.1 shows some examples of heap-allocated data structures and traversal algorithms. In every example, the memory locations that have already been accessed in the traversal are filled with a line pattern. For instance, Fig. 4.1(a) shows a singly-linked list and two ways to traverse it, one based in a loop, the other based on a recursive function. Note that in each step of the traversal (iteration of the loop or recursive call), we follow just one selector, in particular `nxt`. These are examples of the *1-way* traversal pattern. Fig. 4.1(b) shows a *list of lists* data structure traversed by a piece of code made of two nested loops. The outer loop, `L1`, traverses the `header` elements (labeled `H`) through their `nxtH` selector, while the inner loop, `L2`, traverses the `node` elements (labeled `N`, and filled with a different pattern to indicate that they are traversed by a different loop) through their `nxtN` selector. This is another case of the *1-way* traversal pattern, as each loop traverses the structure through one selector for each traversal step.

Fig. 4.1(c) shows a binary tree data structure, traversed by a recursive function that takes the left or right child in each call, depending on some condition. This is another case of the *1-way* traversal pattern, as each function call produces the traversal to advance through only one selector (although the selector may change between `lft` and `rgh`). Note how the selectors traversed in this example are those found through the `r->rgh->lft->lft` path. Finally, Fig. 4.1(d) presents the same binary tree, traversed with a recursive function that calls itself through the left and right children for each recursive call, i.e., for each traversal step, thus establishing a *2-ways* traversal pattern. Here, many more locations have been traversed up to the point shown in the example.

Note that the traversal pattern is not determined solely by the data structure, but it depends as well on the algorithm used to traverse the structure. This way, a tree can be traversed with the *1-way* pattern or the *2-ways* pattern. Of course, if a structure has only one kind of selector to traverse it, such as a singly-linked list, then it only supports *1-way* traversal patterns.

We have elaborated two distinct approaches to detect heap-induced data dependences for the aforementioned traversal patterns. Next, we shall address them separately. It should be noted that in this work we focus on the detection of heap accesses to *data fields*, although the technique described can easily be adapted for *pointer fields* (selectors), as well.

4.2 Data dependence detection for *1-way* traversal patterns

Let us establish now a view of our heap analysis framework specifically configured for dependence analysis of *1-way* traversal patterns. Such a view is displayed at Fig. 4.2. We organize the whole process of data dependences detection for *1-way* traversal patterns in five stages. The figure maps these stages into different modules.

For *stage one* we take the IR resulting from parsing the input program with Cetus, and use it to identify heap accessing statements. These statements are then used to feed *stage two* and *stage three* of the process.

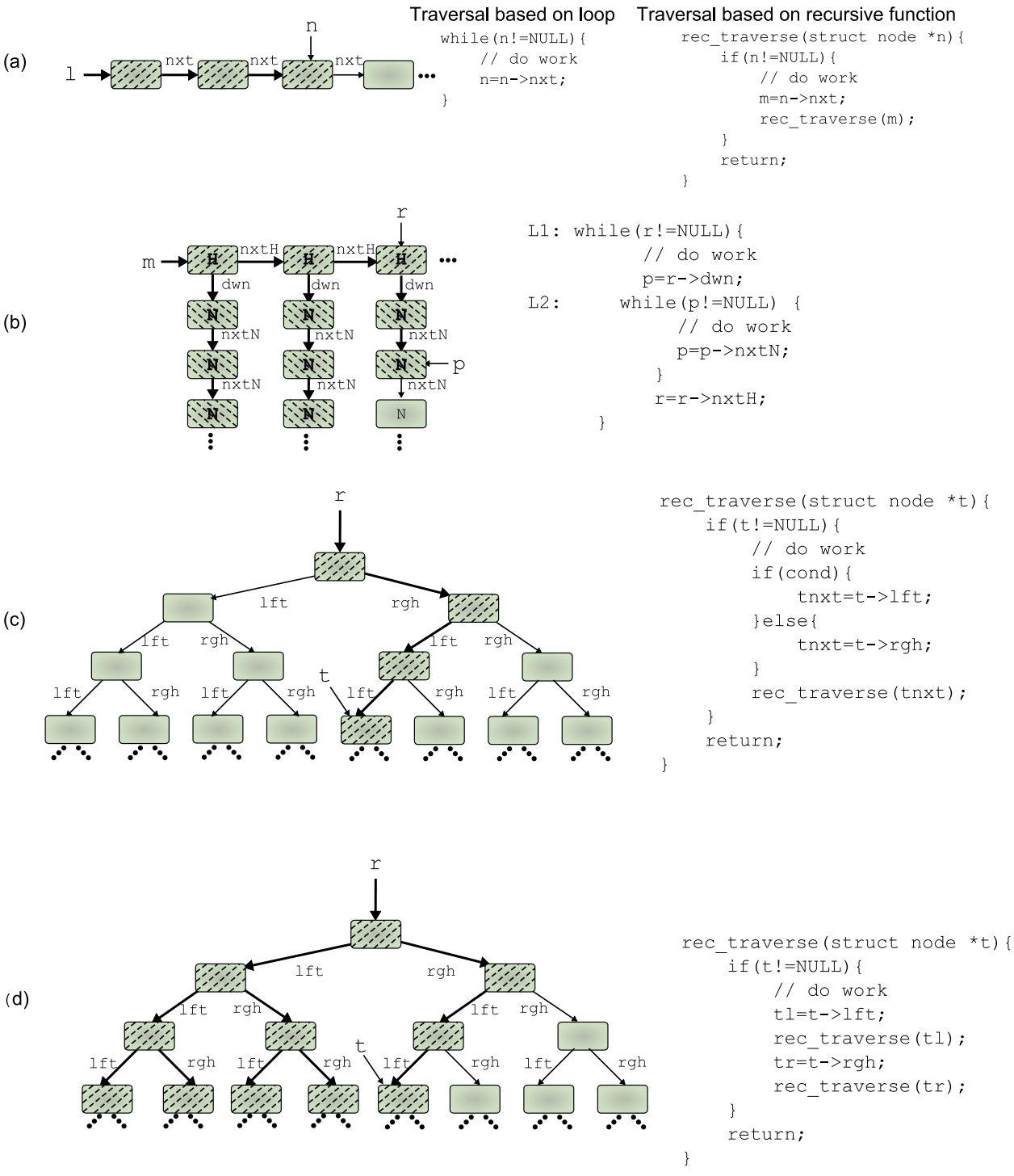

Figure 4.1: Examples of dynamic data structures and traversals. We find the `1-way` traversal pattern for (a), (b) and (c), and the `2-ways` traversal pattern for (d).

Stage two forms groups of possibly conflicting accesses, the *dependence groups*. In *stage three*, we add a touch pseudostatement for every heap accessing statement identified in *stage one*, and found in a dependence group created in *stage two*.

The result of *stage three* is a version of the original program instrumented with touch pseudostatements. This instrumented version of the program is parsed again, to obtain the pointer statements and control flow

information to conduct the shape analysis, in *stage four*. The shape analysis in *stage four* is performed with the touch property enabled so that the touch pseudostatements have effect. Upon abstract interpretation of these pseudostatements, nodes in the shape graphs are annotated with access information. The result is the set of *access pairs*, i.e., pairs of accessing statements registered over the same node.

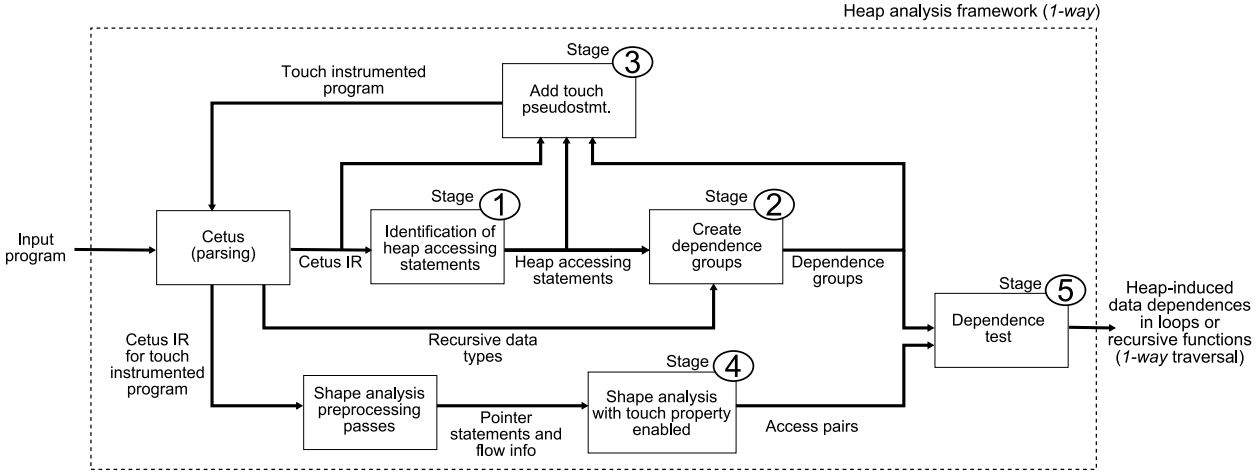

Figure 4.2: Presentation of our heap analysis framework featuring the five stages for data dependence analysis for the *1-way* traversal pattern.

Finally, *stage five* of the process performs the data dependence test, by considering the dependence groups obtained from *stage three*, and the access pairs obtained from *stage four*. As a result, we will obtain the dependences due to heap accesses found for the loops and/or recursive functions in the program.

To explain our approach to detect data dependences due to heap accesses in *1-way* traversal patterns, we will recover the motivating example that we presented in chapter 1 of this dissertation. We will use it to describe the different stages involved in the data dependences detection. Fig. 4.3 shows the code for the example in simple pointer statements.

This simple program traverses a singly-linked list copying the value of one element into the previous one, effectively shifting the values in the list toward its head. The `L1` loop presents a so-called *loop-carried dependence*, which is a dependence that exists between two different iterations in a loop. For this example, the dependence appears as the list element read through pointer `q` in iteration `i` is written through pointer `p` in iteration `i+1`. This produces a *WAR* dependence (Write After Read), also called an *anti-dependence*.

We number the statements that have some abstract semantics function associated to them. The data flow information is embedded in the data-flow equations that drive the analysis. Note that statements accessing the `data` field are not numbered. They are not recognized as one of the pointer statements supported by the technique, as explained in chapter 2. Their effect will be modeled by the touch pseudostatements inserted in *stage two*.

Next, we will describe in detail the five stages involved in the detection of data dependences for *1-way* traversal patterns.

4.2.1 *Stage one (1-way)*: identify heap accessing statements

Stage one of our process to detect data dependences due to heap accesses involves identifying heap accessing statements. These are the simple pointer statements that read or write data fields of recursive data structures, namely `x->field=data` for a write, and `data=x->field` for a read. We assume the input program has been normalized to contain only simple pointer statements as described in chapter 1.

```
   // Declare recursive type "node"
   struct node{
       int data;
       struct node *nxt;
   } *l,*p,*q;
   int main(){
       int val;
1:     l=create_list();
2:     p=l;
3:     q=p->nxt;
   L1: while(q!=NULL){
4:        #pragma SAP.force(q!=NULL)
   R1:    val=q->data;
   W1:    p->data=val;
5:        p=q;
6:        q=p->nxt;
       }
7:     #pragma SAP.force(q==NULL)
8:     return;
   }
```

Figure 4.3: Running example for data dependence detection featuring a *1-way* traversal pattern.

Note that these heap accessing statements do not have associated abstract semantics functions, as described in chapter 2. They do not modify or alter the *shape* of the structure in any way, therefore they are not relevant to create and maintain shape abstractions during the analysis. However, they perform read or write access to heap elements, so they are completely meaningful for the purpose of dependence analysis.

Fig. 4.4 shows `Identify_heap_acc()`, the function used by *stage one* to identify and label heap accessing statements. It is designed as a pass for the Cetus infrastructure. It traverses the program IR looking for pointer accessing statements through pointer fields of recursive data types, i.e., those that have pointer fields to other heap elements. Each of the heap accessing statements is marked with an *access label*, which identifies a heap accessing statement as a heap writing statement (`Wi`) or heap reading statement (`Rj`), also associating a number to it.

For our example, statement `val=q->data` is labeled `R1`, and statement `p->data=val` is labeled `W1`, which is already shown in Fig. 4.3 for convenience.

4.2.2 *Stage two (1-way)*: create dependence groups

Stage two of our process to detect data dependences for *1-way* traversal patterns involves the creation of groups of accesses that may produce a data dependence. Two heap accesses in a program will produce a data dependence if they access the same field in the same heap location, and at least one of them is a write access.

In *stage two* we create *dependence groups*, which are groups of accesses that may produce a dependence. For that they must fulfill two conditions: (i) the access field is the same, and (ii) there is at least one write access in the group. Note that the first condition also involves accessing through pointers of the same data type, as type casting is not allowed.

Fig. 4.5 presents `Create_dep_groups()`, the function used by *stage two* to create the dependence groups. As input it receives the heap accessing statements identified in *stage one* and the recursive data types in the program. First, an empty dependence group for every field in every recursive data type

```
Identify_heap_acc()
  Input: P_IR, RECTYPE          # IR for the analyzed program, and the set of recursive data types
  Output: ACCSTMT               # The set of heap accessing statements

  ACCSTMT=∅
  i=0, j=0
  repeat
      Get stmt, the next statement in P_IR
      Case (stmt)
          stmt is of the kind x->field=data,
              where x is a pointer to type t ∈ RECTYPE and field is a data field of t
                  Add access label Wi to stmt, and increment i
                  ACCSTMT=ACCSTMT ∪ stmt
                  break
          stmt is of the kind data=x->field,
              where x is a pointer to type t ∈ RECTYPE and field is a data field of t
                  Add access label Rj to stmt, and increment j
                  ACCSTMT=ACCSTMT ∪ stmt
                  break
  until (P_IR has no more statements)
  return(ACCSTMT)
end
```

Figure 4.4: The function used by *stage one (1-way)* to identify heap accessing statements.

```
Create_dep_groups()
  Input: ACCSTMT, RECTYPE       # The set of heap accessing stmts, and the set of recursive data types
  Output: DEPGROUP              # The set of all created dependence groups

  DEPGROUP=∅
  forall t ∈ RECTYPE
      forall field, data field in t
          Create DepGroup_field = ∅
          DEPGROUP=DEPGROUP ∪ DepGroup_field
      endfor
  endfor
  forall stmt ∈ ACCSTMT
      Get access field, field, and access label, Li, in stmt
      DepGroup_field=DepGroup_field ∪ Li
  endfor
  forall DepGroup_field ⊂ DEPGROUP
      If (DepGroup_field does not contain any label of kind Wi)      # A heap writing access
          DEPGROUP=DEPGROUP-DepGroup_field
  endfor
  return(DEPGROUP)
end
```

Figure 4.5: The function used by *stage two (1-way)* to create dependence groups.

(DepGroup$_{field}$) is created. Here, we assume that the program has been normalized so that the data fields of the different data types in the program are named differently. For example, if we have two similar types for lists, `list1` and `list2`, then the `nxt` field of both types must be named differently, say `nxt1` and `nxt2`. All the accessing statements gathered in *stage one* contribute with their access label (`Li`, which may be `Ri` or `Wi`) to the corresponding DepGroup$_{field}$. Finally, all the dependence groups where there are no write accessing labels, `Wi`, are removed. Finally, the superset of all dependence groups, `DEPGROUP`, is returned.

For the running example in this section, we create DepGroup$_{data}$={R1,W1}, as both heap accessing statements access data field `data`.

Note that if there are no dependence groups for a program, this means that there are no heap accesses that could potentially lead to a data dependence, so it is straightforward to conclude that it is free of data dependences due to heap accesses.

4.2.3 *Stage three (1-way)*: add touch pseudostatements

We need to reflect in the analysis the effect of heap accessing statements. For this purpose, *stage three* of our scheme adds a touch pseudostatement for every heap accessing statement belonging to a dependence group created in *stage two*. A touch pseudostatement, `touch(x,id)`, is used to annotate the node pointed to by pointer `x` with label `id`. In order to capture the effect of the heap accessing statements, the pointer used for a touch pseudostatement is the accessing pointer and the label is the access label.

```
Add_touch_pseudostmt()
  Input: P_IR, ACCSTMT, DEPGROUP
  # IR for the analyzed program, the set of heap accessing stmts, and the set of dependence groups
  Output: P'_IR
  # IR instrumented with touch pseudostatements

  Create P'_IR=P_IR
  forall stmt ∈ ACCSTMT
     Get access pointer, x, access field field, and access label, Li, in stmt
     If (Li ∈ DepGroup_field ⊂ DEPGROUP)
        Add #pragma SAP.touch(x,Li) directive right after stmt in P'_IR
  endfor
  return(P'_IR)
end
```

Figure 4.6: The function used by *stage three (1-way)* to add touch pseudostatements.

Fig. 4.6 shows the `Add_touch_pseudostmt()` function, which is used by *stage three* to add the touch pseudostatements required by our technique. The way to add a touch pseudostatement is by including it within a `pragma` directive recognizable by our *shape analysis preprocessing pass* (see Fig. 4.2). The input for the `Add_touch_pseudostmt()` algorithm is the program IR, the heap accessing statements gathered in *stage one*, and the dependence groups created in *stage two*. The result is the *touch instrumented* version of the program, i.e., the original source program including touch pseudostatements as `pragma` directives. Note that we only add touch pseudostatements for the heap accesses that belong to a dependence group, and thus may produce a dependence.

Fig. 4.7 shows our running example with the touch pseudostatements inserted by *stage three* in bold typeface. For example, `st.5:#pragma SAP.touch(q,R1)` contains the touch pseudostatement added

```
   // Declare recursive type "node"
   struct node{
       int data;
       struct node *nxt;
   } *l,*p,*q;
   int main(){
       int val;
1:     l=create_list();
2:     p=l;
3:     q=p->nxt;
   L1: while(q!=NULL){
4:        #pragma SAP.force(q!=NULL)
   R1:    val=q->data;
5:        #pragma SAP.touch(q,R1)
   W1:    p->data=val;
6:        #pragma SAP.touch(p,W1)
7:        p=q;
8:        q=p->nxt;
       }
9:     #pragma SAP.force(q==NULL)
10:    return;
   }
```

Figure 4.7: Running example instrumented with touch pseudostatements in bold typeface.

for heap accessing statement `R1:val=q->data`.

4.2.4 *Stage four (1-way)*: shape analysis with touch property

The program instrumented with touch pseudostatements resulting from *stage three*, is emitted as a new source program, then parsed again and processed by our shape analysis preprocessing pass. The result is the pointer statements and flow information that are used for the shape analysis module. This shape analysis must be conducted with the touch property enabled, so that the effect of the touch pseudostatements inserted in *stage three* can be registered. This is the process involved in *stage four* of our data dependence test strategy.

The result that we obtain from *stage four* is the set of *access pairs*, i.e., pairs of accessing statements registered over the same node. Whenever a new access label is annotated in a node, we check for the other accesses previously annotated in the same node. All the possible pairs constructed between previously annotated access labels *and* the newly annotated access label, that contain at least a write access and therefore may induce a dependence, are recorded as access pairs. They represent pairs of accesses that may occur over the same memory location. The order of the accesses is preserved in the access pairs. This information is meaningful for the purpose of discriminating between different types of dependences, as we shall see in *stage five*.

Fig. 4.8 shows the `Touch()` function that expresses the abstract semantics operation for touch pseudostatement, $\texttt{touch(x,touch}_{\texttt{id}}\texttt{)}$. It adds the $\texttt{touch}_{\texttt{id}}$ label to the value of the touch property for the node `ni`, pointed to by `x`. For data dependence detection, $\texttt{touch}_{\texttt{id}}$ is the access label. It also adds ordered access pairs made from the previous labels in the node and the current label. In particular, only the pairs where one of the accesses is a write, and therefore may produce a dependence, are stored. For this purpose, we use the `AccessPairs` set, which was initialized to empty at the beginning of the analysis.

```
Touch()
 Input: sg^1=<N^1,CLS^1>, x ∈ PTR, touch_id        # A shape graph, a pointer and an identifier
 Output: sg^k=<N^k,CLS^k>                          # A shape graph

 Create N^k=N^1
 Create CLS^k=CLS^1
 Find ni ∈ N^k s.t. ∃ pl=<x,ni> ⊂ CLS_ni
 forall touch_id2 ∈ PPM_touch(ni)
    If ( touch_id=Wj ∨ touch_id2=Wj )          # If any of the labels stands for a write access
       AccessPairs=AccessPairs ∪ <touch_id2,touch_id>
 endfor
 PPM_touch(ni)=PPM_touch(ni) ∪ touch_id
 Create sg^k=<N^k,CLS^k>
 return(sg^k)
end
```

Figure 4.8: The `Touch()` function for annotating access labels in nodes. Access pairs are created too.

Fig. 4.9 shows how the abstract interpretation of the touch pseudostatements proceed for the running example. Access labels `R1` and `W1` are annotated into the nodes, and the access pair `<R1,W1>` is created. Coexistent links sets are omitted for simplicity. Next to each graph we find the current value of the `AccessPairs` set.

The shape graph in Fig. 4.9(a) shows a possible abstraction of the singly-linked list when entering the analysis of the `L1` loop. The `AccessPairs` set is empty at this moment. In (b), we show the same shape graph but the nodes pointed to by `p` and `q` has been *touched* by the `W1` and `R1` access labels, respectively. Note that since this is the first $\texttt{touch}_{\texttt{id}}$ for both nodes, no access pairs can be generated. In the shape graph for (c), in the same figure, we display another possible abstraction at the beginning of the second symbolic iteration of `L1`. Pointers `p` and `q` have moved forward in the list. In (d), we have the graph after a new abstract interpretation of the touch pseudostatements. Now, node `n2` has been touched again, this time with $\texttt{touch}_{\texttt{id}}$ `W1`. Since there was another annotation within `n2`, and the new annotation is a write, the access pair `<R1,W1>` is created, and added to the `AccessPairs` set. Note that the access pair preserves the order of the accesses, i.e., the memory location abstracted by `n2` was first *read* with access label `R1`, and then *written* with access label `W1`. Finally, (e) presents the fixed point for the loop, and no more access pairs have been created.

4.2.5 *Stage five (1-way)*: dependence test

The last stage in the process is the actual dependence test, which takes as input the dependence groups created in *stage three* and the access pairs collected in *stage four*. Fig. 4.10 shows the basic algorithm for identifying data dependences in *1-way* traversal patterns.

The `Dep_test_1way()` function traverses the dependence groups available and checks whether any of the collected access pairs, $\texttt{<AccLb}_1\texttt{,AccLb}_2\texttt{>}$, contains both accesses within the same $\texttt{DepGroup}_{\texttt{field}}$. If that is the case, then a data dependence is due. The types of access labels in the pair is then checked to discriminate the kind of dependence: flow dependence, anti dependence or output dependence. An output dependence is registered when two write accesses are found for the same memory location, whether for the same writing statement or for different ones.

The information of the kind of dependence detected can be useful for a subsequent parallelization mod-

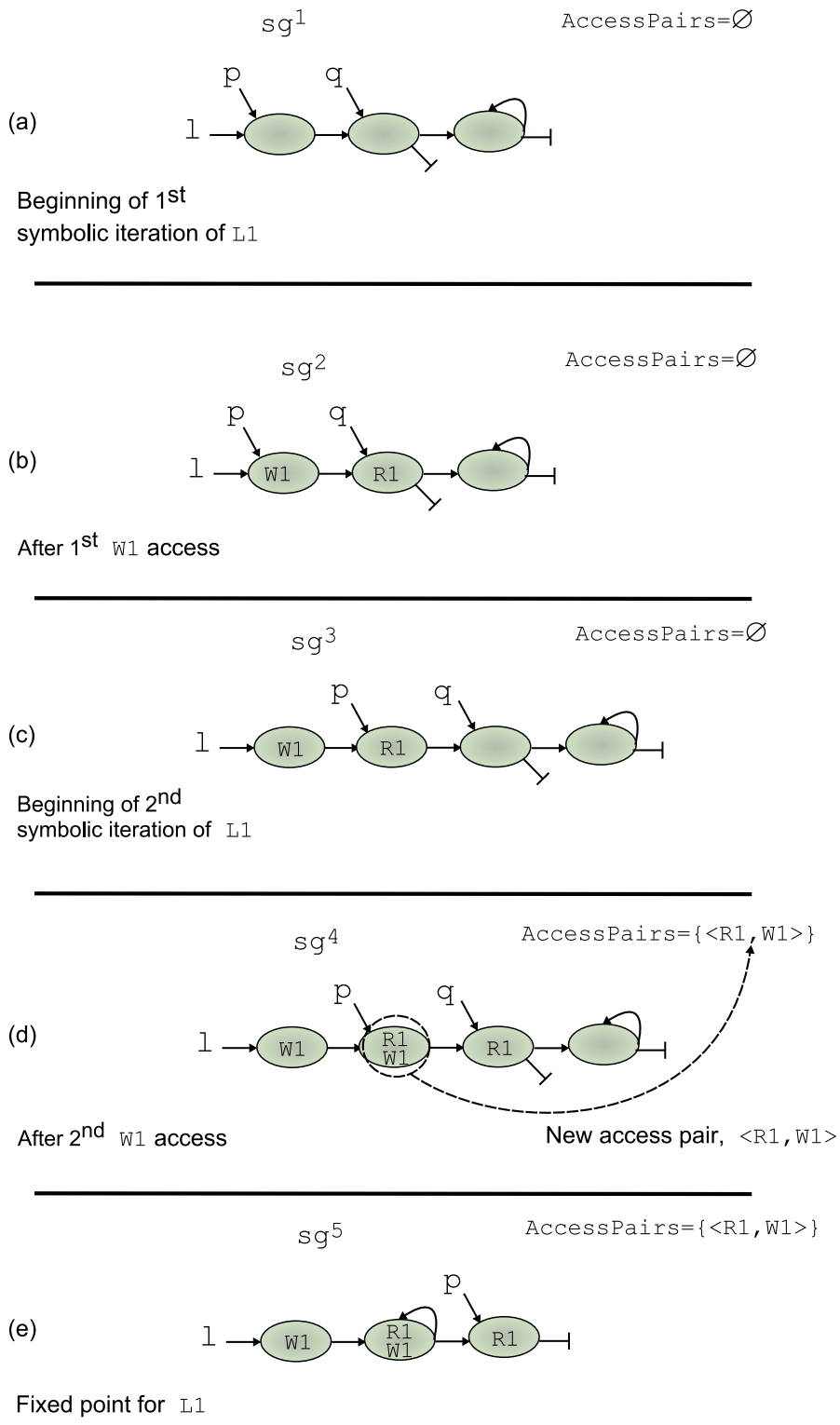

Figure 4.9: The process of access labels annotation in *stage four (1-way)*.

ule, that could transform the program accordingly. For example, in the case of an anti dependence new storage could be added in order to solve the dependence.

Note that all access pairs where the access labels do not belong to the same dependence group, cannot produce a conflict since they do not access the same field, and therefore no conflict due to heap accesses

```
Dep_test_1way()
 Input: DEPGROUP, AccessPairs        # The dependence groups and access pairs
 Output: DEP                         # The set of dependences found for the program

 DEP=∅
 forall DepGroup_field ⊂ DEPGROUP
    Create Dep_field=∅
    forall <AccLb_1,AccLb_2> ∈ AccessPairs
       If (AccLb_1 ∈ DepGroup_field ∧ AccLb_2 ∈ DepGroup_field)   # Acc. in same DepGroup_field
          Case (<AccLb_1,AccLb_2>)
             <AccLb_1=Wi ∧ AccLb_2=Rj>
                Dep_field=Dep_field ∪ FlowDep      # Flow dependence
                break
             <AccLb_1=Rj ∧ AccLb_2=Wi>
                Dep_field=Dep_field ∪ AntiDep      # Anti dependence
                break
             <AccLb_1=Wi ∧ AccLb_2=Wi>
                Dep_field=Dep_field ∪ OutDep       # Output dependence, same writing stmt
                break
             <AccLb_1=Wj ∧ AccLb_2=Wi>
                Dep_field=Dep_field ∪ OutDep       # Output dependence, different writing stmt
                break
    endfor
    DEP=DEP ∪ Dep_field
 endfor
 return(DEP)
end
```

Figure 4.10: The function used by *stage five (1-way)* to identify data dependences due to heap accesses.

may arise.

For our running example, the dependence group $\texttt{DepGroup}_{\texttt{data}}=\{\texttt{R1},\texttt{W1}\}$ was created in *stage three* and access pair <R1,W1> was gathered in *stage four*. By applying the algorithm Dep_test_1way() we can determine that there is an anti dependence due to heap accesses R1 and W1 in loop L1.

4.2.6 Zero distance data dependences

Our technique is able to find, conservatively, every data dependence due to heap accesses that may arise in loops or recursive functions that conform to the *1-way* traversal pattern. On top of that, we can distinguish between flow, anti and output dependences. However, it is often the case that some dependences do not inhibit parallelism of loop iterations or function calls. It is the case of *zero distance* dependences [60]. These are the dependences that arise in the same iteration of a loop or the same function call. These dependences are relevant for compiler transformations that involve statement reordering within a basic block, but they can be ignored for the purpose of coarse-grain parallelization. Since our focus is toward dependence analysis to expose thread-level parallelism in pointer-based programs, then it is important to be able to determine if a dependence found by our technique is a *zero distance* dependence or a *greater-than-zero distance* dependence.

We offer a different approach to detect zero distance dependences for the two possible scenarios in the

1-way traversal pattern: loops and recursive functions. For dependences between iterations of a loop, we add information about the symbolic iteration of the loop to the touch annotation. For dependences between recursive calls, sometimes it is enough with a simple inspection of the function and its heap accesses. If that is not possible or insufficient, we can instrumentate the program further and rerun the analysis for additional information.

4.2.6.1 Detecting zero distance data dependences in loops

We use the concept of *iteration vector* [60] to provide the necessary information to distinguish zero distance dependences due to heap accesses in the context of loops. In the presence of nested loops, an iteration vector describes the current iteration for each loop. In our case, the iteration vector records the symbolic iterations of the loops in the process of abstract interpretation toward a fixed point.

Let us consider now an example to explain how the iteration vector information is used within our approach. Fig. 4.11 shows a variation of the running example in Fig. 4.3. We assume that a *list of lists* kind of data structure has been created and it is pointed to by pointer `m`. Such a structure is represented in the concrete domain in $\texttt{mc}^1$ in Fig. 4.12. The structure is formed by a singly-linked list of `header` elements, each of which points to a singly-linked list of `node` elements. The loop `L1` traverses the list of `header` elements, while the `L2` loop traverses the list of `node` elements, performing the same shifting of values toward the head of the example in Fig. 4.3. Here, the access labels and touch pseudostatements have already been added to the program, i.e., the code shown in Fig. 4.11 is the instrumented program resulting from *stage three*.

In each iteration of the loop `L1`, a different `header` element is first read through access `R1` and then written through access `W2`, thus updating its `value` field. Therefore, there is no loop-carried dependence for the `header` elements in the structure, as each element is only accessed one iteration along the traversal in `L1`. On the other hand, `L2` presents the same loop-carried dependence found for the running example at the beginning of this chapter, i.e., an anti dependence due to the fact that each node element is read in iteration `i` and written in iteration `i+1`.

The following dependence groups are created by *stage two* for this new example: $\texttt{DepGroup}_{\texttt{value}}=\{\texttt{R1,W2}\}$ and $\texttt{DepGroup}_{\texttt{data}}=\{\texttt{R2,W1}\}$. The technique presented so far would report anti dependences due to access pairs <R1,W2> and <R2,W1>. We acknowledge that is important for the purpose of loop parallelization to determine that <R1,W2> is a zero distance dependence. For that, we introduce the information of the iteration vector as part of the touch property information. We just need to maintain a symbolic iteration counter for each loop in the program, and attach the current value of the iteration vector when performing a touch on a node.

Consider $\texttt{sg}^1$ in Fig. 4.12. It abstracts the data structure for this example ($\texttt{mc}^1$), where we maintain *type property* information to separate the nodes for both types of elements in the structure: `node` (`N`) and `header` (`H`). Shape graph $\texttt{sg}^2$ shows the result after executing the first `W1` access. Note that next to the access label, now the touch information also records the value of the iteration vector at the moment of performing the touch (shown below the affected node in bold for readability). Here, we consider an iteration vector of two coordinates, where the first one stands for the symbolic iteration of loop `L1` and the second coordinate stands for `L2`'s symbolic iterations. For instance, node `n1` is touched by `R1<1,0>` which stands for "`R1` access at first iteration of `L1`, outside of body of `L2`", while `n2` is touched by `W1<1,1>` which stands for "`W1` access at the first iteration of `L1`, first iteration of `L2`". The analysis continues and we find $\texttt{sg}^3$, also in Fig. 4.12. This is the resulting graph after the second `W1` access. The `[R2<1,1>,W1<1,2>]` access pair is found over node `n3` and it is stored in the `AcessPairs` set.

Regarding the `header` elements, shape graph $\texttt{sg}^4$ shows how node `n1` features the access pair

```
    // Declare recursive type "node"
    struct node{
        int data;
        struct node *nxtN;
    } *p,*q;
    // Declare recursive type "header"
    struct header{
        int value;
        struct header *nxtH;
        struct node *dwn;
    } *m,*r;
    int main(){
        int total, val;
        // Create structure pointed to by "m"
        [...]
1:      r=m;
   L1:  while(r!=NULL)
2:          #pragma SAP.force(r!=NULL)
   R1:      total=r->value;
3:          #pragma SAP.touch(r,R1)
4:          p=r->dwn;
5:          q=p->nxtN;
   L2:  while(q!=NULL){
6:          #pragma SAP.force(q!=NULL)
   R2:      val=q->data;
7:          #pragma SAP.touch(q,R2)
            total+=val;
   W1:      p->data=val;
8:          #pragma SAP.touch(p,W1)
9:          p=q;
10:         q=p->nxtN;
        }
11:     #pragma SAP.force(q==NULL)
12:     p=NULL;
13:     q=NULL;
   W2:  r->value=total;
14:     #pragma SAP.touch(r,W2)
15:     s=r->nxtH;
16:     r=s;
17:     s=NULL;
    }
18:  #pragma SAP.force(r==NULL)
19:  r=NULL;
20:  return;
   }
```

Figure 4.11: Variation of the running example that presents a zero distance data dependence in loop L1.

[R1<1,0>,W2<1,0>] after the first W2 access. This new access pair is recorded as well in the AcessPairs set. Finally, sg^5 shows the graph at the end of the analysis, at the return statement. It has accumulated the values of the iteration vector over the summary nodes n2, n3 and n4. Note that different values of the iteration vector do not affect node compatibility (i.e., they do not prevent node summarization), but different values of the touch property (i.e., different labels) do affect.

Every time a touch is performed over a node which already had some access label, the resulting access pairs are stored in the AccessPairs set, now including the information of the iteration vectors. At the end of the analysis we have collected the following access pairs:

```
[R2<1,1>,W1<1,2>],[R2<1,2>,W1<1,3>],...,[R2<1,n-1>,W1<1,n>],
[R2<2,1>,W1<2,2>],[R2<2,2>,W1<2,3>],...,[R2<2,n-1>,W1<2,n>],...,
[R2<m,1>,W1<m,2>],[R2<m,2>,W1<m,3>],...,[R2<m,n-1>,W1<m,n>],
[R1<1,0>,W2<1,0>],[R1<2,0>,W2<2,0>],...,[R1<m,0>,W2<m,0>]
```

where m and n are the number of symbolic iterations to reach the fixed point in loops L1 and L2 respectively.

Once the access pairs with iteration vector information are collected, we may perform the dependence test of *stage five*. However, this time we shall consider the information from the iteration vector to discriminate the dependences according to the Dep_test_lcd0() function in Fig. 4.13. The iteration vectors (IV) annotated with each access label (AccLb) are subtracted if the access labels belong to the same dependence group, $\texttt{DepGroup}_{\texttt{field}}$. The leftmost coordinate of the vector that is greater than zero marks the loop for which that access pair provokes a loop-carried dependence. If the vector resulting from the subtraction is zero <0,0,...,0>, then the access pair forms a zero distance loop-carried dependence, lcd0, and it is added to the LCD0 set. The DEP set now contains all the loop-carried dependences found, categorized by

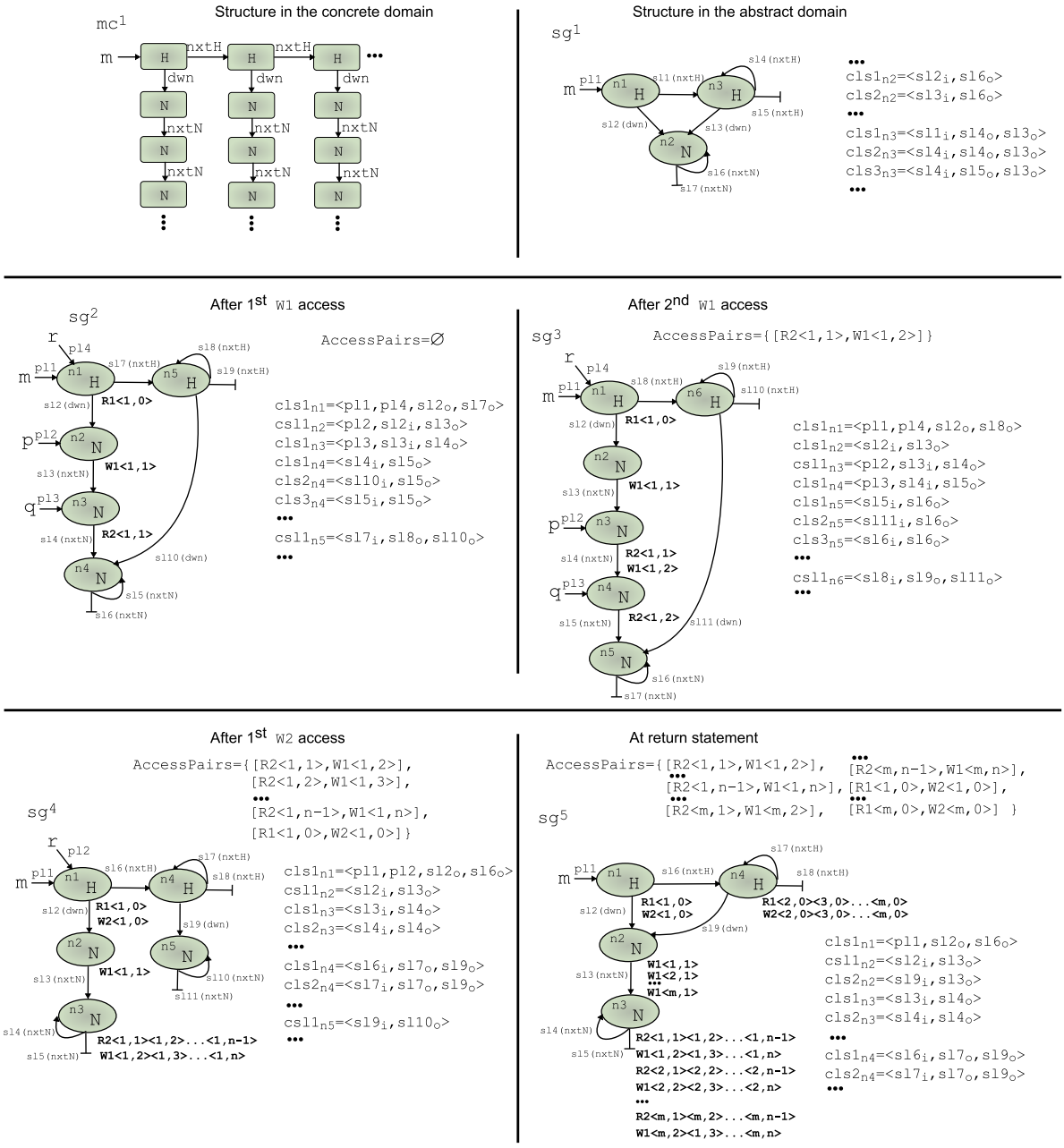

Figure 4.12: A *list of lists* data structure, its abstraction and several shape graphs achieved during the analysis in *stage four (1-way)*. The access labels include the iteration vector information for discriminating zero distance dependences in loops.

loop.

For example, let us consider the effect of the `Dep_test_lcd0()` function over the previously gathered access pair `[R2<1,1>,W1<1,2>]`. The iteration vectors are subtracted yielding `|<1,1>-<1,2>|=<0,1>`. In this way, we detect that the `<R2,W1>` access pair is a zero distance data dependence for loop `L1`, but may produce a loop-carried anti dependence for loop `L2`, which is stored in $\texttt{Dep}_{\texttt{L2}}$. On the other hand, subtracting the iteration vectors in the access pair `[R1<1,0>,W2<1,0>]` results in `<0,0>` meaning that both accesses are performed in the same iteration of `L1`, thus the access pair `<R1,W2>` is not loop-carried, and it is stored in `LCD0`.

```
Dep_test_lcd0()
 Input: DEPGROUP, AccessPairs      # The dependence groups and access pairs
 Output: DEP, LCD0                 # Loop-carried dep. and zero distance dep. in loops

 LCD0=∅
 forall l loop in the program
    Create Dep_l=∅
 endfor
 forall DepGroup_field ⊂ DEPGROUP
    forall [AccLb_1<IV_1>,AccLb_2<IV_2>] ∈ AccessPairs
       If (AccLb_1 ∈ DepGroup_field ∧ AccLb_2 ∈ DepGroup_field)   # Acc. in same DepGroup_field
          IV_3=|IV_1-IV_2|=|<i_1,i_2,...,i_z>-<j_1,j_2,...,j_z>|=<k_1,k_2,...,k_z>
          Get coordinate l, 1≤l≤z, the left-most coordinate in IV_3 greater than 0
          If (l<z)
             Case (<AccLb_1,AccLb_2>)
                <AccLb_1=Wi ∧ AccLb_2=Rj>
                   Dep_l=Dep_l ∪ FlowDep      # Flow dependence
                   break
                <AccLb_1=Rj ∧ AccLb_2=Wi>
                   Dep_l=Dep_l ∪ AntiDep      # Anti dependence
                   break
                <AccLb_1=Wi ∧ AccLb_2=Wi>
                   Dep_l=Dep_l ∪ OutDep       # Output dependence, same writing stmt
                   break
                <AccLb_1=Wj ∧ AccLb_2=Wi>
                   Dep_l=Dep_l ∪ OutDep       # Output dependence, different writing stmt
                   break
          else
             LCD0=LCD0 ∪ <AccLb_1,AccLb_2>
    endfor
 endfor
 DEP=∅
 forall l loop in the program
    If Dep_l ≠ ∅
       DEP=DEP ∪ Dep_l
 endfor
 return(DEP, LCD0)
end
```

Figure 4.13: The `Dep_test_lcd0()` function with further elaboration to detect zero distance loop-carried dependences.

With this technique, not only we can determine what dependences are loop-carried, but we can also distinguish the loop where they hold. The results of the analysis of the program in Fig. 4.11 tell us that loop `L2` has a loop-carried dependence but not `L1`. As a consequence, a subsequent parallelization client could safely decide to exploit coarse-grain parallelism by executing the iterations in `L1` in parallel. Keep in mind though that a valid parallelization scheme, like the one described in [61], would need to solve existing *control dependences*, like the one induced by pointer `r`, which acts as *navigator* or *induction pointer* for loop `L1`.

The approach described here is suitable for perfectly nested loops, but it is easily extensible for imper-

fectly nested loops just by masking the appropriate coordinates when subtracting the iteration vectors. For that, we assign a coordinate for every loop, in program lexicographic order. When subtracting the iteration vector, IV=<...,i,...,j,...,k,...>, it might happen that coordinate k belongs to a loop Lk that is not nested within a loop Lj of coordinate j but both loops are nested inside a loop Li of coordinate i. In that case coordinate j should be masked from the subtraction so that it does not interfere to recognize the greater-than-zero leftmost coordinate that belongs to a loop that contains Lk.

4.2.6.2 Detecting zero distance data dependences in recursive functions

When detecting data dependences due to heap accesses in recursive functions, it is also possible to have zero distance dependences, i.e., data dependences within the same function call. Such a case does not prevent parallelism at function call level, therefore it is important to discriminate these cases from dependences carried across different calls for the purpose of parallelization.

Let us present now an example to explain our approach to detect zero distance dependences in recursive functions. It is displayed in Fig. 4.14. It is the recursive version of the program in Fig. 4.11. Again, the *list of lists* data structure, pointed to by m, is traversed in two axes: the header list through the nxtH selector and the node lists through the nextN selector. This time, instead of using loops for that traversal, we use recursive functions traverse_header() and traverse_node(). Note that this is a case of *1-way* traversal pattern as each recursive function has only one recursive call site traversing the structure through only one selector.

The analysis of this program, by following the described process in five stages, yields the access pair <R1,W1>, for accesses in the header elements, and <R2,W2>, for accesses in the node elements. Note that the dependences found are the same as the iterative version, but the labels for the accesses are now different. The way the code has been rewritten to use recursive functions has changed the order of the accesses in the program code. Remember that the access labels are numbered as they are found by *stage one* (algorithm in Fig. 4.4).

As we mention, whatever the actual names of the access labels, the dependences found are the same as in the iterative version. These results hide the fact that the dependence found for the header elements (indicated by access pair <R1,W1>) is not carried across recursive calls, and thus does not inhibit parallelism at the function call level.

We require additional information to determine whether an access pair stands for a zero distance or greater-than-zero distance data dependence for recursive functions that conform to the *1-way* traversal pattern. We can obtain that information (i) directly from the program source code, or (ii) from the analysis of the program with further instrumentation.

Information obtained directly from the program source code

A quick inspection of the code causing dependences can sometimes provide information about zero distance dependences. All that is required for this on the analysis side is the ability to count the number of times an access pair is found within a node. Remember that in *stage four*, the access pairs found for a node are stored in the AccessPairs set. It is straightforward to add information about the number of times a given access pair is found for *one* node, and we have added that information in our implementation. Note that it is different to find the same access pair in *different* nodes than finding the same access pair repeatedly over the *same* node.

The information about the number of times that an access pair is registered for the same node is considered by the dependence test module (*stage five*), along with some information derived from the source code

```
    // Declare recursive type "node"
    struct node{
        int data;
        struct node *nxtN;
    }*p,*q;
    // Declare recursive type "header"
    struct header{
        int value;
        struct header *nxtH;
        struct node *dwn;
    } *m,*r;
    void traverse_header(struct header *r){
        int total, val;
        struct node *n;
        struct header *s;
 1:     #pragma SAP.excludeRFPTR(n,s)
   R1: total=r->value;
 2:     #pragma SAP.touch(r,R1)
 3:     n=r->dwn;
 4:     val=traverse_node(n);
 5:     n = NULL;
   W1: r->value=total+val;
 6:     #pragma SAP.touch(r,W1)
 7:     s=r->nxtH;
        if(s!=NULL){
 8:        #pragma SAP.force(s!=NULL)
 9:        traverse_header(s);
10:        #pragma SAP.force(r!=NULL)
        }else{
11:        #pragma SAP.force(s==NULL)
        }
12:     return;
    }
```

```
    int traverse_node(struct node *p){
       int val1=0,val2=0,val3=0;
       struct node *q;
       #pragma SAP.excludeRFPTR(q)
13:    q=p->nxtN;
       if(q!=NULL){
14:      #pragma SAP.force(q!=NULL)
    R2:  val1=q->data;
15:      #pragma SAP.touch(q,R2)
    W2:  p->data=val1;
16:      #pragma SAP.touch(p,W2)
17:      val2=traverse_node(q);
         val3=val1+val2;
       }else{
18:      #pragma SAP.force(q==NULL)
       }
19:    return val3;
    }
    int main(){
       // Create structure pointed to by "m"
       [...]
20:    traverse_header(m);
21:    return 1;
    }
```

Figure 4.14: Variation of the running example that presents a zero distance data dependence in recursive function `traverse_header()`.

by the preprocessing stage. In particular, we are able to identify zero distance heap-induced data dependence in recursive functions with *1-way* traversal pattern when:

- The access pair is found only once, at most, for any node.
- Both accesses in the access pair are done with the same pointer, which is local to the function, and it suffers no modification between both accesses.

The first condition ensures that the access pair occurs only once, at most, for any memory location in the program. The second condition ensures that the access pair is registered in the same recursive call. Although this second condition is very restrictive and could be improved with further analysis at the preprocessing stage, it works for simple recursive functions.

Consider the code for function `traverse_header()` in Fig. 4.14. Both accesses in this function, `R1` and `W1`, are done with pointer `r`, which is not modified between both accesses and is a local pointer to `traverse_header()`. Therefore, it is evident that every `header` element is going to be touched first by the `R1` access label, and then by the `W1` access label, forming access pair `<R1,W1>`. The key issue here

is whether a `header` element can be accessed *more than once* during the traversal, i.e., whether it can be accessed by *different instances* of recursive function `traverse_header()`. That would be the case if there was a cycle in the `header` list, for example. However, the access pair `<R1,W1>` is only reported to be found, at most once, for any given node. In such case, we can safely determine that it is a zero distance dependence. Note that the same cannot be said for the `<R2,W2>` access pair, as the `R2` and `W2` accesses are performed with different pointers in the `traverse_node()` function.

Information obtained from the analysis of the program with further instrumentation

Clearly, the information from the source code is sufficient only for recursive functions that perform both accesses in the same access pair with the same pointer. In the case that no useful information can be obtained from the source code to identify zero distance data dependences in recursive functions with the *1-way* traversal pattern, we can instrumentate the program further, to obtain more information about the accesses.

We outline the process involved. First, we perform the analysis and run the dependence test as described. If dependences result, we instrumentate the program with *untouch pseudostatements* and run the analysis a second time. The new access pairs obtained will help us conclude whether a dependence is of zero or greater-than-zero distance.

An untouch pseudostatement performs the opposite operation of a touch pseudostatement, i.e., it removes a label from a node. Fig. 4.15 displays the abstract interpretation function of an untouch pseudostatement.

```
Untouch()
 Input: sg^1=<N^1,CLS^1>, x ∈ PTR, touch_id      # A shape graph, a pointer and an identifier
 Output: sg^k=<N^k,CLS^k>                        # A shape graph

 Find ni ∈ N^1 s.t. ∃ pl=<x,ni> ⊂ CLS_ni
 PPM_touch(ni)=PPM_touch(ni)-touch_id
 Create N^k=N^1
 Create CLS^k=CLS^1
 Create sg^k=<N^k,CLS^k>
 return(sg^k)
end
```

Figure 4.15: The `Untouch()` function for clearing annotations in nodes.

The key idea is to prevent touch information from passing between recursive calls. This is prevented by *untouching* each touch annotation before a recursive call, and reestablishing it upon return. Likewise, before leaving the body of the function, touch annotations are again cleared with untouch pseudostatements before returning to the caller. Access pairs occurring within the same function call will still be recorded in the `AccessPairs` set.

Even though the example presented for the detection of zero distance data dependences in recursive functions (Fig. 4.14) can be successfully analyzed by drawing information from the source code, we will consider it again as example for instrumentation with the untouch pseudostatements. Fig. 4.16 shows the resulting code. The first run is performed without the untouch pseudostatements (code in Fig. 4.14), while the second run uses them (code in Fig. 4.16).

This instrumentation with untouch pseudostatements imposes a check on the program: we must make sure that we *untouch* on the same location that we *touched*. For that we ensure that (i) the untouch is

```
    // Declare recursive type "node"
    struct node{
        int data;
        struct node *nxtN;
    }*p,*q;
    // Declare recursive type "header"
    struct header{
        int value;
        struct header *nxtH;
        struct node *dwn;
    } *m,*r;
    void traverse_header(struct header *r){
        int total, val;
        struct node *n;
        struct header *s;
 1:     #pragma SAP.excludeRFPTR(n,s)
        total=r->value;
 2:     #pragma SAP.touch(r,R1)
 3:     n=r->dwn;
 4:     val=traverse_node(n);
 5:     n = NULL;
        r->value=total+val;
 6:     #pragma SAP.touch(r,W1)
 7:     s=r->nxtH;
        if(s!=NULL){
 8:        #pragma SAP.force(s!=NULL)
 9:        #pragma SAP.untouch(r,R1)
10:        #pragma SAP.untouch(r,W1)
11:        traverse_header(s);
12:        #pragma SAP.touch(r,R1)
13:        #pragma SAP.touch(r,W1)
14:        #pragma SAP.force(r!=NULL)
        }else{
15:        #pragma SAP.force(s==NULL)
        }
16:     #pragma SAP.untouch(r,R1)
17:     #pragma SAP.untouch(r,W1)
18:     return;
    }
```

```
    int traverse_node(struct node *p){
        int val1=0,val2=0,val3=0;
        struct node *q;
        #pragma SAP.excludeRFPTR(q)
19:     q=p->nxtN;
        if(q!=NULL){
20:       #pragma SAP.force(q!=NULL)
          val1=q->data;
21:       #pragma SAP.touch(q,R2)
          p->data=val1;
22:       #pragma SAP.touch(p,W2)
23:       #pragma SAP.untouch(q,R2)
24:       #pragma SAP.untouch(p,W2)
25:       val2=traverse_node(q);
26:       #pragma SAP.touch(q,R2)
27:       #pragma SAP.touch(p,W2)
          val3=val1+val2;
        }else{
28:       #pragma SAP.force(q==NULL)
        }
29:     #pragma SAP.untouch(q,R2)
30:     #pragma SAP.untouch(p,W2)
31:     return val3;
    }
    int main(){
        // Create structure pointed to by "m"
        [...]
32:     traverse_header(m);
33:     return 1;
    }
```

Figure 4.16: Variation of the running example using recursive functions, instrumented with *touch* and *untouch pseudostatements*, displayed in bold typeface.

performed with the same accessing pointer that the touch was performed and (ii) that pointer cannot be modified between both pseudostatements. This aspect can be enforced with additional pointer variables if needed, so it involves no loss of generality.

The reasoning for this approach is simple: in the touch-untouch instrumented version only the access pairs resulting from accesses in the same function call are recorded, while in the touch instrumented version all possible access pairs are registered. Every access pair that is found for the touch instrumented version that is not found for the touch-untouch instrumented version is clearly a greater-than-zero distance data dependence. Likewise, if running with touch pseudostatements does not produce new access pairs with regards to the touch-untouch instrumented version, then it is clear that all access pairs are necessarily registered within the same function call, and therefore they are zero distance data dependences.

We need to define some new sets now. Let $\texttt{AccessPairs}_\texttt{t}$ be the set of access pairs detected as dependences for the first run of the analysis, with the touch instrumented version of the program, and let $\texttt{AccessPairs}_\texttt{tu}$ be the set of access pairs detected as dependences for the second run of the analysis, with the touch-untouch instrumented version of the program. Let $\texttt{CombAccP(AccLb}_1\texttt{,AccLb}_2\texttt{)}$ be the set of possible combinations of access labels $\texttt{AccLb}_1$ and $\texttt{AccLb}_2$ that form a valid access pair, i.e., at least of the accesses must be a write access.

Let us see now how the information gathered from both runs is used. The following assertions hold:

- Every access pair $\texttt{AccP}_\texttt{i} \in \texttt{AccessPairs}_\texttt{t}$ s.t. $\texttt{AccP}_\texttt{i} \notin \texttt{AccessPairs}_\texttt{tu}$, is a greater-than-zero distance data dependence.

- If $\texttt{AccessPairs}_\texttt{t}$=$\texttt{AccessPairs}_\texttt{tu}$, then every access pair $\texttt{AccP}_\texttt{i} \in \texttt{AccessPairs}_\texttt{t}$ is a zero distance data dependence.

- If $|\texttt{AccessPairs}_\texttt{t}| > |\texttt{AccessPairs}_\texttt{tu}|$, then every access pair $\texttt{AccP}_\texttt{i}$=<$\texttt{AccLb}_1$,$\texttt{AccLb}_2$> s.t. $\texttt{AccP}_\texttt{i} \in \texttt{AccessPairs}_\texttt{t} \wedge \texttt{AccP}_\texttt{i} \in \texttt{AccessPairs}_\texttt{tu}$ is a zero distance data dependence if and only if $\nexists\, \texttt{AccP}_\texttt{j} \neq \texttt{AccP}_\texttt{i}$ s.t. $\texttt{AccP}_\texttt{j} \in \texttt{CombAccP(AccLb}_1\texttt{,AccLb}_2\texttt{)} \wedge \texttt{AccP}_\texttt{j} \in \texttt{AccessPairs}_\texttt{t}$.

The two first assertions are intuitive, as introduced earlier. There is a trickier case though, and it affects the access pairs that are found in both versions when the number of access pairs detected is bigger for the touch instrumented version. In that case, the access pairs that are common (belong both to $\texttt{AccessPairs}_\texttt{t}$ and $\texttt{AccessPairs}_\texttt{tu}$) are zero distance data dependences. However, they might be greater-than-zero distance data dependences *as well*. To be certain that they are not greater-than-zero distance data dependences, we must assure that there are no other access pairs in the touch instrumented version that are a combination of the access labels within the access pair considered.

For the example in Fig. 4.16, $\texttt{AccessPairs}_\texttt{t}$={<R1,W1>,<R2,W2>}, and $\texttt{AccessPairs}_\texttt{tu}$= {<R1,W1>}, so applying the rules exposed, we can conclude that: (i) the access pair <R2,W2> is a greater-than-zero data dependence because it is not found for the touch-untouch instrumented version but it exists in the touch instrumented version, and (ii) the access pair <R1,W1> is a zero distance data dependence because it is found in both versions and there is no combination of its access labels that appear as an access pair in the touch instrumented version.

Consider for a moment that the `header` list would include a cycle. In that case, every element of the `header` list within the cycle could be accessed repeatedly by accesses `R1` and `W1` along the traversal in `traverse_header()`. Consider an element that is visited for the first time in recursive call `j`. This element would be touched by access labels `R1` and `W1`, and the access pair <R1,W1> would be created, just as described so far. Now consider that the same element is visited again in the traversal, in recursive call `k>j`. The same memory location is again touched with the `R1` and `W1` access labels, generating new access pairs <W1,R1> and <W1,W1> (which belong to `CombAccP(R1,W1)`). In this case, <R1,W1> would be both a zero distance data dependence and a greater-than-zero distance data dependence, as it may encompass the `R1` and `W1` accesses from the $\texttt{j}^{\texttt{th}}$ call, *and* the `R1` access from the $\texttt{j}^{\texttt{th}}$ call with the `W1` access from the $\texttt{k}^{\texttt{th}}$ call.

Whatever the method used to determine that the <R1,W1> access pair corresponds to a zero distance data dependence (code inspection or further instrumentation and reanalysis), this information can be used by a parallelization client to exploit coarse grain parallelism between different calls to the `traverse_header()` function.

4.3 Data dependence detection for *n-ways* traversal patterns

The approach described so far provides a useful technique for detecting heap-induced data dependences in a variety of programs featuring dynamic data structures with dependences found in loops or recursive functions traversing one selector. This applies even for nested loops or nested recursion. However, in the presence of recursive function that traverse through more than one selector, shape graphs resulting from the analysis of the different recursive flow paths can merge information about the heap accesses. In such case, the approach described is not sufficient to accurately identify conflicting accesses.

For the *n-ways* traversal pattern, we adopt a different approach. We perform *function cloning* [62], using a different version of a recursive function for each traversal path, and then we instrument them with *dynamic touch pseudostatements*, to identify the heap accesses performed through every path. A simple test is performed afterward to identify conflicting accesses in different traversal paths.

The key idea is to transform the program as if we were going to parallelize it, i.e., we decompose the traversal function into different versions for different traversal paths (one for each of the *n-ways* followed in the traversal). Of course, such a decomposition would only be profitable for a parallelization client if the traversal exhibited no dependences. At the moment of performing the program transformation, we do not know if such dependences exist. However, we conduct the analysis over that program configuration to identify possible dependences. If no dependences arise, then the program arranged in such manner can be parallelized without barriers or locks, as it is guaranteed that there are no conflicts due to heap accesses. In other words, we assume a parallel distribution of the traversal, and then we test whether that arrangement is parallel or not. Some implications about this approach are discussed in section 4.3.5, but let us first introduce the mechanisms involved.

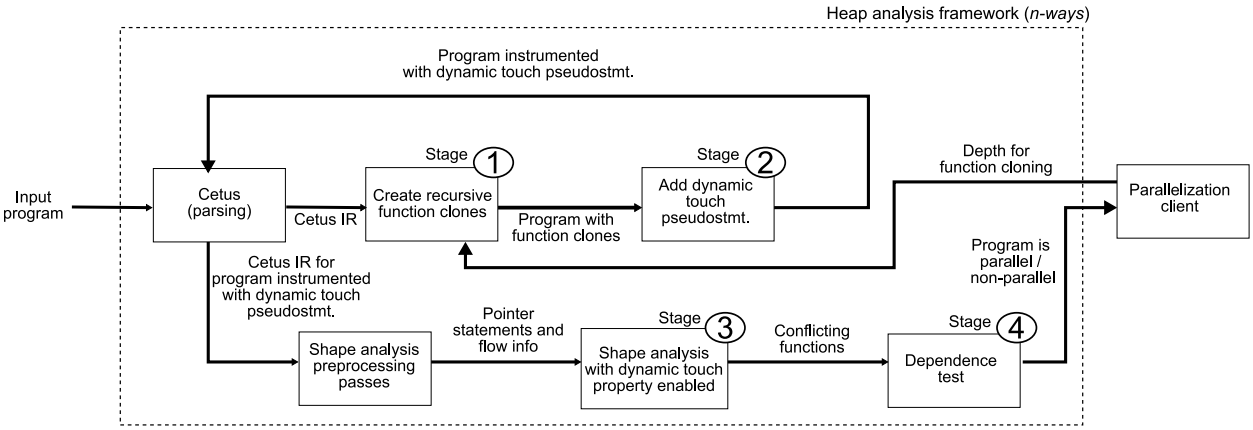

Figure 4.17: Presentation of our heap analysis framework displaying the four stages used for data dependence analysis in *n-ways* traversal patterns.

Fig. 4.17 shows the heap analysis framework configured for the dependence detection of the *n-ways* traversal pattern. Again, we organize the process in stages. However, these stages are different from those used for the *1-way* traversal pattern in Fig. 4.2. *Stage one* takes the program IR as input, and a value for the cloning depth, as specified by a parallelization client external to the framework (the user can provide it, if such client is not available). With this information, *stage one* performs the function cloning of recursive functions that carry out the traversal of the dynamic data structure. The program thus obtained is then fed to *stage two*, which adds *dynamic touch pseudostatements* to the heap accesses involved in the different traversal paths. The program is emitted as source again, then parsed and transformed for its shape analysis in *stage three*. This shape analysis must have the dynamic touch property enabled so that conflicting functions can be identified. This information is then used by the dependence test of *stage four* to report whether

```
   int TreeAdd (struct tree *t){
       int total_val,value,leftval,rightval;
       struct tree *tleft,*tright;
       #pragma SAP.excludeRFPTR(tleft,tright)
       if (t==NULL) {
1:        #pragma SAP.force(t==NULL)
          total_val=0;
       }else{
2:        #pragma SAP.force(t!=NULL)
3:        tleft=t->left;
4:        leftval=TreeAdd(tleft);
5:        #pragma SAP.force(t!=NULL)
6:        #pragma SAP.force(t->left==tleft)
7:        tleft=NULL;
8:        tright=t->right;
9:        rightval=TreeAdd(tright);
10:       #pragma SAP.force(t!=NULL)
11:       #pragma SAP.force(t->right==tright)
12:       tright=NULL;
          value=t->val;
          total_val=value+leftval+rightval;
          t->val=total_val;
       }
13:    return total_val;
   }
```

Figure 4.18: The `TreeAdd()` function used as running example for the `n-ways` traversal pattern.

the program arranged according to the specified cloning depth is parallel or not, regarding heap accesses performed in its traversal.

As running example for this section we will use the `TreeAdd()` function of the benchmark of the same name from the Olden suite [30]. This code was presented in chapter 3 but we display its code again in Fig. 4.18.

The data structure is a binary tree created dynamically. Every call to the `TreeAdd()` recursive function traverses the tree first through the left child of the current element, then through the right child. Therefore, this function presents a *2-ways* traversal pattern, as the tree is traversed through both the left and right selectors in each recursive call.

4.3.1 *Stage one (n-ways)*: perform recursive function cloning

Stage one of the process of data dependence detection for *n-way* traversal patterns performs function cloning of recursive functions involved in the traversal of the dynamic data structure. A suggested number of threads must be specified to drive the function cloning. That is, assuming the function is parallel (which we do not know at this stage), how many threads would we like to generate for the task? The results obtained at the end of the test will tell us if it is safe to parallelize in such manner or if dependences may arise. Although we focus here only on the dependence detection process, this mechanisms is tailored to be used in accordance with a parallelization client that would target the analysis for a specific number of threads. For our purposes, the number of threads to drive the function cloning is determined externally to the framework. For example,

it could be provided by the user.

Function cloning or *function specialization* is the process of creating specialized copies of function bodies [62]. In our approach, we clone the recursive function that performs the traversal of the dynamic data structure. It might be the case that such recursive function also calls to other recursive functions that perform further traversal in the structure. For simplicity, we restrict the function cloning to the uppermost recursive functions in the callgraph that perform the *n-ways* traversal of the data structure.

Fig. 4.19(a) shows an abridged version of the `TreeAdd()` function. It contains two recursive call sites. By cloning with depth one, we obtain two new versions, or clones, for `TreeAdd()`: `TreeAdd_left()` and `TreeAdd_right()`, shown in the callgraph-like diagram of Fig. 4.19(b). The call site in `TreeAdd()` for the traversal over the `left` selector now calls `TreeAdd_left()`, and the call site for the traversal over the `right` selector now calls `TreeAdd_right()`. Now, function `TreeAdd()` is no longer recursive, and as originating function, it is labeled `NTCL`, which stands for *non-terminal clone*. The left subtree from the root is traversed recursively with the `TreeAdd_left()` function, and the right subtree with the recursive function `TreeAdd_right()`. Both these functions are *terminal clones*, meaning that they are the deepest clones created for the originating function `TreeAdd()`, and so are labeled `TCL`.

This scheme is extended for a cloning depth of two, in Fig. 4.19(c), where a total of 6 clones have been generated. Note that only the functions in the last level of cloning (the *terminal clones*) are recursive, the functions in the previous levels change their calls so that they are not recursive anymore.

This example clearly exposes the main drawback of function cloning: it produces an exponential growth in the number of clones as we increase the depth of the cloning. More precisely, a level of cloning depth d, and for a traversal through n selectors, requires n^d clones, i.e., $\sum_{k=1}^{d} n^k$ clones in total up to depth d. Therefore, we advocate for a low depth of function cloning. The process of function cloning is designed so that each resulting terminal clone is processed by one thread. This means that a fewer number of threads, like two or four, is preferred. Although aiming for a higher number of threads is perfectly possible, the analysis cost will rise exponentially, just as the number of function clones.

The algorithm to perform function cloning in *stage one* is displayed in Fig. 4.20. First, the program IR is traversed looking for the right functions to clone. Those are the uppermost recursive functions in the callgraph that conform to the *n-ways* traversal pattern. Additionally, a map is created for each recursive call site in those functions, associating them with the selector that they traverse. The functions gathered with this method are cloned according to the number of recursive calls that they contain and the desired depth of cloning. They are also labeled as *non-terminal* or *terminal clones* (`NTCL` or `TCL`), depending on the case. The functions created at any level of cloning depth are used as cloning source for the next level. For instance, `TreeAdd_left()` is cloned into `TreeAdd_left_left()` and `TreeAdd_left_right()` for the second level of cloning depth (Fig. 4.19(c)).

For the purposes of explaining the stages for data dependence detection in *n-ways* traversal patterns, we will continue the `TreeAdd` example with the version tailored for two threads (Fig. 4.19(b)), which creates two clones of the `TreeAdd()` function, `TreeAdd_left()` and `TreeAdd_right()`.

4.3.2 *Stage two (n-ways)*: add dynamic touch pseudostatements

Once we have performed function cloning of the appropriate recursive functions, the next stage involves adding *dynamic touch pseudostatements* to the terminal, recursive clones. This is done in *stage two*. Since the goal is to discriminate whether the accesses by each thread may reach the same memory location, rather than just annotating all heap accesses, the way we instrument the program now is different to the way it was done for the *1-way* traversal pattern.

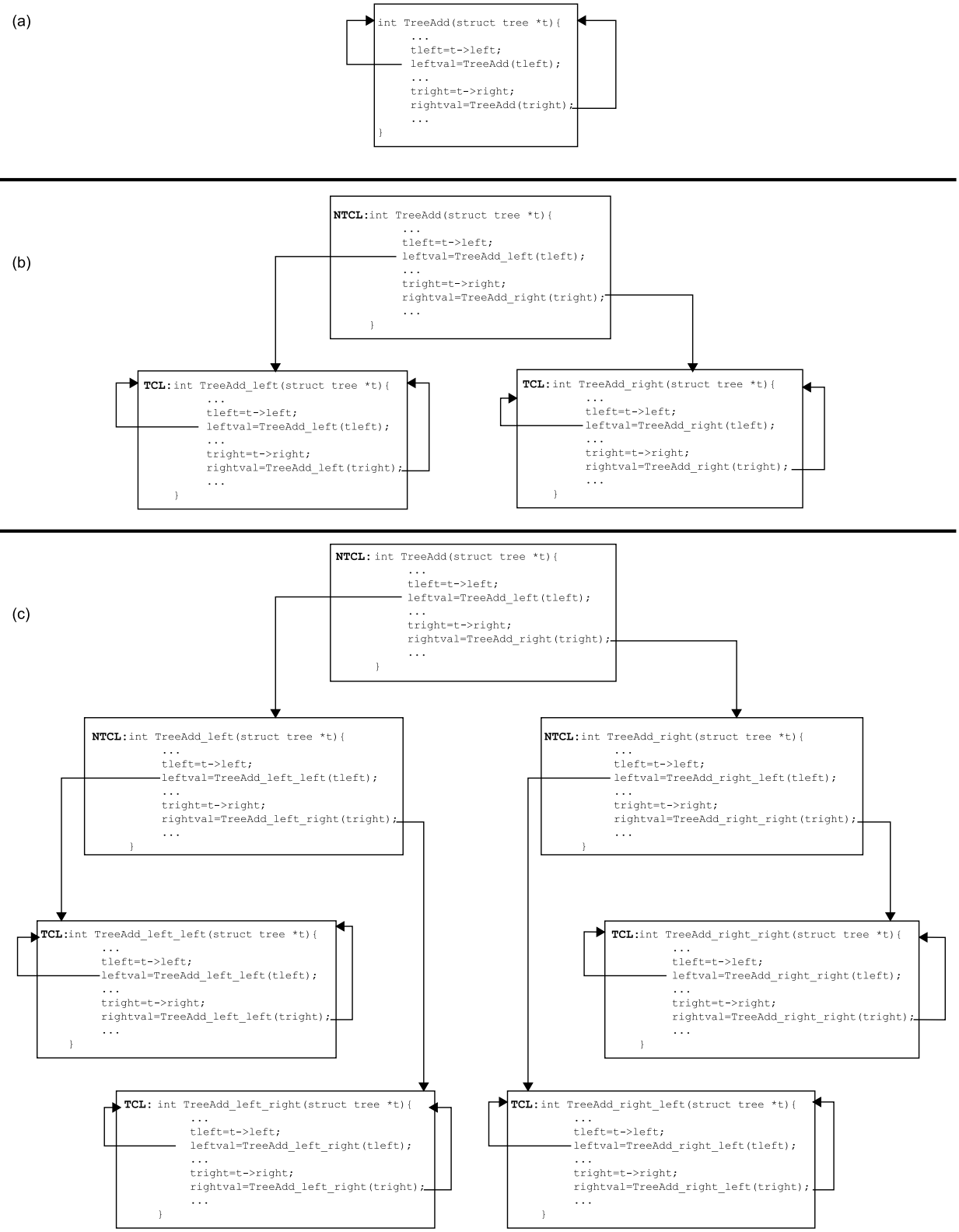

Figure 4.19: The `TreeAdd()` function in (a) the initial version, (b) performing function cloning of depth one, and (c) performing function cloning of depth two.

```
Create_clones()
  Input: P_IR, depth          # IR for the analyzed program, and desired depth of cloning
  Output: P'_IR               # IR with cloned functions

  Create P'_IR=P_IR
  FUN_CL = ∅          # Set of functions to clone
  repeat
      Get fun, the next function declaration statement in P'_IR
      If(fun is a recursive function of n-ways traversal pattern ∧ fun is not nested in fun2 s.t. fun2 ∈ FUN_CL)
          FUN_CL=FUN_CL ∪ fun
          forall rcs, recursive call site in fun
              MAP(rcs)=sel, where sel is the selector traversed for the rcs call
          endfor
  until (P'_IR has no more function declaration statements)
  forall fun ∈ FUN_CL
      NEWFUN_0=fun                # Set of functions to be cloned for depth 1
      d=1
      while (d<depth)
          Create NEWFUN_d=∅        # Set of functions to be cloned for depth d+1
          Increment d
      endwhile
      d=0
      while (d<depth)
          repeat
              Remove cur_fun from NEWFUN_d
              Add label NTCL to cur_fun
              forall rcs, recursive call site in cur_fun
                  Create cur_fun_clone as clone of cur_fun, appending selector MAP(rcs) to the name
                  Update all cloned recursive call sites to call cur_fun_clone recursively
                  Update rcs to call cur_fun_clone
                  If (d<depth-1)
                      NEWFUN_d+1=NEWFUN_d+1 ∪ cur_fun_clone
                  else
                      Add label TCL to cur_fun_clone
                  Add cur_fun_clone to P'_IR
              endfor
          until (NEWFUN_d=∅)
          Increment d
      endwhile
  endfor
  return(P'_IR)
end
```

Figure 4.20: The function used by *stage one (n-ways)* to perform recursive function cloning.

label so that we can distinguish later possible conflicts between the accesses in different terminal clones.

A *dynamic touch pseudostatement* (`dtouch(ptr,field)`) is a touch pseudostatement where the annotation is determined by a label set by a *label setting pseudostatement* (`setLb(label)`), prior to performing a call to a terminal clone. Upon returning from a terminal clone, the label is unset with a *label unsetting pseudostatement* (`unsetLb()`). The terminal function clones add a dynamic touch pseudostatement for every heap accessing statement. These pseudostatements will write the current label in the accessed node. That label is the name of the terminal clone that performs the access, plus the access field, which is included for the purpose of discriminating between accesses to different parts of heap elements. It is also possible to annotate the kind of access performed (read or write), although we do not consider it here, for simplicity.

In Fig. 4.21 we show the `TreeAdd()` function, and its two clones for the 2-threads version, instrumented with the appropriate *label setting*, *label unsetting*, and *dynamic touch* pseudostatements. The previous level to the terminal clones (in this case it is `TreeAdd()` proper) sets a specific label for the dynamic touch annotation before calling to each of the recursive clones. Likewise, the label is unset when returning to the caller. The terminal clones, `TreeAdd_left()` and `TreeAdd_right()`, include the necessary dynamic touch pseudostatements.

The algorithm that implements *stage two* is shown in Fig. 4.22. It first traverses non-terminal clones to set the label for dynamic touchs in the level of clones previous to the terminal level. Then the terminal clones are traversed adding a dynamic touch pseudostatement for each heap accessing statement in them. Also, descendants in the callgraph, i.e., the functions called from the terminal clones are stored in the `CGDESCESDANTS` set. Finally, this set is traversed to add dynamic touch pseudostatements to all descendants in the call graph that perform heap accesses.

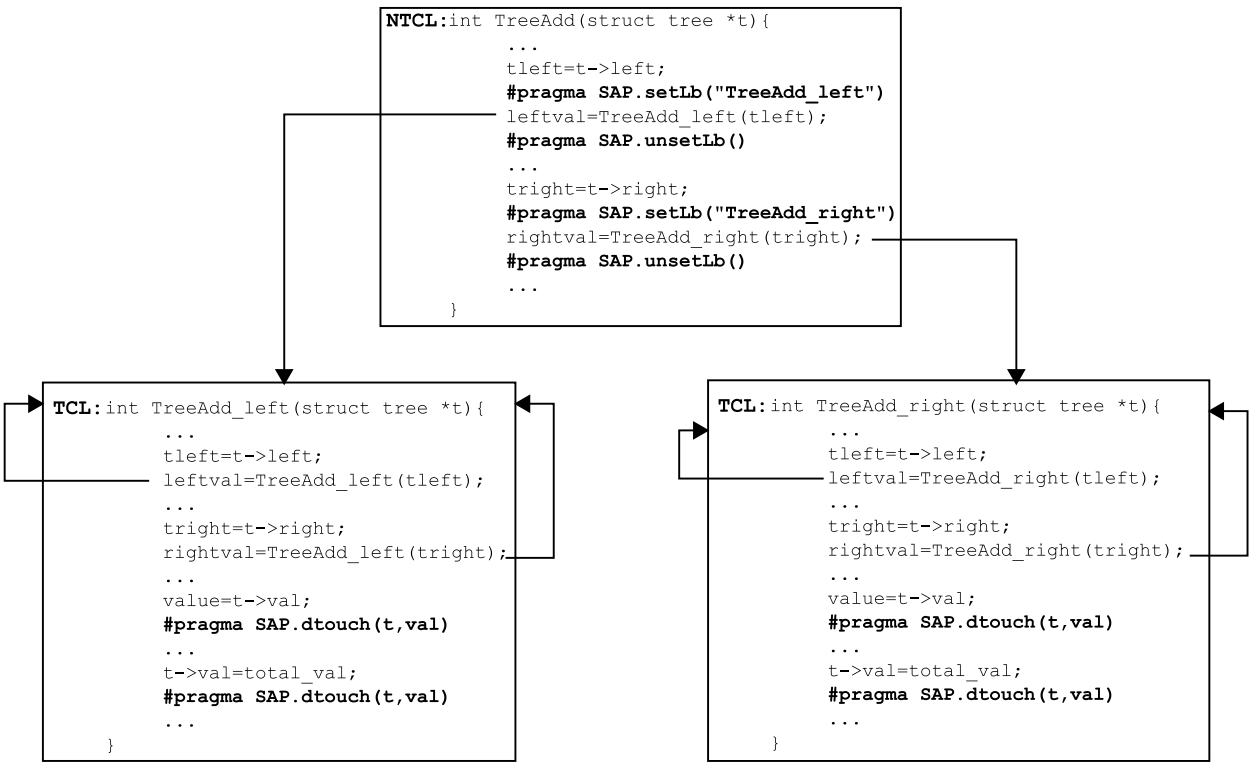

Figure 4.21: The `TreeAdd()` function, and its two clones for the 2-threads analysis, instrumented with *dynamic touch*, *label setting* and *label unsetting* pseudostatements, shown in bold typeface.

```
Add_dtouch_pseudostmts()
  Input: P_IR                  # IR for the analyzed program
  Output: P'_IR                # Instrumented IR

  Create P'_IR =P_IR
  # Process non-terminal clones
  repeat
      Get NTCL_fun, the next function labeled with NTCL in P'_IR
      repeat
          Get fcall, the next function call statement in NTCL_fun, calling function TCL_fun labeled with TCL
              Add #pragma SAP.setLb(TCL_fun) directive right before fcall in P'_IR
              Add #pragma SAP.unsetLb() directive right after fcall in P'_IR
      until (there are no more function calls in the body of NTCL_fun)
  until (there are no more functions labeled NTCL in P'_IR)
  # Process terminal clones
  CGDESCENDANTS= ∅          # Set of callgraph descendants of terminal clones
  repeat
      Get TCL_fun, the next function labeled with TCL in P'_IR
      repeat
          Get stmt, the next statement in TCL_fun
          Case (stmt)
              stmt is a heap accessing statement, i.e., stmt is of the kind x->field=data or data=x->field
                  Add #pragma SAP.dtouch(x,field) directive right after stmt in TCL_fun
              stmt is a function call statement to function CGD_fun, s.t. CGD_fun≠TCL_fun
                  CGDESCENDANTS=CGDESCENDANTS∪CGD_fun
      until (there are no more statements in the body of TCL_fun)
  until (there are no more functions labeled TCL in P'_IR)
  # Process callgraph descendants of terminal clones
  CGDESCENDANTS_done = ∅
  while (CGDESCENDANTS≠ ∅)
      Remove CGD_fun from CGDESCENDANTS
      CGDESCENDANTS_done =CGDESCENDANTS_done ∪ CGD_fun
      repeat
          Get stmt, the next statement in CGD_fun
          Case (stmt)
              stmt is a heap accessing statement, i.e., stmt is of the kind x->field=data or data=x->field
                  Add #pragma SAP.dtouch(x,field) directive right after stmt in CGD_fun
              stmt is a function call statement to function CGD_fun2, s.t. CGD_fun2 ∉ CGDESCENDANTS_done
                  CGDESCENDANTS=CGDESCENDANTS ∪ CGD_fun2
      until (there are no more statements in the body of CGD_fun)
  endwhile
  return(P'_IR)
end
```

Figure 4.22: The function used by *stage two (n-ways)* to add dynamic touch instrumentation.

4.3.3 *Stage three (n-ways)*: shape analysis with dynamic touch property

For *stage three*, we perform shape analysis with the dynamic touch property enabled, over the instrumented version of the program handed by *stage two*.

The clones previous to the terminal level set the name of the terminal clone they call as label for dy-

namic touch annotations. This name is annotated in all heap accesses performed under the call of the terminal clone, adding the access field for purpose of discriminating between accesses to different parts of the heap elements. Once the called terminal clone returns to the calling function, the label for dynamic touch statements is unset, thus disabling the annotation for any heap access not performed within a terminal clone. This process is repeated for all terminal clones called. The result is that nodes in the graph are annotated only with heap accesses performed under the different terminal clones, which are designed to be run by independent threads, and expected not to interfere for proper parallel execution.

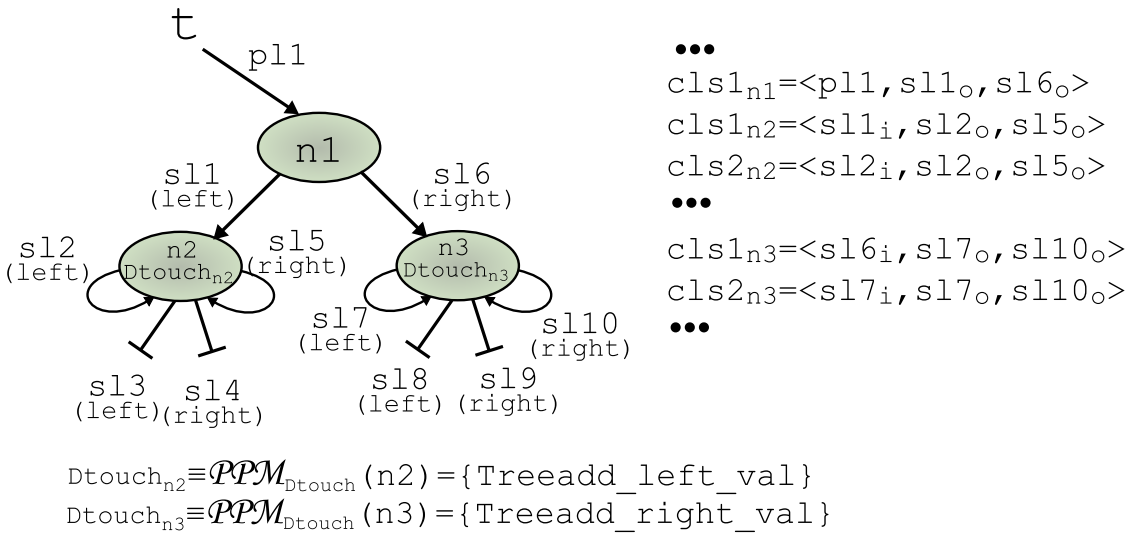

Figure 4.23: The tree resulting from the analysis of `TreeAdd()` with two clones, with nodes annotated with dynamic touch labels.

Fig. 4.23 shows the result of the analysis of the `TreeAdd` program, as instrumented by *stage two*. Here we can see that the memory locations accessed through the different terminal clones are different, as there is no node sharing labels from more than one terminal clone.

Every time a dynamic touch pseudostatement is encountered by the analysis, the algorithm `Dtouch()` (Fig. 4.24) is run. It checks whether the value of the global variable `curFunLb` is set or not. If it is set to any function name, then its value is annotated in the node, appending the access field in the statement (`curFunLb_field`). Otherwise, it means that we are not under the execution of a terminal clone and we should not register heap accesses in nodes. The value of the `curFunLb` variable is set by the `SetDtouchLb()` function and unset by the `UnsetDtouchLb()` function (also Fig. 4.24).

Also, in case that the same node has been previously accessed under a different terminal clone, with the same access field, the pair of terminal clones registered in the previous and current labels is stored in `CONFLICT_FUN`$_{\texttt{field}}$. This set will be checked in *stage four* to draw dependence information.

The domain for the dynamic touch property is the same as the domain for the regular touch property, i.e., a set of labels defined by preprocessing directives, as described in chapter 2. Compatible nodes will need to feature the same value for the dynamic touch property. Remember that all available properties must be compatible for two nodes to be regarded as compatible, and therefore merge them in the summarization process of shape graphs.

```
SetDtouchLb()
 Input: funLb                  # A function name as a label
 Output: none

 curFunLb=funLb               # Set value for global var curFunLb
 return()
end

UnsetDtouchLb()
 Input: none
 Output: none

 curFunLb=NULL                # Unset value for globar var curFunLb
 return()
end

DTouch()
 Input: sg^1=<N^1,CLS^1>, x ∈ PTR, field ∈ FIELD      # A shape graph, a pointer and a data field
 Output: sg^k=<N^k,CLS^k>                             # A shape graph

 Create N^k=N^1
 Create CLS^k=CLS^1
 If (curFunLb is set)          # If the label for dynamic touch is set
    Find ni ∈ N^k s.t. ∃ pl=<x,ni> ⊂ CLS_ni
    If (prevFunLb_field2 ∈ PPM_Dtouch(ni) s.t. prevFunLb≠curFunLb ∧ field2=field)
       CONFLICT_FUN_field =CONFLICT_FUN_field ∪ <curFunLb,prevFunLb>
    PPM_Dtouch(ni)=PPM_Dtouch(ni) ∪ curFunLb_field
 Create sg^k=<N^k,CLS^k>
 return(sg^k)
end
```

Figure 4.24: The `SetDtouchLb()`, `UnsetDtouchLb()`, and `Dtouch()` functions to perform the adequate annotations in nodes for *stage three (n-ways)*.

4.3.4 *Stage four (n-ways)*: dependence test

The last stage in the process of detecting heap-induced data dependences in *n-ways* traversal patterns involves a simple check, as all the necessary information has been gathered in *stage three*. This test is performed by function `Dep_test_nways()` (Fig. 4.25). This function checks the `CONFLICT_FUN` superset, which contains all the sets of conflicting functions sorted by access field.

For that, all data fields of recursive data types are considered. If a pair of conflicting functions is found, then the program exhibits a possible dependence between the reported pair of functions for the specified data field, and for the program configuration obtained by *stage one*. If it does not exist, then it is safe to parallelize the program (regarding heap data accesses) assigning a different thread to each of the terminal clones.

For the `TreeAdd` example, there is no pair of terminal clones whose heap accesses may interfere, so the test would return `FALSE`, meaning that there are no dependences due to heap accesses performed by `TreeAdd_right()` and `TreeAdd_left()`.

```
Dep_test_nways()
 Input: CONFLICT_FUN      # The set of conflicting functions, sorted by accessed field
 Output: TRUE/FALSE       # Is the analyzed program free of heap-induced data dependences?

 forall CONFLICT_FUN_field ⊂ CONFLICT_FUN
    If (CONFLICT_FUN_field ≠ ∅)
       return(TRUE)
 endfor
 return(FALSE)
end
```

Figure 4.25: The function that checks dependences for *n-ways* traversal patterns.

4.3.5 Further considerations

The approach described for the *n-ways* traversal pattern is based on assuming a parallel distribution of the traversal of a dynamic data structure, and then checking for dependences in the arrangement suggested. This is useful even for traversals that perform allocation, deallocation and/or structure modification, as possibly conflicting heap accesses performed under different recursive functions are stored in the $\texttt{CONFLICT\_FUN}_{\texttt{field}}$ sets.

For most cases, if dependences do not arise for a distribution of two terminal clones, they will not arise for any deeper traversal distribution. Cases where this would not hold can be expected to be very rare. However, in the general case, a certain parallel distribution cannot be guaranteed to hold for a larger number of threads than those used for the analysis considered. We are aware that this can limit the applicability of the technique as the analysis for a deep cloning traversal distribution is probably prohibitive for most programs, due to the analysis cost. Most commonly, the analysis will be carried out targeting only two threads. We believe this approach is specially valuable for dual-core architectures.

In the technique presented for the data dependence detection in *n-ways* traversal patterns, we have not considered differentiation between types of dependences. In general, it is not possible to distinguish anti or flow dependences. However, output dependences (two write accesses on the same location) could be identified just by adding, (i) read/write kind of access in the `Dtouch()` function, and (ii) a simple check for two write accesses in `Dep_test_nways()`. This would permit another analysis client to solve that dependence with privatization and synchronization techniques.

The characteristics of this method favors the exploitation of coarse-grain parallelism, i.e., few threads with a substantial workload, as opposed to many threads with little workload. Of course, different programs will benefit from different parallelization schemes. However, it is our intuition that there is a large number of pointer-based applications that can benefit from coarse-grain parallelism. Some other authors suggest the same, like Kulkarni et. al [63], who provide evidence that coarse-grain parallelism is needed for improving the performance of irregular applications that manipulate pointer-based data structures.

Other approaches rely on generation of many small tasks, one per traversal step, and then assign them to available threads. Such is the way favored by the `task` construct in OpenMP 3.0 [16]. This approach can generate an enormous number of threads that can negate speedup for small workloads. Kejariwal et. al [64] show that the threading overhead for the parallelization of innermost loops from SPEC CPU2006 benchmarks [65], renders parallelization of such loops as unprofitable. Although their work is targeted for *thread-level speculation* (TLS), this also holds for loops without dependences. Their work supports our idea that parallelizing recursive functions of small workload per traversal step, as encouraged by `task`-like constructs, is unlikely to be profitable.

4.4 Related work in dependence analysis

Most of the related work in shape analysis that we have discussed in previous chapters is not concerned with data dependence analysis. It is mostly concerned with other client analysis, mainly *verification*.

For instance, the works based in TVLA and 3-valued logic ([29], [45], [46], [47], [48]) are mainly concerned with codifying spatial shape invariant information. Such information is useful for program verification clients, and the experimental results reported by these authors reflect this approach. They test simple programs based on singly-linked lists or binary trees, and use their shape analysis strategy to prove code correctness and the conservation of the shape invariants in the structure in the presence of destructive updating.

Likewise, the works in separation logic ([38], [41], [40], [39]) are concerned in finding predicates that encode spatial relationships between heap locations, and are also targeted by their authors toward program verification. These works also design specific shape analysis strategies for particular target data structures. For instance, Berdine et. al [40] create a shape analyzer specifically targeted for structures based on linked lists (trees are not supported by definition) and use it for a verification client that is tested with a library of a firewire driver.

We believe that the kind of analysis that is suitable for verification is not easily adaptable toward dependence detection. Verification is mainly concerned with proving code correctness and obtaining structure invariants and therefore cannot make any assumptions about the program. In our approach, we assume code correctness, and we are not so much concerned with the structure invariants than actual access relationships, which determine dependences.

Also, in our approach, we do not limit the scope of applicability of the analyzer to a specific type of structure. Rather, we construct shape graphs modeling the heap in the program in a general way. This allows us to obtain good results for programs that manipulate data structures that are not "clean" lists or trees or a combination of some predetermined data type (e.g. lists). Sometimes the analysis may not be precise enough, but often properties can be used to solve such cases.

More related to our approach of using heap analysis for data dependence are the works [56], [37] and [66]. Ghiya and Hendren [56] proposed a test for identifying data dependences, relying on a characterization of the overall data structure as the Tree, DAG or Cycle shapes. Such knowledge is used to identify possible conflicts for the pointer access paths in the statements being analyzed. In the case of the Cycle shape, all precision is lost, and dependences must be conservatively reported.

Also using the overall structure characterization of Tree, DAG or Cycle, Hwang and Saltz [37] devised a dependence test based on the calculation of interprocedural def-use chains for pointer variables. These def-use chains allow them to assess a "shape" for the traversal of the structure. For example, their approach is able to identify a non-cyclic traversal of a cyclic structure. This is useful in the case of having heap accesses that do not follow the cyclic paths in the structure and do not provoke dependences. Such a case could not be identified by [56]. The main drawback in these works is that they are not useful in programs that perform destructive updates in the loops under test, in other words, the structure cannot change within the loop of study.

We first used successfully the idea of annotating heap accesses in nodes in the work of [2]. Marron et. al borrow this idea for dependence analysis on their particular store heap model in [66]. They target Java codes entirely based on manually-tuned collection libraries. Their approach is fast, partly by limiting the discrimination of conflicting accesses. As a consequence, their work is not adaptable to detect different kind of dependences (anti, flow, output) or zero distance data dependences.

In our approach, we annotate the memory locations reached by each heap-directed pointer, with read/write

information. This feature let us capture, more accurately than any other approach, the temporal relationship between the statements that visit the locations of the program heap. Our algorithm supports recursion, and allows the data structure to be modified during the analysis, which enables us to support a wide range of programs that feature loops or recursive functions that traverse and create generic heap-based recursive/dynamic data structures in programs that perform destructive updates. On top of that, we can distinguish among anti, output, and flow data dependences, as well as zero-distance data dependences in loops or recursive functions that comply with the *1-way* traversal pattern.

4.5 Experimental results

We have implemented the algorithms described in this chapter for heap-induced data dependence analysis within our heap analysis framework. We have conducted some experiments that we review next.

4.5.1 Benchmarks and tests

We have considered eight programs for out data dependence detection tests. They are summarized in Table 4.1, and we present them in more detail next. Beside the benchmark name, we include the traversal pattern featured, the data structure and the result obtained by the test. The first two benchmarks are drawn from the running examples of this chapter to provide a baseline for the analysis performance. The rest of benchmarks were introduced in previous chapters.

Benchmark	Traversal pattern	Data structure	Result
1-Running_ex_lcd0	1-way (2 nested loops)	List of lists (each element in the singly-linked header list points to another singly-linked list of heap elements)	2 anti dependences (1 loop-carried dep. for inner loop, 1 zero dist. for outer loop)
2-Running_ex_rec	1-way (2 nested rec. fun.)	List of lists (each element in the singly-linked header list points to another singly-linked list of heap elements)	2 anti dependences (1 loop-carried dep. for inner loop, 1 zero dist. for outer loop)
3-Matrix x Vector(s)	1-way (3 nested loops)	Singly-linked sparse matrix and vector	No dependences
4-Matrix x Matrix(s)	1-way (4 nested loops)	Singly-linked sparse matrices and vector	No dependences
5-Em3d	1-way (2 nested loops)	2 singly-linked lists interconnected forming a bipartite structure	Zero distance anti dependence
6-TreeAdd	2-ways (2 terminal cl.)	Binary tree	No dependences
7-Power	1-way (3 nested rec. fun.)	Multilevel structure of singly-linked lists	Zero distance anti dependences
8-Bisort	2-ways (2 nested rec. fun., 2 terminal cl.)	Binary tree	No dependences

Table 4.1: Summary of benchmark programs used for our data dependence tests.

1. `Running_ex_lcd0`. This is the example of Fig. 4.11. It features a *list of lists* dynamic data structure that is traversed in two nested loops, with a *1-way* traversal pattern. The test identifies two anti dependences, one for the outer loop, and other for the inner loop. The mechanism of the iteration vector described in section 4.2.6 is employed by the analysis to recognize the dependence of the outer loop as a zero distance dependence, and the dependence of the inner loop as a loop-carried dependence. This benchmark is run with the *type* and *touch* properties enabled.

2. `Running_ex_rec`. This is the example of Fig. 4.14. It features the same data structure and obtains the same results as the previous benchmark, but it is based on a traversal with two nested recursive functions, rather than two nested loops. Still, it follows the *1-way* traversal pattern. The dependences identified are the same as above. This benchmark is run with the *previous call*, *paired selectors*, *type*, and *touch* properties.

3. `Matrix x Vector(s)`. This is the sparse matrix by sparse vector benchmark, based on singly-linked lists, presented in chapter 2. This is the version without pruning to improve performance. The pruned part in chapter 2 features the heap accesses and therefore cannot be obviated for the purpose of dependence detection. The analysis reports no dependences for the loop that computes the product, as each read operation is performed over the input matrix or vector, and every writing operation is performed on the output vector. This benchmark is run with the *site* and *touch* properties enabled, and features a *1-way* traversal pattern.

4. `Matrix x Matrix(s)`. This is the sparse matrix by sparse matrix benchmark, based on singly-linked lists, presented in chapter 2. Again, this version allows no pruning. The structure accessed for reading (two input matrices) is different from the structure accessed for writing (output matrix). This is maintained by the analysis, which correctly reports no dependences. This benchmark is run with the *site* and *touch* properties enabled, and again features a *1-way* traversal pattern.

5. `Em3d`. This is the Olden benchmark presented in chapter 2 as well. It is formed by two singly-linked lists, where the elements in a list point to several elements in the other list, forming a bipartite structure. The `compute_nodes()` function, that computes the values of the electric and magnetic field, is instrumented for dependence detection with touch pseudostatements. A zero distance anti dependence is found as each element is read and then updated in every iteration of the traversing loop, in a *1-way* pattern. This benchmark is run with the *site* and *touch* properties enabled.

6. `TreeAdd`. This is another benchmark from the Olden suite. It was introduced in chapter 3, and has been used in the present chapter as example of a *2-ways* traversal pattern in a tree. It is cloned with a depth of one, meaning that only two terminal clones are generated, aiming for a 2-threads traversal distribution. The heap accesses of the terminal clones do not interfere, and so the tested program is reported as parallel due to heap accesses in the traversal. This benchmark is run with the *previous call*, *paired selectors*, and *dynamic touch* properties enabled.

7. `Power`. Another benchmark from Olden, it was presented in chapter 3, and features nested recursion for a *1-way* traversal of a multilevel structure of singly-linked lists. Elements are updated in the traversal, like in `5-Em3d`, thus producing zero distance anti dependences for several data fields. For simplicity, these data fields are grouped for this test, as to reduce the number of required touch pseudostatements. As a result, finer distinction of dependences for different data fields is lost. Nevertheless, zero distance anti dependences are discovered for the analysis. This benchmark is run with the *type*, *previous call*, *paired selectors*, and *touch* properties enabled.

8. `Bisort`. Yet another Olden benchmark, also presented in chapter 3, that features a `2-ways` traversal pattern of a binary tree with two recursive functions that perform nested traversals. Like in `6-TreeAdd`, we perform function cloning of depth one, aiming for a 2-threaded version of the program, separating the traversal of the left subtree from the right in the first function involved in the traversal. Again, the heap accesses from both terminal clones do not interfere, and the program is reported parallel for the analyzed configuration. This benchmark is run with the *previous call*, *paired selectors*, and *dynamic touch* properties enabled.

Benchmarks `6-TreeAdd` and `8-Bisort` feature *2-ways* traversal patterns of a binary tree. The rest feature a variety of data structures and traversals, always conforming to the *1-way* pattern. We are able to obtain accurate dependence information for all the presented benchmarks, identifying the access pairs and conflicting functions where they exist.

We show some measures about performance and problem size for the dependence analysis of these benchmark programs in Table 4.2. We register analysis time, memory consumed, number of analyzable statements, number of analyzed statements until the fixed point is reached, and the number of shape graphs generated. As in previous chapters, the testing platform is a 3GHz Pentium 4 with 1 GB RAM.

Excluding the base test `1-Running_ex_lcd0`, which takes less than one second, we see that some programs need a few seconds (`2-Running_ex_rec`, `6-TreeAdd`, and `7-Power`) while other need a few minutes (`3-Matrix x Vector(s)`, `4-Matrix x Matrix(s)`, and `5-Em3d`), with `8-Bisort` taking the longest time, with more than 45 minutes. Memory consumption seems very reasonable with less than 20 MB in the worst case. The longest programs are more than a hundred analyzable statements long. We only report statements that deal with the heap, either constructing, traversing or accessing the structures. Remember that touch pseudostatements model the effect of heap accessing statements for the data dependence analysis.

The number of analyzed statements until the fixed point is reached stays at just a few thousands, with the exception of `1-Running_ex_lcd0` with the lowest value, and `8-Bisort` with more than 100,000 analyzed statements. The number of shape graphs varies some more, peaking again for `8-Bisort` with more than 350,000 generated graphs.

Benchmark	Time	Space	Code stmts.	Analyzed stmts.	Shape graphs
1-Running_ex_lcd0	0.41 s	1.9 MB	56	585	721
2-Running_ex_rec	20.52 s	11.5 MB	60	3,422	8,416
3-Matrix x Vector(s)	2 m 19.80 s	3.8 MB	97	1,333	16,755
4-Matrix x Matrix(s)	9 m 13.38 s	4.9 MB	131	4,356	30,507
5-Em3d	2 m 5.88 s	11.8 MB	178	1,475	2,454
6-TreeAdd	31.42 s	7.5 MB	67	5,332	11,754
7-Power	12.91 s	8.4 MB	73	3,698	6,454
8-Bisort	45 m 6.88 s	63.6 MB	116	107,165	355,003

Table 4.2: Performance and problem size for the benchmarks used for dependence detection. The testing platform is a 3GHz Pentium 4 with 1GB RAM.

More information about these experiments is gathered in Table 4.3. It gives some metrics related to the complexity of the shape graphs. Next to each benchmark, we list first the number of touch or dynamic touch pseudostatements used for the analysis. Remember that the touch pseudostatements are used for the *1-way* traversal pattern dependence test, while the dynamic touch pseudostatements are used for the *n-ways* traversal pattern dependence test. This number is a small value for all benchmarks as it is usually just one pointer field that may have conflict. This is not the case for `7-Power`, but we group the access fields in this benchmark for simplicity. The highest value is 6 for `8-Bisort`.

Also in Table 4.3, we show the average number of shape graphs per analyzable statement, and the average number of nodes and `cls`'s per graph, with the maximum values in parentheses. The number of shape graphs per code statement is kept as a few hundreds as most, except for `8-Bisort`. The average number of nodes is around 9–10 for most benchmarks, confirming that the use of more properties produces more nodes per graph. The number of `cls`'s in average is quite controlled, with only three programs needing more than a hundred (`4-Matrix x Matrix(s)`, `5-Em3d`, and `8-Bisort`).

Some insights can be drawn from the results displayed in tables 4.2 and 4.3:

- The same program written in different ways can result in very different costs for the analysis. In par-

Benchmark	(D)touchs	Sg's / code stmt.	Avg. nodes / sg (max)	Avg. cls's / sg (max)
1-Running_ex_lcd0	4	12.88	5.98 (10)	19.80 (46)
2-Running_ex_rec	4	140.27	9.91 (13)	33.18 (50)
3-Matrix x Vector(s)	3	172.73	10.38 (14)	82.86 (122)
4-Matrix x Matrix(s)	3	232.88	14.85 (19)	118.47 (168)
5-Em3d	3	13.78	9.38 (13)	128.06 (556)
6-TreeAdd	4	175.43	5.42 (8)	46.95 (116)
7-Power	5	88.41	9.90 (14)	37.80 (66)
8-Bisort	6	3,060.37	8.41 (10)	107.50 (208)

Table 4.3: Shape graph complexity for the benchmarks used for dependence detection.

ticular, recursive algorithms are more costly, for the same traversal over the same data structure, than iterative algorithms. This can be observed for `1-Running_ex_lcd0` and `2-Running_ex_rec`, where the first runs 50 times faster, and takes nearly 6 times less shape graphs to reach the fixed point, while reporting the same dependences.

- The dependence test for *2-ways* traversal patterns is much more costly than the dependence test for `1-way` traversal patterns. For instance, `8-Bisort` takes much more time, and many more shape graphs than `5-Em3d`, even though `5-Em3d` has more nodes and `cls`'s per graph, and more analyzable statements. Added to the cost of analyzing recursive algorithms versus iterative ones, `8-Bisort` also needs to analyze the clones of recursive functions, multiplying the cost of the analysis.

- The number of nodes per graph does not grow too much despite the touch annotations labels. The touch and dynamic touch properties increase the number of nodes per graph. A bigger number of annotation labels can exponentially increase the number of nodes, as they may register different combinations of labels. Luckily, heap accesses occur with a pattern, and memory locations tend to be touched with the same heap accesses, therefore limiting the number of different combinations of annotation labels in nodes. The biggest number of nodes per graph is registered for the sparse matrix benchmarks, with only three touch pseudostatements each. In this case, this is caused by the site property, also used for the analysis. The use of the site property benefits these benchmarks as shown in the experimental section of chapter 2.

These tests show that the techniques presented in this chapter are successful in identifying heap-induced data dependences for a variety of data structures and algorithms, both for the *1-way* and *2-ways* traversal patterns. The cost has increased with regards with the results shown for previous chapters. We shall explore this issue next.

4.5.2 Cost of dependence test over shape analysis

It is clear that the dependence test poses some extra stress on the technique, degrading the performance regarding the analysis that only captures and maintains the shape of data structures. This can be seen at a gross level by comparing the results of tables 4.2 and 4.3, with their homologues from chapters 2 and 3. However, we would like to have a quantification of that extra cost to ponder the real impact of the dependence analysis. That is the purpose of Table 4.4.

Table 4.4 presents the percentage increment in some metrics for the dependence analysis with regards to just the shape analysis. This means that the same program has been analyzed twice: once with the touch or dynamic touch property enabled, and once with that property disabled. The program tested is the same, this means that for the codes with function clones (`6-TreeAdd` and `8-Bisort`), the extended, cloned version

Benchmark	Inc. time	Inc. space	Inc. Sg's	Inc. nodes	Inc. cls's
1-Running_ex_lcd0	55.6%	0%	16.1%	27.2%	31.9%
2-Running_ex_rec	1,059.3%	310.7%	380.1%	71.2%	62.4%
3-Matrix x Vector(s)	5,874.4%	35.7%	594.4%	39.0%	172.0%
4-Matrix x Matrix(s)	1,750.8%	32.4%	217.2%	22.3%	127.2%
5-Em3d	582.3%	114.9%	21.9%	7.2%	11.4%
6-TreeAdd	53.1%	50.0%	18.9%	25.5%	20.3%
7-Power	58.8%	31.2%	4.6%	28.9%	32.4%
8-Bisort	838.1%	285.5%	344.7%	26.9%	60.7%

Table 4.4: Increment in several measures of the shape analysis instrumented for dependence test with regards to just the shape analysis.

is used for both runs. Only the time of the shape analysis stage is considered, i.e., the time for *stage four (1-way)* or *stage three (n-ways)*.

The metrics we have considered for this comparison are: increment in time, increment in memory consumed, increment in the total number of generated shape graphs, increment of nodes per graph in average, and increment in `cls`'s per graph in average. The increments in analysis time vary greatly. For instance, `6-TreeAdd` has a 53.1% increment. This means that if the shape analysis of the cloned version of this program takes 20.52 seconds, the dependence test takes 53.1% *more* time, that is, 31.42 seconds. For `3-Matrix x Vector(s)` however, the increment is nearly 6,000%, showing the great variability of this metric.

The increment in memory consumed is more controlled, but nevertheless it surpasses a 200% increment in memory for two programs (`2-Running_ex_rec` and `8-Bisort`). The increment in shape graphs is again quite variable, ranging from as little as 4.6% for `7-Power` to nearly 600% for `3-Matrix x Vector(s)`. The increment in number of nodes per graph in average is more controlled peaking at 71.2% for `2-Running_ex_rec`. The increment in average number of `cls`'s per graph is also in a similar range but surpassing a 100% increment for the sparse matrix benchmarks.

Some insights can be drawn from the results displayed in Table 4.4:

- The increment in analysis cost in the dependence test benchmarks, mainly analysis time, is significant with regards to performing just the shape analysis, and it also varies a lot between different programs. This makes it hard to predict the behavior of the analysis, though some guidelines apply, as hinted next.
- A strong increment in the number of generated graphs produces an even greater increment in analysis time. This can be seen for `2-Running_ex_rec`, `3-Matrix x Vector(s)`, `4-Matrix x Matrix(s)`, and `8-Bisort`.
- The increment in average number of nodes and `cls`'s per graph is quite moderate for most cases. However, when the shape graph complexity rises above a threshold, the penalty in analysis time is significant. This can be observed for `5-Em3d`, where for a small increment in shape graphs, nodes and `cls`'s per graph, the analysis time increases by 582.3%. In this scenario, the limiting factor is the cost of the abstract semantics operations for a large amount of `cls`'s. Note that `5-Em3d` has the biggest number of `cls`'s per graph, but more importantly the maximum value is more than 4 times the average (Table 4.3). In such a situacion even a moderate increment in `cls`'s, caused by the instrumentation required for the dependence test, has a big impact over the abstract semantics operation. This effect is also noticed in `3-Matrix x Vector(s)`, where the increment in the number of shape graphs is not sufficient to justify the enormous increment in analysis time. This benchmark is affected as well from a significant increment in `cls`'s per graph which places extra burden on the shape analysis internal operations.

We have seen that being able to detect dependences comes at a cost. We are aware that the analysis may be too costly for some purposes. In that sense, we think the approach is suitable to analyze sections of programs, or function libraries. Let us remind the reader too that we are targeting data dependences in heap directed structures that are a challenge for current parallelizing compilers. Related work in shape analysis is not suitable for dependence analysis, and other works in dependence analysis are not sufficiently precise for complicated traversals and/or structures.

It should also be noted that the approach that we have taken for dependence analysis is based solely on the capabilities of the shape analysis technique. This comes as a natural evolution of the foundational work presented in previous chapters. We plan to continue our work to devise a more complete data dependence detection scheme that can benefit from the shape analysis virtues and try to avoid most of its associated costs. Ideas in this respect are discussed as future work in the next chapter.

Now, we would like to complete our experimental results section with information of the role of untouch pseudostatements for zero-distance data dependence discrimination and some measures about the scalability of our approach to data dependence analysis for the *n-ways* traversal pattern.

4.5.3 Further instrumentation with untouch pseudostatements

From the tests reported for dependence analysis, there are a few that feature zero distance dependences. We can apply the techniques presented in section 4.2.6 on them.

`1-Running_ex_lcd0` and `5-Em3d` have *1-way* traversal patterns with loop traversals. The extension of the iteration vector within touch annotations is used in these tests to correctly identify the dependences found as zero distance dependences. That mechanism has been considered for the previous measures. However, `2-Running_ex_rec` and `7-Power` feature *1-way* traversal patterns in recursive functions. In that case, we need one of two mechanisms available: (i) obtain that information from the source program, or (ii) instrument the programs further with untouch pseudostatements. For both `2-Running_ex_rec` and `7-Power`, the source program inspection is enough to determine that the access pairs found are zero distance dependences.

For completion though, we show the measures for the touch-untouch instrumented version for these two tests, which also results in the identification of the dependences as zero distance dependences. Keep in mind that the way of identifying zero distance dependences in recursive *1-way* traversals by obtaining information from the source code is not valid for all programs, while the method of instrumenting with untouch pseudostatements is.

Benchmark	(Un)touchs	Time	Space	Code stmts.	Analyzed stmts.	Shape graphs
2-Running_ex_rec(t)	4	20.52 s	11.5 MB	60	3,422	8,416
2-Running_ex_rec(tu)	16	15.47 s	9.4 MB	72	5,662	9,838
7-Power(t)	5	12.91 s	8.4 MB	73	3,698	6,454
7-Power(tu)	12	11.66 s	7.3 MB	80	5,330	8,027

Table 4.5: Measures for the `2-Running_ex_rec` and `7-Power` benchmarks, considering the touch instrumented version (`t`), and the touch-untouch instrumented version (`tu`).

The measures of the touch-untouch instrumented version (labeled `(tu)`) are shown next to the measures of the touch instrumented version (labeled now with `(t)`, showing the measures presented before), for comparison. They can be found in Table 4.5. We can see that the cost of the second analysis is slightly less than that of the touch instrumented version. This is due to the fact that dependence information is not carried across context changes, which makes the analysis simpler. However, the number of generated graphs is slightly higher, accounting for more possibilities of nodes being touched or untouched by access

annotations.

4.5.4 Scalability of the dependence detection scheme for *n-ways* traversal patterns

Our dependence analysis scheme for *n-ways* traversal patterns involves creating function clones of the main traversing functions. As discussed in section 4.3.5, this scheme requires creating an exponential number of clones for a bigger number of threads intended for parallel execution. That is why we recommend targeting this technique for few threads, unless the cost of the analysis is not an issue.

As a hint on the scalability of the analysis for a higher number of threads, we have conducted the dependence analysis for `6-TreeAdd` and `8-Bisort`, tailored for four threads and measured the results. They are compared in Table 4.6, which features the measures for the 2-threads version (labeled `(2-th)`, showing the results presented before), and the measures of the 4-threads version (labeled `(4-th)`).

Benchmark	Dtouchs	Time	Space	Code stmts.	Analyzed stmts.	Shape graphs
6-TreeAdd(2-th)	4	31.42 s	7.5 MB	67	5,332	11,754
6-TreeAdd(4-th)	8	6 m 15.84 s	19.6 MB	135	10,337	21,134
8-Bisort(2-th)	6	45 m 6.88 s	63.6 MB	116	107,165	355,003
8-Bisort(4-th)	10	1 h 51 m	198.6 MB	170	219,865	734,121

Table 4.6: Measures for the `6-TreeAdd` and `8-Bisort` benchmarks, considering the versions tailored for two threads (`2-th`) and four threads (`4-th`).

As expected, the analysis cost has greatly increased for the version with four recursive function clones. The analysis reports no dependences again due to heap accesses by the different clones, so the programs tested are ready to be parallelized with four threads.

4.6 Summary

This chapter concludes our efforts to apply the shape analysis technique based on the coexistent links sets abstraction, with added interprocedural support, for heap-induced data dependence analysis in applications that deal with dynamic, recursive data structures. Due to the variety of ways to traverse dynamic data structures, through one or more selectors, using loops or recursive functions, we have devised a variety of techniques, all based on annotating heap access information on nodes in shape graphs. Next, we summarize the content presented in this chapter.

- First, we present our motivation for dependence analysis with shape analysis techniques, and introduce the distinction between *1-way* and *n-ways* traversal patterns (section 4.1).
- Next, we focused on dependence analysis on the simpler *1-way* traversal pattern (section 4.2). We organize the process in five stages:
  - *Stage one (1-way)* is in charge of identifying heap accessing statements in the source program (section 4.2.1).
  - *Stage two (1-way)* creates *dependence groups*, identifying heap accesses that may lead to a dependence (section 4.2.2).
  - *Stage three (1-way)* adds *touch pseudostatements*, instrumenting the program for the shape analysis with heap access annotations (section 4.2.3).

  - *Stage four (1-way)* performs the shape analysis proper, creating *access pairs* for heap accesses that have been performed on the same nodes (section 4.2.4).
  - Finally, *stage five (1-way)* considers the access pairs and dependence groups to identify the heap-induced data dependences found, discriminating between *anti*, *output*, and *flow* dependences (section 4.2.5).
  - For the special cases of *zero distance dependences* (section 4.2.6), which are rightfully detected by our scheme, but that do not prevent parallelism, we use the *iteration vector* within annotations for zero distance loop carried dependences, and two methods for discriminating zero distance dependences in recursive functions.

- After tackling the *1-way* traversal pattern, we target the more complicated *n-ways* traversal pattern (section 4.3). This method is organized in four stages:

  - *Stage one (n-ways)* performs recursive *function cloning*, transforming the original program into a version arranged for threaded execution, should it result parallel after the test (section 4.3.1).
  - *Stage two (n-ways)* is dedicated to adding *dynamic touch* instrumentation for the subsequent shape analysis (section 4.3.2).
  - *Stage three (n-ways)* performs the shape analysis, annotating heap accesses performed by the terminal clones (section 4.3.3).
  - *Stage four (n-ways)* considers the results of the shape analysis over the instrumented program to render the arranged program as parallel or non-parallel due to heap accesses (section 4.3.4).

- We also discuss related work in dependence analysis for programs that manipulate dynamic data structures (section 4.4).

- Finally, we present experimental results that test our technique for a variety of dynamic data structures and traversals (section 4.5). We have also characterized the extra cost that the touch or dynamic touch instrumentation poses for the analysis over performing just the shape analysis (section 4.5.2). The tests are completed with information of the impact of using touch-untouch instrumenta

5 Conclusions

5.1 Conclusions

Our research goal lies in parallelizing compilers. In particular, we are interested in uncovering unexploited parallelism in pointer-based applications. For that purpose, we have centered our efforts in the use of shape analysis to design a precise scheme of data dependence analysis. In our approach, we abstract dynamically allocated data structures in the form of shape graphs, and operate on them to annotate heap access information. We use that information to report heap-induced data dependences.

Firstly, we would like to stress the defining characteristics of our work. We have shown that it is possible to use a detailed heap analysis technique for data dependence analysis in programs that are a challenge for current parallelizing compilers. To our knowledge, no other author has used so effectively a shape analysis technique for the purpose of data dependence detection in programs that create and traverse dynamic data structures. We are able to analyze programs even when the defining characteristics of the data structure are changing amidst the traversal. When performing dependence analysis, we are able to determine the kind of data dependences for many cases. This is a very useful feature for optimizations related to parallelism and locality. Let us emphasize that all these characteristics in our analysis are not present in any related work that we are aware of.

Next, we elaborate further on our main contributions:

1. We have designed and implemented a shape analyzer based on the novel concept of *coexistent links sets*, which allow to represent possible connections between memory locations in a compact form. We have provided the necessary abstract semantics for all heap pointer statements, and the adequate scheme of data-flow equations and worklist algorithm for achieving a fixed point for the analysis. We have conducted a complete complexity study that identifies the main sources of limitation for the technique. We have provided experimental evidence that the coexistent links sets abstraction is suitable to accurately represent a variety of dynamic data structures in the form of shape graphs. More information can be found in chapter 2 of this dissertation.

2. We have designed the necessary mechanisms to support the analysis of interprocedural programs, particularly recursive algorithms, within the coexistent links sets shape analyzer. For this purpose,

we have added *recursive flow links* to the shape graphs. They codify flow information that is used by the analysis to setup and recover the appropriate context when analyzing functions. We have added support to reuse the effect of computed functions for certain cases. We have identified some shortcomings that limit the technique when analyzing recursive programs, and have devised solutions for them. Finally, we have conducted tests that provide evidence that our shape analysis approach compares favorably to related work and is able to correctly identify shape information for well-known interprocedural benchmarks. More information can be found in chapter 3 of this dissertation.

3. We have put to use the shape analysis technique based on coexistent links sets and recursive flow links for data dependence analysis in pointer-based applications. We distinguish between two different traversing patterns in recursive, dynamic data structures, namely the *1-way* traversing pattern and the *n-ways* traversing pattern. We have worked on two separate lines to apply the key idea of annotating heap accesses in nodes to provide information about dependences for both kinds of traversing patterns. In the case of *1-way* traversing patterns we are able to distinguish between anti, output and flow dependences. We have also faced the issue of discriminating between zero distance dependences and greater-than-zero distance dependences. A sound technique must be able to detect both types, but for the purposes of parallelization it is important to identify zero distance dependences, which do not hinder parallelism. Regarding the *n-ways* traversing pattern, we have adopted the approach of decomposing the structure traversal, generating a modified version of the program suitable for a subsequent parallelization scheme. We have provided experimental evidence that we can detect heap-induced data dependences in a variety of data structures and traversals for both traversal patterns. Additionally, we have studied the cost inherent to the dependence analysis versus the cost of the shape analysis per se. More information can be found in chapter 4.

5.2 Future work

It is fair to say that there is the generalized feeling within the compiler community that shape analysis has little to say for production compilers. The typical argument lies in the cost of the analysis. Shape analysis is a costly technique by definition as it strives to achieve, at compile time, very detailed information about the arrangement of memory in the heap. The kind of knowledge shape analysis can obtain is beyond the scope of other techniques, such as points-to analysis. However, it is not always clear whether that deep knowledge can be put to use effectively.

In our opinion, the main issue with shape analysis comes from the lack of information from the run time context. It is a compile-time technique, and as such, it has to adopt very conservative assumptions about the analyzed program. Therefore, as a stand-alone technique is probably insufficient for realistic compiler passes.

Despite this defining limitation, there are several directions that we can explore to improve and extend our work:

- **Improvement of internal operations.**
  The internal operations of summarization and materialization lie at the core of our shape analysis strategy. They control the focus of the analysis, whether materializing for accurate updating operations or summarizing for bounding the size of the shape graphs. These two operations must be conservative to preserve correctness of the analysis. However, it is easy to become overly conservative, rendering the analysis worthless for the purposes of data dependence detection.

  We acknowledge a key fact for this limitation: there is information that is available at the moment of summarization that cannot be recovered later during the materialization process. We plan to improve

the efficacy of the materialization, considering information present at the moment of summarization. That information includes: (i) characteristics of the *whole* data structure, and not just *local information*, like in the current approach, and (ii) the *reachability* of its elements by the different pointers. This would allows us to obtain a quicker and more precise way to materialize in shape graphs.

- **Interval analysis.**
  We advocate for the use of shape analysis as a means to analyze only certain parts of a program. In chapter 2 we showed some results that evidence that *pruning* of statements that do not affect the shape of the data structure can greatly improve the analysis performance. There is work in this direction that uses *def-use chains* to drive that pruning in an automated fashion [67]. With this approach we hope to be able to analyze bigger programs.

- **Shape information as a base tool for more sophisticated dependence tests.**
  Our approach to the problem of data dependence detection is entirely based on shape analysis and its inherent capabilities. It is based on performing abstract interpretation of all heap-directed pointer statements in the program, while annotating heap accesses in nodes of the shape graphs. However, we consider this approach just a first approximation to the problem of detecting heap-induced data dependences in programs that manipulate dynamic data structures. Since the engine of abstract interpretation is naturally of exponential complexity, this is a very costly way to uncover heap access conflicts. Alternatively, we can design a more subtle test that tries to avoid the abstract interpretation penalty whenever possible.

  For instance, we can consider shape analysis as a base tool to obtain a shape graph representation of the heap. On top of that, we can use some other technique that relies on the shape abstractions to identify conflicting heap accesses. We are already working in this direction. The key idea is to project, or map, the access paths that can potentially lead to a dependence over the shape graphs abstractions that define the data structure. The major drawback of this approach is that the data structure cannot change in the program section where the access paths are projected over the shape graphs, otherwise its deductions cannot be guaranteed to hold true for every case. Preliminary results with this approach are encouraging, leading us to believe this is the most promising field of applicability of shape analysis for realistic compiler passes.

- **Automatic generation of parallel code.**
  Let us not forget the final goal of our research: the automatic generation of parallel code. The results of our data dependence test strategies can be used by a parallelization pass that generates parallel code. We have already identified UPC (Unified Parallel C) [17], as the language of choice for this task. UPC is one the most promising languages for easy generation of parallel programs. It features parallel constructs that can exploit parallelism in most architectures today. It offers a shared memory programming model, but is able to map tasks in distributed memory architectures, all in a very amenable way for the programmer. It is as simple as sharing the required variables and adding a `upc_forall` construct to parallelize a loop, regardless of the target architecture.

  Although some problems need to be solved for the automatic parallelization of irregular applications, a field yet unexplored with UPC, we are optimistic in the use of a parallel code generation pass based on UPC for exploiting the parallelism reported by our data dependence test strategies.

Appendix A: Shape analysis algorithms

```
XNULL()
  Input: sg^1=<N^1,CLS^1>, x ∈ PTR        # A shape graph, and a pointer variable
  Output: RSSG^k                          # A shape graph in a reduced set of shape graphs

  Create List'[N]=∅
  Create List'[CLS]=∅
  Find ni ∈ N^1 s.t. ∃ pl=<x,ni> ⊂ CLS_ni
  forall cls_ni={PL_ni,SL_ni} ∈ CLS_ni,
     Create PL'_ni=PL_ni-pl                    # Remove the corresponding pl
     Create SL'_ni=SL_ni
     Create cls'_ni={PL'_ni,SL'_ni}
     List'[CLS]=List'[CLS] ∪ cls'_ni
     List'[N]=List'[N] ∪ ni
  endfor
  forall nj ∈ N^1 s.t. nj ≠ ni,
     List'[CLS]=List'[CLS] ∪ CLS_nj
     List'[N]=List'[N] ∪ nj
  endfor
  sg^k=Summarize_SG(List'[N],List'[CLS])   # Summarize compatible nodes
  RSSG^k=sg^k
  return(RSSG^k)
end
```

Figure A.1: The XNULL() function.

```
XNew()
  Input: sg^1=<N^1,CLS^1>, x ∈ PTR        # A shape graph, and a pointer variable
  Output: RSSG^k                          # A shape graph in a reduced set of shape graphs

  RSSG^1=XNULL(sg^1,x) being RSSG^1=sg^2=<N^2,CLS^2>
  Create a new node np
  forall prop ∈ PROP
     PPM_prop (np)=Update_Property(s,prop), where s is the malloc stmt.
  endfor
  Create N^k=N^2 ∪ np
  Create pl=<x,np>
  Create PL_np=pl
  Create SL_np=∅
  forall selj ∈ SEL^t (being T(x)=t)
     Create sl_att=<<np,selj,NULL>,attsl={o}>
     SL_np=SL_np ∪ sl_att
  endfor
  Create cls_np={PL_np,SL_np}
  Create CLS_np=cls_np
  Create CLS^k=CLS^2 ∪ CLS_np
  Create sg^k=<N^k,CLS^k>
  RSSG^k=sg^k
  return(RSSG^k)
end
```

Figure A.2: The `XNew()` function. Statements involved in the management of properties are shown in bold.

```
Update_Property()
  Input: s ∈ STMT, prop ∈ PROP        # A statement s::=x=new(), and a property
  Output: p_prop ∈ P_prop             # The value of the corresponding property

  If (prop==type)
     p_prop=T(x)
  If (prop==site)
     p_prop=s
  If (prop==touch ∨ prop==PC ∨ prop==PS ∨ prop==Dtouch)
     p_prop=∅
  return(p_prop)
end
```

Figure A.3: The `Update_property()` function.

```
XY()
  Input: sg^1=<N^1,CLS^1>, x, y ∈ PTR        # A shape graph, and two pointer variables
  Output: RSSG^k                             # A shape graph in a reduced set of shape graphs

  RSSG^1=XNULL(sg^1,x) being RSSG^1=sg^2=<N^2,CLS^2>
  Find ni ∈ N^2 s.t. ∃ pl1=<y,ni> ⊂ CLS_ni (being CLS_ni ⊂ CLS^2)
  # Modify CLS_ni
  Create CLS'_ni=CLS_ni
  forall cls_ni={PL_ni,SL_ni} ∈ CLS_ni,
      Create pl1'=<x,ni>
      Create PL'_ni=PL_ni ∪ pl1'
      Create SL'_ni=SL_ni
      Create cls'_ni={PL'_ni,SL'_ni}
      CLS'_ni=CLS'_ni-cls_ni ∪ cls'_ni
  endfor
  Create N^k=N^2
  Create CLS^k=CLS^2-CLS_ni ∪ CLS'_ni
  Create sg^k=<N^k,CLS^k>
  RSSG^k=sg^k
  return(RSSG^k)
end
```

Figure A.4: XY() function.

```
FreeX()
  Input: sg^1=<N^1,CLS^1>, x ∈ PTR           # A shape graph, and a pointer variable
  Output: RSSG^k                             # A shape graph in a reduced set of shape graphs

  Find ni ∈ N^1 s.t. ∃ pl=<x,ni> ⊂ CLS_ni (being CLS_ni ⊂ CLS^1)
  Create N^2=N^1-ni                          # Remove the node
  Create CLS^2=CLS^1-CLS_ni                  # Remove the corresponding cls's
  forall nj ∈ N^2,                           # Remove inconsistent sl's from other nodes
      Create CLS'_nj=CLS_nj
      forall cls_nj={PL_nj,SL_nj} ⊂ CLS_nj s.t. ∃ sl_att=<<nj,sel,ni>,attsl> ⊂ cls_nj
          Create sl'_att=<<nj,sel,NULL>,attsl={o}>
          Create SL'_nj=SL_nj-sl_att ∪ sl'_att
          Create PL'_nj=PL_nj
          Create cls'_nj={PL'_nj,SL'_nj}
          CLS'_nj=CLS'_nj-cls_nj ∪ cls'_nj
      endfor
  endfor
  N^k=N^2
  CLS^k=∪_{∀nj∈N^2} CLS'_nj
  Create sg^k=<N^k,CLS^k>
  RSSG^k=sg^k
  return(RSSG^k)
end
```

Figure A.5: FreeX() function.

```
XselY()
  Input: sg^1=<N^1,CLS^1>, x ∈ PTR, sel ∈ SEL, y ∈ PTR     # A shape graph, two pointer vars and a selector field
  Output: RSSG^k                                           # A reduced set of shape graphs in normal form

  Create RSSG^k'=∅
  RSSG^1=Split(sg^1,x,sel)
  forall sg^i=<N^i,CLS^i> ∈ RSSG^1,
     RSSG^2=XSelNULL(sg^i,x,sel)
     forall sg^j=<N^j,CLS^j> ∈ RSSG^2
        Find nk ∈ N^j s.t. ∃ pl1=<x,nk> ⊂ CLS_nk (being CLS_nk ⊂ CLS^j)
        Find np ∈ N^j s.t. (∃ pl2=<y,np> ⊂ CLS_np ∧ np ≠ NULL) (being CLS_np ⊂ CLS^j)
        # Modify CLS_nk
        Create CLS'_nk=CLS_nk
        forall cls_nk={PL_nk,SL_nk} ∈ CLS_nk,
           If (∃ sl_att1=<<nk,sel,NULL>,attsl> ⊂ cls_nk)
              Create sl'_att=<<nk,sel,np>,attsl'>
              If (nk=np)
                 attsl'={c}
              else
                 attsl'={o}
              Create  SL'_nk=SL_nk-sl_att1 ∪ sl'_att
              Create PL'_nk=PL_nk
              Create cls'_nk={PL'_nk,SL'_nk}
              CLS'_nk=CLS'_nk-cls_nk ∪ cls'_nk
        endfor
        # Modify CLS_np
        Create CLS'_np=CLS_np
        forall cls_np={PL_np,SL_np} ∈ CLS_np (being np ≠ nk),
           Create sl'_att=<<nk,sel,np>,attsl'={i}>
           Create SL'_np=SL_np ∪ sl'_att
           Create PL'_np=PL_np
           Create cls'_np={PL'_np,SL'_np}
           CLS'_np=CLS'_np-cls_np ∪ cls'_np
        endfor
        Create N^j'=N^j
        Create CLS^j'=CLS^j-CLS_nk ∪ CLS'_nk-CLS_np ∪ CLS'_np
        Create sg^j'=<N^j',CLS^j'>
        RSSG^k'=RSSG^k' ∪ sg^j'
     endfor
  endfor
  RSSG^k=Summarize_RSSG(RSSG^k')        # Summarize compatible graphs
  return(RSSG^k)
end
```

Figure A.6: `XselY()` function.

```
Summarize_SG()
  Input: List^1[N], List^1[CLS]        # A list of nodes and a list of cls's
  Output: sg^k=<N^k,CLS^k>             # A normalized shape graph

  N^k=∅
  CLS^k=∅
  forall ni ∈ List^1[N] (being CLS_ni ∧ CLS_nj ∈ List^1[CLS]),
     If (∃ nj ∈ N^k s.t. Compatible_Node(ni,nj,CLS_ni,CLS_nj)==TRUE),
        MAP(ni)=nj
     else
        N^k=N^k ∪ ni
        MAP(ni)=ni
  endfor
  forall nr ∈ N^k
     Create CLS'_nr=∅
  endfor
  forall ni ∈ List^1[N],
     nr=MAP(ni)
     forall cls_ni={PL_ni,SL_ni} ∈ List^1[CLS],
        Create PL'_nr=SL'_nr=∅
        forall pl=<x,ni> ∈ PL_ni
           Create pl'=<x,nr>
           PL'_nr=PL'_nr ∪ pl'
        endfor
        forall sl_att1=<<na,sel,nb>,attsl1> ∈ SL_ni      # Compute attsl1 ⊎ attsl2
           If (∃ sl_att2=<<nc,sel,nd>,attsl2> ∈ SL_ni,
           s.t. MAP(na)=MAP(nc) ∧ MAP(nb)=MAP(nd)),
              If (i ∈ attsl1 ∧ i ∈ attsl2)
                 attsl'=attsl1 ∪ attsl2 -i +s
              If (i ∈ (attsl1 ∨ attsl2) ∧ s ∈ (attsl1 ∨ attsl2))
                 attsl'=attsl1 ∪ attsl2 -i
              else
                 attsl'=attsl1 ∪ attsl2
              Create sl'_att=<<MAP(na),sel,MAP(nb)>,attsl'>
              SL'_nr=SL'_nr ∪ sl'_att
           else
              Create sl'_att=<<MAP(na),sel,MAP(nb)>,attsl1>
              SL'_nr=SL'_nr ∪ sl'_att
        endfor
        Create cls'_nr={PL'_nr,SL'_nr}
        CLS'_nr=CLS'_nr ∪ cls'_nr
     endfor
  endfor
  CLS^k=⋃_{∀n∈N^k} CLS'_n
  return(sg^k=<N^k,CLS^k>)
end
```

Figure A.7: `Summarize_SG()` function.

```
XSelNULL()
  Input: sg^1=<N^1,CLS^1>, x ∈ PTR, sel ∈ SEL        # A shape graph, a pointer variable and a selector field
  Output: RSSG^k                                     # A reduced set of shape graphs in normal form

  Create RSSG^k'=∅
  RSSG^1=Split(sg^1,x,sel)
  forall sg^i=<N^i,CLS^i> ∈ RSSG^1,
     sg^j=<N^j,CLS^j>=Materialize_Node(sg^i,x,sel)
     Find nk ∈ N^j s.t. ∃ pl1=<x,nk> ⊂ CLS_nk (being CLS_nk ⊂ CLS^j)
     # Modify CLS_nk
     Create CLS'_nk=CLS_nk
     forall cls_nk={PL_nk,SL_nk} ⊂ CLS_nk,
        If (∃ sl_att1=<<nk,sel,np>,attsl1> ⊂ cls_nk)
           Create sl'_att1=<<nk,sel,NULL>,attsl1'={o}>
           Create SL'_nk=SL_nk - sl_att1 ∪ sl'_att1
           Create PL'_nk=PL_nk
           Create cls'_nk={PL'_nk,SL'_nk}
           CLS'_nk=CLS'_nk - cls_nk ∪ cls'_nk
           # Modify CLS_np
           Create CLS'_np=CLS_np
           forall cls_np={PL_np,SL_np} ⊂ CLS_np (being CLS_np ⊂ CLS^j),
              If (∃ sl_att2=<<nk,sel,np>,attsl2> ⊂ cls_np)
                 Create SL'_np=SL_np - sl_att2
                 Create PL'_np=PL_np
                 Create cls'_np={PL'_np,SL'_np}
                 CLS'_np=CLS'_np - cls_np ∪ cls'_np
           endfor
     endfor
     Create N^j'=N^j
     Create CLS^j'=CLS^j - CLS_nk ∪ CLS'_nk - CLS_np ∪ CLS'_np
     Create sg^j'=<N^j',CLS^j'>
     sg^j''=Normalize_SG(sg^j')
     RSSG^k'=RSSG^k' ∪ sg^j''
  endfor
  RSSG^k=Summarize_RSSG(RSSG^k')        # Summarize compatible graphs
  return(RSSG^k)
end
```

Figure A.8: `XSelNULL()` function.

```
XYSel()
  Input: sg^1=<N^1,CLS^1>, x, y ∈ PTR, sel ∈ SEL      # A shape graph, two pointer variables and a selector field
  Output: RSSG^k                                      # A reduced set of shape graphs in normal form

  Create RSSG^k'=∅
  RSSG^1=XNULL(sg^1,x), being RSSG^1=sg^2=<N^2,CLS^2>
  RSSG^2=Split(sg^2,y,sel)
  forall sg^i=<N^i,CLS^i> ∈ RSSG^2,
     sg^j=<N^j,CLS^j>=Materialize_Node(sg^i,y,sel)
     Find nk ∈ N^j s.t. ∃ pl1=<y,nk> ⊂ CLS_nk (being CLS_nk ⊂ CLS^j)
     Find sl_att1=<<nk,sel,np>,attsl> ⊂ cls_nk
     If (np ≠ NULL)
        # Modify CLS_np
        Create CLS'_np=CLS_np
        forall cls_np={PL_np,SL_np} ∈ CLS_np,
              Create pl'=<x,np>
              Create PL'_np=PL_np ∪ pl'_np
              Create SL'_np=SL_np
              Create cls'_np={PL'_np,SL'_np}
              CLS'_np=CLS'_np-cls_np ∪ cls'_np
        endfor
        Create N^j'=N^j
        Create CLS^j'=CLS^j-CLS_np ∪ CLS'_np
     else       # Case np=NULL
        Create N^j'=N^i
        Create CLS^j'=CLS^i
     Create sg^j'=<N^j',CLS^j'>
     RSSG^k'=RSSG^k' ∪ sg^j'
  endfor
  RSSG^k=Summarize_RSSG(RSSG^k')        # Summarize compatible graphs
  return(RSSG^k)
end
```

Figure A.9: `XYSel()` function.

```
Join_SG()
  Input: sg^1=<N^1,CLS^1>, sg^2=<N^2,CLS^2>      # Two shape graphs
  Output: sg^k=<N^k,CLS^k>                       # A normalized shape graph

  N^k=∅
  CLS^k=∅
  # Join nodes
  forall ni ∈ N^1,
     If (∃ nj ∈ N^2 s.t. Compatible_Node(ni,nj,CLS_ni,CLS_nj)==TRUE),
        # Create a summary node ns
        forall prop ∈ PROP
           PPM_prop(ns)=Join_Property(ni,nj,prop)
        endfor
        N^k=N^k ∪ ns
        MAP(ni)=MAP(nj)=ns
     else
        N^k=N^k ∪ ni
        MAP(ni)=ni
  endfor
  forall nj ∈ N^2,
     If (∄ ni ∈ N^1 s.t. Compatible_Node(nj,ni,CLS_nj,CLS_ni)==TRUE),
        N^k=N^k ∪ nj
        MAP(nj)=nj
  endfor
  # Join cls's
  forall nr ∈ N^k
     Create CLS'_nr=∅
  endfor
  forall ni ∈ N^1 ∨ N^2,
     nr=MAP(ni)
     forall cls_ni={PL_ni,SL_ni} ∈ CLS_ni,
        Create PL'_nr=SL'_nr=∅
        forall pl=<x,ni> ∈ PL_ni
           Create pl'=<x,nr>
           PL'_nr=PL'_nr ∪ pl'
        endfor
        forall sl_att=<<na,sel,nb>,attsl> ∈ SL_ni
           Create sl'_att=<<MAP(na),sel,MAP(nb)>,attsl>
           SL'_nr=SL'_nr ∪ sl'_att
        endfor
        Create cls'_nr={PL'_nr,SL'_nr}
        CLS'_nr=CLS'_nr ∪ cls'_nr
     endfor
  endfor
  CLS^k=⋃_{∀n∈N^k} CLS'_n
  return(sg^k=<N^k,CLS^k>)
end
```

Figure A.10: The `Join_SG()` function. Statements involved in the management of properties are shown in bold.

```
Join_Property()
  Input: n1, n2, prop ∈ PROP        # Two compatible nodes and a property
  Output: p_prop ∈ P_prop           # The value of the corresponding property

  If (prop==type ∨ prop==site ∨ prop==touch ∨ prop==PC ∨ prop==Dtouch)
      p_prop=PPM_prop(n1)   # PPM_prop(n1)==PPM_prop(n2)
  If(prop==PS)
      p_prop=PPM_prop(n1) ∪ PPM_prop(n2)
  return(p_prop)
end
```

Figure A.11: The `Join_Property()` function.

```
Split()
  Input: sg^1=<N^1,CLS^1>, p ∈ PTR, sel ∈ SEL      # A shape graph, a pointer variable and a selector field
  Output: RSSG^k                                   # A set of shape graphs

  RSSG^k=∅
  Find ni ∈ N^1 s.t. ∃ pl=<p,ni> ⊂ CLS_ni
  # Split a graph for each cls_ni ∈ CLS_ni
  forall cls_ni ∈ CLS_ni,
      If (∃ sl_att=<<ni,sel,na>,attsl={o|c}> ∈ cls_ni ∧ na ≠ NULL)
          Create CLS^k'=CLS^1-CLS_ni ∪ cls_ni
          Create N^k'=N^1
          Create sg^k'=<N^k',CLS^k'>
          RSSG^k=RSSG^k ⋃ Normalize_SG(sg^k')
  endfor
  If (∀ ni ∈ N^1, ∄ pl=<p,ni> ⊂ CLS_ni)
      RSSG^k=sg^1
  return(RSSG^k)
end
```

Figure A.12: `Split()` function.

```
Normalize_SG()
  Input: sg^1=<N^1,CLS^1>        # A shape graph
  Output: sg^k=<N^k,CLS^k>       # A normalized shape graph

  Create N_0^k'=N^1
  Create CLS_0^k'=CLS^1
  Create sg_0^k'=sg^1
  i=0
  repeat                  # Iterate until N_i^k' and CLS_i^k' do not change anymore
     Find Nu={nu ∈ N_i^k' s.t. Unreachable(nu,sg_i^k')==TRUE}
     Find Ne={ne ∈ N_i^k' s.t. CLS_ne=∅}
     # Remove unreachable and empty nodes
     N_{i+1}^k'=N_i^k'-Nu-Ne
     # cls's from/to unreachable and empty nodes
     Find cls_nb s.t. ∃ sl_att=<<nf,sel,ng>,attsl> ⊂ cls_nb,
        with (nf ∈ Nu ∪ Ne) ∨ (ng ∈ Nu ∪ Ne)
     # cls's with incoherent selector links
     Find cls_nc s.t. ∃ sl_att1=<<nc,sel,nm>,attsl1> ⊂ cls_nc ∧
        ∧ ∄ sl_att2=<<nc,sel,nm>,attsl2> ⊂ cls_nm
     Find cls_nd s.t. ∃ sl_att3=<<nm,sel,nd>,attsl3> ⊂ cls_nd ∧
        ∧ ∄ sl_att4=<<nm,sel,nd>,attsl4> ⊂ cls_nm
     CLS_{i+1}^k'=CLS_i^k'-⋃_{∀nu∈Nu} CLS_nu-⋃_{∀ne∈Ne} CLS_ne -{cls_nb}-{cls_nc}-{cls_nd}
     sg_{i+1}^k'=<N_{i+1}^k',CLS_{i+1}^k'>
     Increment i
  until (N_i^k'=N_{i-1}^k' ∧ CLS_i^k'=CLS_{i-1}^k')      # Fixed point condition
  N^k=N_i^k'
  CLS^k=CLS_i^k'
  return(sg^k=<N^k,CLS^k>)
end
```

Figure A.13: `Normalize_SG()` function.

```
Materialize_Node() (1/3)
  Input: sg^1=<N^1,CLS^1>, p ∈ PTR, sel ∈ SEL      # A shape graph, a pointer variable and a selector field
  Output: sg^k=<N^k,CLS^k>                          # A shape graph

  Find ni ∈ N^1 s.t. ∃ pl=<p,ni> ⊂ CLS_ni
  Find nj ∈ N^1 s.t. ∃ sl_att1=<<ni,sel,nj>,attsl1> ⊂ CLS_ni
  # Create a new node nm
  forall prop ∈ PROP
    PPM_prop(nm)=PPM_prop(nj)
  endfor
  Create N^k'=N^1 ∪ nm
  forall n ∈ N^k'
    Create CLS'_n=∅
  endfor
  Find {cls_nj ⊂ CLS_nj s.t. ∃ sl_att2=<<ni,sel,nj>,attsl2> ⊂ cls_nj }::={cls_nj s.t. cond. A}
  forall cls_nj={PL_nj,SL_nj} s.t. cond. A,        # Create CLS'_nm
    Create PL'_nm=SL'_nm=∅
    forall pl=<x,nj> ∈ PL_nj
      Create pl'=<x,nm>
      PL'_nm=PL'_nm ∪ pl'
    endfor
    forall sl_att=<<na,sel2,nb>,attsl> ∈ SL_nj
      If (attsl={c})
        Create sl'_att=<<nm,sel2,nm>,attsl>
        SL'_nm=SL'_nm ∪ sl'_att
      If (attsl={o})
        Create sl'_att=<<nm,sel2,nb>,attsl>
        SL'_nm=SL'_nm ∪ sl'_att
      If (attsl={i} ∨ {s})
        Create sl'_att=<<na,sel2,nm>,attsl>
        SL'_nm=SL'_nm ∪ sl'_att
      else          # Cases {i,o},{s,o},{i,c},{s,c}
        Create sl'_att1=<<na,sel2,nm>,attsl-(o|c)>
        Create sl'_att2=<<nm,sel2,nb>,attsl-(i|s)>
        SL'_nm=SL'_nm ∪ sl'_att1 ∪ sl'_att2
    endfor
    Create cls'_nm={PL'_nm,SL'_nm}
    CLS'_nm=CLS'_nm ∪ cls'_nm
  endfor
  ⋮
```

Figure A.14: Part one of three of the `Materialize_Node()` function. Statements involved in the management of properties are shown in bold.

`Materialize_Node()` (2/3)

```
⋮
Create CLS'nj=CLSnj-{clsnj s.t. cond. A}        # Create CLS'nj
forall clsnj={PLnj,SLnj} ∈ CLSnj s.t. ¬ cond. A,
  Create T1=T2=T2'=T3=∅
  Find {slatt6=<<nj,sel2,nj>,attsl6> ⊂ clsnj}::={clsnj s.t. cond. E}
  forall slatt ⊂ clsnj s.t. ¬ cond. E
    T1=T1 ∪ slatt
  endfor
  forall slatt6 ⊂ clsnj s.t. cond. E
    If (attsl6 ≠ {c})
      If (c ∈ attsl6)
        T2=T2 ∪ <<nj,sel2,nj>,attsl6-c>
        T3=T3 ∪ <<nj,sel2,nj>,attsl6-(i|s)>
      else
        If ({i|s,o} ⊂ attsl6)
          T2=T2 ∪ <<nj,sel2,nj>,attsl6-(i|s)> ∪ <<nj,sel2,nj>,attsl6-o>
        else
          T2=T2 ∪ <<nj,sel2,nj>,attsl6>
    forall slatt=<<nj,sel2,nj>,attsl> ∈ T2
      If ((i|s) ∈ attsl)
        Create sl'att=<<nm,sel2,nj>,attsl>
      else
        Create sl'att=<<nj,sel2,nm>,attsl>
      T2'=T2' ∪ sl'att
    endfor
  endfor
    Create PL'nj=PLnj
    SL'nj=T1 ∪ T3
    for P=(00...0):(11...1)        # P is a binary vector of |T2| size
      SL'nj=SL'nj ∪ {P · T2 + ¬ P · T2' }
      Create cls'nj={PL'nj,SL'nj}
      CLS'nj=CLS'nj ∪ cls'nj
    endfor
endfor
⋮
```

Figure A.15: Part two of three of the `Materialize_Node()` function.

`Materialize_Node()` (3/3)

```
  ⋮
  forall nk ∈ N^1 s.t. nk ≠ nj,        # Create CLS'_nk being nk ≠ nj
    forall cls_nk={PL_nk,SL_nk} ∈ CLS_nk,
      Create T1=T2=T2'=T3=∅
      forall sl_att3=<<nk,sel2,nj>,attsl3> ⊂ cls_nk::=cls_nk s.t. cond. B
        Create sl'_att3=<<nk,sel2,nm>,attsl3>
        T2=T2 ∪ sl_att3
        T2'=T2' ∪ sl'_att3
      endfor
      forall sl_att4=<<nj,sel2,nk>,attsl4> ⊂ cls_nk ∧ s ∉ attsl4::=cls_nk s.t. cond. C
        Create sl'_att4=<<nm,sel2,nk>,attsl4>
        T2=T2 ∪ sl_att4
        T2'=T2' ∪ sl'_att4
      endfor
      forall sl_att5=<<nj,sel2,nk>,attsl5> ⊂ cls_nk ∧ s ∈ attsl5::=cls_nk s.t. cond. D
        Create sl'_att5=<<nm,sel2,nk>,attsl5-s+i>
        T3=T3 ∪ sl_att5 ∪ sl'_att5
      endfor
      forall sl_att ⊂ cls_nk s.t. (¬ cond. B ∧ ¬ cond. C ∧ ¬ cond. D)
        T1=T1 ∪ sl_att
      endfor
      Create PL'_nk=PL_nk
      SL'_nk=T1 ∪ T3
      for P=(00...0):(11...1)        # P is a binary vector of |T2| size
        SL'_nk=SL'_nk ∪ {P · T2 + ¬ P · T2'}
        Create cls'_nk={PL'_nk,SL'_nk}
        CLS'_nk=CLS'_nk ∪ cls'_nk
      endfor
    endfor
  endfor
  CLS^k'=⋃_{∀n∈N^k'} CLS'_n
  sg^k'=<N^k',CLS^k'>
  sg^k=Normalize_SG(sg^k')
  return(sg^k)
end
```

Figure A.16: Part three of three of the `Materialize_Node()` function.

```
Force() (1/2)
  Input: sg^1=<N^1,CLS^1>, test_condition      # A shape graph, and a test condition
  Output: sg^k=<N^k,CLS^k>                     # A shape graph

  Case (test_condition)
    test_condition==(x==null)
      If (∃ ni ∈ N^1 s.t. ∃ pl=<x,ni> ⊂ CLS_ni)
        sg^k=∅
      else
        sg^k=sg^1
      break
    test_condition==(x!=null)
      If (∃ ni ∈ N^1 s.t. ∃ pl=<x,ni> ⊂ CLS_ni)
        sg^k=sg^1
      else
        sg^k=∅
      break
    test_condition==(x->sel==null)
      Find ni ∈ N^1 s.t. ∃ pl=<x,ni> ⊂ CLS_ni
      Create CLS'_ni=∅
      forall cls_ni ∈ CLS_ni
        If (∃ sl_att=<<ni,sel,nj>,attsl> ⊂ cls_ni s.t. nj==NULL)
          CLS'_ni=CLS'_ni ∪ cls_ni
      endfor
      Create CLS^k'=CLS^1-CLS_ni ∪ CLS'_ni
      Create N^k'=N^1
      sg^k'=<N^k',CLS^k'>
      sg^k=Normalize_SG(sg^k')
      break
    ⋮
```

Figure A.17: Part one of two of the `Force()` function.

```
Force() (2/2)
   ⋮
   test_condition==(x->sel!=null)
     Find ni ∈ N^1 s.t. ∃ pl=<x,ni> ⊂ CLS_ni
     Create CLS'_ni=∅
     forall cls_ni ∈ CLS_ni,
       If (∃ sl_att=<<ni,sel,nj>,attsl> ⊂ cls_ni s.t. nj ≠ NULL)
         CLS'_ni=CLS'_ni ∪ cls_ni
     endfor
     Create CLS^k'=CLS^1-CLS_ni ∪ CLS'_ni
     Create N^k'=N^1
     sg^k'=<N^k',CLS^k'>
     sg^k=Normalize_SG(sg^k')
     break
   test_condition==(x->sel==y)
     Find ni ∈ N^1 s.t. ∃ pl=<x,ni> ⊂ CLS_ni
     Find nj ∈ N^1 s.t. ∃ pl=<y,nj> ⊂ CLS_nj
     Create CLS'_ni=∅
     forall cls_ni ∈ CLS_ni,
       If (∃ sl_att=<<ni,sel,nj>,attsl> ⊂ cls_ni)
         CLS'_ni=CLS'_ni ∪ cls_ni
     endfor
     Create CLS^k'=CLS^1-CLS_ni ∪ CLS'_ni
     Create N^k'=N^1
     sg^k'=<N^k',CLS^k'>
     sg^k=Normalize_SG(sg^k')
     break
   test_condition==(x->sel!=y)
     Find ni ∈ N^1 s.t. ∃ pl=<x,ni> ⊂ CLS_ni
     Find nj ∈ N^1 s.t. ∃ pl=<y,nj> ⊂ CLS_nj
     Create CLS'_ni=∅
     forall cls_ni ∈ CLS_ni,
       If (∃ sl_att=<<ni,sel,nk>,attsl> ⊂ cls_ni s.t. nk ≠ nj)
         CLS'_ni=CLS'_ni ∪ cls_ni
     endfor
     Create CLS^k'=CLS^1-CLS_ni ∪ CLS'_ni
     Create N^k'=N^1
     sg^k'=<N^k',CLS^k'>
     sg^k=Normalize_SG(sg^k')
     break
 return(sg^k)
end
```

Figure A.18: Part two of two of the `Force()` function.

Appendix B: Shape graph summaries for the `reverse()` function

Here we present all the summary shape graphs generated for the analysis of the recursive function `reverse()`, which reverses a singly-linked list. This recursive function was used in chapter 3 as running example to illustrate the extensions for interprocedural analysis. We reproduce the function again in Fig. B.1 for your convenience.

```
   struct node * reverse(struct node *x){
       struct node *y,*z;
 4:    z=x->nxt;
       if(z!=NULL){
 5:       #pragma SAP.force(z!=NULL)
 6:       y=reverse(z);
 7:       #pragma SAP.force(x!=NULL)
 8:       x->nxt=NULL;
 9:       z->nxt=x;
       }else{
10:       #pragma SAP.force(z==NULL)
11:       y=x;
       }
12:    return y;
   }
```

Figure B.1: The `reverse()` recursive function to reverse a singly-linked list.

As a result of the completion of the analysis of `reverse()`, the analysis generates a shape graph set composed of nine shape graphs. These graphs represent every possible heap state that may be found at the return point of the function, along all stages of recursive analysis. They are shown in Fig. B.2.

These shape graphs will be considered as the overall effect of the `reverse()` function. However, not all graphs are eligible to represent the effect of the non-recursive call of `reverse()`, and therefore not all of them will be converted by the $\mathtt{RTC}_{\mathtt{nrec}}$ rule (see section 3.2.3.4). In particular, only those shape graphs were the recursive flow pointers used are not assigned (i.e., they point to `NULL`), can represent heap states that result from the analysis of the first, non-recursive call to the recursive function. Additionally, the summaries for the recursive analysis of `reverse()` are stored by the tabulation scheme for reuse in

the case of analyzing an equivalent data structure (see section 3.3). Next, let us review all summary shape graphs generated for `reverse()`. To analyze this function we used the *previous call* (PC) and *paired selectors* (PS) properties (see sections 3.4.1 and 3.4.3).

Shape graph sg^1 in Fig. B.2 shows the first output summary shape graph for `reverse()`. It considers the case of a list of length one, and the end of the first recursive call. This shape graph can be used for the RTC_{nrec} rule as capturing the effect of a non-recursive call to `reverse()` with a list of one element.

Shape graph sg^2 shows a list two elements long, at the end of the second recursive call. This represents the memory state of taking the `else` branch in the last recursive call, where there is no next element from the element pointed to by pointer `x`. Pointer `z` points to `NULL`, and therefore there is no pointer link for it. Note that the node pointed to by x_{rfptr} has the `x` value for the previous call (PC) property. It abstracts the element that was pointed to by pointer `x` in the immediately previous recursive call.

Shape graph sg^3 shows the same list of two elements of the previous figure, but one recursive call back in the stack of recursive calls. Such call is the first, non-recursive call to `reverse()`, where the control flow of the program has taken the `if` branch, and thus, `z` is assigned.

Shape graph sg^4 captures a memory state where a list of length equal or greater than three elements reaches the last recursive call. Pointer `z` is not assigned, we have taken the `else` branch, and there are no more elements to traverse forward in the list. Note that `n1` and `n2` have the value of `x` for the previous call (PC) property. In particular, `n2` abstracts the element pointed to by pointer `x` in the previous recursive call, while `n1` abstracts all elements previous to that.

Note that, in addition to the previous call property, nodes `n1` and `n2` feature some values for the paired selectors (PS) property. Node `n2` is related to `n1` with the relation $<nxt_i, x_{rfsel\_o}>$, established by links `sl2=<n1,nxt,n2>` and $rfsl3=<n2, x_{rfsel}, n1>$. Naturally, the `nxt` selector is *incoming* to `n2`, and the x_{rfsel} is *outgoing* from `n2`. For `n1`, the relations of the paired selector property involve `sl1`, `sl2`, `rfsl1`, `rfsl2`, and `rfsl3`, with values $<x_{rfsel\_i}, nxt_o>$ and $<nxt_i, x_{rfsel\_o}>$. These values of the PS property indicate that, within the memory locations abstracted by `n1`: (i) if a recursive flow selector is incoming by x_{rfsel} from one location, then selector `nxt` is also outgoing to that *same* location, or (ii) if a selector from one location is incoming by `nxt`, then a recursive flow selector must be outgoing by x_{rfsel} to the *same* location.

Shape graph sg^5 shows a list of three elements returning from the second recursive call. It shows a list half-reversed, where the element pointed to by `x` (represented by `n2`) is reached simultaneously from the previous and the next element in the traversal order (represented by nodes `n1` and `n3`, respectively).

Shape graph sg^6 shows a list of three or more elements, returning from the first recursive call. This is a suitable graph to capture the effect of a non-recursive call to `reverse()` over an arbitrarily long singly-linked list. It is the shape graph used for the example of the RTC_{nrec} rule in section 3.2.3.4.

Shape graph sg^7 shows a list of three or more elements, returning from the penultimate recursive call. We have followed the `if` branch. The last element, pointed to by `z`, is made to point to the penultimate element, pointed to by pointer `x`, thus starting the reversal of the list. The nodes `n1` and `n2` represent the first part of the list, traversed during the previous recursive calls.

Shape graph sg^8 shows a four-element list, returning from the second recursive call. There is only one previous recursive call, as there are no more nodes reachable from `n1` through x_{rfsel}. The last part of the list (nodes `n3` and `n4`) has already been reversed.

Shape graph sg^9 shows a list of more than three elements, returning from a middle recursive call. A part of two or more elements has already been reversed (nodes `n4`, `n5`, and `n6`), while another part of two or more elements (nodes `n1`, `n2`, and `n3`) is yet to be reversed, upon returning to the previous recursive calls.

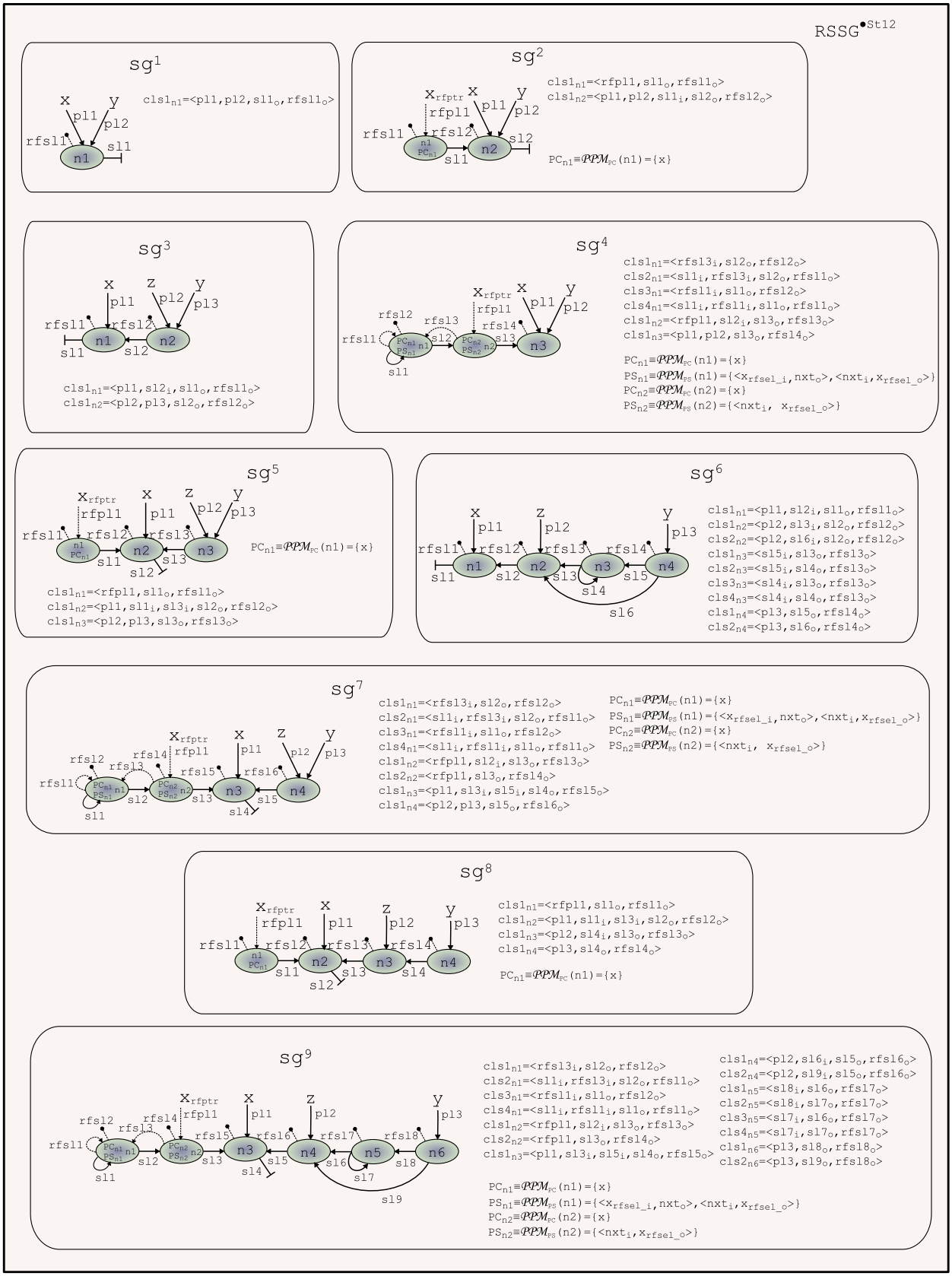

Figure B.2: Output summaries for the recursive analysis of `reverse()`.

Appendix C: Resumen de la tesis doctoral en castellano

C.1 Introducción general

La comunidad científica está de acuerdo en que hemos alcanzado la era multicore: procesadores de 2 y 4 núcleos son ya comunes en ordenadores de sobremesa, y fabricantes como Intel planean procesadores de 80 núcleos. Además, las arquitecturas multiprocesador abundan en las medianas empresas, centros de investigación y organizaciones estatales, conforme se convierten en la tendencia principal en arquitectura de computadores.

Las arquitecturas de un solo núcleo ya no pueden sostener los incrementos de rendimiento de la ley de Moore [14]. Conforme nos acercamos a los límites de las arquitecturas monoprocesador, el incremento en consumo (y su coste de refrigeración asociado) domina a unos menguantes incrementos en rendimiento. Parece haber un modo mejor: usar multiprocesadores. Ejemplos de esta tendencia son la arquitectura Centrino Duo de Intel o el sistema Roadrunner de IBM, el supercomputador más potente en la actualidad[1]. Este último es el primer supercomputador híbrido de la historia, conectando 6.562 chips AMD Opteron de doble núcleo, a la vez que 12.240 chips Cell. Es decir, no solo las arquitecturas multiprocesador se están convirtiendo en comunes, sino que las arquitecturas heterogéneas empiezan a aparecer en sistemas de alta gama. Más que una moda de nuestro tiempo, los multiprocesadores están aquí para quedarse.

La clave ahora es cómo obtener un buen rendimiento software a bajo coste para que podamos explotar todo el hardware disponible. Actualmente, la forma más común de obtener programas para estas arquitecturas multiprocesador consiste en escribir, explícitamente, algoritmos paralelos que se basan en hilos (threads) o librerías de paso de mensajes. No obstante, hay una grave limitación con este enfoque, y se trata del elevado coste de desarrollo. Aunque hay un creciente número de lenguajes y librerías que intentan popularizar la programación paralela (por ejemplo, [15], [16] ó [17]), los programadores paralelos expertos son profesionales muy demandados, más por cuanto la variedad de arquitecturas y paradigmas de programación paralela abundan.

Estamos presenciando un cambio de tendencias: de la *Computación de Alto Rendimiento* (HPC, *High Performance Computing*), que se ocupa principalmente de reducir tiempos de ejecución por medio de cualquier mecanismo, hacia la *Computación de Alta Productividad* (HPC, *High Productivity Computing*), que busca obtener buenos incrementos en rendimiento pero a un coste razonable.

[1] Según la lista Top500 (www.top500.org) en Junio de 2008.

Durante años, un compilador paralelizador versátil y potente ha sido la quimera de la comunidad científica en materia de compilación. El objetivo es ser capaz de identificar y explotar paralelismo en programas secuenciales de forma automática, mediante un proceso completamente controlado por el compilador. Este enfoque ha dado buenos resultados para aplicaciones regulares, sobre todo en Fortran [18], pero las aplicaciones irregulares aún suponen desafíos significativos.

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. Esta limitación se debe principalmente a su incapacidad para extraer la información necesaria del programa fuente. En general, son incapaces de localizar las oportunidades para explotar paralelismo y localidad de las estructuras de datos dinámicas. Para ello, es absolutamente necesario disponer de una descripción precisa de qué localizaciones de memoria, de entre las disponibles en el *heap*, son accedidas y de qué manera. Solo así lograremos avanzar en la paralelización automática de programas irregulares.

C.2 Motivación

El problema que queremos resolver es la paralelización automática de códigos basados en estructuras dinámicas recursivas almacenadas en el *heap*. Se trata de un problema aún no resuelto y desafiante. Encontrar su solución tendría un gran impacto ya que las estructuras de datos dinámicas son ampliamente utilizadas en muchos códigos irregulares y las arquitecturas multiprocesador/multihilo son muy comunes actualmente.

Las estructuras de datos dinámicas son aquellas que se reservan en tiempo de ejecución y son accedidas por punteros dirigidos 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. Estas estructuras son comúnmente utilizadas 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.

El problema de calcular los alias debidos a punteros debe resolverse para que los compiladores puedan desambiguar las referencias de memoria. Un paso básico en el proceso de paralelización automática es la detección de bucles paralelos o llamadas paralelas a funciones utilizando un test de dependencias. Un test de dependencias así requiere información acerca de las propiedades de las estructuras de datos recorridas en bucles o en cuerpos de función. Estamos convencidos de la necesidad de una descripción del *heap* muy precisa, para el propósito del análisis de dependencias en el contexto de aplicaciones que tratan con estructuras de datos dinámicas.

Hay todo un cuerpo de trabajo relacionado con el análisis *points-to*, como [19] ó [20]. Su principal enfoque es hacia la detección de relaciones de alias entre punteros. Por ejemplo, Salcianu [21] construye grafos *points-to* que representan relaciones entre elementos del *heap* y los punteros, incluso para partes incompletas de un programa. La aplicación de su análisis se reduce a algunos clientes simples de lenguajes orientados a objetos, como el descubrimiento de métodos que no modifican los objetos globales (análisis de pureza), o la detección de objetos que son capturados en un método y pueden alojarse, por tanto, en el *stack* (análisis de alojamiento en el *stack*).

En nuestro enfoque, consideramos el *análisis de forma* como la técnica base para conseguir una caracterización de estructuras de datos en el *heap*. Al contrario que las técnicas de análisis *points-to*, que se ocupan principalmente de los conjuntos de *may-alias* y *must-alias*, el análisis de forma se ocupa de la *forma* de la estructura de datos. Esto permite una caracterización más precisa de las estructura de datos en el *heap*. Esta

precisión es necesaria para análisis clientes más complejos, como el análisis de dependencias de datos. Con información de forma, es posible identificar accesos en conflicto en recorridos de elementos en el *heap* que, de otro modo, no serían diferenciados por un análisis de tipo *points-to*.

Vislumbramos un *sistema de análisis del heap*, basado en análisis de forma como su elemento clave, cuyo propósito es obtener información topológica y temporal acerca de estructuras de datos recursivas. Un sistema así estaría orientado hacia la detección de bucles paralelos y llamadas paralelas a función, con la intención de generar una versión paralelizada con hilos de un código secuencial. Este sistema sería muy valioso en el actual escenario repleto de multiprocesadores domésticos, que están llegando rápidamente al usuario medio en la forma de sistemas multinúcleo (*multicore*).

Presentamos una primera aproximación a este *sistema de análisis del heap* en la Fig. C.1. Su función es derivar información de aplicaciones secuenciales basadas en punteros de forma estática. Este sistema debería asociarse con un bloque de transformación de código que haga un uso adecuado de esa información y arroje una versión optimizada del programa original. Para nuestros propósitos, tal optimización está relacionada con el paralelismo automático para obtener aceleración del tiempo de ejecución.

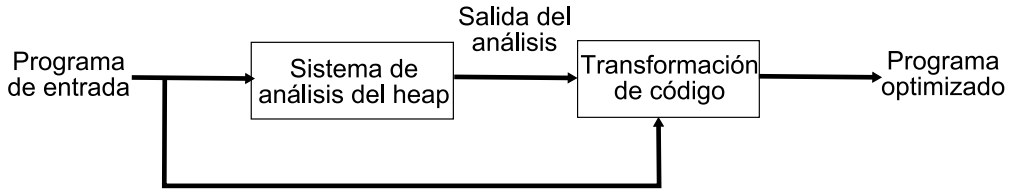

Figure C.1: Sistema de análisis del *heap* para proporcionar información a un bloque de transformación de código.

C.3 Análisis de forma para el análisis de dependencias

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 nuestro enfoque al análisis de forma, usamos 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. Como consecuencia, es habitualmente mucho más costosa que otros enfoques al análisis del *heap*, como el análisis *points-to*.

A continuación presentamos una idea intuitiva acerca del modo en que un análisis de forma basado en grafos puede usarse para encontrar conflictos en los accesos en un típico bucle de recorrido por punteros. La idea principal en nuestro esquema de detección de dependencias de datos es la interpretación abstracta de las sentencias del bucle analizado, abstrayendo las localizaciones del *heap* accedidas con nodos de grafos de forma y anotando estos nodos con información de lectura/escritura.

El código en la Fig. C.2 crea una lista simplemente enlazada y luego la recorre, copiando el campo `data` del elemento apuntado por el puntero `q`, al elemento apuntado por el puntero `p`. El efecto global de este algoritmo es desplazar los valores en la lista una posición hacia el comienzo de la lista. Hay una posible dependencia de datos entre `S3:val=q->data`, que lee el campo `data`, y `S4:p->data=val`, que escribe en él.

Nuestro test ejecuta simbólicamente el código abstrayendo las estructuras de datos en grafos de forma. Por ejemplo, sg^1 es el grafo de forma (`sg`, *shape graph*) que abstrae la lista creada en la sentencia `S1`. Usando la interpretación abstracta [22], la *semántica abstracta* de cada sentencia actualiza el grafo de forma resultante de la sentencia previa. En este proceso, las localizaciones de memoria que se leen y/o escriben se anotan adecuadamente. En este ejemplo, el acceso de lectura de la sentencia `S3` es anotada como `RS3` en los grafos de forma, mientras que el acceso de escritura de la sentencia `S4` es anotada como `WS4`. La segunda ejecución simbólica de la sentencia `S4:p->data=val` produce el grafo de forma sg^8. Dentro de este grafo de forma podemos detectar que una localización de memoria ha sido leída en una iteración y escrita en la siguiente, causando una dependencia acarreada por lazo debida a un acceso WAR (*write-after-read*, escritura tras lectura).

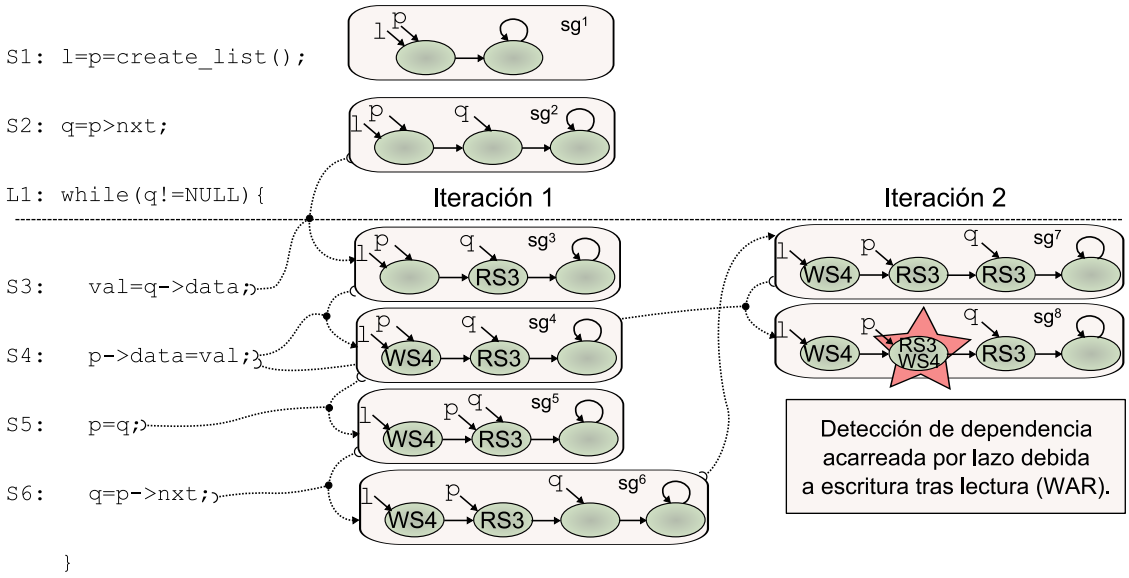

Figure C.2: El uso del análisis de forma para la detección de dependencias de datos en el *heap*.

C.3.1 El análisis de forma dentro del *sistema de análisis del heap*

A continuación, expandimos el concepto de *sistema de análisis del heap* introducido anteriormente. La Fig. C.3 presenta un esquema de diferentes módulos interactuando dentro del *sistema de análisis del heap*.

En primer lugar, el programa de entrada entra en el módulo SAP. Sus siglas significan *Preprocesador para el Análisis de Forma* (*Shape Analysis Preprocessor*). Como su nombre sugiere, este módulo es responsable de realizar las tareas de preproceso sobre el programa requeridas para su análisis de forma. El resultado de este módulo es el conjunto de sentencias de punteros que actúa sobre *heap*, y el flujo de información que gobierna el modo en que esas sentencias se ejecutan en el programa.

Esa información es la entrada para la herramienta del *analizador de forma*, dentro del *paquete del analizador de forma*. También dentro de este paquete encontramos la *herramienta de visualización* [23], que se usa para visualizar los grafos de forma obtenidos y ayudar a depurar la técnica.

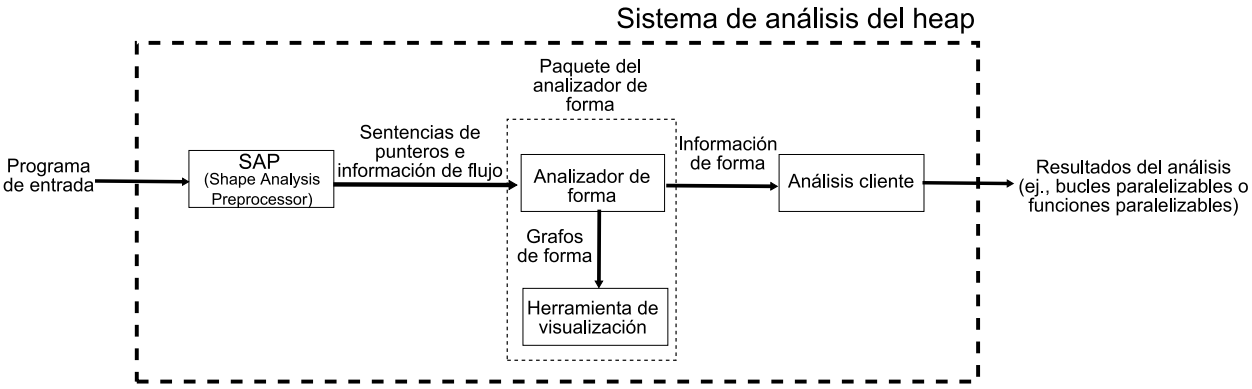

Figure C.3: Preprocesado del programa, análisis de forma y análisis cliente dentro del *sistema de análisis del heap*.

Como resultado de la ejecución del analizador de forma, obtendremos una *caracterización de forma* de estructuras de datos dinámicas. Esa información puede ser usada por *análisis clientes*. Por ejemplo, un test de dependencias de datos podría ser ese cliente: puede considerar información de forma combinada con información de accesos al *heap* para detectar dependencias en aplicaciones basadas en punteros. Los resultados del cliente son ofrecidos como salida del sistema. Por ejemplo, un cliente de detección de dependencias de datos podría informar de bucles paralelizables o funciones paralelizables a un sistema de paralelización externo al *sistema de análisis del heap*.

C.4 Análisis de forma intraprocedural

Nuestro acercamiento al problema del 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, etcétera.

Nuestro algoritmo de análisis de forma está diseñado como un análisis iterativo de tipo *data-flow*. Las sentencias en el programa se ejecutan simbólicamente de forma iterativa, según las ramas y bucles del programa para la parte intraprocedural. En este proceso los grafos de forma son transformados según la semántica abstracta de las sentencias analizadas. Este proceso continúa hasta que los grafos de forma alcanzan un estado estacionario, donde continuar la interpretación abstracta no produce nueva información. Este estado se conoce como el *punto fijo* del algoritmo.

Estrechamente ligado a la noción de punto fijo del algoritmo, está la operación de *sumarización*. La sumarización es el proceso que mezcla nodos en grafos de forma cuando se estima que son *suficientemente similares*. La similitud 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.

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. C.4. 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.

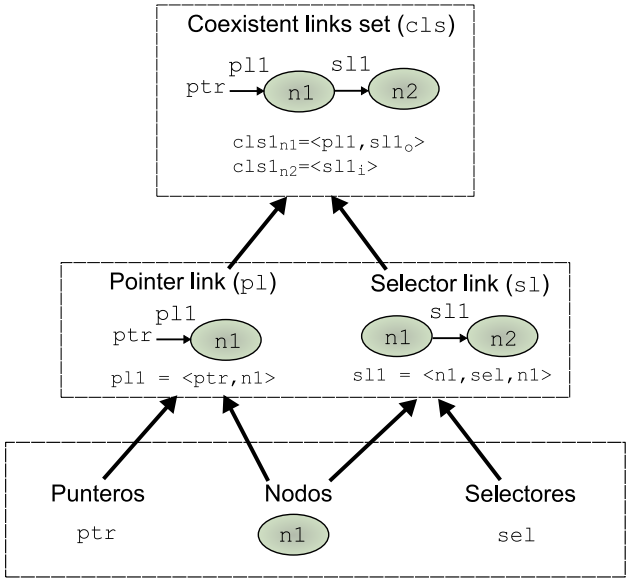

Figure C.4: Vista jerárquica de los elementos de un grafo de forma.

C.5 Análisis de forma interprocedural

El soporte para los programas interprocedurales en el análisis de forma es aún un desafío, especialmente en presencia de funciones recursivas. Sin embargo, el recorrido de estructuras de datos recursivas con algoritmos recursivos es muy común, ya que algunas estructuras de datos, como los árboles, se expresan en una forma que hace natural el hecho de recorrerlas de forma recursiva. El principal problema que afrontamos cuando analizamos funciones recursivas es el problema de registrar el estado de los parámetros puntero en los cambios de contexto. Para cambios de contexto no recursivos, es suficiente conocer la relación entre los punteros que actúan como parámetros reales o formales. En tal caso, el cambio de contexto puede trasladarse fácilmente al dominio de los grafos de forma.

Sin embargo, cuando manejamos parámetros formales puntero en funciones recursivas, no es tan simple: la misma variable puntero debe ser seguida o registrada a lo largo de la secuencia de longitud indefinida de llamadas recursivas. El nombre del puntero es el mismo, pero dependiendo de la llamada, puede apuntar a localizaciones distintas. Estas localizaciones deben registrarse para saber donde apuntaba un puntero cuando volvemos de una llamada recursiva.

Por tanto, para mantener la pista de un parámetro formal puntero necesitamos cambiar su *esquema de nombres*: no solo necesitamos conocer su nombre, sino también alguna información que lo relacione a la llamada a la que pertenece. Lo mismo se necesita para los punteros definidos en el cuerpo de la función recursiva, los *punteros locales*. Estos punteros se redefinen en cada llamada, es decir, su ámbito pertenece solo a una cierta llamada recursiva, y deben ser correctamente asignados al volver de las llamadas recursivas.

En tiempo de ejecución, esto se realiza manteniendo diferentes entradas en el *Registro de Activación* o *Activation Record Stack* (ARS). Entre otra información, el ARS mantiene el estado de los parámetros reales punteros y los punteros locales antes de una llamada. De este modo, cuando volvemos de la llamada estos

punteros pueden ser correctamente reasignados. Hay que recordar que una técnica de análisis en tiempo de compilación no puede conocer el número de veces que una función recursiva se llamará. Aún así el análisis debe encontrar un punto fijo, incluso en presencia de parámetros reales y actuales puntero y punteros locales. Esto hace difícil alcanzar el punto fijo en la abstracción de forma para funciones recursivas, al tiempo que se mantiene la precisión en la estructura.

En nuestro enfoque, abstraemos la información del ARS usando un nuevo tipo de enlace sobre los grafos de forma. Denominamos a esos enlaces *enlaces de flujo recursivo* o *recursive flow links*. Estos enlaces no representan enlaces reales en la estructura de datos del programa sino que trazan el camino de los parámetros formales puntero y los punteros locales a lo largo del flujo interprocedural recursivo.

Adicionalmente, proporcionamos ecuaciones *data-flow* extendidas para el análisis interprocedural, las sentencias de llamada y retorno de función y una reglas que determinan el cambio de contexto en los grafos cuando entran o salen de una función.

En el caso de repetitivos análisis de funciones con entradas iguales o similares, es útil disponer de un mecanismo de reutilización de resultados obtenidos previamente, en especial en una técnica costosa como el análisis de forma. En respuesta a esta demanda, hemos diseñado un sistema que reutiliza los grafos sumarios obtenidos en análisis previos para grafos similares con un mecanismo de tabulación.

C.6 Análisis de dependencias

Hemos realizado nuestra investigación para desarrollar una técnica de análisis de forma versátil y precisa que pueda ser utilizada como herramienta base para un test de dependencias. Nos centramos en la detección de dependencias de datos debidas a punteros que apuntan al *heap* en dos escenarios habituales: (i) bucles que recorren estructuras de datos dinámicas recursivas, identificando dependencias que pueden aparecer entre dos iteraciones del bucle, y (ii) llamadas a funciones recursivas que recorren estructura de datos dinámicas, identificando accesos en conflicto en diferentes llamadas recursivas.

En nuestro enfoque, anotamos información acerca de accesos de lectura/escritura durante la interpretación abstracta de sentencias de punteros que puedan estar en conflicto. Esto se realiza con la *propiedad touch*. Esta propiedad registra la historia de los accesos en los nodos. Esta historia se crea durante la interpretación abstracta, conforme el análisis progresa hacia el punto fijo. Los accesos al *heap* obtenidos de esta manera se usan para detectar dependencias de datos.

Identificamos dos patrones de recorrido en estructuras de datos dinámicas: el patrón de recorrido *1-way* y el patrón de recorrido *n-ways*, según se siga un selector o más de un selector en el recorrido de la estructura. Hemos diseñado diferentes técnicas de detección de dependencias de datos, según el patrón de recorrido identificado.

Para el patrón de recorrido *1-way*, hemos ideado un sistema en 5 fases: (i) identificación de sentencias de accesos al *heap*, (ii) creación de grupos de dependencia, (iii) incorporación de pseudosentencias *touch*, (iv) análisis de forma con propiedad *touch*, y (v) test de dependencias. Esta técnica nos permite no solo detectar todas las posibles dependencias de datos debidas a accesos al *heap* en bucles y funciones recursivas con recorridos de tipo *1-way*, sino que nos permite además distinguir entre dependencias de flujo, de salida o antidependencias. Adicionalmente, hemos diseñado técnicas para identificar dependencias de distancia cero, que no inhiben el paralelismo.

Para el patrón de recorrido *n-ways*, hemos ideado un sistema en 4 fases: (i) creación de clones de funciones recursivas, (ii) incorporación de pseudosentencias de *touch* dinámico, (iii) análisis de forma con propiedad de *touch* dinámico, y (iv) test de dependencias. En este enfoque, descomponemos la función objeto de nuestro estudio en clones y registramos los accesos al *heap* de cada clon. Si los accesos de los

distintos clones (etiquetados de modo diferente) no aparecen en un mismo nodo, entonces el código así descompuesto será paralelo, no siéndolo en caso contrario.

C.7 Conclusiones

Nuestro objetivo en investigación son los compiladores paralelizadores. En particular, estamos interesados en desvelar paralelismo desaprovechado en aplicaciones basadas en punteros. Para este propósito, hemos centrado nuestros esfuerzos en el uso del análisis de forma para el diseño de un esquema preciso de análisis de dependencias de datos. En nuestro enfoque, abstraemos las estructuras de datos que son reservadas dinámicamente en la forma de grafos de forma, y operamos sobre ellos para anotar información acerca de los accesos al *heap*. Usamos esa información para informar de dependencias de datos en el *heap*.

En este punto nos gustaría destacar las características definitorias de nuestro trabajo. Hemos mostrado, en los resultados experimentales registrados, que es posible usar una técnica de análisis del *heap* de alta precisión para un efectivo análisis de dependencias en programas que suponen un desafío para los compiladores paralelizadores actuales. Según nuestro conocimiento, ningún otro autor ha usado de forma tan efectiva las técnicas de análisis de forma para el propósito de la detección de dependencias de datos en programas que crean y recorren estructuras de datos dinámicas. Somos capaces de analizar programas incluso cuando las características de forma de la estructura de datos cambian en mitad de un recorrido. Cuando realizamos análisis de dependencias, somos capaces de detectar el tipo de dependencias en un gran número de casos, lo que es de suma utilidad para optimizaciones de paralelismo y localidad. Nos gustaría enfatizar que todas estas peculiaridades de nuestro análisis no están presentes en los trabajos relacionados que conocemos.

A continuación detallamos nuestras principales contribuciones:

1. Hemos diseñado e implementado un analizador de forma basado en el novedoso concepto de *conjuntos coexistentes de enlaces (coexistent links sets)*, que permiten representar las conexiones posibles entre localizaciones de memoria de un modo compacto. Hemos provisto la semántica abstracta necesaria para todas las sentencias de punteros al *heap*, y el esquema adecuado de ecuaciones *data-flow* y algoritmos *worklist* para conseguir el punto fijo para el análisis. Hemos realizado un completo estudio de complejidad para identificar las principales causas de limitación de la técnica. Hemos aportado evidencia experimental de que la abstracción de los conjuntos coexistentes de enlaces es adecuada para representar con exactitud una variedad de estructuras de datos dinámicas en la forma de grafos de forma. Para más información sobre estos temas, por favor consulte el capítulo 2 de esta tesis (en inglés).

2. Hemos diseñado los mecanismos necesarios para soportar el análisis de programas interprocedurales, particularmente algoritmos recursivos, dentro del analizador de forma basado en conjuntos coexistentes de enlaces. Para este propósito, hemos añadido *enlaces de flujo recursivo (recursive flow links)* a los grafos de forma. Estos codifican información de flujo que es utilizada por el análisis para preparar y recuperar el contexto apropiado en el análisis de funciones. Hemos añadido soporte para reutilizar el efecto de funciones ya analizadas para ciertos casos. Hemos identificado algunos problemas que limitan la técnica en el análisis de programas recursivos, y hemos creado soluciones para ellos. Finalmente, hemos realizado experimentos que evidencian que nuestro enfoque de análisis de forma se compara favorablemente con trabajo relacionado y es capaz de identificar correctamente información de forma para conocidos *benchmarks* interprocedurales. Más información sobre estas aportaciones está disponible en el capítulo 3 (en inglés).

3. Hemos usado la técnica de análisis de forma basada en conjuntos coexistentes de enlaces y enlaces

de flujo recursivo para análisis de dependencias de datos en aplicaciones basadas en punteros. Distinguimos entre dos patrones de recorrido en estructuras de datos dinámicas recursivas, en concreto, los patrones de recorrido *1-way* y *n-ways*. Hemos trabajado en dos líneas separadas para aplicar la idea clave de anotar acceso al *heap* en nodos para proporcionar información acerca de dependencias para ambos tipos de patrones de recorrido. En el caso del patrón de recorrido *1-way*, somos capaces de distinguir entre dependencias de flujo, salida y antidependencias. También nos hemos enfrentado a la cuestión de discriminar entre dependencias de distancia cero y dependencias de distancia mayor que cero. Una técnica sólida debe ser capaz de detectar ambos tipos, pero para los propósitos de paralelización es importante identificar dependencias de distancia cero, que no inhiben el paralelismo. En cuanto al patrón de recorrido *n-ways*, hemos adoptado el enfoque de descomponer el recorrido de la estructura, generando una versión modificada del programa, adecuada para un esquema de paralelización subsiguiente. Hemos aportado evidencia experimental de que podemos detectar dependencias de datos inducidas en el *heap* en una variedad de estructura de datos y recorridos para ambos patrones de recorrido. Adicionalmente, hemos estudiado el coste inherente al análisis de dependencias frente al coste del análisis de forma en sí mismo. Más información al respecto está disponible en el capítulo 4 de esta tesis (en inglés).

C.8 Trabajo futuro

Es justo decir que existe la sensación generalizada dentro de la comunidad de compilación de que el análisis de forma tiene poco que decir para los compiladores en producción. El argumento típico está en el elevado coste del análisis. El análisis de forma es una técnica costosa por definición porque se esfuerza en conseguir, en tiempo de compilación, información muy detallada acerca de la configuración de la memoria en el *heap*. El tipo de conocimiento que el análisis de forma puede obtener está más allá del enfoque de otras técnicas, como el análisis *points-to*. Sin embargo, no está siempre claro si ese conocimiento tan profundo puede ser usado con eficiencia.

En nuestra opinión, el problema principal respecto al análisis de forma proviene de la falta de información respecto al tiempo de ejecución. Es una técnica que opera exclusivamente en tiempo de compilación, y como tal, tiene que adoptar decisiones muy conservativas en los programas que analiza. Por tanto, como técnica aislada es probablemente insuficiente para pases de compilación realistas.

A pesar de esta limitación característica, hay varias direcciones que podemos explorar para mejorar y extender nuestro trabajo:

- **Mejora de las operaciones internas.**
  Las operaciones internas de sumarización y materialización yacen en el núcleo de nuestra estrategia de análisis de forma. Controlan el enfoque del análisis, ya sea en la materialización para mayor precisión en operaciones de actualización, o ya sea en la sumarización para acotar el tamaño de los grafos de forma. Estas dos operaciones deben ser conservativas para preservar la corrección del análisis. Sin embargo, es fácil que el análisis se vuelva excesivamente conservador, dejando el análisis inservible para el propósito de la detección de dependencias de datos.

  Reconocemos el hecho clave para esta limitación: hay información que está disponible en el momento de la sumarización que no puede ser recuperada más adelante en el proceso de materialización. Planeamos mejorar la eficacia de la materialización, considerando información presente en el momento de la sumarización. Esa información incluye: (i) características de la estructura de datos *en su conjunto*, no solo información *local* como en el enfoque actual, y (ii) el *alcance (reachability)* de sus elementos a través de los diferentes punteros. Esto nos permitiría obtener una forma más rápida y

precisa de materializar en los grafos de forma.

- **Análisis parciales.**
  Somos partidarios del uso del análisis de forma para analizar únicamente ciertas partes de un programa. En el capítulo 2 mostramos algunos resultados que evidencian que la eliminación de sentencias que no afectan a la forma de la estructura de datos puede mejorar en gran medida el rendimiento del análisis. Hay trabajo en esta dirección que utiliza *cadenas de definición y uso* para dirigir ese proceso de eliminación de sentencias de forma automática [67]. Con este enfoque esperamos poder analizar programas mayores.

- **Información de forma como herramienta base para tests de dependencias más sofisticados.**
  Nuestro enfoque del problema de la detección de dependencias de datos está basado enteramente en el análisis de forma y sus capacidades inherentes. Está basado en la interpretación abstracta de todas las sentencias de punteros al *heap*, mientras que anotamos los accesos en nodos de grafos de forma. Sin embargo, consideramos este enfoque como una primera aproximación al problema de la detección de dependencias de datos en el *heap* en programas que manipulan estructuras de datos dinámicas. Como el motor de la interpretación abstracta es de complejidad exponencial por su propia naturaleza, se trata de un modo muy costoso de revelar conflictos de accesos al *heap*. Alternativamente, podemos diseñar un test más sutil que intente evitar la penalización de la interpretación abstracta siempre que sea posible.

  Por ejemplo, podemos considerar el análisis de forma como la herramienta base para obtener una representación del *heap* en forma de grafos. Adicionalmente, podemos utilizar otra técnica que se base en la abstracción de forma para identificar accesos al *heap* en conflicto. Ya hemos realizado trabajo en esta dirección. La idea clave es proyectar, o mapear, las rutas de acceso que pueden conducir potencialmente a una dependencia sobre los grafos de forma que definen la estructura de datos. El mayor inconveniente de este enfoque es que la estructura de datos no puede cambiar en la sección del programa donde las ruta de acceso se proyectan sobre los grafos de forma. En caso contrario, sus deducciones no podrían garantizarse para todos los casos. Los resultados preliminares con este enfoque son esperanzadores, llevándonos a creer que este es el campo más prometedor para la aplicación del análisis de forma para pases de compilación realistas.

- **Generación automática de código paralelo.**
  No nos olvidemos de que el objetivo final de nuestra investigación es la generación automática de código paralelo. Los resultados de nuestras estrategias de detección de dependencias de datos pueden usarse para un pase de compilación que genere código paralelo. Ya hemos identificado UPC (*Unified Parallel C*) [17], como el lenguaje adecuado para esta tarea. UPC es uno de los lenguajes más prometedores para la generación sencilla de programas paralelos. Dispone de construcciones paralelas que pueden explotar paralelismo en la mayoría de arquitecturas actuales. Ofrece un modelo de programación de memoria compartida, pero es capaz de mapear tareas en arquitecturas de memoria distribuida, y todo de una manera muy accesible al programador. Es tan simple como compartir las variables requeridas y añadir una construcción `upc_forall` para paralelizar un bucle, independientemente de la arquitectura de destino.

  Algunos problemas aún deben ser resueltos para la paralelización automática de aplicaciones irregulares en UPC, un campo aún inexplorado. No obstante, somos optimistas acerca del uso de un pase de generación de código paralelo basado en UPC para explotar el paralelismo encontrado por nuestras estrategias de detección de dependencias.

Bibliography

[1] Francisco Corbera. *Detección automática de estructuras de datos basadas en punteros*. PhD thesis, Dpt. Computer Architecture, University of Malaga, Spain, 2001.

[2] A. Navarro, F. Corbera, R. Asenjo, A. Tineo, O. Plata, and E.L. Zapata. A new dependence test based on shape analysis for pointer-based codes. *Languages and Compilers for High Performance Computing 2004 (LCPC'04) - Lecture Notes in Computer Science*, 3602:394–408, May 2005.

[3] F. Corbera, A. Navarro, R. Asenjo, A. Tineo, and E.L. Zapata. A new loop-carried dependence detection approach for pointer-based codes. In *XV Jornadas de Paralelismo*, pages 432–437, Almería, Spain, September 2004.

[4] A. Tineo, F. Corbera, A. Navarro, R. Asenjo, and E.L. Zapata. A novel approach for detecting heap-based loop-carried dependences. In *The 2005 International Conference on Parallel Processing (ICPP'05)*, Oslo, Norway, June 2005.

[5] A. Navarro, F. Corbera, A. Tineo, R. Asenjo, and E.L. Zapata. Detecting loop-carried depedences in programs with dynamic data structures. *Journal of Parallel and Distributed Computing*, 67:47–62, 2007.

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

[7] A. Tineo, F. Corbera, A. Navarro, R. Asenjo, and E.L. Zapata. Shape analysis for dynamic data structures based on Coexistent Links Sets. In *12th Workshop on Compilers for Parallel Computers, CPC 2006*, A Coruña, Spain, 9-11 January 2006.

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

[9] A. Tineo. Speculative parallelization of pointer-based applications. In *Science and Supercomputing in Europe - Report 2006*, pages 306–308. CINECA, 2007.

[10] Adrian Tineo, Marcelo Cintra, and Diego R. Llanos. Speculative parallelization of pointer-based applications (poster). In *Transnational Access Meeting 2007 (TAM'07)*, Bologna, Italy, June 14-15 2007.

[11] A. Tineo, F. Corbera, A. Navarro, R. Asenjo, and E.L. Zapata. A compiler framework for automatic parallelization of pointer-based codes (poster). In *3rd International Summer School on Advanced Computer Architecture and Compilation for Embedded Systems (ACACES 2007)*, L'Aquila, Italy, July 15-20 2007.

[12] A. Tineo, F. Corbera, A. Navarro, R. Asenjo, and E.L. Zapata. Tracing recursive flow paths for interprocedural shape analysis (poster). In *20th International Workshop on Languages and Compilers for Parallel Computing (LCPC'07)*, Urbana, Illinois, October 11-13 2007.

[13] R. Asenjo, R. Castillo, F. Corbera, A. Navarro, A. Tineo, and E.L. Zapata. Parallelizing irregular C codes assisted by interprocedural shape analysis. In *2nd IEEE International Parallel & Distributed Processing Symposium (IPDPS'08)*, Miami, Florida, USA, April 2008.

[14] Gordon Moore. Cramming more components onto integrated circuits. *Proceedings of the IEEE*, 86:82–85, January 1998.

[15] B.L. Chamberlain, D. Callahan, and H.P. Zima. Parallel programmability and the Chapel language. *International Journal of High Performance Computing Applications*, 21(3):291–312, 2007.

[16] OpenMP Architecture Review Board. *OpenMP Application Program Interface - Version 3.0*, May 2008. http://www.openmp.org/mp-documents/spec30.pdf.

[17] UPC Consortium. *UPC Language Specifications, v1.2*. Lawrence Berkeley National Lab, 2005. Tech Report LBNL-59208.

[18] W. Blume, R. Eigenmann, K. Faigin, J. Grout, J. Hoeflinger, D. Padua, P. Petersen, W. Pottenger, L. Rauchwerger, P. Tu, and S. Weatherford. Parallel programming with Polaris. *IEEE Computer*, 29(12):78–82, 12 1996.

[19] Bjarne Steensgaard. Points-to analysis in almost linear time. In *POPL '96: Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 32–41, New York, NY, USA, 1996. ACM.

[20] Samuel Z. Guyer and Calvin Lin. Client-driven pointer analysis. In *International Static Analysis Symposium (SAS 2003)*, San Diego, California, USA, June 2003.

[21] Alexandru D. Sălcianu. *Pointer Analysis for Java Programs: Novel Techniques and Applications*. PhD thesis, Massachusetts Institute of Technology, 2006.

[22] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Fourth ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 238–252, Los Angeles, California, USA, 1977.

[23] Jordi Juan Segura Domínguez. Interfaz para la optimización y paralelización de código C. Master's thesis, Dept. Computer Architecture, July 2007.

[24] Troy A. Johnson, Sang-Ik Lee, Long Fei, Ayon Basumallik, Gautam Upadhyaya, Rudolf Eigenmann, and Samuel 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)*, West Lafayette, Indiana, USA, September 2004.

[25] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice Hall, Inc., 2nd edition, 1988.

[26] 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.

[27] 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.

[28] D. Chase, M. Wegman, and F. Zadek. Analysis of pointers and structures. *In SIGPLAN Conference on Programming Languages Design and Implementation*, pages 296–310, 1990.

[29] M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. *ACM Transactions on Programming Languages and Systems*, 2002.

[30] Martin C. Carlisle and Anne Rogers. Software caching and computation migration in Olden. In *ACM Symposium on Principles and Practice of Parallel Programming (PPoPP)*, July 1995.

[31] R.L. Graham, D.E. Knuth, and O. Patashnik. *Concrete Mathematics: A Foundation for Computer Science. Chapter 6: Stirling numbers*. Addison-Wesley, 2nd edition, 1994.

[32] F. Corbera, R. Asenjo, and E.L. Zapata. Towards compiler optimization of codes based on arrays of pointers. In *Proc. 15th Int'l Workshop on Languages and Compilers for Parallel Computing (LCPC'02)*, College Park, Maryland, July 2002.

[33] L. Lovász. *Combinatorial Problems and Exercises*. North-Holland Publishing Co., Amsterdam, 2nd edition, 1993.

[34] M. Hind and A. Pioli. Which pointer analysis should I use? In *Int. Symp. on Software Testing and Analysis (ISSTA '00)*, 2000.

[35] R. Wilson and M.S. Lam. Efficient context-sensitive pointer analysis for C programs. In *ACM SIGPLAN'95 Conference on Programming Language Design and Implementation*, La Jolla, CA, June 1995.

[36] 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.

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

[38] D. Distefano, P.W. O'Hearn, and H. Yang. A local shape analysis based on separation logic. *Lecture Notes in Computer Science*, 3920:287–302, 2006. Springer-Verlag.

[39] B. Guo, N. Vachharajani, and D. August. Shape analysis with inductive recursion synthesis. In *Programming Language Design and Implementation (PLDI'07)*, June 2007.

[40] J. Berdine, C. Calcagno, B. Cook, D. Distefano, P. O'Hearn, T. Wies, and H. Yang. Shape analysis for composite data structures. *Lecture Notes in Computer Science*, 4590:178–192, 2007.

[41] S. Magill, A. Nannevski, E. Clarke, and P. Lee. Inferring invariants in separation logic for imperative list-procesing programs. In *Workshop on Semantics, Program Analysis and Computing Enviroments for Memory Management (SPACE)*, January 2006.

[42] N. Jones and S. Muchnick. A flexible approach to interprocedural data flow analysis and programs with recursive data structures. *In Proceedings of the ACM Symposium on Principles of Programming Languages*, pages 66–74, 1982.

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

[44] S. Horwitz, P. Pfeiffer, and T. Reps. Dependence analysis for pointer variables. *ACM SIGPLAN Notices*, 1989.

[45] T. Lev-Ami and M. Sagiv. TVLA: A system for implementing static analyses. In *Static Analysis Symp. (SAS'00)*, pages 280–301, 2000.

[46] N. Rinetzky and M. Sagiv. Interprocedural shape analysis for recursive programs. In *10th International Conference on Compiler Construction (CC'01)*, pages 1433–1449, Genova, Italy, April 2001.

[47] B. Jeannet, A. Loginov, T. Reps, and M. Sagiv. A relational approach to interprocedural shape analysis. In *Proceedings of the 11th International Static Analysis Symposium (SAS'04)*, Verona, Italy, August 2004.

[48] N. Rinetzky, M. Sagiv, and E. Yahav. Interprocedural shape analysis for cutpoint-free programs. In *12th International Static Analysis Symposium (SAS'05)*, London, England, September 2005.

[49] Gilad Arnold. Specialized 3-valued logic shape analysis using structure-based refinement and loose embedding. In *Static Analysis Symposium 2006 (SAS06)*, Seoul, Korea, August 2006.

[50] Igor Bogudlov, Tal Lev-Ami, Thomas Reps, and Mooly Sagiv. Revamping TVLA: Making parametric shape analysis competitive. In *Computer Aided Verification 2007 (CAV07)*, Berlin, Germany, July 2007.

[51] Mark Marron, Deepak Kapur, Darko Stefanovic, and Manuel Hermenegildo. A static heap analysis for shape and connectivity. Unified memory analysis: The base framework. In *The 19th International Workshop on Languages and Compilers for Parallel Computing (LCPC'06)*, New Orleans, Louisiana, USA, November 2006.

[52] 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.

[53] Brian Hackett and Radu Rugina. Region-based shape analysis with tracked locations. In *Proceedings of the ACM SIGPLAN Symposium on Principles of Programming Languages (POPL'05)*, pages 310–323, Long Beach, California, USA, 12-14 January 2005.

[54] F. Corbera, R. Asenjo, and E.L. Zapata. A framework to capture dynamic data structures in pointer-based codes. *Transactions on Parallel and Distributed System*, 15(2):151–166, 2004.

[55] 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.

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

[57] A. Loginov, T. Reps, and M. Sagiv. Abstraction refinement via inductive learning. *Lecture Notes in Computer Science*, 3576, 2005. Springer-Verlag.

[58] S. Cherem and R. Rugina. Maintaining doubly-linked list invariants in shape analysis with local reasoning. In *Proceedings of the ACM Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI '07)*, Nice, France, January 2007.

[59] Alexey Gotsman, Josh Berdine, and Byron Cook. Interprocedural shape analysis with separated heap abstractions. In *In Proceedings of the 13th International Static Analysis Symposium (SAS'06), LNCS 4134*, pages 240–260, Seoul, Korea, 2006.

[60] Steven S. Muchnick. *Advanced compiler design and implementation*. Morgan Kaufmann, 1997.

[61] R. Ghiya and L. J. Hendren. 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.

[62] Keith D. Cooper, Mary W. Hall, and Ken Kennedy. Procedure cloning. In *Computer Languages*, pages 96–105, 1992.

[63] M. Kulkarni, K. Pingali, B. Walter, G. Ramanarayanan, K. Bala, and L.P Chew. Optimistic parallelism requires abstractions. In *Programming Language Design and Implementation (PLDI'07)*, pages 211–222, June 2007.

[64] A. Kejariwal, X. Tian, M. Girkar, W. Li, H. Saito, U. Banarjee, A. Nicolau, A.V. Veidenbaum, and C.D. Polychronopoulos. Tight analysis of the performance potential of thread speculation using SPEC CPU2006. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'07)*, pages 215–225, San Jose, California, March 2007.

[65] Standard Performance Evaluation Corporation (SPEC). *SPEC CPU2006 Documentation*, 2006. http://www.spec.org/cpu2006/Docs/.

[66] Mark Marron, Darko Stefanovic, Deepak Kapur, and Manuel Hermenegildo. Identification of heap-carried data dependence via explicit store heap models. In *Languages and Compilers for Parallel Computing (LCPC'08)*, Alberta, Canada, 2008.

[67] R.Castillo, F. Corbera, A. Navarro, R. Asenjo, and E.L. Zapata. 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.