– Cover Page –

A framework to capture dynamic data structures in pointer-based codes *

F. Corbera R. Asenjo E. Zapata

Computer Architecture Department. University of Málaga. Spain.
e-mail: {corbera,asenjo,ezapata}@ac.uma.es

Abstract

To successfully exploit all the possibilities of current computer/multicomputer architectures, optimization compiling techniques are a must. However, for codes based on pointers and dynamic data structures these optimization techniques have to be necessarily carried out after identifying the characteristics and properties of the data structure used in the code. In this paper we describe the framework and the analyzer we have implemented to capture complex data structures generated, traversed, and modified in codes based on pointers. Our method assigns a *Reduced Set of Reference Shape Graphs* (RSRSG) to each statement to approximate the shape of the data structure after the execution of such a statement. With the properties and operations that define the behavior of our RSRSG, the method can accurately detect complex recursive data structures such as a doubly linked list of pointers to trees where the leaves point to additional lists. Several experiments are carried out with real codes to validate the capabilities of our analyzer.

Keywords: Shape Analysis, Pointers, Recursive Data Structures, Shape Graphs, Optimizing compiler, irregular codes

*This work was supported by the Ministry of Education and Science (CICYT) of Spain (TIC96-1125-C03), by the European Union (BRITE-EURAM III BE95-1564), by APART: Automatic Performance Analysis: Resources and Tools, EU Esprit IV Working Group No. 29488

1 Introduction

Programming languages such as C, C++, Fortran90, or Java are widely used for non-numerical (symbolic) and numerical applications. All these languages allow the use of complex data structures usually based on pointers and dynamic memory allocation. The use of complex data structures is very helpful in order to speedup code development and, besides this, it also may lead to reducing the program execution time. However, compilers are not able to successfully optimize codes based on these complex data structures for current computers or multicomputers.

More precisely, when dealing with pointer-based data structures built at run time, current compilers are not able to capture, from the code text, the necessary information to exploit locality, automatically parallelize the code, or carry out other important optimizations. In other words, if the compiler is not aware of the properties fulfilled by the data structure used in the code, it is impossible to apply certain optimizations. For instance, if the compiler does not know that a certain loop is traversing a doubly linked list, then important techniques such as data prefetching, locality exploiting or parallelism detection, cannot be applied.

With this motivation, the goal of our research line is to propose and implement new techniques that can be included in compilers to allow for the automatic optimization of real codes based on dynamic data structures. As a first step, we have selected the shape analysis subproblem, which aims at estimating at compile time the shape the data will take at run time. Given this information, subsequent analysis (not implemented yet) would focus on particular optimizations, for example, to exploit the memory hierarchy or to detect whether or not certain sections of the code can be parallelized because they access independent data regions.

There are other open research lines dealing with the analysis of codes in the presence of pointers, such as alias analysis or points-to analysis. Basically, these analyses are designed to determine the superset of locations to which a pointer *must* or *may* point (points-to sets). In this context, there are flow- and context-insensitive approaches such as Steensgaard's algorithm [24] (fast but imprecise), Andersen's approach [1] (more precise but non-scalable), and tradeoff solutions [23, 8]. There are also context-sensitive approaches such as [25] or flow-sensitive analysis [12].

These kinds of pointer analysis provide enough information to allow for some scalar optimizations, such as Common Subexpression Elimination, Loop Invariant Removal, or Location Invariant Removal [11]. However, the information provided by the points-to sets is not accurate enough to allow for more ambitious optimizations such as loop-level automatic parallelization, automatic data distribution, and locality exploiting. Currently, the majority of research groups rely on manual annotations when dealing with such complex code optimizations in the presence of pointers, due to points-to analysis is not sufficient. For instance, Chilimbi et al. ask the programmer to annotate the code to exploit cache locality [5] or a previous execution profile is needed in order to exploit cache prefetching [4]. In the area of distributed memory locality exploiting and communication optimization, Zhu and Hendren [26, 27] also rely on code annotations with special compiler directives. Similarly, Rogers et al. [20] propose a thread-level parallelism in codes annotated with directives such as *futurecall* and *touch*.

However, some groups are trying to automatically extract more information from the code text to optimize codes based on pointers. For example, Ghiya [9] have implemented the McCAT compiler to put pointer analysis to work. Basically, this compiler uses points-to analysis to deal with stack-directed pointers and connection analysis and shape analysis to deal with heap-directed pointers. This analysis is used for exploiting two parallelism levels in codes based on recursive data structures: at the function level when routines traverse disjoint sub-tree structures; and at the loop level in two cases; list traversing and array of pointers to disjoint structures traversing. However, their shape analysis [10] is too simple and conservative leading to a serious lack of parallelism exploitation. This shape analysis determines the shape attribute, TREE, DAG or CYCLE, for the data structures pointed to by pointer variables. The problem is that the CYCLE attribute, which prevents any kind of parallelism exploitation, can too easily be associated with a data structure; for example, when a single linked list grows by inserting new items in the middle instead of appending them at the end, or because the data structure has a temporary cycle, but returns to being acyclic in the rest of the code. Besides this, doubly linked lists cannot be parallelized even when they are traversed by only one kind of selector (next or previous). In summary, all these drawbacks arise due to the fact that their "shape analysis" does not keep information about the topological structure of the links in the analysed data structures. For example, you may know that it is a DAG data structure, but you do not know which paths may lead to the same heap node of the DAG data structure.

Thus, we have to emphasise that our final goal is to allow for the automatic optimization of codes based on recursive data structures, but it is clear that, first of all, better shape analysis techniques have to be proposed. That is, new approaches to automatically capture the essential characteristics and properties of heap-allocated data structures are essential. The goal of this paper, in this initial step, is to describe a new and very accurate shape analysis algorithm able to identify from the code text the essential properties of the data structures of the code.

With this in mind, our proposal is based on approximating all the possible memory configurations that can arise after the execution of a statement by a set of graphs: the *Reduced Set of Reference Shape Graphs* (RSRSG). We see that each RSRSG is a collection of *Reference Shape Graphs* (RSG) each one containing several non-compatible nodes. Finally, each node represents one or several memory locations and the edges between nodes represents links between the represented memory locations. Compatible nodes are "summarized" into a single one. Two nodes are compatible if they share the same reference properties. With this framework we can achieve accurate results in a reasonable analysis time and expending a reasonable ammount of memory. Besides this, we cover situations that were previously unsolved, such as detection of complex structures (lists of trees, lists of lists, etc.) and structure permutation, as we will see in this article.

The rest of the paper is organized as follows. Section 2 briefly describes the whole framework, introducing the key ideas of the method and presenting the data structure example that will help in understanding node properties and operations with graphs. These properties are described in Sect. 3 where we show how the RSG can accurately approximate a memory configuration. In Sect. 4 we formalize when two RSGs are compatible and how they can be joined, among other operations related with the RSGs included in an RSRSG. These operations have been implemented in an analyzer which is experimentally validated, in Sect. 5, by analyzing several codes based on complex data structures. In Sect. 6 we compare our proposal with previous related works. Finally, we summarize the main contributions and future work in Sect. 7.

2 Method overview

Basically, our method is based on approximating by graps all possible memory configurations that can appear after the execution of a statement in the code. We call a collection of dynamic structures a *memory configuration*. These structures comprise several memory chunks, that we call *memory locations*, which are linked by references. Inside these memory locations there is room for data and for pointers to other memory locations. These pointers are called *selectors*.

Note that due to the control flow of the program, a statement could be reached by following several paths in the control flow. Each "control path" has an associated memory configuration which is modified by each statement in the path. Therefore, a single statement in the code modifies all the memory configurations associated with all the control paths reaching this statement. Each memory configuration is approximated by a graph we call *Reference Shape Graphs* (RSG). So, taking all this into account, we conclude that each statement in the code will have a set of RSGs associated with it.

The RSGs are graphs in which nodes represent memory locations which have similar reference patterns. To determine whether or not two memory locations should be represented by a single node, each one is annotated with a set of properties. Now, if several memory locations share the same properties, then all of them will be represented by the same node. This way, a possibly unlimited memory configuration can be represented by a limited size RSG, because the number of different nodes is limited by the number of properties of each node. These properties are related to the reference pattern used to access the memory locations represented by the node. Hence the name *Reference Shape Graph*. These properties are described in Sect. 3.

As we have said, all possible memory configurations which may arise after the execution of a statement are approximated by a set of RSGs. We call this set *Reduced Set of Reference Shape Graphs* (RSRSG), since not all the different RSGs arising in each statement will be kept. On the contrary, several RSGs related to different memory configurations will be fused when they represent memory locations with similar reference patterns. As we will see, there are also several properties related to the RSGs, and two RSGs should share these properties to be joined. Therefore, besides the number of nodes in an RSG, the number of different RSGs associated with a statement are limited too. This union of RSGs greatly reduces the number of RSGs and leads to a practicable analysis.

To move from the "memory domain" to the "graph domain", the calculation of the RSRSGs associated with a statement is carried out by the **symbolic execution** of the program over the graphs. In this way, each program statement transforms the graphs to reflect the changes in memory configurations derived from statement execution. The **abstract semantic** of each statement states how the analysis of this statement must transform the graphs.

Let us illustrate all this with an example. In Fig. 1 we can see a simple code with seven pointer statements. Our analyzer symbolically executes each statement to build the RSRSG associated with them. Actually, after the execution of the third statement we obtain an RSRSG with a single RSG which represents three different memory locations by three nodes; all of them of the same type, with the same *nxt* selector, but pointed to by different pointer variables (*pvars)*. Now, this RSRSG is modified in three different ways because there are three different paths in the control flow graph, each one with a different pointer statement. All these paths join in statement 7, and after the execution of this statement we obtain an RSRSG with two RSGs. This is because the RSGs coming from statements 5 and 6 are compatible and can be summarized into a single one.

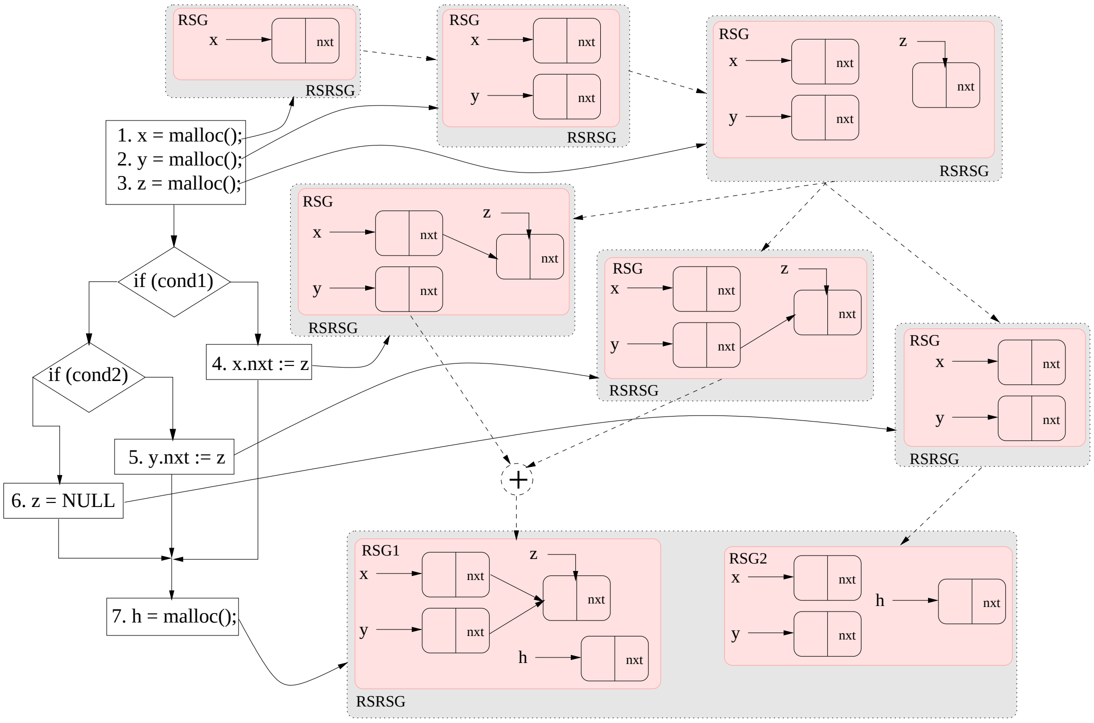


Figure 1: Building an RSRSG for each statement of an example code.

As we have seen, the symbolic execution of the code consists in the abstract interpretation of each statement in the code. This abstract interpretation is carried out iteratively for each statement until we reach a fixed point in which the resulting RSRSG associated with the statement does not change any more [7]. This way, for each statement that modifies dynamic structures, we have to define the abstract semantics which describes how these statements modify the RSRSG. We consider six simple instructions that deal with pointers: $x = NULL$, $x = malloc$, $x = y$, $x \rightarrow sel = NULL$, $x \rightarrow sel = y$, and $x = y \rightarrow sel$. More complex pointer instructions can be built upon these simple ones and temporal variables. Due to space constraints we cannot formally describe the abstract semantics of each one of these statements. However, we intuitively present the modifications involved in an RSG after the statement symbolic execution:

- The $x = NULL$ statement leads to elimination of the references from the pointer variable x to any memory location. Therefore we have to remove these references from the RSG.

• The $x = malloc$ statement simply initializes new memory locations represented in the RSG by a node referenced by x.

• The $x = y$ modifies the RSG such that all memory locations pointed to by y are now also pointed to by x. Before this statement and the previous one, we always automatically insert the $x = NULL$ statement to ensure that before assigning a new value to x, x is not pointing to any place.

• The statement $x \to sel = NULL$ is the most complex one, because it break links between nodes. This leads to many changes in the properties of the nodes. In order to obtain an accurate output RSG before removing the $x \to sel$ link we divide the RSG into several rsg_j. This division is carried out by taking into account that each rsg_j should have a single destination (node) for the $x \to sel$ link. In this way we can better focus on the several memory configurations represented by the RSG regarding this $x \to sel$ link. Each rsg_j is pruned after the division to remove redundant or inexistent nodes or links which have been conservatively inherited from the parent RSG. We also increase the accuracy of the method by materializing from the node, the memory location which is the real target of the $x \to sel$ link. After this, this link can be safely removed.

All these processes can be better illustrated by a simple example. In Fig. 2 (a) we see an RSG representing a doubly linked list of two or more elements. Actually, n_1 represents the first element in the list, n_2 the middle elements, and n_3 the last one. Let us suppose that this RSGs is an input rsg_i to the $x \to nxt = NULL$ statement where x is pointing to the memory location represented by node n_1. As we said, the first step in the abstract interpretation of this statement is the division operation. Figure 2 (b) shows the resulting $rsg\prime_1$ and $rsg\prime_2$ after the division. Note that in each one of these graphs there is a single destination for $x \to nxt$. This division process is formally described in Sect. 4.1. In Fig. 2 (c) we show the result of the pruning process in which the compiler removes nodes and links which do not fulfill the graphs' properties. In fact, $rsg\prime\prime_1$ represents a list of three or more elements and $rsg\prime\prime_2$ is clearly a list of just two items. This pruning process is formally described in Sect. 4.2. Now, before removing the $x \to nxt$ link in both graphs, the compiler has to focus more on one of the RSGs. More precisely, in $rsg\prime\prime_1$, we have to materialize from node n_2 the node n_4 which represents the single list item referenced by $x \to nxt$, as we can see in Fig. 2 (d). Finally, we see in Fig. 2 (e) how we safely remove the link $x \to nxt$ in both graphs to obtain the final rsg_1 and rsg_2.

• The $x \to sel = y$ statement implies the execution of the same procedure just

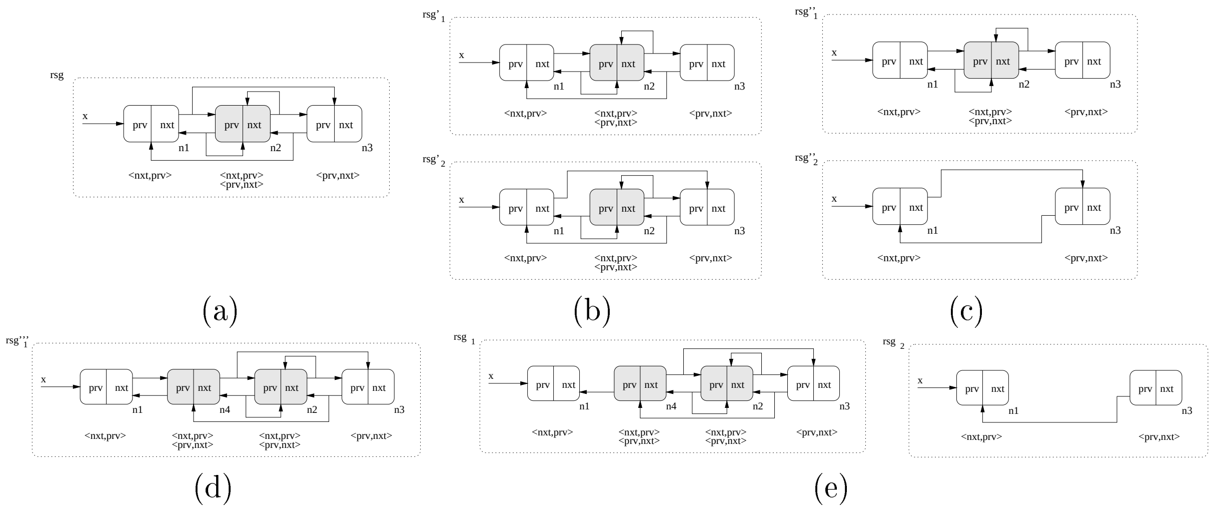


Figure 2: Complete process of the abstract interpretation required by the $x \rightarrow nxt = NULL$ statement.

described for the $x \rightarrow sel = NULL$ statement (RSG division, pruning, and node materialization) followed by the generation of a link by selector sel from the nodes pointed to by x to the nodes pointed to by y.

• Finally, the statement $x = y \rightarrow sel$ leads to the inclusion of a new reference from the x pvar to all the memory locations pointed to by $y \rightarrow sel$. Now, the RSG division, pruning, and node materialization are carried out for the $y \rightarrow sel$ link. In this way we will point with x to the exact memory locations pointed to by $y \rightarrow sel$ and to no other.

From a higher perspective, the whole symbolic execution process can be seen by looking at Fig. 3. For each statement in the code we have an input RSRSG_i and the corresponding output RSRSG_o resulting from the modifications which take place in the memory configurations after the statement execution. As we said, the RSRSG_i comprises several rsg_{ij} to capture all the memory configurations associated with each path in the control flow graph. During the symbolic execution of the statement all these rsg_{ij} are going to be updated. The first step comprises the graph division and pruning processes after which we obtain several rsg_{ijk}. Then the abstract interpretation of the statement takes place and usually the complexity of the RSGs grows. In order to counter this effect, the compiler carries out a compression of the graph phase in which each RSG is simplified by the summarization of compatible nodes in the RSG, to obtain the rsg^*_{ijk} graphs. This step is formally described in Sect. 3.2. Furthermore, some of the rsg^*_{ijk}s can be fused into a single rsg_{ok} if they represent similar memory configurations. This graph union operation is described in Sect. 4.3.

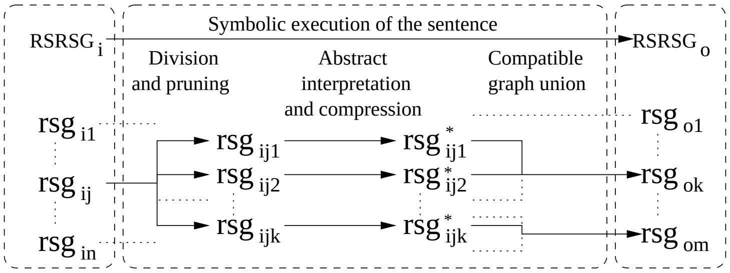
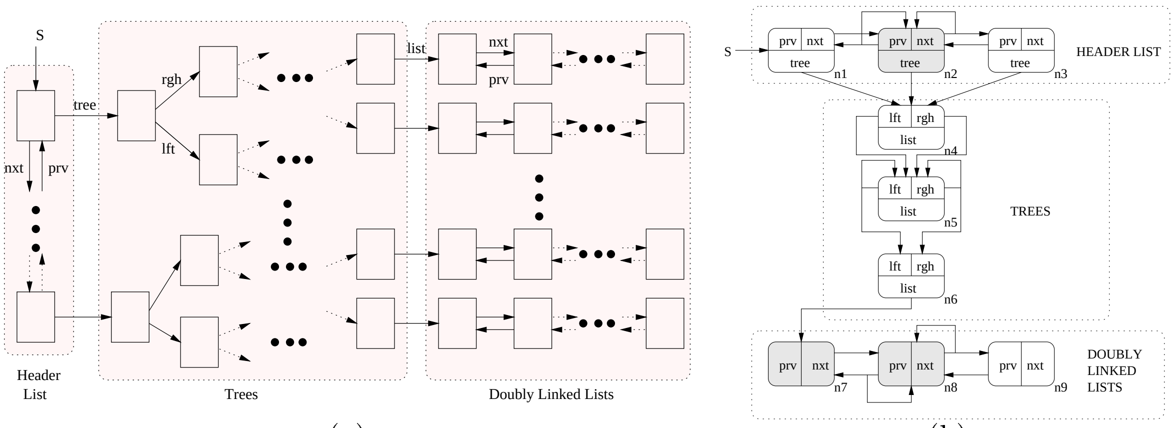


Figure 3: Schematic description of the symbolic execution of a statement.

2.1 Working example

We present here an hypothetical data structure which will be referred to during the framework and operations description in the next sections. The data structure, presented in Fig. 4 (a), is a doubly linked list of pointers to trees. Besides this, the leaves of the trees have pointers to doubly linked lists. The pointer variable S points to the first element of the doubly linked list (header list). Each item in this list has three selectors: nxt, prv, and $tree$. This $tree$ selector points to the root of a binary tree in which each element has the lft and rgh selectors. Finally, the leaves of the trees point to additional doubly linked lists. All the trees pointed to by the header list are independent and do not share any element. In the same way, the lists pointed to by the leaves of the same tree or different trees are also independent.



(a) (b)

Figure 4: A complex data structure (a), and compact representation of the resulting RSRSG (b). The real RSRSG generated by the analyzer provides more information than the one represented.

This data structure is built by a C code that also traverses the elements of the header list with two pointers and eventually can permute two trees. Our analyzer has processed this code obtaining an RSRSG for each statement in the program. Figure 4 (b) shows a compact represetation of the RSRSG obtained for the last statement of the code after the analysis process. As we will see in the next sections, from the RSRSG represented in Fig. 4 (b) we can infer the actual properties of the real data structure: the trees and lists do not share elements and therefore they can be traversed in parallel.

3 Reference Shape Graph

We present here our framework with a formal description of the RSGs. First, we need to present the notation used to describe the different memory configurations that may arise when executing a program.

Definition 3.1 *We call a collection of dynamic structures a memory configuration. These structures comprise several memory chunks, that we call memory locations, which are linked by references. Inside these memory locations there is room for data and for pointers to other memory locations. These pointers are called selectors.*

We represent the memory configurations with the tuple $M = (L, P, S, PS, LS)$ where: **L** *is the set of memory locations;* **P** *is the set of pointer variables (pvars) used in the program;* **S** *is the set of selectors declared in the data structures;* **PS** *is the set of references from pvars to memory locations, of the type $< pvar, l >$, with $pvar \in P$ and $l \in L$; and* **LS** *is the set of links between memory locations, of the form $< l_1, sel, l_2 >$ where $l_1 \in L$ references $l_2 \in L$ by selector $sel \in S$.*

We will use L(m), P(m), S(m), PS(m), and LS(m) to refer to each one of these sets for a particular memory configuration, m. □

Therefore, we can assume that the RSRSG of a program statement is an approximation of the memory configuration, M, after the execution of this statement. But let us first describe the *RSGs* now that we know how a memory configuration is defined.

Definition 3.2 *An RSG is a graph represented by the tuple $RSG = (N, P, S, PL, NL)$ where:* **N** *is the set of nodes. Each node can represent several memory locations if they fulfill certain common properties;* **P** *is the set of pointer variables (pvars) used in the program;* **S** *is the set of declared selectors;* **PL** *is the set of references from pvars to nodes, of the type $< pvar, n >$ with $pvar \in P$ and $n \in N$; and* **NL** *is the set of links between nodes, of the type $< n_1, sel, n_2 >$ where $n_1 \in N$ references $n_2 \in N$ by selector $sel \in S$.*

We will use N(rsg), P(rsg), S(rsg), PL(rsg), and NL(rsg) to refer to each one of these sets for a particular RSG, rsg. □

Note that the P(m) and P(rsg) sets are the same in both memory and graph domains, so from now on we will refer to them as P. The same can be applied to S(m) and S(rsg).

To obtain the RSG which approximates the memory configuration, M, an abstraction function is used, $F : M \rightarrow RSG$. This function maps memory locations into nodes and references to memory locations into references to nodes at the same time. In other words, F, translates the memory domain into the graph domain. Function F extracts some important properties from a memory location and, depending on these, this location is translated into a node. Besides this, if several memory locations share the same properties then this function maps all of them into the same node of the RSG. These properties are: Type, Structure, Reference pattern, Share information, Simple paths, Touch information, and Cycle links. These are now described.

3.1 Node properties

1.- The node **TYPE** property states the data type of the memory locations represented by this node. Therefore, two memory locations can be mapped into the same node if they share the same `TYPE` value. This property leads to the situation where, for the data structure presented in Fig. 4(a), the nodes representing list memory locations will not be summarized with those nodes representing tree locations, as we can see in Fig. 4 (b).

2.- The **STRUCTURE** information avoids the summarization of two nodes which represent memory locations of the same type but which do not share any element (they are non-connected components). Two memory locations have the same `STRUCTURE` value if there is a path between them. Again, two memory locations can be represented by the same node if they have the same `STRUCTURE` value.

3.- The **Reference patterns** are introduced to represent by different nodes the memory locations with different reference patterns (type of selectors which point to and from this memory location). This keeps singular memory locations of the data structure in separate nodes. For example, the roots and the leaves of the data structure in Fig. 4 (a) are two kinds of memory locations. The root of one of the trees is referenced by the header list and the leaves do not point to tree items but to a doubly linked list. Thanks to the reference patterns, the method results in the RSRSG of Fig. 4 (b), where the root of the tree, the leaves, and the other tree items are clearly identified.

In order to obtain this behavior, we define two sets `SELINset` and `SELOUTset` which contain the set of input/output selectors for a certain location: sel_i is in the $\texttt{SELINset}(l_1)$ if l_1 is referenced from somewhere by selector sel_i, or sel_i is in $\texttt{SELOUTset}(l_1)$ if $l_1.sel_i$ points to somewhere outside. Our method can eventually lead to being unable to exactly determine whether or not a certain selector is in an input or output set for a node, n. This way, we also define $\texttt{PosSELINset}(n)$ and $\texttt{PosSELOUTset}(n)$ which contain the selectors which "may" point to/from node n. To know if two nodes can be summarized we have defined a Boolean function $\texttt{C\_REFPAT}(n_1, n_2)$ that returns true if two nodes represent memory locations with a similar reference pattern. Basically, we allow the summarization of two nodes if both of them have the same input and output selectors or we do not know the input or output selectors of one or both nodes.

4.- The **Share information** can tell whether at least one of the locations represented by a node is referenced more than once from other memory locations. Due to the relevance of this property to inform the compiler about the potential parallelism exhibited by the analyzed data structure, we use two kinds of attributes for each node: (i) $\texttt{SHARED}(n)$ with $n \in N(rsg)$, is a Boolean function which returns "true" if any of the locations, l_1, represented by n can be referenced by other locations, l_2 and l_3, by different selectors, sel_i and sel_j; and (ii) $\texttt{SHSEL}(n, sel)$ with $n \in N(rsg)$ and $sel \in S$, is a Boolean function which returns "true" if any of the memory locations, l_1, represented by n can be referenced more than once by selector sel from other locations, l_2 and l_3.

Let's illustrate these SHARED and SHSEL properties using the compact representation of the RSRSG presented in Fig. 4 (b). In this figure, shaded nodes have the SHARED property set to true. For example, in the header list the middle node n_2 is shared, $\texttt{SHARED}(n_2)$=1, because n_2 is referenced by selectors nxt and prv. However, the $\texttt{SHSEL}(n_2, nxt)$=$\texttt{SHSEL}(n_2, prv)$ =0 which means that by following selector nxt or prv it is not possible to reach an already visited memory location. Actually, in this example, there are no selectors with the SHSEL property set to true. Thus, the same happens for node n_8 which represents the middle items of the doubly linked lists. We can also see in Fig. 4 (b), that node n_4 is not shared, which states that, in fact, from memory locations represented by n_1, n_2, and n_3 we always reach different trees which do not share any elements (as we see in Fig. 4 (a)). Finally, node n_7 is shared because it is pointed to by selectors $list$ and prv. However, due to $\texttt{SHSEL}(n_7, list)$=0 we can ensure that two different leaves of the trees will never point to the same doubly linked list.

Now, two nodes can be summarized if they have the same SHARED and SHSEL attributes.

5.- Simple paths denominates the access path from a pointer variable (pvar) to a location or node if the length of this path is less than or equal to 1. An example of a simple path is $p \to s.sel \to t$ in which the pvar p points to the location s which points to the location t using the selector sel. Note that, in this example, the simple path for t is $< p, sel >$ and the simple path for s is $< p, \emptyset >$. There are codes in which the shape of the data structure is better approximated by the RSG if we avoid the summarization of nodes which are "near" to a pvar. With the simple path we consider that a node is "near" to a pvar if this pvar points to the node directly, or by a path of length one (in other words, there is a single hop from the pvar to the node).

We define, for a memory location $l \in L(m)$, $\texttt{SPATH}(l) = \{sp_1, ..., sp_n\}$ where $sp_i =< pvar, sel^* >$ with $pvar \in P$. In other words, $\texttt{SPATH}(l)$ is the set of simple paths (pointer variables and selectors) which point to l directly ($sel^* = \emptyset$) or through an intermediate location, l_i ($< pvar, l_i > \in PS(m) \wedge < l_i, sel^*, l > \in LS(m)$). The length of a simple path, $sp_i =< pvar, sel_i >$, $\texttt{LEN}(sp_i) = 0$ if $sel_i = \emptyset$ or 1 if $sel_i = sel \in S$.

To determine when two nodes can be summarized, we define a Boolean function $\texttt{C\_SPATH}(n_1, n_2, m)$ which returns true if nodes n_1 and n_2 have compatible SPATH. The parameter m imposes the constraints in the comparison of nodes. If $m = 0$ there are less restrictions to summarize two nodes. In this case, two node SPATHs are compatible if they comprise the same zero-length simple paths. This particular case of the C_SPATH function is called C_SPATH0. On the contrary, if $m > 0$, the two SPATHs also have to share at least m one-length simple paths. We have found in our experiments that a tradeoff value for the m parameter is one, for which we can obtain a good description of the data structure without an excessive growth of nodes. In this case the function is called C_SPATH1.

6.- Touch information. Until now, we have seen properties which capture some important issues of the memory configuration and which change according to the current statement. However, sometimes it is necessary to keep track of the memory locations for several statements to increase the accuracy of the RSRSGs. For example, if we deal with a list data structure, we know that all the middle elements (not pointed to by any pvar) are going to be summarized in a single node. Now, when traversing this list in a loop, the same summary node will represent non-visited locations as well as visited ones. On the other hand, during one acyclic traversal of the list, it would be better to keep the visited locations in a separate node in such a way that new changes only affect non-visited nodes.

In order to achieve this behavior in the analyzer we assign to each node a new property called TOUCH. This property is taken into account only inside loop bodies. In this case, the TOUCH information of a node is the set of pvars from which the memory locations represented by the node have been visited. We understand by "pvar x visit node n" that the node n has been referenced by x. For example, in $x = y$, the node pointed to by y is visited by x. In the same way, in $x = y \rightarrow sel$, the node pointed to by selector sel from node y is visited by x.

Now, two memory locations can be summarized in the same node if they have been "touched" by the same set of pvars. In this way we can keep visited nodes apart from non-visited ones. However, clearly this new restriction in the summarization will increase the number of nodes in the RSRSGs. In order to avoid the explosion in the number of nodes we have to constrain the kind of pvar which can appear in the TOUCH set. More precisely, only those pvars which are used to traverse dynamic data structures (called induction pointers by Yuan-Shin Hwang [15] or navigators by Rakesh Ghiya [9]) are eligible to be included in the set. Taking all this into account, we can finally define for $n \in N(rsg)$: $\mathtt{TOUCH}(n) = \{ipvar | ipvar \in iP\}$ where iP is the set of induction pvars found in the code. Clearly, there should be a preprocessing analyzer pass to identify inductions pvars in the code. Due to space constraints we cannot describe this preprocessing pass but it is based on Access Path Expressions [15].

Finally, the compiler also implements an additional improvement to save space and time. The idea is that after exiting a loop body the TOUCH information regarding the ipvars of this loop are not needed any more. This way, the compiler removes those ipvars associated with this loop from the TOUCH set of the nodes.

7.- The **Cycle links** property is introduced to increase the accuracy of the data structure representation by avoiding unnecessary edges that can appear during the RSG updating process. The cycle links of a node, n, are defined as the set of pairs of references $< sel_i, sel_j >$ such that when starting at node n and consecutively following selectors sel_i and sel_j, the n node is reached again.

This property is very useful for dealing with doubly linked structures. For example, in the data structure presented in Fig. 2 (a), the elements in the middle of the doubly linked lists, represented by node n_2, have two cycle links: $< nxt, prv >$ and $< prv, nxt >$, due to starting at a list item and consecutively following selectors nxt and prv (or prv and nxt) the starting item is reached.

We conclude here that the CYCLELINKS property is used during the pruning process. Thus, in contrast with the other six properties already described, the CYCLELINKS property does not prevent the summarization of two nodes which do not share the CYCLELINKS sets.

3.2 Compression of graphs

As we explained in Sect. 2 the code analysis implies symbolically executing each statement applying the abstract semantics associated with this statement. This symbolic execution modifies the RSGs included in the input RSRSG. This way, these graphs may contain new nodes representing new memory locations and/or the properties for certain nodes may have changed. In other words, after the symbolic execution of a statement over an input RSRSG, the resulting RSRSG may contain RSGs with redundant information, which can be removed due to node summarization or compression of the RSG.

In order to do this, after the symbolic execution of a statement, the method applies the `COMPRESS` function over the just modified RSGs. However, before explaining this `COMPRESS` function, we need to define the `C_NODES_RSG` one, which identifies the compatible nodes that will later be summarized. This Boolean function just has to check whether or not the first six properties previously described are the same for both nodes (as we said in the previous subsection, the `CYCLELINKS` property does not affect the compatibility of two nodes). This way

$\texttt{C\_NODES\_RSG}(n_i, n_j) = true$ if $(\texttt{TYPE}(n_i) = \texttt{TYPE}(n_j)) \wedge (\texttt{STRUCTURE}(n_i) = \texttt{STRUCTURE}(n_j)) \wedge (\texttt{SHARED}(n_i) = \texttt{SHARED}(n_j)) \wedge (\texttt{SHSEL}(n_i, sel_k) = \texttt{SHSEL}(n_j, sel_k) \forall sel_k \in S) \wedge (\texttt{TOUCH}(n_i) = \texttt{TOUCH}(n_j)) \wedge (\texttt{C\_REFPAT}(n_i, n_j) = 1) \wedge (\texttt{C\_SPATH}(n_i, n_j, m) = 1)$.

Now, compatible nodes of the same RSG are summarized by the function $\texttt{COMPRESS}(rsg) = rsg_c$ which does not change the set of pvars, P, nor the set of selectors, S, in the new rsg_c. However, the other sets involved in the new rsg_c composition are:

• The set of nodes of the compressed RSG, $N(rsg_c)$, will contain the nodes which cannot be summarized with any other plus the nodes resulting from the summarization of compatible nodes. We use the `MERGE_COMP_NODES` function to generate a summary node from a group of compatible nodes, as we will see later. Formally:

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

• The new set of pvar references, $PL(rsg_c)$, is basically the same set of the uncompressed RSG, $PL(rsg)$, but which maps all the nodes into the new ones. This is done with the $n_c = \texttt{MAP\_RSG}(n)$ function which maps the old node $n \in N(rsg)$ into the new node $n_c \in N(rsg_c)$:

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

• The same idea applies to the set of references for the compressed graph:

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

Regarding the MERGE_COMP_NODES function used for the summarization of several compatible nodes, we define:

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

The $\texttt{MERGE\_NODES}(n_1, n_2)$ function takes care of the summarization of nodes n_1 and n_2 into node n. The properties of this new node, n, are set in order to preserve the description of the data structures represented by the original nodes. This way, $\texttt{MERGE\_NODES}(n_1, n_2) = n$, where:

$\texttt{TYPE}(n) = \texttt{TYPE}(n_1) = \texttt{TYPE}(n_2)$; $\texttt{SHARED}(n) = \texttt{SHARED}(n_1) = \texttt{SHARED}(n_2)$; $\texttt{SHSEL}(n, sel_i) = \texttt{SHSEL}(n_1, sel_i) = \texttt{SHSEL}(n_2, sel_i) \forall sel_i \in S$; $\texttt{TOUCH}(n) = \texttt{TOUCH}(n_1) = \texttt{TOUCH}(n_2)$; $\texttt{PosSELOUTset}(n) = \texttt{PosSELOUTset}(n_1) \cup \texttt{PosSELOUTset}(n_2)$; $\texttt{PosSELINset}(n) = \texttt{PosSELINset}(n_1) \cup \texttt{PosSELINset}(n_2)$; $\texttt{SELINset}(n) = \texttt{SELINset}(n_1) \cap \texttt{SELINset}(n_2)$; $\texttt{SELOUTset}(n) = \texttt{SELOUTset}(n_1) \cap \texttt{SELOUTset}(n_2)$;

$$\texttt{CYCLELINKS}(n) = \{< sel_i, sel_j > | \\ \begin{cases} < sel_i, sel_j > \in (\texttt{CYCLELINKS}(n_1), \texttt{CYCLELINKS}(n_2)) \lor \\ < sel_i, sel_j > \in \texttt{CYCLELINKS}(n_1) \land \nexists n_k \in N(rsg), < n_2, sel_i, n_k > \in NL(rsg) \lor \\ < sel_i, sel_j > \in \texttt{CYCLELINKS}(n_2) \land \nexists n_k \in N(rsg), < n_1, sel_i, n_k > \in NL(rsg) \end{cases}$$

This means that the new node will have the same TYPE, SHARED, SHSEL, and TOUCH properties which actually should be the same in n_1 and n_2 to allow the summarization of both nodes. However, the new reference pattern information behaves conservatively. If sel_i is an input/output selector in both nodes, n_1 and n_2, then it will remain as an input/output selector in the resulting summarized node, n. The same happens if sel_i is not an input/output selector in both nodes. However, if we are not sure about the sel_i selector in one of the original nodes, then the same uncertainty exists regarding the resulting summarized node.

Finally, regarding the CYCLELINKS sets, the resulting node n keeps the common cycle links sets from the original nodes, n_1 and n_2. In addition, a cycle link, $< sel_i, sel_j >$, from one of the nodes, for example, n_1, is also included in the cycle link set of the node n if the first selector sel_i is not a link selector in the other node, n_2.

4 Reduced Set of RSGs

We have already seen that an RSG describes a memory configuration by a finite graph. The execution of each statement in the program can modify the memory configuration and thus the RSG. However, due to the control flow of the program, the same statement could be reached by several control paths leading to several memory configurations depending on the path. This way, there could also be several RSGs for the same program statement. In our method, we maintain the representation of all these RSGs with the *Reduced Set* *of Reference Shape Graphs* (RSRSG). Therefore, each statement in the program has an associated RSRSG which approximately describes all possible memory configurations that may arise after the execution of this statement.

However, the number of RSGs stored in an RSRSG could be prohibitive if we do not apply some simplifications while keeping reasonable accuracy in the representation. Actually, this simplification consists in allowing the union of some of the RSGs which fulfill certain conditions. After the union, the resulting RSG should represent all the locations approximated by the original RSGs and the relevant shape information should be maintained.

More precisely, two graphs, rsg_1 and rsg_2, can be joined in a single one if they are compatible. Thus, we define $\texttt{COMPATIBLE}(rsg_1, rsg_2) = \text{true}$ if $\texttt{ALIAS}(rsg_1) = \texttt{ALIAS}(rsg_2)) \wedge \texttt{COMP\_NODES}(rsg_1, rsg_2) = 1$. We see that two graphs, rsg_1 and rsg_2, are compatible if they fulfill two conditions: i) the alias relation between pvars of both graphs are the same; and ii) certain nodes in both graphs are compatible. This leads us to define the alias relation between pvars and the compatibility condition between certain nodes in the graphs:

- $\texttt{ALIAS}(rsg)$ is the set of alias relations, alr_i, in the rsg graph, where each alr_i identifies all the pvars pointing to the same node n_i:

$\texttt{ALIAS}(rsg) = \{alr_1, ..., alr_n\}$ where $alr_i = \{pvar_1, .., pvar_m \in P\}$ if $\exists n_i \in N(rsg) | < pvar_1, n_i >, ..., < pvar_m, n_i > \in PL(rsg)$

- $\texttt{COMP\_NODES}(rsg_1, rsg_2)$ is a Boolean function which returns true if the nodes directly pointed to by the same pvar are compatible, which happens when they have similar properties (and therefore these nodes can be summarized). This function is formalized in two steps: the first one identifies the nodes from both RSGs, $n_j \in N(rsg_1)$ and $n_k \in N(rsg_2)$, which are pointed to by the same pvar; the second step determines whether these nodes have similar properties using an additional Boolean function `C_NODES`:

$\texttt{COMP\_NODES}(rsg_1, rsg_2) = \text{true}$ if $\forall pvar_i \in P,\ < pvar_i, n_j > \in PL(rsg_1) \wedge < pvar_i, n_k > \in PL(rsg_2) \wedge \texttt{C\_NODES}(n_j, n_k) = 1$.

where $\texttt{C\_NODES}(n_1, n_2) = \text{true}$ if $(\texttt{TYPE}(n_1) = \texttt{TYPE}(n_2)) \wedge (\texttt{SHARED}(n_1) = \texttt{SHARED}(n_2)) \wedge (\texttt{SHSEL}(n_1, sel_i) = \texttt{SHSEL}(n_2, sel_i) \forall sel_i \in S) \wedge (\texttt{TOUCH}(n_1) = \texttt{TOUCH}(n_2)) \wedge (\texttt{C\_REFPAT}(n_1, n_2) = 1) \wedge (\texttt{C\_SPATH}(n_1, n_2, m) = 1)$.

We can see here that `C_NODES_RSG` and `C_NODES` are quite similar, but they differ in that the latter does not check the `STRUCTURE` property. This is because of the excessive complexity involved in checking this property for all the nodes of two different RSGs.

In the following subsections, we describe the operations that can be carried out with the RSGs of an RSRSG. In the worst case, the sequence of operations that the analyzer carries out in order to symbolically execute a statement, as we have seen in Fig. 3, are: graph division, graph prune, statement symbolic execution (RSG modification), RSG compression, and RSG union to build the final RSRSG. In fact, the RSG compression has already been presented in Sect. 3.2, so we focus on the other operations.

4.1 Graph division

The division operation takes place for the $x \rightarrow sel = NULL$, $x \rightarrow sel = y$ and $y = x \rightarrow sel$. The goal of this operation is to split the input rsg into several graphs, such that for each one of these graphs, the node directly pointed to by x points to a single node by selector sel. Basically, with the division, we try to recover the individual characteristics of memory configurations that were fused into a single RSG in a previous statement. Going back to Fig. 2 (a) and (b), we can see how the rsg is divided into graphs $rsg\prime_1$ and $rsg\prime_2$.

The original graph is presented in Fig. 2(a). Note that the node directly pointed to by x, n_1, points to two different nodes by selector nxt. Therefore, we divide this graph into two different ones in such a way that in each resulting graph, n_1 points to a single node by nxt. We see these two graphs in Fig. 2(b), where in $rsg\prime_1$ the node n_1 only points to node n_2 and in $rsg\prime_2$ this node only points to node n_3.

We define $\texttt{DIVIDE}(rsg, x, sel) = \{rsg_1, .., rsg_n\}$ which divides the rsg in the set $\{rsg_1, .., rsg_n\}$ regarding the pvar x and selector sel. This division is carried out in the following way. If $n \in N(rsg) | < x, n > \in PL(rsg)$, then, $\forall < n, sel, n_i > \in NL(rsg)$, we create a $rsg\prime_i$ such that $N(rsg\prime_i) = N(rsg)$, $PL(rsg\prime_i) = PL(rsg)$ and $NL(rsg\prime_i) = NL(rsg) \setminus \{< n, sel, n_j > \in NL(rsg), \forall n_j \neq n_i\}$. Each $rsg\prime_i$ can contain a single node n_i pointed to by n by selector sel.

4.2 Graph pruning

After graph division, there can be nodes or links which are not compliant with the new properties of these graphs. For example, if a node n belongs to an rsg_1 (which results from the division of rsg) in which $\texttt{SELIN}(n, sel) = 1$ there should be a node n_i pointing to n by selector sel ($< n_i, sel, n > \in NL(rsg_1)$). If such a node, n_i, cannot be found we know that node n does not belong to this rsg_1 and therefore the node and all its links can be removed from this graph. Actually, this node n will be present in other RSGs resulting from the graph division and in some of these RSGs the node will fulfill the SELIN property.

In our previous example we can see how the two divided graphs of Fig. 2(b) are simplified into the graphs presented in Fig. 2(c). Note that rsg''_1 is obtained after the pruning of rsg'_1, in which we can safely remove the link $< n_3, prv, n_1 >$ due to it not fulfilling the cycle link properties of node n_3. This property states that subsequently following prv and nxt from node n_3, this n_3 should be reached, but this does not happen for the above-mentioned link. Regarding rsg''_2, note that the same happens for the link $< n_2, prv, n_1 >$. Besides this, because node n_3 is not shared by selector nxt and we are sure that $< n_1, nxt, n_3 >$ exists, we can conclude that $< n_2, nxt, n_3 >$ should be removed. This implies the elimination of $< n_3, prv, n_2 >$ due to cycle link properties. After this elimination, node n_2 cannot be reached and is therefore removed from rsg''_2. In our implementation, these latter steps actually take place during the node materialization included in the abstract interpretation. In any case, it is important to note that the false value in share attributes leads to a more aggressive pruning which simplifies the RSRSGs and greatly contributes to avoid an explosion in the number of nodes.

To formalize the pruning process, our method uses two Boolean functions to determine whether a node, `N_PRUNE`(n), or a link, `NL_PRUNE`$(< n_1, sel, n_2 >)$, fulfill the graph properties:

• A certain node, n, is removed from the graph only taking into account the reference pattern property. That is, if the `SELIN/SELOUT` functions assert that the node is referenced by selector sel or that this node references by sel to another node, these conditions should be hold. In other cases, the node should be eliminated from the graph. More formally

`N_PRUNE`(n) = true if $(\exists sel_i \in$`SELOUTset`$(n) \ \wedge \ sel_i \notin$`posSELOUTset`$(n) \ \wedge \ \nexists n_2 \in N(rsg), < n, sel_i, n_2 > \in NL(rsg)) \ \vee \ (\exists sel_i \in$`SELINset`$(n) \ \wedge \ sel_i \notin$`posSELINset`$(n) \ \wedge \ \nexists n_2 \in N(rsg), < n_2, sel_i, n > \in NL(rsg))$.

• On the other hand, the link restrictions arise due to the `CYCLELINKS` property. For example, let's assume a certain node, n, has a set of `CYCLELINKS` which comprises this particular one: $< sel_1, sel_2 >$. This cycle link points out that the memory locations represented by the node points to others by selector sel_1, and those ones point to the original locations by selector sel_2. Therefore, in our example, if the node n_2 pointed to by n_1 does not point again to n_1 using selectors sel_1 and sel_2, respectively, we can safely remove the link $< n_1, sel_1, n_2 >$. Formally

`NL_PRUNE`$(< n_1, sel_i, n_2 >)$ = true if $\exists < sel_i, sel_j > \in$ `CYCLELINKS`$(n_1) \mid \ < n_2, sel_j, n_1 > \notin NL(rsg)$.

Additionally, we should note that this pruning is an iterative process, because after the elimination of some nodes or links there may appear more nodes or links which do not fulfill the rules and should be removed too. This way, the pruning process ends when all the nodes and links fulfill the graph properties. The whole process can be expressed as $\texttt{PRUNE}(rsg) = rsg_n$. This iterative function starts with $rsg_0 = rsg$. Next, $\forall i = 1..n$:

$N(rsg_i) = N(rsg_{i-1}) \setminus \{n \in N(rsg_{i-1}) | \texttt{N\_PRUNE}(n) = 1\}$; $PL(rsg_i) = PL(rsg_{i-1}) \setminus \{< pvar, n > \in PL(rsg_{i-1}) | \texttt{N\_PRUNE}(n) = 1\}$ and $NL(rsg_i) = NL(rsg_{i-1}) \setminus \{< n_1, sel, n_2 > \in NL(rsg_{i-1}) | (\texttt{N\_PRUNE}(n_1) = 1) \vee (\texttt{N\_PRUNE}(n_2) = 1) \vee (\texttt{NL\_PRUNE}(< n_1, sel, n_2 >) = 1)\}$
where for each iteration we remove the nodes and links for which functions N_PRUNE and NL_PRUNE return true. The process ends when we reach the graph rsg_n which holds $\forall n \in N(rsg_n), \texttt{N\_PRUNE}(n) = 0 \wedge \forall < n_1, sel, n_2 > \in NL(rsg_n), \texttt{NL\_PRUNE}(< n_1, sel, n_2 >) = 0$.

4.3 Graph union

Two compatible graphs rsg_1 and rsg_2, ($\texttt{COMPATIBLE}(rsg_1, rsg_2) = 1$), can be fused in a single graph, rsg, which captures the data structure information represented by the two original graphs. This union of graphs is carried out by the $\texttt{JOIN}(rsg_1, rsg_2) = rsg$ function which builds the new graph, rsg, from the original ones, rsg_1 and rsg_2. This function does not modify the pvars set, P, nor the selectors set, S. On the contrary, the nodes set, N, and link sets, PL and NL, should be updated.

In particular, some of the nodes of rsg_1 and rsg_2 are going to be summarized if they are compatible. Now, using the function MERGE_NODES described in Sect. 3.2 we can describe the sets N, PL, and NL of the new RSG, resulting from the union of rsg_1 and rsg_2:

• The set of nodes, N, for the new graph, rsg, comprises three subsets: the non-compatible nodes from rsg_1, the non-compatible nodes from rsg_2, and the nodes resulting from the union of compatible nodes (MERGE_NODES):

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

We define the $\texttt{MAP}(n_i)$ function which points out which node of the new graph rsg is now representing a certain node of the rsg_1 or rsg_2. By using this MAP function it is easy to describe the new $PL(rsg)$ and $NL(rsg)$ sets.

• The set of references from pvars to nodes $PL(rsg)$ are obtained by translating the old references from rsg_1 and rsg_2 to the new graph using the MAP function.

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

• Similarly, we obtain the set of links between nodes:

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

This way, in the new graph, rsg, we keep all the references and links existing in the original graphs, rsg_1 and rsg_2, just changing the source and destination nodes for the corresponding ones of the new graph using the `MAP` function. Thus, the new rsg resulting from the union of two compatible graphs has been completely defined. We emphasize here that due to this RSG union we can save a great amount of memory space, but at the same time we enable the representation of several memory configurations (which are not completely equal) with the same RSG.

5 Experimental results

All the previously described operations and properties have been implemented in an analyzer written in C which can be fed with an input code to generate the RSRSG associated with each statement of the code. The codes have to be preprocesed in a first step to just keep the statements dealing with pointers. In addition, since our analyzer only considers the six pointer statements described in Sect. 2, the preprocessor has to translate other complex pointer statements into the allowed ones.

Furthermore, before the symbolic execution of the code, the preprocessor can also extract some important information from the program in a previous pass. For example, a quite frequent pattern arising in C codes based on dynamic data structures is: `while (x != NULL)` $\{\ldots\}$. In this case the analyzer can assert that at the entry of the while body the pvar $x \neq NULL$. Besides this, if we have not exited the while body with a `break` statement, we can also ensure that just after the while body the pvar $x = NULL$. This information is used to simplify the analysis and increase the accuracy of the method. More precisely, we can reduce the number of RSGs and/or reduce the complexity of this RSG by diminishing the number of memory configurations represented by each RSG. Other statements from which we can extract useful information are `IF-THEN-ELSE`, `FOR` loops, or any conditional statement.

The implementation of this idea has been carried out by the definition of certain *pseudoinstructions* that we call `FORCE`. These pseudoinstructions are inserted in the code by the preprocessor of the analyzer and will be symbolically executed as a regular statement. Therefore, each one of these `FORCE` statements has its own abstract semantics and its own associated RSRSG. The `FORCE` pseudoinstructions we have taken into account are: $\texttt{FORCE}_{[x==NULL]}(rsg)$, $\texttt{FORCE}_{[x!=NULL]}(rsg)$, $\texttt{FORCE}_{[x==y]}(rsg)$, $\texttt{FORCE}_{[x!=y]}(rsg)$, $\texttt{FORCE}_{[x\rightarrow sel==NULL]}(rsg)$.

In addition, we have found that the $\texttt{FORCE}_{[x!=NULL]}$ pseudoperation can be also placed just before the following statements: $x \rightarrow sel = NULL$, $x \rightarrow sel = y$ y $y = x \rightarrow sel$, under the assumption that the code is correct. That is, it makes sense to assume that before the execution of all these three statements, x is not NULL (in other cases the code would produce an error at execution time). The same can be assumed for any statement with an occurrence of the type $x \rightarrow val$, where val is a non-pointer field of the structure pointed to by x. All this preprocessing is now done by hand but we plan to have an automatic preprocessor soon.

5.1 Progressive analysis

As we have seen, the set of properties associated with a node allows the analyzer to keep in separate nodes those memory locations with different properties. Obviously, the number of nodes in the RSRSGs depends on the number of properties and also on the range of values these properties can take. The higher the number of properties the better the accuracy in the memory configuration representation, but also the larger the RSRSGs and memory wastage.

Fortunately, not all the properties are needed to achieve a precise description of the data structure in all the codes. That is, simpler codes can be successfully analyzed taking into account fewer properties, and complex programs will need more compilation time and memory due to all the properties have to be considered. Bearing this in mind, we have implemented the analyzer to carry out a progressive analysis which starts with fewer constraints to summarize nodes, but, when necessary, these constraints are increased to reach a better approximation of the data structure used in the code.

More precisely, the analysis comprises three levels: L_1, L_2, and L_3, from less to more complexity as we explain next:

- L_1: In this level the `TOUCH` sets are not built nor taken into account and only the `C_SPATH0` condition is used.
- L_2: This level is based on the previous one but now using the `C_SPATH1`.
- L_3: This is the higher level in which all the properties including the `TOUCH` one are taken into account.

	Working Ex.	S.Mat-Vec	S.Mat-Mat	S.LU	Barnes-Hut
Level	L_1 / L_2 / L_3	L_1 / L_2 / L_3	L_1 / L_2 / L_3	L_1	L_1 / L_2 / L_3
Time	0'03"/0'05"/0'06"	0'01"/0'02"/0'03"	0'20"/0'38"/1'00"	7'50"	5'56"/0'34"/2'06"
MBytes	2.11/2.78/3.02	1.37/1.85/2.17	8.13/11.45/12.68	99.46	37.82/8.82/8.94
Lines	213	104	156	164	216

Table 1: Time and space required by the analyzer to process several codes with different number of code lines.

With this tool we have analyzed several codes: the one described in Sect. 2.1 (working example), the sparse Matrix by vector multiplication, the sparse Matrix by Matrix multiplication, the Sparse LU factorization, and the Barnes-Hut code. The first four codes were successfully analyzed in the first level of the analyzer, L_1. However, for the Barnes-Hut program the highest accuracy of the RSRSGs was obtained in the last level, L_3, as we explain in Sect. 5.4. All these codes where processed by our analyzer in a Pentium 4 1.6 GHz with 128 MB main memory. The time and memory required by the analyzer are summarized in Table 1. In this table we also show the number of code lines after the preprocessing of the original C codes. The particular aspects of these codes are described next.

5.2 Working example's RSRSG

We refer in this subsection to the code that generates, traverses, and modifies the data structure presented in Fig. 4 (a). A compact representation of the resulting RSRSG for the last statement of the code can be seen in Fig. 4 (b). Although we do not show the code due to space constraints, we have to say that this code presents an additional difficulty due to some tree permutations being carried out during data structure modification. The problem arising during structure permutation is that it is very easy to temporally assign the `SHARED`=true property to the root of one of the trees that we are permutating, when this root is temporally pointed to by two different locations from the header list. If this shared property remains true after the permutation we would have a shaded n_4 node in Fig. 4 (b). This would imply that two different items from the header list can point to the same tree (which would prevent the parallel execution of traversing the trees). However, this problem is solved because, after the permutation, the method reassigns false to the shared property thanks to the combination of our properties and the division of graph operations. Summarizing, after the analysis of this code, the compact representation of the resulting RSRSG for the last statement of the program (Fig. 4 (b)) accurately describes the data structure depicted in Fig. 4 (a) in the sense that: (i) The analyzer successfully detects the doubly linked list which is acyclic by selectors nxt or prv and whose elements point to binary trees; (ii) As SHSEL($n_4, tree$)=0, we can say that two different items of the header list cannot point to the same tree; (iii) At the same time, since no tree node (n_4, n_5 and n_6) is shared, we can say that different trees do not share items; (iv) The same happens for the doubly linked list pointed to by the tree leaves: all the lists are independent, there are no two leaves pointing to the same list, and these lists are acyclic by selectors nxt or prv.

Besides this, our analyzer has also processed four codes which generate, traverse, and modify complex dynamic data structures which we describe next.

5.3 Sparse matrix codes

Here we deal with some irregular codes which implements sparse matrix operations: the sparse matrix by vector multiplication, $r = M \times v$; the sparse matrix-matrix multiplication, $A = B \times C$; and the sparse LU factorization, $A = LU$.

The sparse matrices are stored in memory as a header doubly linked list with pointers to other doubly linked lists which store the matrix rows (if the matrix is row-wise) or columns (for column-wise matrices). The sparse vectors, v and r are also doubly linked lists. In Fig. 5(a) we can see the main data structures for the matrix vector multiplication code.

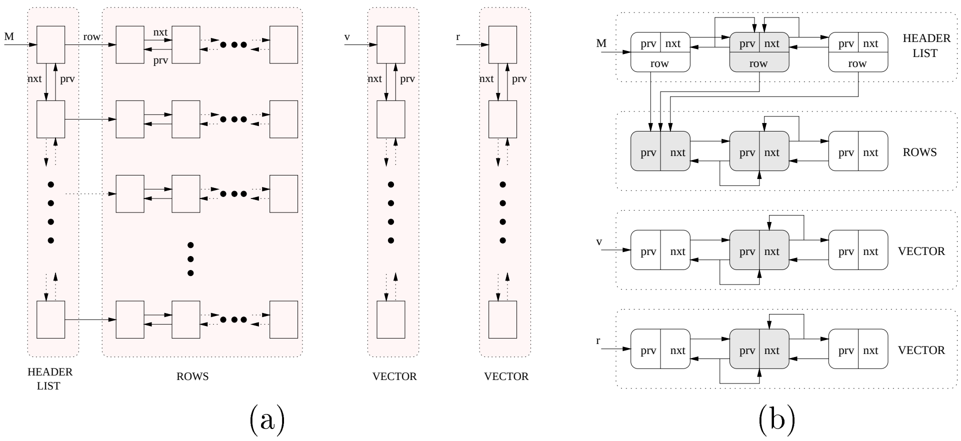


Figure 5: Sparse matrix-vector multiplication data structure and compacted RSRSG.

After the analysis process, carried out by our analyzer, the resulting RSRSG accurately represents the data structures. Actually, in Fig. 5(b) we present a compact representation of the resulting RSRSG for the last statement of the matrix vector code. First, note that the three structures involved in this code are kept in separate subgraphs. Even when the data type for vectors v and r and rows of M, is the same, the `STRUCTURE` property avoids the union of these graphs into a single one. This RSRSG states that the rows of the matrix are pointed to from different elements of the header list (there is no selector with the shared property set to true). Also, the doubly linked lists which store the rows of M and the vectors v and r are acyclic by selectors nxt and prv. Regarding the sparse matrix matrix multiplication, a similar RSRSG is also obtained, but instead of one matrix and two vectors representation we have three different matrix graphs.

On the other hand, the sparse LU factorization code solves non-symmetric sparse linear systems by applying the LU factorization of the sparse matrix. Here, the sparse matrix is a list of pointers to sparse columns. However, this code is much more complex to analyze due to the partial pivoting and column permutation which takes place in the factorization in order to provide numerical stability and preserve the sparseness. The compact representation of the corresponding RSRSG represents a sparse matrix like matrix M in Fig. 5(b). In this case, a pointer variable A (instead M) points to a doubly linked list, the header list. Each node of this list points to a single doubly linked list which represents a matrix column. Due to `SHSEL` being false for all selectors we conclude that the header list and the column lists are acyclic structures when they are traversed by a single selector type and that different elements of the header list should point to different columns. A subsequent analysis of the code and the RSRSG associated with each statement would be able to state that several sparse matrix columns can be updated in parallel during factorization and, in addition, it is also possible to update each column in parallel.

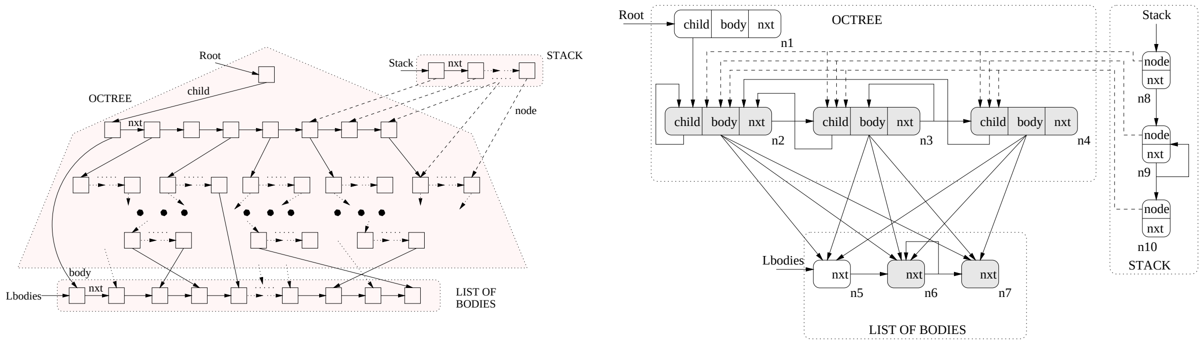
5.4 Barnes-Hut N-body simulation

This code deserves more attention due to it being a good example in which the analyzer has to sequentially carry out the three levels of compilation in the progressive analysis.

This code is based on the algorithm presented in [2] which is used in astrophysics. In fact, this application simulates the evolution of a system of bodies under the influence of gravitational forces. It is a classical gravitational N-body simulation, in which each body is modeled as a point mass. The simulation proceeds over time-steps, each step computing the net force on every body and thereby updating that body's position and other attributes. The data structure used in this code is based on a hierarchical octree representation of space in three dimensions.

In Fig. 6(a) we present a schematic view of the data structure used in this code.

The bodies are stored by a single linked list pointed to by the pvar *Lbodies*. The octree represents the several subdivisions of the 3D space. That is, the root of the tree represents the whole space, each one of its children represents a single subsquare of this space, and so on. This way, each leaf of the octree represents a subsquare which contains a single body and therefore points to this body stored in the *Lbodies* list. Each octree node which is not a leaf has a pointer *child* pointing to the first of its eight children which are linked by selector *next*.



(a) (b)

Figure 6: Barnes-Hut data structure and compacted RSRSG.

The three main steps in the algorithm are: (i) The creation of the octree and the pointers from the elements of the octree to the elements of the *Lbodies* list; (ii) for each subsquare in the octree, compute the center of mass and total mass for all the particles it contains; and (iii) for each particle, traverse the tree to compute the forces on it.

All the traversals of the octree are carried out in the code in recursive calls. Due to the fact that our analyzer is still not able to perform an interprocedural analysis, we have manually carried out the inline of the subroutine and the recursivity has been transformed into a loop. This loop uses a stack pointing to the nodes which are referenced during the octree traversal. This stack is also considered in Fig. 6 (a) and obtained in the corresponding RSRSG, Fig. 6 (b).

After the L_1 analysis of the code, the resulting RSRSG for the last statement of the code does not correspond with the real properties of the data structure used in the code. The problem is that the summary node n_6, which represents the middle elements of the *Lbodies* list fulfill `SHSEL`(body) = true. This would imply that there may be different leaves in the octree pointing to the same body in the *Lbodies* list. This inaccuracy is due to an imprecise analysis during the generation of the list of children in the octree.

This inaccuracy is avoided moving on to the L_2 level, thanks to the use of `C_SPATH1`.

The resulting RSRSG can be seen in Fig. 6(b), where the summary node n_6 fulfills $\texttt{SHSEL}(n_6, body)$ = false, in line with the real data structure. However, due to the stack we use to assist the octree traversal, there is a problem that cannot be solved in L_2 for the (iii) step of the algorithm: nodes n_2, n_3, and n_4 are shared by selector *node* from the *Stack* data structure. This would prevent the parallel traversal of the octree and does not fit with the real data structure. The lack of accuracy is now due to the fact that, during the traversal of the octre, visited nodes are summarized with non-visited ones.

However, by switching to the L_3 compilation level this problem is solved thanks to the TOUCH property which keeps in separated summary nodes visited and non-visited nodes. A subsequent analysis of the code can state that the tree can be traversed and updated in parallel.

However, regarding the Table 1, there is a paradoxical behavior that deserves an explanation: L_2 and L_3 expend less time and memory than L_1. As was briefly described in the example in Sect. 4.2, when SHARED and SHSEL are false there are more nodes and links pruned during the abstract interpretation and graph compression phase of the symbolic execution of a statement. In this code, for the L_2 and L_3 levels, the $\texttt{SHSEL}(n_6, body)$ = false leads to more pruning and graph simplifications counteracting the increase in complexity due to the use of the C_SPATH1 and TOUCH properties.

6 Related works

There are several ways the shape analysis problem can be approached. The simplest strategy consists in asking the programmer to annotate the sequential code, so helping the analyzer with this extra information [14] following a semiautomatic approach. On the other hand, user interaction is avoided in many studies, such as the ones based on "access paths" or on graphs. For example, the method proposed by Matsumoto et al. [18] uses "normalized" path expressions to maintain the "alias-pair" between pointers. However, we discarded this set of methods as they cannot handle cyclic structures such as double linked lists or trees with pointers from leaves to parents.

On the other hand, in the graph-based methods the "storage chunks" are represented by nodes, and edges are used to represent references between them. For example, Chase et al. [3] define the "Storage Shape Graph" which contains one node for each variable and one for each allocation site in the program. With this abstraction their method can detect a single linked list even when new items are appended to the end of the list. However, this method is not powerful enough to detect insertion of elements in the middle of the list.

Plevyak et al. [19] try to solve Chase's problem by extending the "Storage Shape Graphs" to the "Abstract Storage Graph" (ASG). However, the improvements in accuracy are paid for with too much complexity in comparisons and compression operations. On the other hand, the method presented by Sagiv et al. [21] is based on what they call "Static Shape Graphs" (SSG). The main difference between this method and the previous ones lie in the node-name scheme they use for the nodes, where all the memory locations not directly pointed to by a pointer variable (pvar) are fused into a single summary node.

In a previous work [6] we saw that ASG or SSG were not sufficient to deal with the complex data structures we presented in the previous section. In that way, we combined and extended Plevyak and Sagiv's methods allowing for more than a summary node per graph among other extensions. However, we keep the restriction of one graph per statement in the code. This way, since each statement of the code can be reached after following several paths in the control flow, the associated graph should approximate all the possible memory configurations arising after the execution of this statement. This restriction leads to memory and time savings, but at the same time it significantly reduces the method's accuracy. Since we are mainly concerned with the precision achieved by our method, in the current work we have changed our previous direction by selecting a tradeoff solution: we consider several graphs with more than a summary node, while fulfilling some rules to avoid an explosion in the number of graphs and nodes in each graph.

Among the first relevant studies which allowed several graphs per statement were those developed by Jones et al. [16] and Horwitz et al. [13]. These approaches are based on a "k-limited" approximation in which all nodes beyond a k selectors path are joined in a summary node. The main drawback to these methods is that the node analysis beyond the "k-limit" is very inexact and therefore they are unable to capture complex data structures. A more recent work that also allows several graphs and summary nodes is the one presented by Sagiv et al.[22]. They propose a parametric framework based on a 3-valued logic. To describe the memory configuration they use 3-valued structures defined by several predicates. These predicates determine the accuracy of the method. However, as far as we know the currently proposed predicates do not suffice to deal with the complex data structures that we handle in this paper.

There are three main differences between our shape analysis method and that of Sagiv et al. [22]. The first is that in our work we consider important properties, such as the "reference patterns" and "touch information", among others, which lead to a more precise description of the data structure used in the code. Second, we join similar reference shape graphs (RSGs) to build a reduced set of RSGs for each program point, while in [22] they keep all the graphs. We think that this may explain why their Three-Valued-Logic Analyzer (TVLA) runs out of memory for simple codes such as the singly linked list insert sort and bubble sort using the multiple structure approach [17]. Third, they recognize that their TVLA engine is only useful to analyse small programs and report experimental results for small, singly linked list operations (insert, reverse, sort, etc.). However, they have not published experimental results successfully dealing with real codes based on the combination of complex data structures such as doubly linked lists pointing to trees or to other lists, etc. Besides, it is probably unfair to compare their Java-written TVLA running on a Pentium II-400MHz with our C-written analyzer on a Pentium III-500MHz, but to give an idea of performance we can point out that in [17] they report 186 sec. to analyse the bubble sort routine using the single structure approach. Our analyzer precisely captures the memory configuration at each statement of this routine in less than 3 sec. In summary, and to the best of our knowledge, our analyzer is the only one able to accurately identify the complex data structures described in the previous section.

7 Conclusions and future work

We have developed an analyzer which can be fed with a code to determine the RSRSG associated with each statement of the code. Each RSRSG contains several RSGs, each one representing the different data structures which may arise after following different paths in the control flow graph of the code. However, several RSGs can be joined if they represent similar data structures, in this way reducing the number of RSGs associated with a statement. Every RSG contains nodes which represent one or several memory locations. To avoid an explosion in the number of nodes, all the memory locations which are similarly referenced are represented by the same node. This reference similarity is captured by the properties we assign to the memory locations. In comparison with previous works, we have increased the number of properties assigned to each node. This leads to more nodes in the RSG because the nodes now have to fulfill more properties to be summarized. However, by avoiding the summarization of these nodes, we keep a more accurate representation of the data structure. This is a key issue when analyzing the parallelism exhibited by a code.

Our analyzer symbolically executes each statement in the code, transforming the RSGs to reflect the modifications in the data structure that are carried out due to the execution of the statement. We have validated the analyzer with several codes which generate, traverse, and modify complex dynamic data structures, such as a doubly linked list of pointers to trees where the leaves point to other doubly linked lists. Other structures, such as a list of lists or the n-body data structure, have been also accurately identified by the analyzer, even in the presence of structure permutations (for example, column permutations in the sparse LU code). As far as we know, there is no analyzer achieving such successful results for these kinds of data structures appearing in real codes. We are currently optimizing the analyzer implementation to further reduce the analysis time and memory requirements presented in table 1.

In the near future we will develop an additional analyzer pass that will automatically analyze the RSRSGs and the code to determine the parallel loops of the program and allow the automatic generation of parallel code.

References

[1] L. Andersen. *Program analysis and specialization for the C programming language.* PhD thesis, DIKU, University of Copenhagen, May 1994. DIKU report 94/19.

[2] J. Barnes and P. Hut. A hierarchical O(n· log n) force calculation algorithm. *Nature*, 324, December 1986.

[3] D. Chase, M. Wegman, and F. Zadek. Analysis of pointers and structures. In *SIGPLAN'90 Conference on Programming Language Design and Implementation*, pages 296–310, 1990.

[4] T. M. Chilimbi. Efficient representations and abstractions for quantifying and exploiting data reference locality. In *ACM SIGPLAN '01 Conference on Programming Language Design and Implementation (PLDI)*, 2001.

[5] T. M. Chilimbi, M. D. Hill, and J. R. Larus. Cache-conscious structure layout. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 1–12, 1999.

[6] F. Corbera, R. Asenjo, and E. Zapata. New shape analysis for automatic parallelization of c codes. In *ACM International Conference on Supercomputing*, pages 220–227, Rhodes, Greece, 1999.

[7] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Fourth ACM Symposium on Principles of Programming Language*, pages 238–252, Los Angeles, California, 1977. ACM Press, New York.

[8] M. Das. Unification-based pointer analysis with directional assignments. *ACM SIGPLAN Notices*, 35(5):35–46, 2000.

[9] R. Ghiya. *Putting Pointer Analysis to Work.* PhD thesis, School of Computer Science, McGill University, Montreal, May 1998.

[10] R. Ghiya and L. Hendren. Is it a tree, a dag, or a cyclic graph? a shape analysis for heap-directed pointers in C. In *23rd ACM Symposium on Principles of Programming Languages*, pages 1–15, 1996.

[11] R. Ghiya and L. Hendren. Putting pointer analysis to work. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 121–133, 1998.

[12] M. Hind and A. Pioli. Assessing the effects of flow-sensitivity on pointer alias analyses. In *Static Analysis Symposium*, pages 57–81, 1998.

[13] S. Horwitz, P. Pfeiffer, and T. Reps. Dependence analysis for pointer variables. *ACM SIGPLAN Notices*, 24(7):28–40, 1989.

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

[15] Y. Hwang and J. Saltz. Identifying DEF/USE information of statements that construct and traverse dynamic recursive data structures. In *10th International Workshop on Languages and Compilers for Parallel Computing*, pages 131–145, 1997.

[16] N. Jones and S. Muchnick. *Flow Analysis and Optimization of Lisp-like Structures*, chapter Flow Analysis: Theory and Applications,Chapter 4, pages 102–131. Prentice Hall, 1981.

[17] T. Lev-Ami and M. Sagiv. TVLA: A system for implementing static analyses. In *Static Analysis Symposium*, pages 280–301, 2000.

[18] A. Matsumoto, D. Han, and T. Tsuda. Alias analysis of pointers in Pascal and Fortran 90: Dependence analysis between pointer references. *Acta Informatica*, 33:99–130, 1996.

[19] J. Plevyak, A. Chien, and V. Karamcheti. Analysis of dynamic structures for efficient parallel execution. In *Languages and Compilers for Parallel Computing*, pages 37–57, Berlin, 1993. Sprienger-Verlag.

[20] A. Rogers, M. C. Carlisle, J. H. Reppy, and L. J. Hendren. Supporting dynamic data structures on distributed-memory machines. *ACM Transactions on Programming Languages and Systems*, 17(2):233–263, March 1995.

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

[22] M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. In *Symposium on Principles of Programming Languages*, pages 105–118, 1999.

[23] M. Shapiro and S. Horwitz. Fast and accurate flow insensitive points-to analysis. In *Symposium on Principles of Programming Languages*, pages 1–14, 1997.

[24] B. Steensgaard. Points-to analysis in almost linear time. In *23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 32–41, St. Petersburg, Florida, 1996.

[25] R. Wilson and M. Lam. Efficient context-sensitive pointer analysis for C programs. In *ACM SIGPLAN'95 Conference on Programming Language Design and Implementation*, pages 18–21, La Jolla, California, 1995.

[26] Y. Zhu and L. Hendren. Locality analysis for parallel C programs. *IEEE Transactions on Parallel and Distributed Systems*, 10(2), 1999.

[27] Y. Zhu and L. J. Hendren. Communication optimizations for parallel C programs. *Journal of Parallel and Distributed Computing*, 58(2):301–332, 1999.