New shape analysis and interprocedural techniques for automatic parallelization of C codes * †

F. Corbera R. Asenjo E.L. Zapata

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

Abstract

Automatic parallelization of codes with complex data structures is becoming very important. These complex, and often recursive, data structures are widely used in scientific computing. Shape analysis is one of the key steps in the automatic parallelization of such codes. In this paper we extend the Static Shape Graph (SSG) method to enable the successful and accurate detection of complex doubly-linked structures. We have also included an interprocedural analysis in our framework. These techniques have been implemented in a compiler, which has been validated for several C codes. In particular, we present the results the compiler achieves for the C sparse LU factorization algorithm and for a recursive code generating a tree data structure. The output SSG perfectly describes the complex data structure used in these codes.

1 Introduction

Regarding high performance computing, it is clear that compilers represent a key tool that should take care of optimizing the applications that must be executed efficiently in parallel computers. A good deal of work has been done in the area of array dependence analysis [2] with notable success. However, non-numerical and numerical applications based on complex and dynamic data structures are becoming more and more widely used lately. These complex, and often recursive, data structures are based on dynamic allocation and references.

To successfully optimize these applications, a fundamental compiler task is the analysis of dynamic structures which are generated at execution time. Parallelization of any application requires the compiler's special knowledge about the underlying semantic of the data structure. With these assumptions, shape analysis becomes a first step in the data dependence test for such kinds of codes. The aim of this phase is to find out at compile time the shape of the heap.

In this work we present some important modifications to the shape analysis method developed by Sagiv et al. [13]. On the other hand, we have also extended our framework to deal with interprocedural analysis. The relevance of this analysis is justified due to procedures with recursive calls appear frequently in C codes, mainly when managing recursive data structures. In addition, we check our improvements with a real numerical application, in particular the non-symmetric sparse LU decomposition based on a one-dimensional doubly-linked list, and with a recursive program generating a tree.

The organization of the paper is as follows. In the next section we revise the approaches currently available to solve shape analysis problems. Section Three presents the motivating example. Our shape

*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

†This paper is an extension of the one presented in the ACM International Conference on Supercomputing, Rhodes, Greece, June 1999 [4]

analysis techniques are described in Section Four. Finally, the interprocedural analysis is presented in Section Five. Conclusions and future work close the paper.

2 Related Work

There are several methods addressing the shape analysis problem. Some of these are based on explicit annotations, as in Hummel et al. [8]. These methods are based on programmer annotations describing the data structure.

Other approximations are based on access paths. Hendren et al. [6] use "path matrix analysis" that contains "access paths" between pointers. Matsumoto et al. [11] use "normalized" path expressions to maintain the "alias-pair" between pointers. These methods cannot handle cyclic structures like double linked lists and trees with parent pointers.

Finally, there are methods based on graphs. In the graph, the nodes represent "storage chunks", and the edges represent references between them.

One of the first relevant works on this topic was developed by Jones et al. [9]. In this work, the authors focus on the shape analysis of programs with destructive updating. They bind, to each program point, a set of graphs which describe all potential alias relationships that can arise at execution time. In addition, they use a "k-limited" approximation in which all nodes beyond a k selectors path are joined in a summary node. Horwitz et al. [7] presented another variation on k-limited graphs, called "storage graphs". Also, the authors maintain a set of storage graphs at each program point. The main drawbacks of these methods are: (1) the number of shape graphs that can arise for each program point is very high, leading to a great deal of computational and memory overhead; and (2) the node analysis beyond the "k-limit" is very inexact.

On the other hand, there are approximations in which each program point has an associated graph which covers all possible shape graphs combinations, instead of all these possible graphs independently [3, 10, 12, 13]). The result of joining all the information, previously represented by different shape graphs, in a single one, is a lack of accuracy in the representation, but on the other hand, it leads to a practical shape analysis algorithm. Larus et al. [10], use a variation of "k-limited" graphs called "alias graphs", and introduce summary nodes using "s-l limiting". This method works well only for simple data structures like trees and lists. It is expensive due to its complex meet, node summarization and node labeling operations.

The algorithm presented by Chase et al. [3] is not "k-limited". Their abstraction "Storage Shape Graph" contains one node for each variable and one for each allocation site in the program. In this method, the count of references from the heap to a node (0, 1, inf), is stored for each node in the graph. This way, the k-limited drawback is avoided. This algorithm is able to detect a single linked list even when new elements are appended to the end of the list. However, it is not powerful enough to detect insertions of elements in the middle of the list.

Plevyak et al. [12] work is based on the Chase's method. They extend the previous "Storage Shape Graph" into the "Abstract Storage Graph" (ASG) in order to solve the main problems arising in the first one. However, in the same way as Chase's method, their comparison and compression operations are complex and expensive.

The method presented by Sagiv et al. [13] is based on what they call "Static Shape Graphs" (SSG). The main difference between this method and previous ones lie in the node-name scheme they use for the nodes. Their graph contains nodes only for heap locations pointed to by program variables. Some very interesting properties are: (a) alias relationships are easier to find; (b) the determinism is better preserved in the graph due to they always carry out a "strong nullification" which is equivalent to a "strong update" (substitution of a reference by another one, without keeping the previous one).

In SSG, the union and comparison operations are very simple due to this node-naming scheme. For example, to know whether or not two nodes can be summarized, the compiler will just look at the name of the nodes, which actually state the set of pointer variables pointing to this node. On the contrary, in previous approaches it was necessary to look also at another node characteristics making the analysis

```
do k = 1, n
    Find pivot=A_kj
    if (j ≠ k)
        swap A(1 : n, k) and A(1 : n, j)
    endif
    A(k + 1 : n, k) = A(k + 1 : n, k)/A(k, k)
    do j = k + 1, n
        do i = k + 1, n
            A(i, j) = A(i, j) − A(i, k)A(k, j)
        enddo
    enddo
enddo
```

Figure 1: LU algorithm (General approach, right-looking version)

more complex.

However, the SSG method cannot analyze doubly-linked structures which are widely used in C codes, like Sparse LU factorization. In this case, the SSG is not able to accurately represent the data structure of the Sparse LU factorization. This data structure, called LLCS (Linked List Column Storage)[1], is presented in Fig. 2 (b). We can see that each column of the sparse matrix is represented by a doubly-linked list. In addition, a different doubly-linked list is needed to point to the first element of each list (column).

We propose combining the Sagiv's method [13] and the Abstract Storage Graph (ASG) proposed by Plevyak et al. [12] to achieve a more precise shape analysis of this type of structures. In the near future we will develop an additional compiler pass to automatically parallelize C codes with complex data structures.

The extended SSG proposed in this work introduces two main modifications:

- There will be several summary nodes in the SSG, allowing us to summarize different structure and type elements into different summary nodes, each one with its own properties.
- We include a "shared" attribute assigned to each selector, and keep additional information regarding pairs of selectors called "cycle links". With this modification we achieve a more accurate representation of the doubly-linked structures.

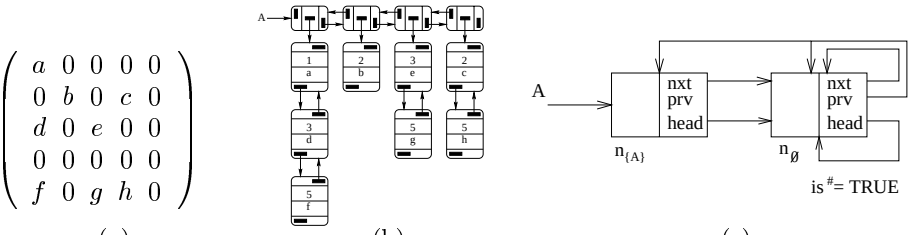
Besides the regular control flow structures (sequential, conditional and iterative structures), we have also taken into account the procedure and recursive call structures, with which we face the interprocedural analysis problem. We have followed a new approach, different from the one presented in [13], which we believe may produce more accurate SSGs.

3 Motivating example: sparse LU factorization

The kernel of many computer-assisted scientific applications is to solve large sparse linear systems. We find examples of these kinds of applications in optimization problems, linear programming, simulation, circuit analysis, fluid dynamic computation, and numeric solutions of differential equations in general.

Furthermore, this problem presents a good case study and is a representative computational code for many other irregular problems. Actually, this problem represents those in which the computational load grows with the execution time (fill-in) and matrix coefficients change their coordinates due to row/column permutations (pivoting).

More precisely, our working example application solves non-symmetric sparse linear systems by applying the LU factorization of the sparse matrix, computed by using a general method [1, 5]. These methods directly solve the sparse problem and share the same loop structure of the corresponding dense code (the one we see in Fig. 1).



(a) (b) (c)

Figure 2: (a) Sparse matrix. (b) LLCS data structure. (c) Sparse LU SSG

In this Fig. 1, we show an in-place code for the direct right-looking LU algorithm, where an n-by-n matrix A is factorized. The code includes a row pivoting operation (partial pivoting) to provide numerical stability and preserve the sparsity.

Usually, in order to save both memory and computation overhead, zero entries of sparse matrices are not explicitly stored. A wide range of methods for storing the nonzero entries of sparse matrices have been developed [5]. Here, we will consider only linked list data structures. The partial-pivoting LU decomposition stores the coefficient matrix in a one-dimensional doubly-linked list (see Fig. 2 (b)), to facilitate the insertion of new entries and to allow column permutations.

Analyzing the sparse LU algorithm with Sagiv's method, the resulting SSG is shown in Fig. 2 (c). Here, we can see that the same summary node, "$n_{\emptyset}$", refers to both the elements belonging to the header list and the ones in the column linked lists. This way it is impossible to discern between the two different data structures (header and columns).

Furthermore, the summary node is shared, ($is^{\#}$ = true), which means that the summarized nodes are referenced from the heap more than once, when actually, they are referenced by different selectors (nxt, prv). Therefore, there is no way to detect that no node is referenced twice by the same selector. Or in other words, the previous graph points out that the data structure may be cyclic and that there may be shared elements in different columns. This information, given in the SSG, prevents the compiler from automatically generating a parallel code that traverses and updates each column in parallel.

4 A modified shape analysis algorithm

This section focuses on the description of the new techniques which solve the previous problem. However, before this, it is necessary to briefly introduce the compiler preprocessing stage and the SSG notation used in [13].

Regarding the preprocessing pass, the compiler has to transform the C codes to fulfil the normalization assumptions according to the abstract semantic of the SSG method:

- Only one constructor or selector is applied per assignment statement.
- All allocation statements are of the form $x := new$ ($x.sel := new$ is not allowed).
- In each assignment statement, the same variable does not occur on both the left-hand and right-hand side.
- Each assignment statement of the form $lhs := rhs$ in which $rhs \neq nil$ is immediately preceded by an assignment statement of the form $lhs := nil$.

4.1 SSG Notation.

An SSG is a finite, labeled, directed graph that approximates the actual stores that can arise during program execution. The shape-analysis algorithm itself is an iterative procedure that computes an SSG at every program point.

An SSG, $SG^{\#}$, consists of two kinds of nodes, *pointer variables (PVar)* and *shape-nodes*, and two kinds of edges, *variable-edges* and *selector-edges*. An SSG is represented by a pair of edges sets, $< E_v^{\#}, E_s^{\#} >$, where:

- $E_v^{\#}$ is the graph's set of variable-edges, each of which is denoted by a pair of the form $[x, n]$, where $x \in PVar$ and n is a shape-node.
- $E_s^{\#}$ is the graph's set of selector-edges, each of which is denoted by a triple of the form $< s, sel, t >$ where s and t are shape-nodes, and sel is a selector.

Shape-nodes are named using a (possibly empty) set of pointer variables, X. The set $shape\_nodes(SG^{\#})$ is a subset of $\{n_X \mid X \subseteq PVar\}$. A shape-node n_X, where $X \neq \emptyset$, represents the cons-cell (storage chunk) pointed to by exactly the pointer variables in the set X, in any given concrete store. The shape-node $n_\emptyset$ (summary node) can represent multiple cons-cells of a single concrete store.

Each shape-node n in an SSG has an associated Boolean flag, denoted by $is^{\#}(n)$ (is shared). The equation $is^{\#}(n)$=$true$, indicates that the cons-cells represented by n may be the target of pointers emanating from two or more distinct cons-cells fields. On the other hand, $is^{\#}(n)$=$false$ means that, if several selector edges in an SSG point to n, they represent concrete edges that never point to the same cons-cell in any concrete store. The function $is^{\#}$ is therefore of type $shape\_nodes(SG^{\#}) \rightarrow \{false, true\}$.

Two different shape-nodes n_X and n_Y, such that $X \neq Y$ and $X \cap Y \neq \emptyset$, represent *incompatible* configurations of variables. Thus, for all $< n_X, sel, n_Y > \in E_s^{\#}$, either $X = Y$ or $X \cap Y = \emptyset$. The function $compatible^{\#}(n_{Z_1}, ..., n_{Z_k})$ means $\forall i, j : Z_i = Z_j \vee Z_i \cap Z_j = \emptyset$.

The "Abstract Interpretation" presents the modifications on an SSG that take place when executing the six kind of statements that manipulate pointer variables ($x := nil$, $x.sel := nil$, $x := new$, $x := y$, $x.sel := y$ and $x := y.sel$). $E_v^{\#\prime}$, $E_s^{\#\prime}$ and $is^{\#\prime}$ are $E_v^{\#}$, $E_s^{\#}$ and $is^{\#}$ after statement execution. The equivalent statements in C for the six kinds of statement presented before are: $x = NULL$, $x \rightarrow sel = NULL$, $x = allocate()$, $x = y$, $x \rightarrow sel = y$ and $x = y \rightarrow sel$. Control structures (if, loops, ..) also have their abstract interpretation but, since we closely follow the approach presented in [13], we will just provide, in section 4.5, a brief introduction of the compiler details to deal with these sentences.

With all these definitions we can move on to the main part of this section: the description of the new techniques we propose.

4.2 Several Summary Nodes

The SSG method [13] can contain only one summary node: the one which represents the whole storage chunk in a certain program point which is not referenced directly by any variable. However, to improve the data structure representation in many cases, we allow the existence of more than one summary node. More precisely, in our SSG there may be a summary node for each pointer type and connected component, as we describe now.

4.2.1 A Summary Node per pointer type.

If the method is constrained to a single summary node, then nodes of different structure type may be summarized in a single node. This way, all of these nodes will have the same "is shared",$is^{\#}$, attribute. Obviously, "$is^{\#}$" may turn to be true at a certain program point, but this is less likely to happen when the summary nodes are representing less nodes.

For instance, by allowing only one summary node, two different structures, like the ones we see in figure Fig. 3 (a), are going to be represented by the same summary node, Fig. 3 (b). This way, even when only one of the structures has several references to the same node, $is^{\#}$ becomes true for the whole summary node. Therefore, there is no way to know which structure (or if both of them) is actually sharing elements. This can be solved allowing a summary node for each different type of structure, as we can see in Fig. 3 (c). More precisely, we will consider that two structures have different type if the pointers pointing to elements of them have different type in the pointer declaration.

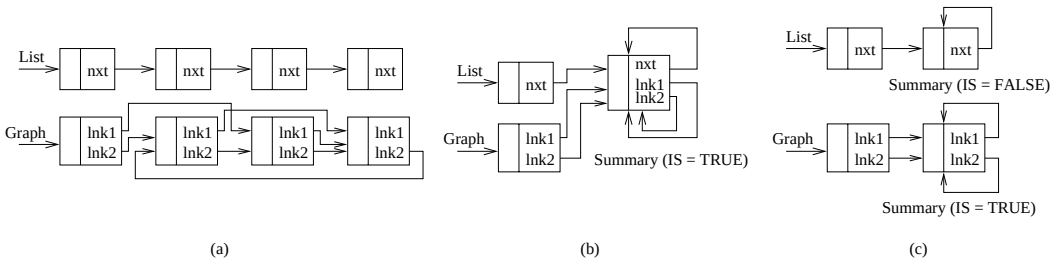


Figure 3: (a) Real structures. (b) Shape Graph without "type". (c) Shape Graph with "type". In (b) List and Graph may share nodes and List may be shared.

In order to do this, apart from the $is^{\#}$ attribute, we associate to each node the type information ($type^{\#}$). For each pointer variable we keep its type (type_var), which is taken from the declaratory part. With all these assumptions, the abstract semantic of the following statements should be modified:

1. Statement [x:= new]
   The $type^{\#}$ of the new node ($n_{\{x\}}$) is set to the $type^{\#}$ of the variable that points to it.
   $$type^{\#\prime}(n_{\{x\}}) = type\_var(x)$$
2. Statement [x:= y]
   All the nodes preserve their type, and the new nodes (now referenced by "x") take the type of the nodes pointed to by "y".
   $$type^{\#\prime}(n_Z) = type^{\#}(n_{Z-\{x\}})$$
3. Statement [x:= y.sel]
   A node materialization takes place. The type of the new node is the same as the type of the node from which it is materialized.
   $$type^{\#\prime}(n_Z) = type^{\#}(n_{Z-\{x\}})$$
4. The summarization of nodes not directly referenced by pointer variables, varies as well. Now, only nodes of the same type and not pointed to by any variable can be summarized.
5. During node matching for the union or comparison of graphs, apart from the set of variables referenced by the node, it is now also necessary to match their $type^{\#}$. This only affects the summary nodes since the others will have the $type^{\#}$ equal to that of the variables which reference them.

4.2.2 A Summary Node per connected component.

With the previous modifications we can maintain, for each graph, different summary nodes for different "types" of structures. Now, if we deal with several structures of the same type, the corresponding nodes not directly pointed to by variables, will be summarized in a single summary node. Again, it may be of importance to explicitly distinguish these structures, even when they have the same "type". For instance, in Fig. 4 (a) we can see two different structures which do not share any element. However, the method proposed in [13] leads to the SSG presented in Fig. 4 (b). Since there is a single summary node, there is no way to detect that one of the structures should have the "is shared" attribute set to false.

To solve this problem and let the method reach an SSG like the one presented in Fig. 4 (c), each node is annotated with an additional attribute: the structure to which this node belongs, $structure^{\#}$. This $structure^{\#}$ attribute has the same value for all nodes connected by a path. More precisely, we define the set of nodes connected with a given node "n", as the set of nodes that can be found in any path toward or from node "n":

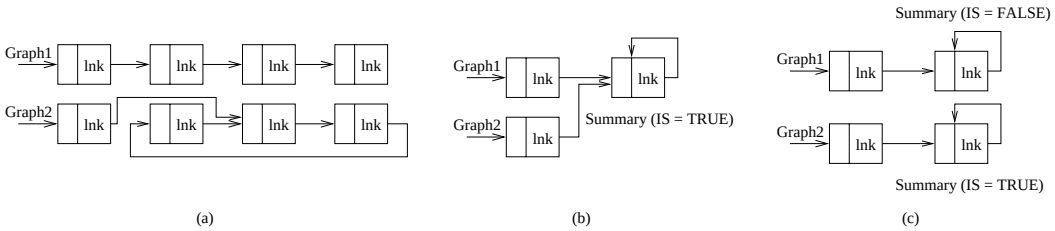


Figure 4: (a) Real structures. (b) Shape Graph without $structure^{\#}$. (c) Shape Graph with $structure^{\#}$. In (b) Graph1 and Graph2 may share nodes and Graph1 may be shared.

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

Again, the abstract semantic of the following statements is modified:

1. Statement [x:= y]
   The graph connectivity does not change for this statement. The new nodes (now pointed to by "x") will have the same $structure^{\#}$ as the nodes pointed to by "y".

   $$structure^{\#\prime}(n_Z) = structure^{\#}(n_{Z-\{x\}})$$

2. Statement [x.sel:= nil]
   This statement can break a connected component creating two new ones.
   $\forall n_X, x \in X, < n_X, sel, n_Z > \in Es^{\#}$:

   - if $C[Es^{\#\prime}](n_X) \cap C[Es^{\#\prime}](n_Z) = \emptyset$ then $\forall n \in C[Es^{\#\prime}](n_X), structure^{\#\prime}(n) = new\_structure$, $\forall m \in C[Es^{\#\prime}](n_Z), structure^{\#\prime}(m) = new\_structure$
   - if $C[Es^{\#\prime}](n_X) \cap C[Es^{\#\prime}](n_Z) \neq \emptyset$, $structure^{\#}$ does not change.

   When the connected components of the nodes n_X and n_Z do not have any node in common, it is clear that we are actually dealing with two different connected components. Therefore, the $structure^{\#}$ attribute is changed for all nodes in each connected component.

3. Statement [x.sel:= y]
   This statement can merge two previously unconnected components. Since any assignment to "x" or "x.sel" is always preceded by "x:= nil" or "x.sel:= nil" respectively, this statement cannot break any connection.

   $\forall n_X, n_Y, [x, n_X], [y, n_Y] \in Ev^{\#\prime}, < n_X, sel, n_Y > \in Es^{\#\prime}, compatible^{\#}(n_X, n_Y)$:

   - $\forall n \in C[Es^{\#\prime}](n_X), \forall m \in C[Es^{\#\prime}](n_Y) structure^{\#}(n) = structure^{\#}(m) = new\_structure$

   The $structure^{\#}$ information will be equal for all nodes connected to n_X and n_Y, since now they belong to the same connected component.

4. Statement [x:= y.sel]
   This statement does not break any connection in the graph. The $structure^{\#}$ attribute of the new materialized node will be the same as the one of the node from which it is materialized.

   $$structure^{\#\prime}(n_Z) = structure^{\#}(n_{Z-\{x\}})$$

5. Now, during the node summarization, only those nodes not pointed to by any variable and with the same $structure^{\#}$ attribute, can be summarized in a single node.

6. Regarding the node matching, the method also takes into account that the $structure^{\#}$ attributes must match as well.

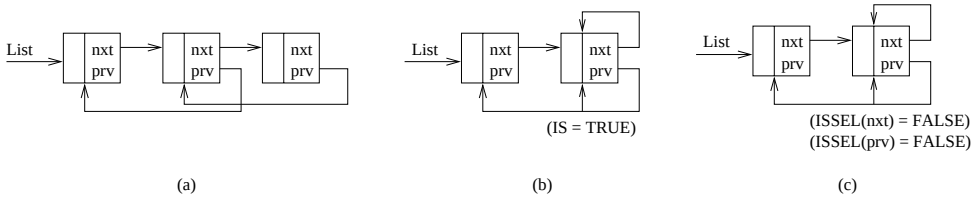


Figure 5: (a) doubly-linked list. (b) Shape Graph without is_sel attributes. (c) Shape Graph with is_sel attributes. In (c), when a loop traverses the List only by selector "nxt" or "prv", we can conclude that the same node cannot be visited twice.

4.3 Share Information per Selector

In the original method, each node keeps its own $is^{\#}$ attribute, which tells the compiler whether or not the node is referenced more than once from the heap. However, since this method does not take into account the selector used to reach the node, there is a potential lack of accuracy during the shape analysis. For example, Fig. 5 (a) shows a doubly-linked list. Even when this list is traversed in a loop only by selector "nxt" or "prv", the original method results in a single summary node with $is^{\#} = \text{true}$, Fig. 5 (b). In these kinds of situations, it is very important to keep the shared attribute for each selector, as we see in Fig. 5 (c).

Our shape analysis algorithm follows this last approach, assigning a shared attribute to each selector. Therefore, in addition to $is^{\#}(n)$ we also introduce:

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

which indicates whether the node "n" is referenced from the heap more than once by using the selector "sel". This leads to a less conservative and more accurate shape analysis for many data structures.

In order to accomplish these requirements, the abstract semantic of the following statements needs to be modified:

1. Statement [x:= nil]
   The summarized nodes(no longer referenced by "x") keep their $is\_sel^{\#}$ attributes.

   $$is\_sel^{\#\prime}(n_Z, sel) = is\_sel^{\#}(n_Z, sel) \vee is\_sel^{\#}(n_{Z\cup\{x\}}, sel)\ \forall sel$$

2. Statement [x:= new]
   When a new node is created, the corresponding $is\_sel^{\#}$ information is initially set to "false" for all the types of selectors.

   $$is\_sel^{\#\prime}(n_{\{x\}}, sel) = false\ \forall sel$$

3. Statement [x:= y]
   This statement does not change $is\_sel^{\#}$ attribute, since the connections in the graph are not changed.

   $$is\_sel^{\#\prime}(n_Z, sel) = is\_sel^{\#}(n_{Z-\{x\}}, sel)\ \forall sel$$

4. Statement [x.sel:= nil]
   This statement may break references from node "x" by selector "sel", and therefore nodes with $is\_sel^{\#}(n, sel)$ "true" may turn to be "false".

   To properly update the $is\_sel^{\#}$ attribute, we define the following function: $iss\_sel^{\#}[Es^{\#}](n, sel)$: $\exists n_{Z1}, n_{Z2}, compatible^{\#}(n_{Z1}, n_{Z2}, n) \wedge < n_{Z1}, sel, n >, < n_{Z2}, sel, n > \in Es^{\#} \wedge n_{Z1} \neq n_{Z2}$

$iss\_sel^{\#}$ becomes "true" for node "n" and selector "sel" when (a) there are two different nodes n_{Z1} and n_{Z2} which are compatible (using the $compatible^{\#}$ function) with "n" and (b) both of them reference the node "n" by selector "sel".

We extend the semantic of this statement as follows:
$is\_sel^{\#\prime}(n, sel) =$

- $is\_sel^{\#}(n, sel) \wedge iss\_sel^{\#}[Es^{\#\prime}](n, sel)$ if $\exists n_X, [x, n_X] \in Ev^{\#} \wedge < n_X, sel, n > \in Es^{\#}$
- $is\_sel^{\#}(n, sel)$ otherwise

That is, after breaking references to nodes pointed to by variable "x" using selector "sel", we check whether or not these referenced nodes maintain the shared attribute for selector "sel".

5. Statement [x.sel = y]
This statement can change the $is\_sel^{\#}$ information of the nodes directly pointed to by variable "y", since they are going to be referenced by selector "sel" from the heap.
$is\_sel^{\#\prime}(n, sel) =$

- $is\_sel^{\#}(n, sel) \vee iss\_sel^{\#}[Es^{\#\prime}](n, sel)$ if $[y, n] \in Ev^{\#}$
- $is\_sel^{\#}(n, sel)$ otherwise

With the $iss\_sel^{\#}$ function we check if the nodes pointed to by variable "y" are referenced more than once by selector "sel". The other nodes do not change.

6. Statement [x:= y.sel]
The changes induced by this statement are twofold. First, we note that the $is\_sel^{\#}$ attribute does not change. However, like in the statement [x:= y], we need to take into account the new nodes, which are now pointed to by variable "x".

$$is\_sel^{\#\prime}(n_Z, sel) = is\_sel^{\#}(n_{Z-\{x\}}, sel)\ \forall sel$$

On the other hand, the $is\_sel^{\#}$ attribute must be taken into account during the node materialization in order to avoid the creation of unnecessary references.

Therefore, we have modified the following functions:

- $compat\_in^{\#}([y, n_Y], < n_Y, sel, n_Z >, < n_W, sel', n_Z >)$:

$$compatible^{\#}(n_Y, n_Z, n_W) \wedge [y, n_Y] \in Ev^{\#} \wedge < n_Y, sel, n_Z >, < n_W, sel', n_Z > \in Es^{\#} \wedge$$
$$n_Z \neq n_W \wedge ((n_Y =^{\#} n_W \wedge sel = sel') \vee is\_sel^{\#}(n_Z, sel) \wedge sel = sel' \vee is^{\#}(n_Z) \wedge sel \neq sel')$$

  Note that in this previous function, we use $is\_sel^{\#}$ in addition to $is^{\#}$. This function, $compat\_in^{\#}$, is used to set new references from the already existing nodes to the materialized one. These new references are of two types: (a) references from selector "sel" from nodes pointed to by "y" variable (corresponding to the statement [x:= y.sel]), and (b) other node references, which are going to be taken into account only if $is\_sel^{\#}(n_Z, sel)$ or $is^{\#}(n_Z)$ is true.

- $compat\_self^{\#}([y, n_Y], < n_Y, sel, n_Z >, < n_Z, sel', n_Z >)$:

$$compatible^{\#}(n_Y, n_Z) \wedge [y, n_Y] \in Ev^{\#} \wedge < n_Y, sel, n_Z >, < n_Z, sel', n_Z > \in Es^{\#} \wedge$$
$$((n_Y =^{\#} n_Z \wedge sel = sel') \vee is\_sel^{\#}(n_Z, sel) \wedge sel = sel' \vee is^{\#}(n_Z) \wedge sel \neq sel')$$

  Similarly to $compat\_in$ function, we use $is\_sel^{\#}$ and $is^{\#}$. This function creates "self references" in the materialized node. In order to do this, we only consider those references for which node n_Z is shared.

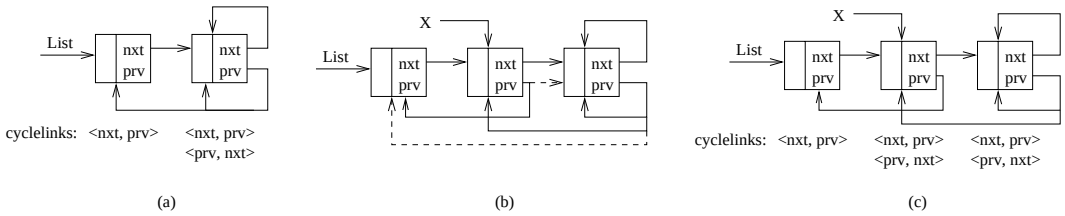


Figure 6: (a) doubly-linked list with "cyclelinks". (b) Shape graph without "cyclelinks" after executing statement "x:= list.nxt". (c) Shape graph with "cyclelinks" after statement "x:= list.nxt". Note that in (b) there are two superfluous references (dashed).

4.4 Cycle links

In order to reduce the number of unnecessary edges in the SSG, we assign a new attribute to each node: $cyclelinks^{\#}$. This attribute is actually a set of pairs of references $< sel1, sel2 >$. For a certain node, the pairs in $cyclelinks^{\#}$ fulfil the following property: when taking $sel1$ and $sel2$ subsequently from this node, the resulting reference points to the original node. This set maintains similar information to that of "identity paths" in the Abstract Storage Graph (ASG) [12], which is very useful to deal with doubly-linked structures.

Again, the following modifications of the abstract interpretation are needed:

1. Statement [x:= nil]
   This statement may produce the summarization of the nodes pointed to by variable "x". When two nodes are joined, we keep the compatible $cyclelinks^{\#}$ of both of them. A "cycle link" $< sel1, sel2 >$ belonging to $cyclelinks^{\#}(n1)$, is compatible with $cyclelinks^{\#}(n2)$, if (1) $< sel1, sel2 >$ belongs to $cyclelinks^{\#}(n2)$ as well, or (2) the node "n2" does not reference any node by selector "sel1".

$$\begin{aligned} cyclelinks^{\#\prime}(n_Z) = \{ &< sel1, sel2 > | < sel1, sel2 > \in (cyclelinks^{\#}(n_Z), cyclelinks^{\#}(n_{Z\cup\{x\}})) \vee \\ &< sel1, sel2 > \in cyclelinks^{\#}(n_Z) \wedge \neg\exists n, < n_{Z\cup\{x\}}, sel1, n > \in Es^{\#} \vee \\ &< sel1, sel2 > \in cyclelinks^{\#}(n_{Z\cup\{x\}}) \wedge \neg\exists n, < n_Z, sel1, n > \in Es^{\#} \} \end{aligned}$$

2. Statement [x:= new]
   For this sentence we create a new node with an empty $cyclelinks^{\#}$.

$$cyclelinks^{\#\prime}(n_{\{x\}}) = \emptyset$$

3. Statement [x:= y]
   The $cyclelinks^{\#}$ of the nodes pointed to by "y" are preserved. In addition, these nodes are also pointed to by "x"

$$cyclelinks^{\#\prime}(n_Z) = cyclelinks^{\#}(n_{Z-\{x\}})$$

4. Statement [x.sel:= nil]
   This statement results in the deletion of some elements in the $cyclelinks^{\#}$ set. First, elements of type $< sel, sel_i >$ of $cyclelinks^{\#}(n_X)$ are deleted, where n_X refers to the nodes directly pointed to by the "x" variable. On the other hand, we also delete elements $< sel_j, sel >$ from the $cyclelinks^{\#}(n_Z)$, where n_Z are the nodes referenced from n_X by selector "sel". For this case, n_Z should reference n_X by "sel_j".

   $cyclelinks^{\#\prime}(n) =$

   - $cyclelinks^{\#}(n) \setminus < sel, sel_i >$
     if $[x, n] \in Ev^{\#} \wedge < sel, sel_i > \in cyclelinks^{\#}(n)$

- $cyclelinks^{\#}(n) \setminus < sel_j, sel >$
  if $[x, n_X] \in Ev^{\#} \wedge < n_X, sel, n > \in Es^{\#} \wedge < n, sel_j, n_X > \in Es^{\#} \wedge$
  $< sel_j, sel > \in cyclelinks^{\#}(n)$
- $cyclelinks^{\#}(n)$ otherwise

5. Statement [x.sel:= y]
   This statement may create elements in the $cyclelinks^{\#}$ (n_X) and $cyclelinks^{\#}(n_Y)$ sets. Here, n_X is pointed to by "x" and n_Y is pointed to by "y". Regarding $cyclelinks^{\#}(n_X)$, we extend this set with $< sel, sel_i >$ element if n_Y only points, by selector "sel_i", to nodes directly pointed to by variable "x". In a similar way, we include $< sel_i, sel >$ in $cyclelinks^{\#}(n_Y)$.

   $cyclelinks^{\#\prime}(n) =$

   - $cyclelinks^{\#}(n) \cup < sel, sel_i >$
     if $[x, n], [y, n_Y] \in Ev^{\#} \wedge compatible^{\#}(n, n_Y) \wedge < n_Y, sel_i, n > \in Es^{\#} \wedge$
     $\neg \exists n_Z, compatible^{\#}(n, n_Z), n \neq n_Z, < n_Y, sel_i, n_Z > \in Es^{\#}$
   - $cyclelinks^{\#}(n) \cup < sel_i, sel >$
     if $[x, n_X], [y, n] \in Ev^{\#} \wedge compatible^{\#}(n, n_X) \wedge < n, sel_i, n_X > \in Es^{\#} \wedge$
     $\neg \exists n_Z, compatible^{\#}(n_X, n_Z), n_X \neq n_Z, < n, sel_i, n_Z > \in Es^{\#}$
   - $cyclelinks^{\#}(n)$ otherwise

6. Statement [x:= y.sel]
   The $cyclelinks^{\#}$ of the nodes pointed to by "y.sel" are preserved. In addition, these nodes are also pointed to by "x". The materialized nodes will have the same set of "cycle links" as the node from which it has been materialized.

$$cyclelinks^{\#\prime}(n_Z) = cyclelinks^{\#}(n_{Z-\{x\}})$$

Once we have applied all these updates in the SSG, it is necessary to check whether or not the references in the graph correspond to the information provided by the cycle links sets. Actually, the method breaks the references which are not compliant with the cycle links sets.

Let be

$$A = \{n \mid ([x, n] \in Ev^{\#\prime}) \vee ((< n_X, sel, n > \in Es^{\#\prime}) \vee (< n, sel, n_X > \in Es^{\#\prime}), [x, n_X] \in Ev^{\#\prime})\}$$

The new set of selector edges is

$$Es^{\#\prime\prime} = Es^{\#\prime} \setminus \{< n, sel1, n_Z > \mid n \in A, < sel1, sel2 > \in cyclelinks^{\#\prime}(n), < n, sel1, n_Z > \in Es^{\#\prime}, < n_Z, sel2, n > \notin Es^{\#\prime}\}$$

In Fig. 6 we can see an example showing the improvement that can be achieved by the use of these cycle links set. In Fig. 6 (a) we show a doubly-linked list and their corresponding cycle links sets. Figures 6 (c) and (b) show the resulting SSG after executing the sentence "x:=list.nxt", taking the cyclelink information into account or not, respectively. We can see that there are two artificial references in case (b), that can be avoided by considering the cyclelinks information (c), which leads to more accurate SSGs.

4.5 Sparse LU modified SSG

All these previously described techniques have been implemented in a simple compiler which reads C code and returns the SSG for each program point. To generate each one of these SSGs, the compiler takes the output SSG, SSG_o, of the previous sentence as the input SSG, SSG_i, for the current sentence. Now, this input SSG is modified according to the abstract semantic of the current sentece, generating the output SSG, which will be the input SSG for the next sentence, and so on. Since one sentence can be reached after following several paths in the control flow graph, the SSG for this sentence is the union of all the possible SSGs after following all these paths. That way, if the compiler reach an already visited

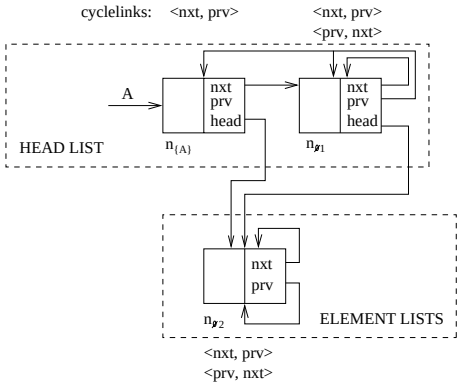


Figure 7: LU algorithm Shape Graph modified.

sentence, the new output SSG is summarized with the previous SSG of the sentence. For instance, the SSG associated with a loop head is the union of the SSG associated with the sentence just before this loop head and the SSG associated with the last sentence in the loop body. As in [13], the analysis of the sentences inside a loop body does not end until the SSG associated with the loop head does not change from one iteration to the next one (the analysis reach a fixed point). Regarding the IF sentence, the SSG_i for the first sentence of the THEN and ELSE parts are the SSG_o of the sentence just before the IF. In a similar way, the first sentence after the IF branch take the SSG_i as the union of the SSG_o of the last sentences of the THEN and ELSE parts. The compiler has been written in C, taking special care over memory management and in the selection of a proper data structure to store the SSG.

The resulting SSG in the last sentence of the Sparse LU code is shown in Fig. 7. As we can see, variable A points to a doubly-linked list, summary node $n_{\emptyset 1}$, with the shared attribute set to false. Every node in this list points to a different doubly-linked list, represented by summary node $n_{\emptyset 2}$, by selector "*head*". There are two summary nodes, because there are two types of pointers ("head list" and "element list").

For all nodes, $is\_sel^{\#}(n, sel)$ is "FALSE" for all selectors. Therefore, we conclude that the "head list" and the "element list" are acyclic structures when they are traversed by a single selector type. In addition, we note that there is no shared node among different "element lists". With all these results, it is clear that several sparse matrix columns can be updated in parallel during factorization. Furthermore, the resulting SSG points out that each column (element list) is acyclic when traversed using a single selector type. So, if they are actually traversed in this way, it is also possible to update each column in parallel.

For instance, our sparse data structure is traversed in our algorithm by using the "nxt" selector. The "prv" selector is only used to simplify the deletion operation of an entry. In other words, "nxt" can be seen as a "traversing link" whereas "prv" as a "referencing link". However, this "traversing" information should be inferred in a subsequent compiler stage which uses the SSG to perform the data dependence test step. We do not cover this step yet, but we will address this topic in the near future.

It is important to note here that the same SSG we see in Fig. 7 is achieved in two program points: after the data structure initialization (initial sparse matrix A) and after the sparse LU factorization (data structure for the factorized matrix LU). In other words, the in-place LU factorization code does not change the sparse data structure representation and characteristics.

Compared to the SSG method presented by Sagiv [13], we see that our SSG is more accurate than the one they obtain (already presented in Fig. 2 (c)). The main reasons leading to their results are: (a) they can only represent a summary node in the SSG whereas we can include many of them for each structure type, (b) their summary node has $is^{\#} =$ true due to the fact that their method is not able to detect that there is not more than one reference from the heap to each node by the same selector. Furthermore, by considering the "is_sel" and "cycle_links" attributes, we are able to infer that there are no shared elements in the list, and also we keep the SSG as accurate as possible avoiding superfluous references. We have also tested the compiler with other simple C codes based on trees achieving succesful results.

Actually, in the next section we can see how the compiler deals with a recursive code which builds a tree.

5 Interprocedural analysis

The shape-analysis described here is based on "abstract interpretation", where, for each sentence, the algorithm stores a graph (SSG) which represents all possible shapes of the data structure that may arise after the execution of any control flow path which ends in this sentence. Up until this point, we have dealt with three types of control flow: sequential, conditional (if, case) and loops. The most complex is the last one (loops) as it is necessary to analyze it iteratively until the compiler reaches a fixed point in which the SSG at the loop entry does not change any more [13].

In this section we also extend the compiler to deal with procedures and recursivity, since these constructions are widely used, mainly by codes based on dynamic data structures like trees and lists.

5.1 Variables and parameters

When dealing with procedures the compiler should take into account several issues related to variables and parameters. First, the compiler has to distinguish between variables with the same name in different contexts. In order to do this, a local variable inside a procedure will be given its name followed by the procedure name as a suffix. This should be done during code transformation to the intermediate form (first pass).

On the other hand, the compiler can manage the argument pass by the insertion of sentences to assign actual arguments to formal arguments before the call and after it [3]. For parameters passed by value, the compiler assigns the formal parameters to the actual ones before the call. For parameters passed by reference, the compiler also assigns the actual parameters to the formal ones after the call.

Figure 8 shows a small example of these transformations in a code.

Original code

```
Global pvar list, a;
...
a = list;
b = a;
call p1(a, b);
...
```

```
p1(val. pvar a, ref. pvar b)
local pvar c;
...
c = a;
...
end procedure;
```

Transformed code

```
Global pvar list, a;
...
a = list;
b = a;
a_p1 = a;
b_p1 = b;
call p1();
b = b_p1;
...
```

```
p1(val. pvar a_p1, ref. pvar b_p1)
local pvar c_p1;
...
c_p1 = a_p1;
...
end procedure;
```

Figure 8: Code Transformation for procedure call (`pvar a` is passed by value and `pvar b` by reference)

```
...
if (...)
  ...
  (creation of a shared structure pointed to by L)
  ...
  s1: call P(L)
  ...
else
  ...
  (creation of a non-shared structure pointed to by L)
  ...
  s2: call P(L)
  ...
end if
```

Figure 9: A situation which can be managed thanks to `SSGcd`

5.2 Procedure call

Clearly, a procedure can be called from different program points and therefore with a different SSG at each procedure entry. In our procedure analysis, we want the compiler to discriminate between the different call points, in such a way that a given SSG at the procedure entry point is not mixed up with an SSG corresponding to other calling points. With this approach, the compiler can achieve better accuracy for the graphs associated with sentences of the caller procedure.

In order to do this, the compiler assigns two different graphs to the sentences inside a procedure:

- **SSGcd**: SSG call dependent. This graph approximates all possible concrete stores which can appear for a given sentence when we start with a single SSG corresponding to a particular calling point to the procedure.

  This `SSGcd` is obtained following the abstract semantic, starting from the SSG for the actual call. That is, given the pointer statement `st` associated with a procedure:

$$SSGcd(st) = [st](JOIN(SSGcd(stp)))\ \forall\ stp \in \{predecessors\ of\ st\ in\ the\ CFG\}$$

- **SSGci**: SSG call independent. This graph approximates all possible concrete stores which can appear for a given sentence inside a procedure, taking into account all possible calling points in all caller procedures.

  The `SSCGci` is obtained by joining all SSGcd arising through the program analysis.

$$SSGci(st) = JOIN(SSGcd(st), SSGci(st))$$

The `SSGcd` are needed during the analysis to avoid mixing up information due to different contexts but which, at the end of the analysis, do not provide any information. On the other hand, the `SSGci` keeps an approximation of all concrete stores which may appear for a sentence in a procedure. Therefore, these ones are used for a subsequent analyisis of the code. However, the importance of the `SSGcd` is illustrated in Figure 9, where we see that procedure `P` is called from the `then` part (where a shared structure pointed to by `L` is created), and also from the `else` part (where the structure pointed to by `L` is not shared). Thanks to the `SSGcd` the compiler will know that the SSG for sentences before `s2` are not adversely affected by sentence `s1` where `L` points to a shared structure.

5.3 Recursivity

Once we have dealt with the analysis of procedure calls, we can move on to the recursivity call problem. When calling recursively to a procedure, there are several instances of the same code running at the

same time. Therefore there are several copies of different pointer variables (`pvar`) belonging to each one of these instances. In order to differentiate the `pvar` from different calls, the compiler uses new `pvar` for each nested level [10]. We call `PVD` to the set of these "pvars-depth" which are built by appending a suffix to the local `pvar` name or parameter that shows the depth of the recursive call to which it belongs.

```
                                    0  procedure p1(val. pvar a_p1)
                                    1  local pvar c_p1;
                                       ...
procedure p1(val. pvar a_p1)        2  c_p1 = a_p1;
local pvar c_p1;                    3  a_p1_depth = a_p1;
...                                 4  c_p1_depth = c_p1;
c_p1 = a_p1;                        5  a_p1 = ...;
a_p1 = ...;                         6  conditional recursive call p1();
recursive call p1();                7  c_p1 = c_p1_depth;
...                                 8  a_p1 = a_p1_depth;
end procedure;                      9  c_p1_depth = nil;
                                    10 a_p1_depth = nil;
                                       ...
                                    11 end procedure;
```

Figure 10: Code Transformation for recursive procedure call.

In figure 10 we see the necesary code transformations for a recursive call. Notice that the compiler first assigns the local `pvar` to the corresponding `pvars-depth`. Just after exiting the procedure the compiler performs the reverse assignment. This way, when returning from a recursive call, all local `pvars` keep their previous values (before the call). The `depth` suffix is managed by the compiler, incrementing the value when calling (depth + 1) or decrementing when returning (depth - 1) from a recursive call.

Now the problem is quite similar to the one we faced when dealing with loops: the control flow graph is not bounded because we do not know at compile time how many times the recursive call will take place. So, we analyze the recursive call in a similar way to loops.

From a high-level point of view, we can think of a recursive call as a sequence of two loops: a forward loop and backward loop.

- **Forward Loop**: The body of this loop contains the sentences which are between the beginning of the procedure and the sentence before the procedure call (in figure 10 these are the sentences between 0 and 5). Analyzing this code involves building an SSG for each sentence in the loop body iteratively until the compiler reaches a fixed point (the SSG at the procedure entry does not change any more). The problem here is that due to the new `pvars-depth` that are created for each iteration the fixed point will never be reached.

  In order to solve this problem we have to take note that the function of `pvars-depth` is to keep track of the concrete stores pointed to by the local `pvars` in previous calls. Therefore they are not involved in the data structure modification during the procedure execution. This way, the data structure which can be "seen" by the procedure at a certain depth is the one pointed to by local `pvars` at that depth.

  So, the compiler implements the function $VIEW : SSG \rightarrow SSG$

  $$VIEW(SSG) = [pv = nil](SSG)\ \forall\ pv \in PVD$$

  This function nullifies all `pvars-depth` belonging to previous calls to take only local `pvars` into account.

  The `SSGci` are now generated by joining the different $VIEW(SSGcd)$ appearing in each sentence:

  $$SSGci(st) = JOIN(VIEW(SSGcd(st)), SSGci(st))$$

Now, the "loop" analysis ends when $SSGci$ is equal at the entry of one level (depth) and at the entry of the next level (`depth`+1). In the same way in which the fixed point is reached for loops, it is easy to prove that it is always reached for recursive calls also (once the compiler applies the $VIEW$ function, the number of `pvars` is finite, and therefore the number of `SSGs` too).

- **Backward Loop**: The body of this loop contains the sentences which are between the next sentence to the recursive call and the last sentence of the procedure (control flow between sentences 7 and 12 in figure 10). We call post-process to the operations which take place in each iteration of this backward loop. These operations may change the shape of the data structure and should be taken into account to obtain an accurate SSG for the call sentence.

  The number of iterations that this backward loop is executed is equal to the maximum depth, `md`, achieved during the forward loop until reaching the fixed point. However, it may happen that we do not reach a fixed point by just post-processing the SSG `md` times. We can solve this by following this algorithm: if we do not reach a fixed point when post-processing the SSG `md` times, we have to iterate the forward loop one more time to get a new maximum depth `md`=`md`+1, and now check whether this is enough to reach a fixed point in the backward loop. This can be summarized in the code presented in figure 11, where `stin` represents the entry statement to the procedure and `stout` the output one.

```
Begin
  While New SSGci(stin) do
    Process Forward Loop (depth+1)

  Process Backward Loop to depth 0
  While New SSGci(stout) do
    Process Forward Loop (depth+1)
    Process Backward Loop to depth 0
End
```

Figure 11: Algorithm to reach a fixed point in a recursive call

This analysis has been implemeted in our compiler and has been evaluated with the code presented in figure 12. This code just traverses and inserts entries in a binary tree by recursive calls. These kinds of procedures are widely used in N-body simulations, molecular dynamics, astrophysics, cuad-trees and oc-trees problems.

The resulting SSG obtained at the call sentence in the main program is presented in figure 13. We can notice that the `pvar tree` points to a binary tree with shared information false for all selectors.

For the sake of clarity of the recursive analysis we show next an step by step example in table 1. We assume in this example that the compiler is analyzing the code of figure 12 traversing the sentences in the order they are written. The rows in table 1 show the input SSG_i for the recursive call sentence (line number 6). This call sentence correspond to the traversing of the left branches of the tree. In figure 14 we can see the transformed code of the first six lines in procedure `ins`. In this example we suppose that the input SSG_i for the zero call (first row in the table) is the graph corresponding to a tree with more than two left children. In the next rows of table 1 we can see the SSG corresponding to three recursive calls, and the SSGci resulting after applying the VIEW function to the SSG. Remember that the SSGci is the one used to determine whether or not we have reach a fixed point.

As we can see, in the second row (call 1), the pvars *tree_ins*_1 and *aux_ins*_1 are pointing to the nodes previously pointed to by *tree_ins* and *aux_ins*, respectively, following sentences 5.1 and 5.2 of the code in figure 14. After applying the VIEW function to the previous SSG we obtain a similar SSG (in the SSGci column) but removing the depth pvars, this is, just keeping the local pvars to the current call. In the next call (third row in table), the same happens for pvars *tree_ins*_2 and *aux_ins*_2, and we see that the *tree_ins* pvar follows the left branch. Again, by applying the VIEW function the depth pvars disappear. Notice that the node previously pointed to by *tree_ins*_2 and *aux_ins*_1, this is the node named $n_{\{aux\_ins\_1, tree\_ins\_2\}}$, changes to be $n_{\emptyset}$ after the depth pvars elimination. Since all the

```
                              1  procedure ins(tree, val)
                              2  local pvar aux;
                              3    if (COMPARE(tree.val, val))
                              4      if (tree.left != nil)
                              5        aux = tree.left;
                              6        call ins(aux, val);
                              7      else
...                           8        aux = new;
tree = new;                   9        aux.val = val;
tree.val = ...;               10       tree.left = aux;
do while ...                  11     end if
  val = ...;                  12   else
  call ins(tree, val);        13     if (tree.right != nil)
  ...                         14       aux = tree.right;
end do                        15       call ins(aux, val);
                              16     else
                              17       aux = new;
                              18       aux.val = val;
                              19       tree.right = aux;
                              20     end if
                              21   end if
                              22 end procedure
```

Figure 12: Creation of a binary tree

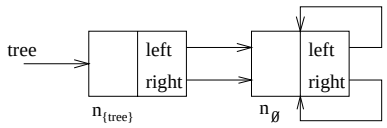


Figure 13: SSG for a binary tree

nodes with the same name are summarized, this new $n_\emptyset$ node is summarized with the old one. In the next call (last row) we note that after applying the VIEW function we obtain the same SSGci of the previous call, which means that we have reach a fixed point and that subsequent calls will just produce the same SSGci.

In this example we have focus in the `Forward Loop` analysis. In this case the `Backward Loop` does not affect the resulting SSG due to after the recursive calls there are no additional sentences in the CFG (after returning from the call, the procedure exits).

6 Conclusions and future work

In this work, we have implemented a shape-analysis algorithm based on the methods of Sagiv et al. [13] (SSG), and Plevyak et al. [12] (ASG), improving accuracy and dealing with more complex data structures. In addition, we have extended our shape analysis algorithm to make an interprocedural analysis without applying inlining. Since there are many codes which use recursive calls to traverse the data structure, this is also an important topic.

We have validated the implementation of the method with a real code, the Sparse LU Factorization, and for a simple recursive code, for which we have achieved good results. The resulting shape graph provides a great deal of information at compile time. Summarizing, this information describes the data structure used in the algorithm. For the LU code, the resulting SSG states that the columns of the

```
1  procedure ins(tree_ins, val)
2  local pvar aux_ins;
3    if (COMPARE(tree_ins.val, val))
4      if (tree_ins.left != nil)
5        aux_ins = tree_ins.left;

5.1      aux_ins_depth = aux_ins;
5.2      tree_ins_depth = tree_ins;
5.3      tree_ins = aux_ins;
5.4      aux_ins = null;

6        call ins(aux, val);
7      else
       ....
       ....
       ....

end procedure
```

Figure 14: Section of the transformed code for the creation of a binary tree

sparse matrix are stored in memory as doubly-linked lists which do not share elements. A subsequent data dependence test phase would determine that the algorithm traverses the columns of the reduced submatrix for each k-loop iteration (see Fig. 1) and that each column can be updated in parallel. On the other hand, for the recursive program, the method achieves the same SSG as if the generation and traversing of a binary tree were implemented by loops. Thanks to there being no shared=true for any selector, in the data dependence phase the compiler will assume the tree can be processed in parallel.

This data dependence phase is one of the topics on which we need to focus next. But first, we plan to enhance the method in order to handle more complex data structures, like Acyclic Direct Graphs (ADGs), where there may exist more than one reference to a node by the same selector.

References

[1] R. Asenjo. Sparse LU Factorization in Multiprocessors. Ph.D. Dissertation, Dept. Computer Architecture, Univ. of Málaga, Spain, 1997.

[2] U. Banerjee. Loop Transformations for Restructuring Compilers: The Foundations. Kluwer, 1993.

[3] D. Chase, M. Wegman and F. Zadeck. *Analysis of Pointers and Structures.* In SIGPLAN Conference on Programming Language Design and Implementation, 296-310. ACM Press, New York, 1990.

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

[5] I.S. Duff, A.M. Erisman and J.K. Reid (1986), *Direct Methods for Sparse Matrices*, Oxford University Press, NY, 1986.

[6] L. Hendren and A. Nicolau. *Parallelizing Programs with Recursive Data Structures.* IEEE Transactions on Parallel and Distributed Systems,1(1):35-47, January 1990.

[7] S. Horwitz, P. Pfeiffer, and T. Reps. *Dependence Analysis for pointer variables.* In Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation, 28-40, June 1989.

[8] J. Hummel, L. J. Hendren and A. Nicolau *A General Data Dependence Test for Dynamic, Pointer-Based Data Structures.* In Proceedings of the SIGPLAN Conference on Programming Language Desing and Implementation, pages 218-229. ACM Press, 1994.

[9] N. Jones and S. Muchnick. *Flow Analysis and Optimization of Lisp-like Structures.* In Program Flow Analysis: Theory and Applications, S. Muchnick and N. Jones, Englewood Cliffs, NJ: Prentice Hall, Chapter 4, 102-131, 1981.

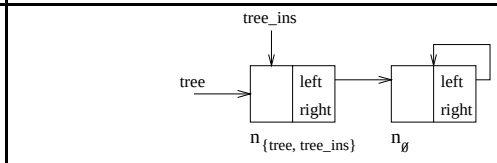
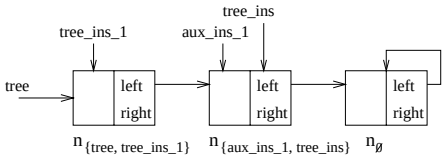
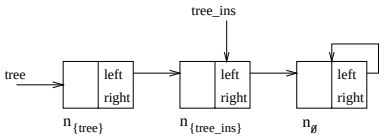
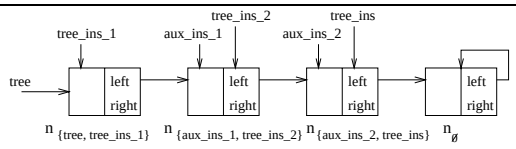
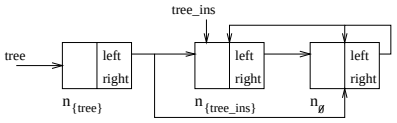
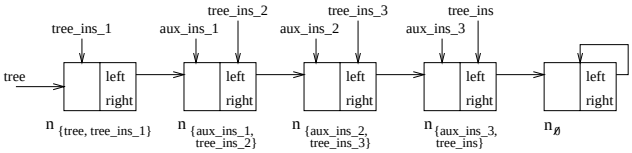
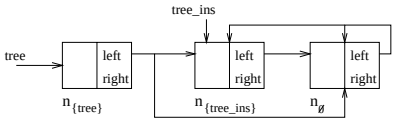
Call	SSG	SSGci (VIEW)
0		
1		
2		
3		

Table 1: Step by step example of a recursive procedure analysis.

[10] J. R. Larus and P. N. Hilfinger. *Detecting Conflicts between Structure Accesses.* In SIGPLAN Conference on Programming Language Design and Implementation, 21-33. ACM Press, New York, 1988.

[11] A. Matsumoto, D. S. Han and T. Tsuda. *Alias Analysis of Pointers in Pascal and Fortran 90: Dependence Analysis between Pointer References.* Acta Informatica 33, 99-130. Berlin Heidelberg New York: Springer-Verlag, 1996.

[12] J. Plevyak, A. Chien and V. Karamcheti. *Analysis of Dynamic Structures for Efficient Parallel Execution.* In Languages and Compilers for Parallel Computing, U. Banerjee, D. Gelernter, A. Nicolau and D. Padua, Eds. Lectures Notes in Computer Science, vol 768, 37-57. Berlin Heidelberg New York: Springer-Verlag 1993.

[13] 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):1-50, January 1998.