Complexity study of the shape analysis based on CLSs

A. Navarro, F. Corbera, R. Asenjo, A. Tineo and E.L. Zapata*

February 22, 2007

Abstract

In this paper, we focus on the complexity study of a recently proposed shape analysis technique. That technique is based on a compact representation for the shape of data structures by using *Coexistent Links Sets* -CLSs- for nodes in a graph. In this study we have found that two parameters, named the number of graphs and the number of CLSs drive the analysis times of the approach. Besides, we have conducted several experimental tests which seem to indicate that, in spite of the theoretical exponential behaviour of these parameters, in practice the number of generated graphs and specially, the number of generated CLSs are quite manageable. In fact, the number of generated CLSs is an extremely low fraction when compared to the theoretical estimated maximum values. Through these experiments we have found that the analyses run in reasonable times, specially for such a complex technique.

1 INTRODUCTION

In the context of optimizing compilers for heap-directed pointer-based codes, information about how memory locations are arranged in the heap is essential for data dependence analysis or software verification. For instance, with proper shape and data dependence information we can reveal parallelism as well as perform data-cache optimizations or loop transformations, optimizations which are typically ignored by compilers when dealing with such codes.

A promising way to overcome the difficulties of analyzing the dynamic data structures that appear in these codes has emerged with the shape analysis approaches [6], [4], [3]. However, existing shape analysis methods face a dilemma: either they are too costly to be useful for real compilers or they are too imprecise to be useful for real programs. In this paper, we focus on the complexity study of a recently proposed shape analysis technique [7]. This technique is based on a compact representation for the shape of data structures by using *Coexistent*

*The authors are with the Dept. of Computer Architecture, University of Málaga, Complejo Tecnológico, Campus de Teatinos, E-29071. Málaga, Spain.E-mail:angeles,corbera,asenjo,tineo,ezapata@ac.uma.es

Links Sets -CLSs- for nodes in a graph. The interested reader can find more details about the technique in [7].

In our study we have found that two parameters, named the number of graphs and the number of CLSs drive the analysis times of the approach. Besides, we have conducted several experimental tests which seem to indicate that, in spite of the theoretical exponential behaviour of these parameters, in practice the number of generated graphs and specially, the number of generated CLSs are quite manageable. In fact, the number of generated CLSs is an extremely low fraction when compared to the theoretical estimated maximum values. Through these experiments we have found that the analyses run in reasonable times, specially for such complex technique.

The rest of the paper is organized as follows: Section 2 gives some details about the main operations of our shape analysis approach. Next, in Section 3 we identify and compute the parameters that model the complexity of our approach. Then, in Section 4 we conduct several experiments that help us to gain insight about the behaviour of our approach and to discuss the applicability of our approach to real codes. Finally, in Section 5 we present our main conclusions.

2 SHAPE ANALYSIS OPERATIONS

In this section, we succinctly describe the shape analysis operations and the compilation algorithms that implement them. We report here the algorithms, because they will be used to compute the complexity -of each operation- in the next section. Fig. 1 presents the phases that each graph (SG_{in}) of the input set of graphs undergoes when a pointer statement is processed.

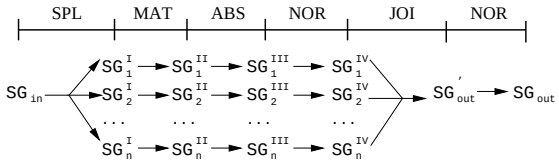


Figure 1: Phases for graph processing at a pointer statement.

In our analysis, we consider six simple pointer statements (see Table 1), because other complex pointer statements can be transformed into several of these simple pointer statements in a preprocessing stage. Not all phases are required by the abstract semantic of each statement, as we see in Table 1. In general, there are 5 different phases, each involving different transformations:

- **Graph splitting (SPL)**. Some statements require that the access path indicated by a pointer and selector (`x->sel`) leads to just one node. This way, the next transformations are easier to perform. Each graph of the input set of graphs, is split into as many different graphs, as different nodes can be reached through the `x->sel` reference. Each of the resulting graphs, is processed in the same way by the MAT, ABS and NOR phases.

Table 1: Pointer statements and operations they involve for each phase.

Statement	SPL	MAT	ABS	NOR/JOI/NOR
`x = NULL`	-	-	`XNull(x)`	`norm()`
`x = malloc()`	-	-	`XNull(x),norm(),XNew(x)`	-
`x = y`	-	-	`XNull(x),norm(),XY(x,y)`	-
`x = y->sel`	`split(y,sel)`	`nm = mater(y,sel)`	`XNull(x),norm(),XNode(x,nm)`	`norm(),join(),norm()`
`x->sel = y`	`split(x,sel)`	`nm = mater(x,sel)`	`XSelNodeY(x,sel,nm,y)`	`norm(),join(),norm()`
`x->sel = NULL`	`split(x,sel)`	`nm = mater(x,sel)`	`XSelNodeNull(x,sel,nm)`	`norm(),join(),norm()`

- **Node materialization (MAT)**. If the access path of a statement leads to a summary node (node that represents several actual memory locations), then a new singular node is materialized. This node represents just the memory location where the access occurs, thus providing precision in the analysis.
- **Abstract semantics (ABS)**. Each pointer statement produces a different effect in the graph according to its meaning in the program (e.g., a malloc statement creates a new node, a pointer linking statement establishes links between nodes, etc).
- **Normalization (NOR)**. After the changes directed by the abstract semantics of the statement have been applied to each graph, it is possible that some nodes can be summarized. This phase checks for summarizable nodes and merges them, a crucial process for bounding the graphs.
- **Graph joining (JOI)**. All graphs representing memory configurations with the same pointer arrangement (those that present the same pointer alias relationships and whose nodes pointed by pointers are compatible) are joined into a single graph. Resulting graphs must also be normalized (NOR), before proceeding to next statement.

Each phase may involve several operations depending on the pointer statement considered. Table 1 presents the operations involved for each statement, and the phase where they occur. Table 2 describes at high level the `split()`, `mater()`, `norm()` and `join()` functions.

From Table 1, we notice that statements that involve following a path through a pointer and selector (i.e., `x=y->sel`, `x->sel=y`, and `x->sel=NULL`), require that the input set of graphs is split, as described by the `split()` function (Table 2). From a descriptive point of view, this function creates separate graphs that account for all possible memory configurations in the input set of graphs.

Right after splitting, these statements that follow a path, require that a node be materialized out of the node pointed to by `x->sel`, if it was a summary node. The `mater()` function sketched in Table 2 does just that. Basically, if `n2` is the node reached through `x->sel`, it creates a new node `nm` out of `n2`, inheriting all

Table 2: Overview of the `split()`, `mater()`, `norm()` and `join()` functions.

`split(x,sel)`	`mater(x,sel)`
1. Apply steps 2-7 to each graph of the input set of graphs 2. Apply steps 3-7 to all nodes `n1` pointed by `x` 3. Apply steps 4-7 to all nodes `n2` belonging to a `SL=<n1,sel,n2>` 4. Create a copy of the graph 5. Remove `CLSs(ni)` containing `PL=<x,ni>`, for all `ni!=n1`, and remove those PLs 6. Remove `CLSs(n1)` containing `SL=<n1,sel,ni>`, for all `ni!=n2` 7. Remove unused elements 8. Return all generated graphs	1. Find node `n1` pointed by `x` and node `n2` found in `SL=<n1,sel,n2>` 2. Create node `nm` as a copy of `n2`, with the same properties (if any) 3. Make `nm` inherit all SLs and CLSs from `n2` 4. Remove `SL=<n1,sel,n2>` and drop it from `CLSs(n1)` and `CLSs(n2)` 5. Add `SL=<n1,sel,nm>` to `CLSs(n1)` and `CLSs(nm)` 6. Make `nm` a singular node and remove unnecessarily conservative links 7. Remove unused elements
`norm()`	`join()`
1. Find compatible nodes and summarize them, calculating merged values for properties (if any) 2. Remove duplicated PLs and SLs 3. Update CLSs exchanging nodes, PLs and SLs 4. Remove duplicated CLSs	1. Find graphs whose nodes pointed by pointers are compatible and that present the same alias relationship 2. For those graphs, add all nodes, PLs, SLs and CLSs, into a working graph

its links appropriately. Then the function removes unnecessarily conservative links and updates CLSs accordingly, so as the materialized node is a singular node that represents just the actual memory location being accessed through `x->sel`. This is crucial for achieving precision when traversing a structure.

Now, let us consider the abstract semantics phase (ABS). For the nullification statement, `x=NULL`, the ABS phase only involves the `XNull(x)` function call, which performs pointer nullification for `x` in the current graph. The `XNull()` function is described in Fig. 2. Basically, this function finds all PLs of the form `PL=<x,ni>`, removes those PLs from any `CLS(ni)` that references it, and finally removes those PLs from the graph. In the pseudo-code illustrating the functions, `PLs`, `SLs` and `CLSs` are the collections (of type `PLCol`, `SLCol` and `CLSCol` respectively), which contain the corresponding elements for the current graph.

The `XNull()` function is also called as the first operation in the ABS phase for all statements that involve a write access to the location pointed by a pointer (`x`), namely `x=malloc()`, `x=y` and `x=y->sel`. These statements may involve destructive updating. In order to keep the abstract semantics of these statements simpler, they assume that the assigned pointer has been nullified. This is to

```
fun XNull(Ptr x)
  PLCol PLsWithX = PLs.getPLsWithPtr(x);
  foreach(PL in PLsWithX)
    Node n = PL.getNode();
    CLSCol CLSsWithN = CLSs.getCLSsWithNode(n);
    foreach(CLS in CLSsWithN)
      CLS.dropPL(PL);
    PLs.remove(PL);
```

```
fun XNew(Ptr x)
  Node n = new Node(x.type);
  PointerLink PL = new PointerLink(x,n);
  PLs.add(PL);
  CoexistentLinkSet CLS = new CoexistentLinkSet(n);
  CLS.addPL(PL);
  foreach(sel in x.type.sels)
    SelectorLink SL = new SelectorLink(n,sel,NI);
    SLs.add(SL);
    CLS.addSLAttr(SL,'o');
  CLSs.add(CLS);
```

```
fun XY(Ptr x, Ptr y)
  PLCol PLsWithY = PLs.getPLsWithPtr(y);
  foreach(PL in PLsWithY)
    Node n = PL.getNode();
    XNode(x,n);
```

```
fun XNode(Ptr x, Node n)
  PointerLink PL = new PointerLink(x,n)
  PLs.add(PL);
  CLSCol CLSsWithN = CLSs.getCLSsWithNode(n);
  foreach (CLS in CLSsWithN)
    CLS.add(PL);
```

Figure 2: Pseudo-code for the `XNull()`, `XNew()`, `XY()` and `XNode()` functions.

ensure that `x` does not point anywhere before continuing with graph transformations. Then, after nullifying a pointer, some nodes could be summarized and the `norm()` function must be called. In Table 2, in the description of the `norm()` function, we see that so-called *compatible nodes* are found and merged. Compatible nodes are those that present similar access patterns, according to the summarization criteria applied. So far, in the previous section, we have explained a basic mechanism to summarize nodes: those with the same *pvars* set. This provokes the summarization of two kind of nodes: (i) nodes pointed to by the same pointers, and (ii) nodes not pointed to by any pointer. If we want to attain more precision for (ii), then we can use *properties*. Properties become a valuable instrument to keep nodes apart when they represent different access patterns and they are not directly accessed by pointers.

The abstract semantics for statement `x=malloc()` (Table 1) also includes a call to the `XNew()` function, which is described in pseudo-code in Fig. 2. It creates a new node `n`, a new `PL=<x,n>`, new initialized SLs of the form `SLk=<n,selk,NI>` (where `NI` stands for non-initialized node), and a new CLS for the new node of the form `CLS(n)={PL,SL1(o),SL2(o),...}`. On the other hand, the abstract semantics for statement `x=y` performs a call to the `XY()` function (Fig. 2 again). That function is the key to describing the behavior of the aliasing statement. For each node `ni` that `y` points to, a new PL of the form `PL=<x,ni>` is created and added to `CLSs(ni)`, by means of the `XNode()` function.

Fig. 3 presents the functions needed in the ABS phase for the statements `x->sel=y` and `x->sel=NULL`. Both involve nullifying the `SL=<n1,sel,n2>` with the `nullifySL(n1,sel,n2)` function call, being `n1` the node reached through `x`, and `n2` the node reached through `x->sel`. This removes the SL from the SL collection in the current graph, after dropping it from CLSs for `n1` (`CLSsWithN1`) and `n2` (`CLSsWithN2`). Recall that at this point, `n1` and `n2` are unique. Therefore, it is safe to recover the only element in the PL collection of PLs that contain `x` (i.e., PLs of the form `PL=<x,ni>`), via `PLsWithX[0]`, in `XSelNodeNull()`

```
fun XSelNodeNull(Ptr x, Sel sel, Node n2)
  PLCol PLsWithX = PLs.getPLsWithPtr(x);
  Node n1 = PLsWithX[0].getNode();
  nullifySL(n1,sel,n2);
  CLSCol CLSsWithN1 = CLSs.getCLSsWithNode(n1);
  SL = new SelectorLink(n1,sel,NU);
  SLs.add(SL);
  foreach(CLS in CLSsWithN1)
    CLS.addSL(SL,'o')

fun nullifySL(Node n1, Sel sel, Node n2)
  SelectorLink SL = SLs.getSL(n1,sel,n2);
  CLSCol CLSsWithN1 = CLSs.getCLSsWithNode(n1);
  foreach(CLS in CLSsWithN1)
    CLS.dropSL(SL,'o');
  CLSCol CLSsWithN2 = CLSs.getCLSsWithNode(n2);
  foreach(CLS in CLSsWithN2)
    CLS.dropSL(SL,'i');
  SLs.remove(SL);
```

```
fun XSelNodeY(Ptr x, Sel sel, Node n2, Ptr y)
  PLCol PLsWithX = PLs.getPLsWithPtr(x);
  Node n1 = PLsWithX[0].getNode();
  nullifySL(n1,sel,n2);
  CLSCol CLSsWithN1 = CLSs.getCLSsWithNode(n1);
  PLCol PLsWithY = PLs.getPLsWithPtr(y);
  foreach(PL in PLsWithY)
    Node n2 = PL.getNode();
    SelectorLink SL = new SelectorLink(n1,sel,n2);
    SLs.add(SL);
    if (n1 == n2) // Cyclic-link
      foreach (CLS in CLSsWithN1)
        CLS.addSL(SL,'c');
    else
      foreach (CLS in CLSsWithN1)
        CLS.addSL(SL,'o');
      CLSCol CLSsWithN2 = CLSs.getCLSsWithNode(n2);
      foreach(CLS in CLSsWithN2)
        CLS.addSL(SL,'i');
```

Figure 3: Pseudo-code for the `XSelNodeNull()`, `nullifySL()`, and `XSelNode()` functions.

and `XSelNodeY()`. `NU` is the null node, so when calling `nullifySL()` from `XSelNodeNull()`, the `CLSs.getCLSsWithNode(n2)` call will return an empty collection and the subsequent foreach loop will have no effect. Note that in the `XSelNodeY()` function, a *cyclic link* is created if `n1==n2` (meaning that `x` and `y` were pointing to the same memory location).

Finally, after the ABS phase (Table 1), the function `norm()` (Table 2) is called when necessary. In this phase, compatible nodes, according to the basic summarization mechanism and available properties, are summarized. Next, for statements `x=y->sel`, `x->sel=y` and `x->sel=NULL`, the `join()` function (Table 2) is used for joining the split graphs after the MAT, ABS and NOR phases. This function creates several working graphs adding only the information (nodes, PLs, SLs and CLSs) of graphs with the same pointer arrangement. Each resulting graph is then normalized in the next NOR phase.

3 COMPLEXITY OF THE APPROACH

In this section, we will focus firstly on the computation of the main parameters which will help us to find the complexity of the method. One of these parameters is the maximum number of graphs generated by our approach. At a given program point, such number of graphs depends on the number of ways of partitioning the live pointer variables at that point. For instance, if the set of live pointer variables is {`p1, p2, p3`}, i.e. three live pointer variables, we could find the following graphs:

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

  - {p1,p3} $\bigcup$ {p2}
  - {p2,p3} $\bigcup$ {p1}
- One graph with three nodes n1 $\bigcup$ n2 $\bigcup$ n3, pointed to by {p1} $\bigcup$ {p2} $\bigcup$ {p3}, respectively.

Therefore, we firstly have to compute the number of ways of partitioning a set of j elements (in our case, j live pointer variables) into k blocks (in this case, nodes). Such a number is named the j-th number of Bell, $B(j)$, and can be computed from $B(j) = \sum_{k=1}^{j} S(j,k)$, where $S(j,k)$ is the Stirling number of the second kind [2],

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

As we are interested in computing the maximum number of graphs generated by our approach, we should consider all the possibilities due to different control flow paths: for instance, a path could generate graphs with just one pointer variable, another path could generate graphs with two pointer variables, etc. Assuming that nv represents the maximum number of live pointer variables at any program point, the maximum number of graphs generated at a point should be the sum of all the ways of partitioning j live pointer variables, from $j = 1$ till $j = nv$, i.e., $\sum_{j=1}^{nv} B(j)$. In addition, we should consider the number of properties evaluated in the shape analysis, np, as well as the range of the values for each property p_j, range that we define as $0 : rp_j$. In this case, each value for each property can contribute with a new graph, therefore the number of graphs should be multiplied by $\left[2^{\sum_{j=1}^{np} rp_j}\right]$. In the case that no properties are considered in the analysis, then $np = 1$ and $rp = 0$.

Let's not forget that we are computing the maximum number of graphs for a program point, i.e. for a statement. With all of this, the **maximum number of graphs per statement**, which we name NG_{stmt}, could be estimated as we indicate in Eq. 1. An obvious way to compute the **maximum number of graphs** generated for the analyzed code, which we will name NG, would be obtained multiplying NG_{stmt} by the number of statements, $nstmt$, as we see in Eq. 2.

$$NG_{stmt} = \left[2^{\sum_{j=1}^{np} rp_j}\right] \cdot \sum_{j=1}^{nv} B(j) \quad (1)$$

$$NG = nstmt \cdot NG_{stmt} = nstmt \cdot \left[2^{\sum_{j=1}^{np} rp_j}\right] \cdot \sum_{j=1}^{nv} B(j) \quad (2)$$

There are other interesting parameters that give us more detailed information about how complex the graphs are and that are measurable: for instance how many nodes does a graph have and how interconnected these nodes are. About the number of nodes, we are interested in computing an upper bound, i.e. the maximum size of a graph. In other words, the **maximum number**

of nodes per graph, which we will name NN. It depends on the maximum number of live pointer variables, nv, because, in a worst case, when none of the pointers are aliased, then each one could point to a different node. NN depends too on the number of properties considered, np and the range of the values for each property p_j, i.e. $0 : rp_j$, because each value for each property can contribute as a new node. With all of this, NN can be estimated as we show in Eq. 3.

$$NN = nv + 2^{\sum_{j=1}^{np} rp_j} \tag{3}$$

About how interconnected the nodes are, we should compute the maximum number os SLs -Selector Links- and the maximum number of CLSs -Coexistent Links Sets-, which are precisely the parameters that encode this information in our approach. We will name the **maximum number of SLs per node**, as NSL_{node} and the **maximum number of SLs per graph**, as NSL. The former depends on the maximum number of links or pointer fields declared in the most complex data structure, nl. It depends too on the maximum number of nodes, to which any node can be connected through a selector link, i.e. $NN - 1$. As the links that can coexist in a given node can be incoming from any other node, outgoing to any other node, and a link to/from itself, then the maximum number of selector links of a given type could be $2 \cdot NN - 1$. Therefore, NSL_{node} can be computed as we see in Eq. 4. $NSL_{node}(NN)$ denotes the maximum number of selector links when we consider that the number of nodes is NN. The maximum number of SLs per graph should be the sum of all the selector links per node when we iteratively incorporate $NSL_{node}(j))$ for each new node, from $j = 1$ till NN, as we see in Eq. 6.

$$NSL_{node} = NSL_{node}(NN) = nl \cdot (2 \cdot NN - 1) \tag{4}$$

$$NSL = \sum_{j=1}^{NN} NSL_{node}(j) = \sum_{j=1}^{NN} nl \cdot (2 \cdot j - 1) = \tag{5}$$

$$= nl \cdot (2 \cdot NN - 1) \cdot (NN - 1) \tag{6}$$

However, the most important parameter is the maximum number of CLSs. For a node, the **maximum number of CLSs** depends on the combination of the maximum number of selector links that can coexist in the node (excluding the links from/to itself, i.e. $2^{NSL_{node}-nl}$, see Eq. 4), as well as the number of variations that can occur for the selector links that are from/to itself, it is, 5^{nl}. The reason of this last parameter is that in a CLS there could be five different states for each selector link from/to the same node: i) it does not appear, ii) it is just incoming, iii) it is just outgoing, iv) it is incoming and outgoing (io, for a summary node), v) it is cyclic (c, for a summary node). Therefore, we could compute the **maximum number of CLSs for a node**, named $NCLS_{node}$, by Eq. 7. Clearly, the **maximum number of CLSs per graph** named $NCLS$, can be computed from Eq. 7 and NN (the maximum number of nodes) as we see in Eq. 8.

$$NCLS_{node} = \left(2^{NSL_{node}-nl}\right) \cdot 5^{nl} = \left(2^{2 \cdot nl \cdot (NN-1)}\right) \cdot 5^{nl} \quad (7)$$

$$NCLS = NCLS_{node} \cdot NN = \left[\left(2^{2 \cdot nl \cdot (NN-1)}\right) \cdot 5^{nl}\right] \cdot NN \quad (8)$$

Eq. 7 is a first approximation that gives us a worst case upper bound for the estimation of the maximum number of CLSs for a node when there is not available information about the data structures. However, such a number can be greatly reduced when we have some information about the data structures. Till now, we have assumed that all the selector links can be incoming to and outgoing from a node. But, in a CLS that represents a real data structure, there is as most, a maximum number of "real" incoming selector links. We will call nli to this important piece of information. For instance, in a singly-linked list $nli = 1$, in a doubly-linked list $nli = 2$, or in a binary tree $nli = 1$. With this information we have to compute all the CLSs that are combinations due to the selector links that are incoming in a node, multiplied by combinations due to the selector links that can be outgoing from the node. In a node, we know that there could be at most: a) $nl \cdot (NN - 1)$ selector links from other (different) nodes, plus b) nl selector links from the same node with attribute io (incoming and outgoing in a summary node), plus c) nl selector links from the same node with attribute c (cyclic in a summary node). Thus, there could be $nl \cdot (NN+1)$ selector links in a node. From them, at most, only nli would appear as incoming selector links in a CLS, therefore, for the computation of the combination of the selector links that are incoming in a node we can do,

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

From the $nl \cdot (NN+1)$ selector links that there could be in a node, we know that in a CSL could be from 0 till nl outgoing links. Thus, for the computation of the combination of the selector links that are outgoing from a node we can do,

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

In other words, a more feasible estimation for the computation of the maximum number of CLSs, $NCLS_{node}$, is given by Eq. 9. Again, the maximum number of CLSs per graph, named $NCLS$, can be computed from Eq. 9 and the maximum number of nodes, NN, as we see in Eq. 10.

$$NCLS_{node} = \sum_{j=1}^{nli} \binom{nl \cdot (NN+1)}{j} \cdot \sum_{k=0}^{nl} \binom{nl \cdot (NN+1)}{k} \quad (9)$$

$$NCLS = NCLS_{node} \cdot NN \quad (10)$$

Table 3: Parameters of our complexity study.

Parameter	Definition	Value
$nstmt$	number of statements to be analyzed	
nv	maximum number of live pointer variables at any program point	
nl	maximum number of links - or pointer fields- declared in the data structures	
nli	maximum number of "real" incoming links in the data structures	
np	number of properties considered in the shape analysis	by default 1
rp_j	upper value in the range of the values for property j, $0 : rp_j$	by default 0
NG_{stmt}	maximum number of graphs per statement	Eq. 1
NG	maximum number of graphs	Eq. 2
NN	maximum number of nodes per graph	Eq. 3
NSL_{node}	maximum number of SLs per node	Eq. 4
NSL	maximum number of SLs per graph	Eq. 6
$NCLS_{node}$	maximum number of CLSs per node	Eq. 9
$NCLS$	maximum number of CLSs per graph	Eq. 10
NPL_{node}	maximum number of PLs per node	Eq. 11
NPL	maximum number of PLs per graph	Eq. 12

For instance, working with a singly-linked lists, we know that $nl = 1$ and $nli = 1$, so applying Eq. 10 we could get $O(NN^3)$ as the maximum number of different CLSs per graph. With a doubly linked list, where $nl = 2$ and $nli = 2$, for Eq. 10 we could get $O(NN^5)$, whereas for a binary tree we should get $O(NN^4)$.

Other parameter of our abstraction, that could be interesting to compute is the **maximum number of PLs per node**, and we will name it as NPL_{node}. It depends on the number of live pointer variables, nv, and it can be easily computed as we can see in Eq. 11. The **maximum number of PLs per graph**, named NPL, is represented in Eq. 12. It is clearly the same, because each pointer variable can appear only once on each graph.

$$NPL_{node} = nv \tag{11}$$

$$NPL = NSL_{node} = nv \tag{12}$$

Table 3 summarizes the main parameters used in our complexity study, as well as their definitions and their values.

Now, our goal is to estimate the worst theoretical performance of our shape analysis framework. Roughly, from Table 1 we see that the cost of analyzing a pointer statement will depend on the cost of the shape analysis operations that the statement invokes. We can start taking a look at the Table 2 and Figures 2

and 3, and estimating the dominant cost for each operation. For the estimation of these dominant costs, we assume a worst case scenario: the maximum number of graphs (i.e. NG) are present at the input program point where our shape analyzer is going to perform the most costly abstract interpretation for the pointer statement. Let's see the cost for each operation:

- The graph splitting operation (SPL) calls to the `split()` function which requires $O(NG \cdot NN \cdot NCLS_{node})$, due to steps (1), (3) and steps (5,6), as we see in the code of Fig. 2.

- The node materialization operation (MAT) calls to the `mater()` function, and when inspecting the code of Fig. 2 we see that this function requires $O(NG \cdot (NSL_{node} + 3 \cdot NCLS_{node}))$, due to steps (1) and steps (4,6). As we know from Eqs. 4 and 9, $NCLS_{node} >> NSL_{node}$, therefore the cost of this function is dominated by $O(NG \cdot NCLS_{node})$.

- The normalization operation (NORM) calls to the `norm()` function. From the code of Fig. 2 we see that this function requires $O(NG \cdot (NN \cdot log(NN) + NCLS + NCLS \cdot log(NCLS)))$, due to steps (1), (3) and (4). In steps (1) and (4) a merge sort algorithm has been implemented to find compatible nodes and to remove the duplicated CLSs. Therefore, the computational cost in this function is dominated by $O(NG \cdot (NN \cdot log(NN) + NCLS \cdot log(NCLS)))$. As we know from Eqs. 3 and 10, $NCLS >> NN$, therefore, the cost of this operation is $O(NG \cdot NCLS \cdot log(NCLS))$.

- The graph joining operation (JOI) calls to the `join()` function. From the code of Fig. 2 we see that this function requires in step (1) $O(NG \cdot log(NG) \cdot NPL \cdot log(NPL))$, because we have to find compatible nodes with the same pointer links (PLs) in all the input graphs.

- The abstract semantic operation (ABS) calls to several functions, depending on the statement being analyzed. From the Fig. 2 we find that the cost of the `XNull()` function is dominated by $O(NG \cdot NCLS_{node})$, the cost of the `XNew()` is dominated by $O(NG \cdot nl)$, the cost of the `XY()` function is dominated by $O(NG \cdot NCLS_{node})$ and the cost of the `XNode()` function is dominated by $O(NG \cdot NCLS_{node})$. From the Fig. 3 we guess that the cost of the `XSelNodeNull()`, `nullifySL()` and `XSelNodeY()` functions are dominated by $O(NG \cdot NCLS_{node})$. Summarizing, the worst case has a complexity of $O(NG \cdot NCLS_{node})$.

From Table 2 we notice that the abstract interpretation of a statement could require the sequence of operations SPL+MAT+ABS+NOR+JOI+NOR, in other words, the complexity of that analysis could be $O(NG \cdot NN \cdot NCLS_{node} + NG \cdot NCLS_{node} + NG \cdot NCLS_{node} + NG \cdot NCLS \cdot log(NCLS) + NG \cdot log(NG) \cdot NPL \cdot log(NPL) + NG \cdot NCLS \cdot log(NCLS))$. Clearly, the complexity is dominated by the NORM operation, so our method has a complexity of $O(NG \cdot NCLS \cdot log(NCLS))$.

The fixed point requires that these operations be done until none can be applied any more. However, we have considered the maximum number of possible graphs, nodes, SLs and CLSs so the complexity to reach the fixed point is included in the previous discussion.

From the previous discussion, we find that the complexity of our approach depends on the number of NG and $NCLS$. From Eqs. 2 and 10 which represent the theoretical maximum values for NG and $NCLS$, we can notice that our approach would have an exponential behaviour as a worst case. However, we think that the issues are: is the worst case reached in practice, and how often? We will address these questions in the next section.

4 EXPERIMENTAL RESULTS

We think that if we want to demonstrate the real applicability of the shape analysis techniques, in particular, the applicability of our approach, then representative benchmarks that use complex data structures should be analyzed. We report here our experiences with such codes.

We have created a Java implementation of our shape analyzer and we have used it to conduct several experiments. The shape analyzer is coupled with a front-end module based on Cetus [5], a compiler infrastructure for source-to-source translation. Cetus is used to parse the input program to an IR that can be easily manipulated. We have designed a custom pass over Cetus IR to translate input program to the format needed by the shape analyzer.

4.1 RESULTS WITH BENCHMARKS THAT MANAGE COMPLEX DATA STRUCTURES

We have considered five complete programs. The first three are codes representative of typical recursive data structures found in heap-directed pointer-based codes. The tests that consider linked lists (singly-linked and doubly-linked) first create the lists, then traverse them. The test working with trees (a binary tree) perform structure traversing during the tree creation. For the last two tests, we have selected the sparse matrix-vector product. Sparse structures are usually built with linked list pointing to another linked lists. In the 4th test, the structures are based in simply-linked lists (s), while on the 5th test, they are based on doubly-linked lists (d). In these two tests, first the input matrix and vector are created, then the output vector is built as the matrix and input vector are traversed. Table 4 shows the structures tested and displays some detailed metrics for the analysis performed. The last two rows of the table present the results for the matrix-vector product codes for which all traversal statements that are not involved in the output vector creation have been *pruned* (p). The output for each test is a set of graphs, for which we have checked that they accurately capture the structures created and traversed on the corresponding code. The complete codes and resulting graphs are available through our web-

Table 4: Structures tested in the shape analyzer, number of analyzed statements, time (sec.) spent on the analysis, total number of generated graphs, and measured number of nodes, selector links and CLSs per graph, in average (and maximum) values.

Code #	Data structure	# stmts	Time	NG	NN, NSL & NCLS
1	Singly-linked list	17	0.61	66	2.42 (4) / 3.54 (7) / 4.65 (13)
2	Doubly-linked list	19	0.65	76	2.55 (4) / 6.85 (8) / 4.56 (13)
3	Binary tree	25	1.25	439	2.72 (4) / 10.52 (20) / 23.71 (65)
4	Matrix-vector(s)	83	6.45	2639	5.64 (9) / 23.57 (39) / 28.63 (68)
5	Matrix-vector(d)	97	6.70	2719	5.64 (9) / 27.89 (51) / 28.33 (68)
6	Matrix-vector(s,p)	66	1.04	590	4.22 (6) / 16.32 (24) / 15.55 (30)
7	Matrix-vector(d,p)	77	1.09	666	4.33 (6) / 18.86 (29) / 15.70 (30)

site[1]. We should mention that no properties were considered in any of the tests performed, so $np = 1$ and $rp = 0$.

The first column of Table 4 identifies each test, the second column the main data structure created/traversed in the corresponding test, while the third column holds the number of analyzed statements. The fourth column shows times for the tests, as measured in a Pentium IV 2.4 GHz with 1 GB RAM, with the `time()` command. The fifth column indicates the total number of graphs generated for each test. In the next column, we show the measured number of nodes, selector links and CLSs per graph, as average values with the measured maximum in brackets.

An interesting finding at this point was to compare, for the tested codes, the behaviour of the analysis times (given by the fourth column in the table) vs the complexity of our method given by expression $O(NG \cdot NCLS \cdot log(NCLS))$, expression which was deduced in the previous section. For the computation of the complexity value for each code, we used the maximum measured values given in Table 4 for NG and $NCLS$. Fig. 4 represents the normalized values for the analysis times (which range from 0.61 sec. to 6.7 sec) and those complexity values (which range from $2.2 \cdot 10^3$ to $7.8 \cdot 10^5$). As we see from the figure, the expression for the complexity can approximate the behaviour of our method and it could be used as a rough estimation of how well is going to perform our approach for different codes.

Anyway, let's see in more detail the parameters which are in the complexity expression, and that control the analysis times of our method. Within the tests codes we see that the number of measured graphs range from a few dozens to a few thousands. For instance, we see that the matrix by vector products (4th and 5th codes) achieve the highest no. of graphs, and take longest, clocking at more than 6 seconds. That's basically due to their bigger number of live pointer variables (nv), most of those are associated with the navigation of different loops in deep nestings. Precisely, nv is the parameter which control the theoretical maximum number of graphs given by Eq. 2. For instance, in the loop that carries out the product in the 5th code (a three level loop nesting), there are

[1] http://www.ac.uma.es/∼asenjo/research/codes.html

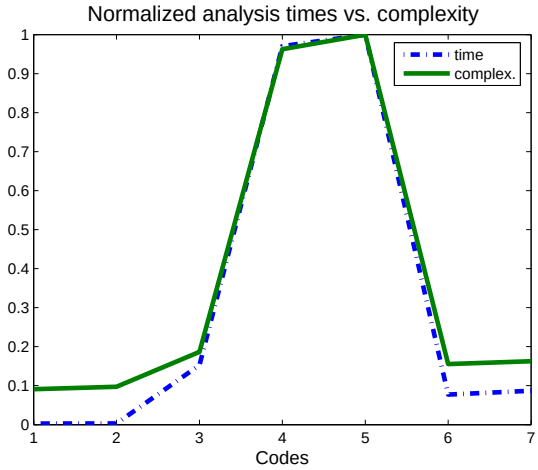


Figure 4: Normalized analysis times (dotted line) vs complexity (solid line).

9 live pointer variables. So, applying Eq. 2, that could suppose a generation of NG=2.56 $\cdot$ 10^6 graphs, as a worst case. Instead of that, the real number of generated graphs is 2.7 $\cdot$ 10^3 (three order of magnitude less). Therefore in this case, our theoretical estimation of NG is a very conservative upper bound. In order to see how accurate our theoretical worst case estimation of NG is, we can compare the measured # graphs given by Table 4 against the worst case estimation of NG given by Eq. 2 for all the codes. Fig. 5 represents the fraction (expressed in %) of measured vs. worst case estimated NG for our codes, where we see that the measured values range from less than 0.15% for the 4th-5th codes to the 77% for the 3rd code. So depending on the code, specially in codes with a high level of control flow nesting, our worst case estimation of NG could be less accurate. But in these cases, the measured real no. of graphs is just a small fraction of the theoretical worst case estimation.

In any case, the most determining factor in the complexity expression, and therefore in the analysis times, seems to be the number of CLSs ($NCLS$). They describe how nodes and links can combine to create all possible memory configurations arising in the program. Eq. 10 computes the maximum theoretical worst case, and again, we get the highest value for the 5th code. In fact, the number of worst case estimated $NCLS$ ranges from 180 for the 1st code, to 20.2 $\cdot$ 10^6 for the mentioned 5th code. Again, we could compare the measured # CLSs per graph given by Table 4 against the theoretical wost case estimation of $NCLS$ given by Eq. 10. Fig. 6 represents the fraction (expressed in %) of measured vs. worst case estimated $NCLS$ for our codes. All the fractional values are lower than 10%. Clearly, the measured no. of CLSs is very low compared to the theoretical worst case, and our measurements seems to indicate that, in real codes, this parameter does not have the exponential behaviour which we had estimated.

In fact, we computed the complexity values for the complexity expression

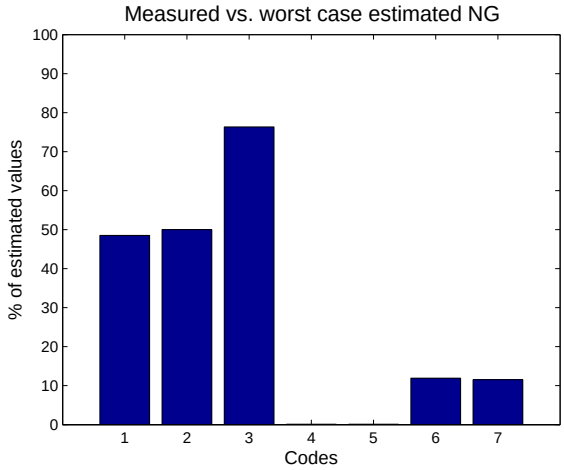


Figure 5: Fraction of measured vs. worst case estimated NG.

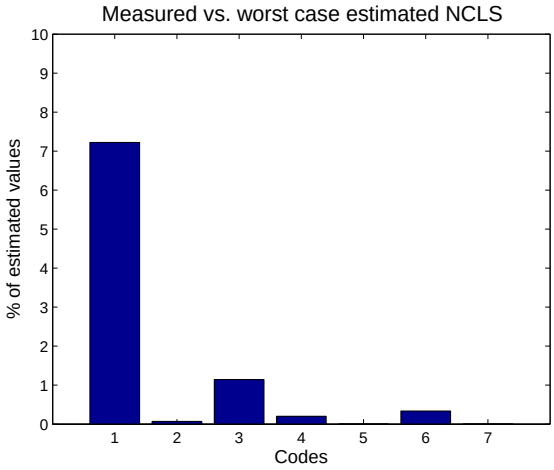


Figure 6: Fraction of measured vs. worst case estimated $NCLS$.

using two set of parameters: in the first one, the measured values of NG and $NCLS$ given in Table 4; in the second one, the wost case estimated values of NG and $NCLS$ given by Eqs. 2 and 10. Fig. 7 represents now the fraction (expressed in %) of the obtained measured vs. worst case estimated values of the complexity, for the codes. In all the cases, the fraction of measured values is under the 5% of the estimated ones. In fact, for the more complex codes (the matrix-vector products), the obtained measured values of the complexity are under the 0.01% of estimated ones. Although our estimation seems to be very conservative, the important point is that our experiments with real codes confirm that our method never reach the worst case, and that in practice, even for the more complex codes, only a very small subset of possibly graphs and

CLSs is generated.

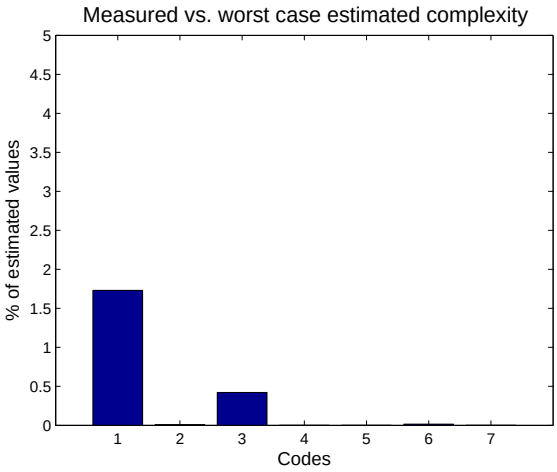
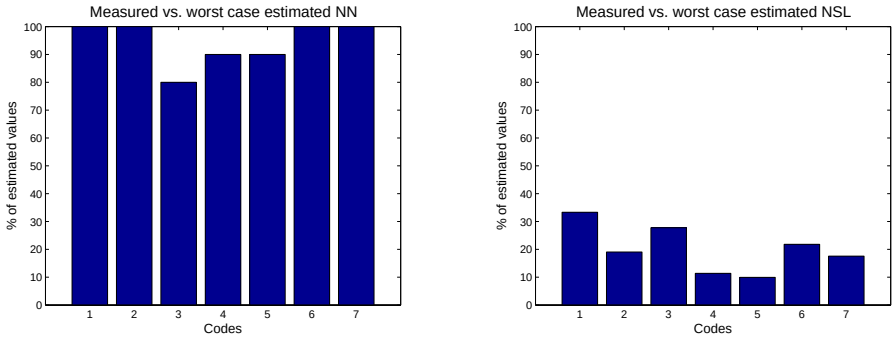


Figure 7: Fraction of measured vs. worst case estimated complexity values.

Other interesting parameters which we have measured and that could give us more idea about the size of the graphs is the number of nodes and number of selector links per graph. The number of nodes depends mostly on the number of live pointers (nv) and the properties considered ($np = 1$ in our tests). The maximum theoretical worst case can be computed by Eq. 3. Fig. 8(a) represents the fraction (expressed in %) of measured vs. worst case estimated NN for our codes. The measured # of nodes against the estimated NN values, range from the 80% (code # 3) to the 100% (codes # 1, 2, 6 and 7). The other measured parameter, the number of selector links depends on the amount of different links (or pointer fields) that the data structures have (nl), as well as the number of nodes per graph. Eq. 6 give us the maximum theoretical worst case. Again, Fig. 8(b) represents the fraction (expressed in %) of measured vs. worst case estimated NSL for our codes. The measured # of selector links against the estimated NSL values, range from the 10% (5th code) to the 35% (1st code). For these two measured parameters, our estimations seems to be quite accurate, but in any case, they do not seem to have such an important impact in the analysis times of our experiments.

To sum up, we can say that our shape analyzer can effectively analyze complex data structures that frequently arise in heap-directed pointer-based codes. We have checked that generated graphs accurately capture the heap structures. In this study, we have found that the complexity of our approach is driven by the no. of graphs and the no. of CLSs, and we have given some expressions to estimate the worst cases. The experimental results seem to indicate that, in spite of the theoretical exponential behaviour of the approach, in practice the no. of generated graphs and specially, the no. of generated CLSs are quite manageable and we can see that the analyses run in reasonable times, specially for such a complex technique. Despite this encouraging results, it is clear that



(a) (b)

Figure 8: (a) Fraction of measured vs. worst case estimated NN; (b) Fraction of measured vs. worst case estimated NSL.

this is a costly technique which is not likely to succeed if used for whole program analysis on bigger codes. Instead, it would be better used within a client analysis module that would focus on a *local analysis* of selected parts of the code.

In this regard, we recently discovered that def-use information can be used to identify the statements directly involved in the creation of recursive data structures. A def-use chain establishes a relationship between the definition point where a value is created and points where it is used. With that information we can automatically determine what are the statements that actually define the shape of dynamic memory and discard all other statements. The shape analysis only needs to analyze these statements to build the graphs that represent the data structure in the program. With this approach we avoid analyzing irrelevant statements that would slow down the shape analysis, as well as we eliminate live pointer variables that could increase the no. of graphs and therefore the complexity of the approach. We have tried this approach on the matrix by vector examples. We removed all traversal statements that were not involved in the output vector creation, and the resulting codes were analyzed again. The results are shown in the last two rows of Table 4 (the 7th and 8th codes). We found that pruned tests produce the same output graphs as their original counterparts, thus capturing memory configuration without any loss in precision. These results prove that def-use driven shape analysis works much better, as the no. of graphs has been greatly decreased, and in a less extend, the no. of CLSs has been decreased too. As a consequence, the analysis times have been reduced importantly. These examples motivate us to tightly integrate shape analysis within client analysis that focuses on the statements of interest. More details of this framework can be found in [1].

5 CONCLUSIONS

In this paper we have discussed several complexity issues related with a shape analysis approach based on Coexistent Links Sets (CLSs). We have found where

the complexity of our analysis resides and through several experiments we have demonstrated that our analysis runs in manageable times when processing complex data structures. However, we have understood that shape analysis works better when combined with high level compiler passes that drive the analysis, as for instance a def-use pass that can be used to focus the shape analysis on the statements of interest.

References

[1] R. Castillo, A. Tineo, F. Corbera, A. Navarro, R. Asenjo, and E.L. Zapata. Towards a versatile pointer analysis framework. *Lecture Notes in Computer Science*, 4128:323–333, 2006.

[2] R.L. Graham, D.E. Knuth, and O. Patashnik. *Concrete Mathematics: A Foundation for Computer Science*, chapter 6: Stirling numbers, pages 257–267. Addison-Wesley, 2nd edition, 1994.

[3] B. Hackett and R. Rugina. Region-based shape analysis with tracked locations. In *ACM SIGPLAN Symposium on Principles of Programming Languages (POPL'05)*, pages 310–323, Long Beach, California, USA, January 2005.

[4] B. Jeannet, A. Loginov, T. Reps, and M. Sagiv. A relational approach to interprodedural shape analysis. In *11th International Static Analysis Symposium*, Verona, Italy, August 2004.

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

[6] M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. *ACM Transactions on Programming Languages and Systems*, 2002.

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