# – Cover Page –

## A Formal Presentation of Shape Analysis graphs and operations

F. Corbera, A. Navarro, R. Asenjo, A. Tineo and E. Zapata
Department of Computer Architecture. University of Málaga,
Spain.
{corbera,angeles,asenjo,tineo,ezapata}@ac.uma.es

**Abstract**

**Keywords:**

## 1 Introduction

To formalize the description of our model, we use the simple statements and definitions shown in Fig. 1. We only consider statements dealing with pointers as the ones shown in the figure (they are C-like imperative statements with dynamic allocation), because other complex pointer statements can be transformed into several of these simple pointer statements in a preprocessing stage. We assume that the types of all pointer variables and objects are explicitly declared. Each object type has a set of pointer fields associated with it, and the set of all these pointer fields that are defined in the program is what we call $SEL$.

| | |
|---|---|
| programs: | $prog \in P,\ P =< STMT, PTR, TYPE, SEL >$ |
| statements: | $s \in STMT,\ s ::= x = NULL \mid x = malloc() \mid free(x) \ \mid x = y$ $\mid x \rightarrow sel = NULL \mid x \rightarrow sel = y \mid x = y \rightarrow sel$ |
| pointer variables: | $x, y \in PTR$ |
| type objects: | $t \in TYPE$ |
| selectors fields: | $sel \in SEL$ |

Figure 1: Simple statements and definitions.

## 2 Concrete Heap

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

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

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

| | |
|---|---|
| Pointer Map (in the concrete domain): | $\mathcal{PM}^c$: $PTR \longrightarrow L$ |
| Selector Map (in the concrete domain): | $\mathcal{SM}^c$: $L \ \times \ SEL \longrightarrow (L \cup null)$ |

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

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

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

- $\mathcal{SM}^c$ models points to relations between locations $l_1$ and $l_2$, through selector fields $sel$:

  $\forall l_1 \in L \;\; s.t. \;\; \mathcal{T}(l_1) = t \wedge \forall sel \in \mathcal{SF}(t), \exists l_2 \in (L \cup null) \;\mid\; \mathcal{SM}^c(l_1, sel) = l_2.$

  We use a tuple $slc =< l_1, sel, l_2 >$, which we name *concrete selector link* to represent this relation. The set of all concrete selector links is called $SLc$.

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

## 3 Abstract Heap

Our abstract domain is based on a heap graph model. Each node may represent a set of concrete memory locations, whereas each edge may represent a pointer variable or a set of selectors with the same field name.

The abstract domain for the nodes, $N = \mathcal{P}(PTR) \cup \{null\}$ (which includes a special node named $null$) indicates that the nodes are distinguishable through the set of pointer variables which point to them.

Now we define three mapping functions $\mathcal{LM}$, $\mathcal{PM}^a$, $\mathcal{SM}^a$ to model the relationship between memory locations and nodes in the concrete and abstract domain, as well as the connections of pointers variables and selector fields to nodes in the abstract heap. The mapping functions $\mathcal{LM}$ and $\mathcal{PM}^a$ are total functions, while $\mathcal{SM}^a$ is a multivalued function. They can be defined as follows:

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

- $\mathcal{LM}$ assigns a node $n$ to a concrete memory location $l$:

  $\forall l \in L, \exists n \in N \;\mid\; \mathcal{LM}(l) = n.$

- $\mathcal{PM}^a$ maps a pointer variable $x$ which points to a location $l$ in the concrete domain, to a node $n$ in the abstract domain:

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

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

Usually, we use the tuple $pl =< x, n >$, which we name *pointer link*, to represent this binary relation. The set of all pointer links is named now $PL$.

- $\mathcal{SM}^a$ models points to relations between locations $l_i$ and $l_j$ through selector field $sel$ in the concrete domain, as relations between nodes $n1$ and $n2$:

  $\forall \mathcal{SM}^c(l_i, sel) = l_j \subset MC, \exists n_1 \in (N - null) \wedge \exists n_2 \in N \;\; s.t. \;\; \mathcal{LM}(l_i) = n_1 \wedge \mathcal{LM}(l_j) = n_2 \mid \mathcal{SM}^a(n_1, sel) = n2.$

  Again, we use a tuple $sl =< n_1, sel, n2 >$, which now we name *selector link* to represent this relation. The set of all selector links is called $SL$.

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

*Selector Links with attributes.*

We define a set of attributes, $ATT = \{i, o, c, s\}$, where each element $att \in ATT$ codifies information about the direction and nature of a selector link when it is related to a node. Intuitively, $att = i$ stands for an input link, $att = o$ for an output link, $att = c$ for a cyclic link, and $att = s$ for a shared one. They will be defined more formally later on. From the set $ATT$ we define a new domain $ATTSL = \mathcal{P}(ATT)$, where each element of this new domain $attsl \in ATTSL$ represents a possible combination of attributes that describe the characteristics of a selector link when it is associated to a node. The join operation in the $ATTSL$ domain, $\uplus$, will be defined in Section 4.

In particular, from the set of all selector links, $SL$ and from $ATTSL$ we define the domain $SL_{att} = SL \times ATTSL$. An element $sl_{att}$ in this domain, which we call a *selector link with attributes*, is represented as a tuple $sl_{att} =< sl, attsl >$, where $sl \in SL$ and $attsl \in ATTSL$.

*Coexistent Links Set.*

The key feature of our model is to be able to maintain the connectivity and aliasing information that can coexist in an abstract node, even when the node represents different memory locations with different connection patterns. This is achieved through the Coexistent Links Set abstraction. The domain of our Coexistent Links Set abstraction $CLS^a = (\mathcal{CLM})$ is defined in terms of a mapping function $\mathcal{CLM}$ as follows:

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

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

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

where:

$$\begin{aligned}
PL_n &= \{pl \in PL \;\; s.t. \;\; pl =< x, n >\} \\
SL_n &= \{sl_{att} \in SL_{att} \;\; s.t. \;\; sl_{att} =<< n_1, sel, n_2 >, attsl >, \;\; being \; (n_1 = n \vee n_2 = n)\}
\end{aligned}$$

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

Let $cls_n = \{PL_n, SL_n\}$ be. For each $sl_{att} =<< n_1, sel, n_2 >, attsl > \in SL_n$ we can find one or more of the following cases:

If $l_1 \neq l_2$ and $\exists slc_1(l_1, sel, l) \wedge \exists slc_2(l_2, sel, l) \;\; s.t. \;\; (\mathcal{LM}(l_1) = \mathcal{LM}(l_2) = n_1 \wedge \mathcal{LM}(l) = n_2 = n) \Longrightarrow s \in attsl$

else

If $l_1 \neq l_2$ and $\exists slc =< l_1, sel, l_2 > \;\; s.t. \;\; (\mathcal{LM}(l_1) = n_1 \wedge \mathcal{LM}(l_2) = n_2 = n) \Longrightarrow i \in attsl.$

If $l_1 \neq l_2$ and $\exists slc =< l_1, sel, l_2 > \;\; s.t. \;\; (\mathcal{LM}(l_1) = n_1 = n \wedge \mathcal{LM}(l_2) = n_2) \Longrightarrow o \in attsl.$

If $l_1 = l_2 = l$ and $\exists slc =< l, sel, l > \;\; s.t. \;\; (\mathcal{LM}(l) = n_1 = n_2 = n) \Longrightarrow c \in attsl.$

The set of all the $cls_n$ associated to a node $n$ is called $CLS_n$, and it codifies all the possible patterns of aliasing and connectivity that can coexist in a given node $n$. In addition, for all the nodes $n$ defined in our abstract heap, we can create the set $CLS = \{CLS_n, \forall n \in N\}$.

Shape Graph

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

We restrict this abstract domain by defining a *normal form* of the shape graphs. We will need the auxiliary functions `Compatible_Node()` and `Path()`, that are described in Fig. 2. We say that a shape graph $sg^i =< N^i, CLS^i >$ is in normal form if:

1. It has not compatible nodes: $\nexists n_1, n_2 \in N^i\ \ s.t.\ \ Compatible\_Node(n_1, n_2, CLS_{n1}, CLS_{n2}) = TRUE$
2. It has not unreachable nodes: $\forall n_1 \in N^i, \exists pl_1 =< x, n_1 >\subset CLS_{n1} \vee (\exists n_2 \in N^i\ \ s.t.\ \ \exists pl_2 =< x, n_2 >\subset CLS_{n2} \wedge Path(n_2, n_1, CLS^i) = TRUE)$
3. A pointer variable unambiguously points to one node: $\forall n_1, n_2 \in N^i\ \ s.t.\ \ n_1 \neq n_2$, If $\exists pl_1 =< x, n_1 >\subset CLS_{n1} \Longrightarrow \nexists pl_2 =< x, n_2 >\subset CLS_{n2}$
4. The selector links of connected nodes, are coherent: $\forall n_1, n_2 \in N^i\ \ s.t.\ \ n_1 \neq n_2$, If $\exists sl_{att} =<< n_1, sel_k, n_2 >, attsl >\subset CLS_{n1} \Longrightarrow \exists sl_{att} =<< n_1, sel_k, n_2 >, attsl' >\subset CLS_{n2}$

```
Compatible_Node()
 Input: $n_1, n_2, CLS_{n1}, CLS_{n2}$     # two nodes and their CLS's
 Output: $TRUE/FALSE$

 If ($\forall pl_1 =< x, n_1 >\subset CLS_{n1}, \exists pl_2 =< x, n_2 >\subset CLS_{n2} \wedge$
     $\forall pl_2 =< y, n_2 >\subset CLS_{n2}, \exists pl_1 =< y, n_1 >\subset CLS_{n1}$ ),
        return($TRUE$)
 else
        return($FALSE$)
end
```

(a)

```
Path()
 Input: $n_1, n_2, CLS$   # two nodes and a CLS set
 Output: $TRUE/FALSE$

 If ($\exists sl_{atti} =< n_1, sel_0, n_a >, attsli >\subset CLS_{n1}$,
     $sl_{attj} =< n_a, sel_1, n_b >, attslj >\subset CLS_{na}, \ldots$
     $\ldots, sl_{attk} =< n_k, sel_k, n_2 >, attslk >\subset CLS_{nk}$),
        return($TRUE$)
 else
        return($FALSE$)
end
```

(b)

Figure 2: (a) Check when two nodes are compatible; (b) Compute if exists a path between two nodes.

Reduced Set of Shape Graphs

As we mentioned previously, our abstract heap is modeled as a multi-graph. We call *reduced set of shape graphs* to the set of shape graphs that represents the state of the heap at a given program point s: $RSSG^s = \{sg^i \in SG \ \ s.t. \ \ sg^i \ \ is \ \ in \ \ normal \ \ form\}$

Again, we impose a restriction in this set of graphs, and it is that the set is in normal form. We say that a reduced set of shape graphs, $RSSG^s = \{sg^i\}$ is in normal form if:

1. It has not compatible shape graphs: $\nexists sg^1, sg^2 \in RSSG^s \ \ s.t. \ \ \texttt{Compatible\_SG}(sg^1, sg^2) = TRUE$.

   The auxiliary function $\texttt{Compatible\_SG}(sg^1, sg^2)$ is described now in Fig. 3. The function checks that for each node of graph $sg^1$ pointed to by a pointer (or group of pointer variables), there is another node of graph $sg^2$ pointed to by the same pointer (or group of pointer variables). The same check is done for all the nodes in graph $sg^2$. In other words, the function checks that all the nodes pointed to by pointer variables in graphs $sg^1$ and $sg^2$ are compatible. In this case, we would say that the two graphs are compatible, and they could be joined in a new summary graph (see function 15). Clearly, only the graphs with the same alias relationships can be joined.

The constraint that a reduced set of shape graphs $RSSG^s$ is in normal form ensures that each graph $sg^i \in RSSG^s$ represents a different alias configuration. This issue will become very useful when implementing the abstract semantics of several statements.

```
Compatible_SG()
 Input: $sg^1 =< N^1, CLS^1 >, sg^2 =< N^2, CLS^2 >$          # two shape graphs
 Output: $TRUE/FALSE$

 If $\Big( (\forall n_i \in N^1 \ \ s.t. \ \ \exists pl =< x, n_i > \subset CLS_{ni} \wedge \exists n_j \in N^2 \ \ s.t. \ \ \texttt{Compatible\_Node}(n_i, n_j, CLS_{ni}, CLS_{nj}) = TRUE) \wedge$
    $(\forall n_j \in N^2 \ \ s.t. \ \ \exists pl =< y, n_j > \subset CLS_{nj} \wedge \exists n_i \in N^1 \ \ s.t. \ \ \texttt{Compatible\_Node}(n_j, n_i, CLS_{nj}, CLS_{ni}) = TRUE) \Big)$,
        return($TRUE$)
 else
        return($FALSE$)
end
```

Figure 3: Check when two shape graphs are compatible.

## 4 Abstract Semantics and Operations

In this section we describe the abstract semantic associated to each statement and present the principal algorithms used in the analysis.

### 4.1 Abstract Semantics

We formulate our analysis as a dataflow analysis that computes a reduced set of shape graphs at each program point. For each statement in the program, $s \in STMT$, we define two program points: $\bullet s$ is the program point before $s$, and $s\bullet$ is the program point after $s$. Therefore, the result of the analysis is a reduced set of shape graphs, $RSSG^{\bullet s}$ before $s$, and $RSSG^{s\bullet}$ after that. Let `pred()` map statements to their predecessor statements in the control flow (these can be easily computed from the syntactic structure of control statements). Fig. 4 shows the dataflow equations.

$$
\begin{array}{ll}
\text{[JOIN]:} & RSSG^{\bullet s} = \bigsqcup^{RSSG}_{s' \in pred(s)} RSSG^{s'\bullet} \\
\text{[TRANSF]:} & RSSG^{s\bullet} = AS_s(RSSG^{\bullet s}), \text{ where} \\
& \quad AS_{s::=\ x=null}(RSSG^{\bullet s}) = \bigsqcup^{RSSG}_{sg^i \in RSSG^{\bullet s}} XNull(sg^i, x) \\
& \quad AS_{s::=\ x=malloc()}(RSSG^{\bullet s}) = \bigsqcup^{RSSG}_{sg^i \in RSSG^{\bullet s}} XNew(sg^i, x) \\
& \quad AS_{s::=\ free(x)}(RSSG^{\bullet s}) = \bigsqcup^{RSSG}_{sg^i \in RSSG^{\bullet s}} FreeX(sg^i, x) \\
& \quad AS_{s::=\ x=y}(RSSG^{\bullet s}) = \bigsqcup^{RSSG}_{sg^i \in RSSG^{\bullet s}} XY(sg^i, x, y) \\
& \quad AS_{s::=\ x\rightarrow sel=null}(RSSG^{\bullet s}) = \bigsqcup^{RSSG}_{sg^i \in RSSG^{\bullet s}} XselNull(sg^i, x, sel) \\
& \quad AS_{s::=\ x\rightarrow sel=y}(RSSG^{\bullet s}) = \bigsqcup^{RSSG}_{sg^i \in RSSG^{\bullet s}} XselY(sg^i, x, sel, y) \\
& \quad AS_{s::=\ x=y\rightarrow sel}(RSSG^{\bullet s}) = \bigsqcup^{RSSG}_{sg^i \in RSSG^{\bullet s}} XYsel(sg^i, x, y, sel)
\end{array}
$$

Figure 4: Dataflow equations.

We model the analysis of individual statements computing a transfer function for each one. To simplify the formal definitions of the transfer functions we use the functions `XNull`(), `XNew`(), `FreeX()`, `XY`(), `XselNull`(), `XselY`() and `XYsel`() to describe the transformations that take place in the abstract heap when a simple statement `s` is interpreted (see Figures 7, 8, 9, 10, 11, 12, 13 respectively). The operator $\bigsqcup^{RSSG}$ represents the join operation in the $RSSG$ domain. It is described as a function too, in Fig. 6. Basically, the transfer functions for the `x=null`, `x=malloc()`, `free(x)` and `x=y` statements, take each shape graph from the input set $RSSG^{\bullet s}$, transform it according to the statement semantic, and later join all the transformed graphs to build the output set $RSSG^{s\bullet}$. On the other hand, the transfer functions for the `x->sel=null`, `x->sel=y` and `x=y->sel` statements, take each shape graph from the input set $RSSG^{\bullet s}$, split it (following the `x->sel` or `y->sel` path) in a temporal set of graphs (generating a intermediate $RSSG^1$); next for each one of the temporal graphs in that intermediate set, the transfer functions materialize an individual node (the one unambiguously pointed to by `x->sel` or by `y->sel`), transform the graph according to the statement semantic, normalize it, summarize compatible temporal graphs, and finally join all the resultant RSSG's to build the output set $RSSG^{s\bullet}$. More details about the functions and the operations that involve can be found in Section 4.2.

We present in Fig. 5 a worklist algorithm for solving the dataflow equations presented in Fig. 4. The input of our worklist algorithm is a program $P$ and an initial $RSSG^{in} = \emptyset$, whereas the output is the $RSSG^{out}$ resultant at the exit program point, assuming that the exit point is statement $sr \in STMT$. This algorithm also computes the resultant $RSSG^{s\bullet}$ at each program point. Lines 1-3 perform the initialization, where the $RSSG$ at the input of the program entry point (in our case statement $se \in STMT$) is initialized with $RSSG^{in}$. Next, the algorithm processes the worklist using the loop defined in lines 4-12. At each iteration, it removes, in program lexicographic order, a statement for the worklist, computes the join of the $RSSG$'s from the predecessors as the statement input (`pred(s)`), and then it applies the corresponding transfer function. In the case in which the resultant RSSG has changed, the algorithm adds the successors of the statement under consideration (`succ(s)`) to the worklist (line 10).

## 4.2 Operations

As we have mentioned previously, to simplify the formal definitions of the join operators and transfer functions, we have incorporated them in the paper as functions. In addition, we have incorporated other useful instrumental functions. We describe all of them here in more detail. The bold face lines represent actions that only take place when, in order to avoid aggressive summarizations, properties are considered in the analysis (see Section 4.4 for details about the properties supported).

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

1:  Create $W = STMT$
2:  $RSSG^{\bullet se} = RSSG^{in}$
3:  $\forall s \in STMT \rightarrow RSSG^{s\bullet} = \emptyset$
4:  repeat
5:      Remove $s$ from $W$ in lexicographic order
6:      $RSSG^{\bullet s} = \bigsqcup^{RSSG}_{s' \in pred(s)} RSSG^{s'\bullet}$
7:      $RSSG^{s\bullet} = AS_s(RSSG^{\bullet s})$
8:      If ($RSSG^{s\bullet}$ has changed),
9:          forall $s' \in succ(s)$,
10:             $W = W \cup s'$
11:         endfor
12: until ($W = \emptyset$)
13: $RSSG^{out} = RSSG^{sr\bullet}$
14: return($RSSG^{out}$)
end
```

Figure 5: The worklist algorithm. It computes the $RSSG^{s\bullet}$ at each program point.

```
Join_RSSG() ( $\bigsqcup^{RSSG}$ )
  Input: $RSSG^1, RSSG^2$   # two reduced sets of shape graphs
  Output: $RSSG^k$   # a reduced set of shape graphs in normal form

  $RSSG^k = \emptyset$
  Create $RSSG^{k'} = RSSG^1 \cup RSSG^2$
  $RSSG^k = Summarize_{RSSG}(RSSG^{k'})$
  return($RSSG^k$)
end
```

Figure 6: The operator $\bigsqcup^{RSSG}$ as the Join_RSSG() function.

```
XNull()
  Input: $sg^1 =< N^1, CLS^1 >, x \in PTR$          # a shape graph, and a pointer variable
  Output: $RSSG^k$                                  # a graph in a reduced set of shape graphs

  Create $List'[N] = \emptyset$; Create $List'[CLS] = \emptyset$
  Find $n_i \in N^1 \;\; s.t. \;\; \exists pl =< x, n_i > \subset CLS_{ni}$
  forall $cls_{ni} = \{PL_{ni}, SL_{ni}\} \in CLS_{ni}$,
      Create $PL'_{ni} = PL_{ni} - pl$                  # Remove the corresponding pl
      Create $SL'_{ni} = SL_{ni}$
      Create $cls'_{ni} = \{PL'_{ni}, SL'_{ni}\}$
      $List'[CLS] = List'[CLS] \cup cls'_{ni}$
      $List'[N] = List'[N] \cup n_i$
  endfor
  forall $n_j \in N^1 \;\; s.t. \;\; n_j \neq n_i$,
      $List'[CLS] = List'[CLS] \cup CLS_{nj}$
      $List'[N] = List'[N] \cup n_j$
  endfor
  $sg^k =$Summarize_SG$(List'[N], List'[CLS])$        # Summarize compatible nodes
  $RSSG^k = sg^k$
  return($RSSG^k$)
end
```

Figure 7: `XNull()` function.

```
XNew()
  Input: $sg^1 =< N^1, CLS^1 >, x \in PTR$          # a shape graph, and a pointer variable
  Output: $RSSG^k$                                  # a graph in a reduced set of shape graphs

  $RSSG^1 =$XNull$(sg^1, x)$   being   $RSSG^1 = sg^2 =< N^2, CLS^2 >$
  # Create a new node $n_p$
  $\forall \mathbf{prop} \in \mathbf{PROP} \Longrightarrow \mathcal{PPM}_{\mathbf{prop}}(\mathbf{n_p}) =$ Update_Property$(\mathbf{s}, \mathbf{prop})$
  Create $N^k = N^2 \cup n_p$
  Create $pl =< x, n_p >$
  Create $PL_{np} = pl$; Create $SL_{np} = \emptyset$
  forall $sel_j \in SEL$
      Create $sl_{att} =<< n_p, sel_j, null >, attsl = \{o\} >$
      $SL_{np} = SL_{np} \cup sl_{att}$
  endfor
  Create $cls_{np} = \{PL_{np}, SL_{np}\}$
  Create $CLS_{np} = cls_{np}$
  Create $CLS^k = CLS^2 \cup CLS_{np}$
  Create $sg^k =< N^k, CLS^k >$
  $RSSG^k = sg^k$
  return($RSSG^k$)
end
```

Figure 8: `XNew()` function.

```
FreeX()
  Input: $sg^1 =< N^1, CLS^1 >, x \in PTR$          # a shape graph, and a pointer variable
  Output: $RSSG^k$                                   # a graph in a reduced set of shape graphs

  Find $n_i \in N^1$ s.t. $\exists pl =< x, ni >\subset CLS_{ni} (being\ \ CLS_{ni} \subset CLS^1)$
  Create $N^2 = N^1 - n_i$                           # Remove the node
  Create $CLS^2 = CLS^1 - CLS_{ni}$                  # Remove the corresponding CLS
  forall $n_j \in N^2$,                 # Remove inconsistent sel. links from other nodes
      Create $CLS'_{nj} = CLS_{nj}$
      Find $\{cls_{nj} \subset CLS_{nj}\ \ s.t.\ \ \exists sl_{att} \subset cls_{nj}\ \ being\ \ sl_{att} =<< n_j, sel, n_i >, attsl >\} ::= \{cls_{nj}\ \ s.t.\ \ cond.A\}$,
      forall $cls_{nj} = \{PL_{nj}, SL_{nj}\}\ \ s.t.\ \ cond.A$
          Create $sl'_{att} =<< n_j, sel, null >, attsl = \{o\} >$
          Create $SL'_{nj} = SL_{nj} - sl_{att} \cup sl'_{att}$
          Create $PL'_{nj} = PL_{nj}$
          Create $cls'_{nj} = \{Pl'_{nj}, SL'_{nj}\}$
          $CLS'_{nj} = CLS'_{nj} - cls_{nj} \cup cls'_{nj}$
      endfor
  endfor
  $N^k = N^2$ ; $CLS^k = \cup_{\forall n_j \in N^2} CLS'_{nj}$
  Create $sg^k =< N^k, CLS^k >$
  $RSSG^k = sg^k$
  return($RSSG^k$)
end
```

Figure 9: FreeX() function.

```
XY()
  Input: $sg^1 =< N^1, CLS^1 >, x, y \in PTR$        # a shape graph, and two pointer variables
  Output: $RSSG^k$                                   # a graph in a reduced set of shape graphs

  $RSSG^1$ =XNull($sg^1, x$)  being  $RSSG^1 = sg^2 =< N^2, CLS^2 >$
  Find $n_i \in N^2\ \ s.t.\ \ \exists pl_1 =< y, n_i >\subset CLS_{ni}\ \ (being\ \ CLS_{ni} \subset CLS^2)$
  #  Modify $CLS_{ni}$
  Create $CLS'_{ni} = CLS_{ni}$
  forall $cls_{ni} = \{PL_{ni}, SL_{ni}\} \in CLS_{ni}$,
      Create $pl'_1 =< x, n_i >$                     # Update PL
      Create $PL'_{ni} = PL_{ni} \cup pl'_1$
      Create $SL'_{ni} = SL_{ni}$
      Create $cls'_{ni} = \{PL'_{ni}, SL'_{ni}\}$
      $CLS'_{ni} = CLS'_{ni} - cls_{ni} \cup cls'_{ni}$
  endfor
  Create $N^k = N^2$; Create $CLS^k = CLS^2 - CLS_{ni} \cup CLS'_{ni}$
  Create $sg^k =< N^k, CLS^k >$
  $RSSG^k = sg^k$
  return($RSSG^k$)
end
```

Figure 10: XY() function.

```
XselNull()
  Input: $sg^1 =< N^1, CLS^1 >, x \in PTR, sel \in SEL$        # a shape graph, a pointer variable and a selector field
  Output: $RSSG^k$                                             # a reduced set of shape graphs in normal form

  Create $RSSG^{k'} = \emptyset$
  $RSSG^1 =$Split$(sg^1, x, sel)$
  forall $sg^i =< N^i, CLS^i >\in RSSG^1$,
      $sg^j =< N^j, CLS^j >= Materialize\_Node(sg^i, x, sel)$
      Find $n_k \in N^j \ s.t. \ \exists pl_1 =< x, n_k >\subset CLS_{nk} \ (being \ CLS_{nk} \subset CLS^j)$
      # Modify $CLS_{nk}$
      Create $CLS'_{nk} = CLS_{nk}$
      forall $cls_{nk} = \{PL_{nk}, SL_{nk}\} \subset CLS_{nk}$,
          If $(\exists sl_{att1} \subset cls_{nk} \ being \ sl_{att1} =<< n_k, sel, n_p >, attsl1 >)$,
              Create $sl'_{att1} =<< n_k, sel, null >, attsl1' = \{o\} >$
              Create $SL'_{nk} = SL_{nk} - sl_{att1} \cup sl'_{att1}$
              Create $PL'_{nk} = PL_{nk}$
              Create $cls'_{nk} = \{PL'_{nk}, SL'_{nk}\}$
              $CLS'_{nk} = CLS'_{nk} - cls_{nk} \cup cls'_{nk}$
              # Modify $CLS_{np}$
              Create $CLS'_{np} = CLS_{np}$
              forall $cls_{np} = \{PL_{np}, SL_{np}\} \subset CLS_{np} \ (being \ CLS_{np} \subset CLS^j)$,
                  If $(\exists sl_{att2} \subset cls_{np} \ being \ sl_{att2} =<< n_k, sel, n_p >, attsl2 >)$,
                      Create $SL'_{np} = SL_{np} - sl_{att2}$
                      Create $PL'_{np} = PL_{np}$
                      Create $cls'_{np} = \{PL'_{np}, SL'_{np}\}$
                      $CLS'_{np} = CLS'_{np} - cls_{np} \cup cls'_{np}$
              endfor
      endfor
      Create $N^{j'} = N^j$
      Create $CLS^{j'} = CLS^j - CLS_{nk} \cup CLS'_{nk} - CLS_{np} \cup CLS'_{np}$
      Create $sg^{j'} =< N^{j'}, CLS^{j'} >$
      $sg^{j''} =$Normalize_SG$(sg^{j'})$
      $RSSG^{k'} = RSSG^{k'} \cup sg^{j''}$
  endfor
  $RSSG^k =$Summarize_RSSG$(RSSG^{k'})$          # Summarize compatible graphs
  return($RSSG^k$)
end
```

Figure 11: XselNull() function.

```
XselY()
  Input: $sg^1 =< N^1, CLS^1 >, x \in PTR, sel \in SEL, y \in PTR$      # a shape graph, two pointer vars and a selector field
  Output: $RSSG^k$                                                       # a reduced set of shape graphs in normal form

  Create $RSSG^{k'} = \emptyset$
  $RSSG^1 =$Split$(sg^1, x, sel)$
  forall $sg^i =< N^i, CLS^i > \in RSSG^1$,
      $RSSG^2 =$XselNull$(sg^i, x, sel)$
      forall $sg^j =< N^j, CLS^j > \in RSSG^2$
          Find $n_k \in N^j \;\; s.t. \;\; \exists pl_1 =< x, n_k > \subset CLS_{nk} \;\; (being \;\; CLS_{nk} \subset CLS^j)$
          Find $n_p \in N^j \;\; s.t. \;\; (\exists pl_2 =< y, n_p > \subset CLS_{np} \wedge n_p \neq null) \;\; (being \;\; CLS_{np} \subset CLS^j)$
          # Modify $CLS_{nk}$
          Create $CLS'_{nk} = CLS_{nk}$
          forall $cls_{nk} = \{PL_{nk}, SL_{nk}\} \in CLS_{nk}$,
              If $(\exists sl_{att1} \subset cls_{nk} \;\; being \;\; sl_{att1} =<< n_k, sel, null >, attsl >)$,
                  Create $sl'_{att} =<< n_k, sel, n_p >, attsl' >$
                  If $(n_k = n_p) \rightarrow attsl' = \{c\}$
                  else $\;\rightarrow attsl' = \{o\}$
                  Create $SL'_{nk} = SL_{nk} - sl_{att1} \cup sl'_{att}$
                  Create $PL'_{nk} = PL_{nk}$
                  Create $cls'_{nk} = \{PL'_{nk}, SL'_{nk}\}$
                  $CLS'_{nk} = CLS'_{nk} - cls_{nk} \cup cls'_{nk}$
          endfor
          # Modify $CLS_{np}$
          Create $CLS'_{np} = CLS_{np}$
          forall $cls_{np} = \{PL_{np}, SL_{np}\} \in CLS_{np} \; (being \;\; n_p \neq n_k)$,
              Create $sl'_{att} =<< n_k, sel, n_p >, attsl' = \{i\} >$
              Create $SL'_{np} = SL_{np} \cup sl'_{att}$
              Create $PL'_{np} = PL_{np}$
              Create $cls'_{np} = \{PL'_{np}, SL'_{np}\}$
              $CLS'_{np} = CLS'_{np} - cls_{np} \cup cls'_{n_l}$
          endfor
          Create $N^{j'} = N^j$
          Create $CLS^{j'} = CLS^j - CLS_{nk} \cup CLS_{nk} - CLS_{np} \cup CLS_{np}$
          Create $sg^{j'} =< N^{j'}, CLS^{j'} >$
          $RSSG^{k'} = RSSG^{k'} \cup sg^{j'}$
      endfor
  endfor
  $RSSG^k =$Summarize_RSSG$(RSSG^{k'})$          # Summarize compatible graphs
  return($RSSG^k$)
end
```

Figure 12: XselY() function.

```
XYsel()
  Input: $sg^1 =< N^1, CLS^1 >, x, y \in PTR, sel \in SEL$          # a shape graph, two pointer variables and a selector field
  Output: $RSSG^k$                                                   # a reduced set of shape graphs in normal form

  Create $RSSG^{k'} = \emptyset$
  $RSSG^1$ =XNull($sg^1, x$)  being  $RSSG^1 = sg^2 =< N^2, CLS^2 >$
  $RSSG^2$ =Split($sg^2, y, sel$)
  forall $sg^i =< N^i, CLS^i > \in RSSG^2$,
      $sg^j =< N^j, CLS^j >$=Materialize_Node($sg^i, y, sel$)
      Find $n_k \in N^j$ $s.t.$ $\exists pl_1 =< y, n_k > \subset CLS_{nk} (being\ CLS_{nk} \subset CLS^j)$
      If ($\exists sl_{att1} \subset cls_{nk}$ $s.t.$ $sl_{att1} =<< n_k, sel, n_p >, attsl > \wedge n_p \neq null\}$),
          # Modify $CLS_{np}$
          Create $CLS'_{np} = CLS_{np}$
          forall $cls_{np} = \{PL_{np}, SL_{np}\} \in CLS_{np}$,
                  Create $pl' =< x, n_p >$; Create $PL'_{np} = PL_{np} \cup pl'_{np}$
                  Create $SL'_{np} = SL_{np}$
                  Create $cls'_{np} = \{PL'_{np}, SL'_{np}\}$
                  $CLS'_{np} = CLS'_{np} - cls_{np} \cup cls'_{np}$
          endfor
          Create $N^{j'} = N^j$; Create $CLS^{j'} = CLS^j - CLS_{np} \cup CLS'_{np}$
      else        # Case $n_p = null$
          Create $N^{j'} = N^i$; Create $CLS^{j'} = CLS^i$
      Create $sg^{j'} =< N^{j'}, CLS^{j'} >$
      $RSSG^{k'} = RSSG^{k'} \cup sg^{j'}$
  endfor
  $RSSG^k$ =Summarize_RSSG($RSSG^{k'}$)        # Summarize compatible graphs
  return($RSSG^k$)
end
```

Figure 13: XYsel() function.

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

  $RSSG^k = \emptyset$
  forall $sg^i \in RSSG^1$
      If ($\exists sg^j \in RSSG^k$ $s.t.$ Compatible_SG($sg^i, sg^j$) $= TRUE$),
          $RSSG^k = RSSG^k - sg^i \cup$ Join_SG($sg^i, sg^j$)
      else
          $RSSG^k = RSSG^k \cup sg^i$
  endfor
  return($RSSG^k$)
end
```

Figure 14: Summarize_RSSG() function.

```
Join_SG()
  Input: $sg^1 =< N^1, CLS^1 >, sg^2 =< N^2, CLS^2 >$          # two shape graphs
  Output: $sg^k =< N^k, CLS^k >$                          # a normalized shape graph

  $N^k = \emptyset; CLS^k = \emptyset$
  # Compute $N^1 \sqcup N^2$
  forall $n_i \in N^1$,
      If ($\exists n_j \in N^2$ $s.t.$ Compatible_Node($n_i, n_j, CLS_{ni}, CLS_{nj}) = TRUE$),
          # Create a summary node $n_s$
          $\forall \mathbf{prop} \in \mathbf{PROP} \Longrightarrow \mathcal{PPM}_{\mathbf{prop}}(\mathbf{n_s}) =$ Join_Property($\mathbf{n_i}, \mathbf{n_j}, \mathbf{prop}$)
          $N^k = N^k \cup n_s$
          $MAP(n_i) = MAP(n_j) = n_s$
      else
          $N^k = N^k \cup n_i$
          $MAP(n_i) = n_i$
  endfor
  forall $n_j \in N^2$,
      If ($\nexists n_i \in N^1$ $s.t.$ Compatible_Node($n_j, n_i, CLS_{nj}, CLS_{ni}) = TRUE$),
          $N^k = N^k \cup n_j$
          $MAP(n_j) = n_j$
  endfor
  # Compute $CLS^1 \sqcup CLS^2$
  $\forall n_r \in N^k \rightarrow$ Create $CLS'_{nr} = \emptyset$
  forall $n_i \in N^1 \vee N^2$,
      $n_r = MAP(n_i)$
      forall $cls_{ni} = \{PL_{ni}, SL_{ni}\} \in CLS_{ni}$,
          Create $PL'_{nr} = SL'_{nr} = \emptyset$
          $\forall pl =< x, n_i >\in PL_{ni} \Longrightarrow$ Create $pl' =< x, n_r >; PL'_{nr} = PL'_{nr} \cup pl'$
          $\forall sl_{att} =<< n_a, sel, n_b >, attsl >\in SL_{ni} \Longrightarrow$
              Create $sl'_{att} =< MAP(n_a), sel, MAP(n_b) >, attsl >; SL'_{nr} = SL'_{nr} \cup sl'_{att}$
          Create $cls'_{nr} = \{PL'_{nr}, SL'_{nr}\}$
          $CLS'_{nr} = CLS'_{nr} \cup cls'_{nr}$
      endfor
  endfor
  $CLS^k = \bigcup_{\forall n \in N^k} CLS'_n$
  return($sg^k =< N^k, CLS^k >$)
end
```

Figure 15: The Join_SG() function.

```
Summarize_SG()
  Input: $List^1[N], List^1[CLS]$          # A list of nodes and a list of CLS's
  Output: $sg^k =< N^k, CLS^k >$        # a normalized shape graph

  $N^k = \emptyset; CLS^k = \emptyset$
  forall $n_i \in List^1[N]$,            # being $CLS_{ni} \wedge CLS_{nj} \in List^1[CLS]$
    If ($\exists n_j \in N^k\ \ s.t.$ Compatible_Node$(n_i, n_j, CLS_{ni}, CLS_{nj}) = TRUE$),
      $MAP(n_i) = n_j$
    else
      $N^k = N^k \cup n_i$
      $MAP(n_i) = n_i$
  endfor
  $\forall n_r \in N^k \rightarrow$ Create $CLS'_{nr} = \emptyset$
  forall $n_i \in List^1[N]$,
    $n_r = MAP(n_i)$
    forall $cls_{ni} = \{PL_{ni}, SL_{ni}\} \in List^1[CLS]$,
      Create $PL'_{nr} = SL'_{nr} = \emptyset$
      $\forall pl =< x, n_i >\in PL_{ni} \Longrightarrow$ Create $pl' =< x, n_r >; PL'_{nr} = PL'_{nr} \cup pl'$
      $\forall sl_{att1} =<< n_a, sel, n_b >, attsl1 >\in SL_{ni} \Longrightarrow$          # Compute $attsl1 \uplus attsl2$
        If ($\exists sl_{att2} =<< n_c, sel, n_d >, attsl2 >\in SL_{ni}$
          $being\ \ MAP(n_a) = MAP(n_c) \wedge MAP(n_b) = MAP(n_d)$),
            If $(i \in attsl1 \wedge i \in attsl2) \rightarrow attsl' = attsl1 \cup attsl2 - i + s$
            If $(i \in (attsl1 \vee attsl2) \wedge s \in (attsl1 \vee atts2)) \rightarrow attsl' = attsl1 \cup attsl2 - i$
            else $\rightarrow attsl' = attsl1 \cup attsl2$
            Create $sl'_{att} =< MAP(n_a), sel, MAP(n_b) >, attsl' >; SL'_{nr} = SL'_{nr} \cup sl'_{att}$
        else
            Create $sl'_{att} =< MAP(n_a), sel, MAP(n_b) >, attsl1 >; SL'_{nr} = SL'_{nr} \cup sl'_{att}$
      Create $cls'_{nr} = \{PL'_{nr}, SL'_{nr}\}$
      $CLS'_{nr} = CLS'_{n_r} \cup cls'_{nr}$
    endfor
  endfor
  $CLS^k = \bigcup_{\forall n \in N^k} CLS'_n$
  return($sg^k =< N^k, CLS^k >$)
end
```

Figure 16: Summarize_SG() function.

```
Split_SG()
  Input: $sg^1 =< N^1, CLS^1 >, p \in PTR$          # a shape graph, and a pointer variable
  Output: $RSSG^k$                                    # a set of shape graphs

  $RSSG^k = \emptyset$
  Find $n_i \in N^1$  $s.t.$  $\exists pl =< p, n_i > \subset CLS_{ni}$
  #  Split a graph for each $cls_{ni} \in CLS_{ni}$
  forall $cls_{ni} \in CLS_{ni}$,
      Create $CLS^{k'} = CLS^1 - CLS_{ni} \cup cls_{ni}$
      Create $N^{k'} = N^1$
      Create $sg^{k'} =< N^{k'}, CLS^{k'} >$
      $RSSG^k = RSSG^k \bigcup \texttt{Normalize\_SG}(sg^{k'})$
  endfor
  If $\forall n_i \in N^1$,  $\nexists pl =< p, n_i > \subset CLS_{ni}$,
      $RSSG^k = sg^1$
  return($RSSG^k$)
end
```

Figure 17: Split_SG() function.

```
Normalize_SG()
  Input: $sg^1 =< N^1, CLS^1 >$          # a shape graph
  Output: $sg^k =< N^k, CLS^k >$        # a normalized shape graph

  Create $N_0^{k'} = N^1$
  Create $CLS_0^{k'} = CLS^1$
  Create $sg_0^{k'} = sg^1$
  repeat                    # Iterate until $N_i^{k'}$ and $CLS_i^{k'}$ do not change anymore
      Find $N_u = \{n_u \in N_i^{k'}\ \ s.t.\ \ Unreachable(n_u, sg_i^{k'}) = TRUE\}$
      Find $N_e = \{n_e \in N_i^{k'}\ \ s.t.\ \ CLS_{ne} = \emptyset\}$
      # Remove unreachable and empty nodes
      $N_{i+1}^{k'} = N_i^{k'} - N_u - N_e$
      # cls's from/to unreachable and empty nodes
      Find $\{cls_{nb}\ \ s.t.\ \ \exists sl_{att} \subset cls_{nb}\ \ being\ \ sl_{att} =<< n_f, sel, n_g >, attsl >,$
          $with\ \ (n_f \in N_u \cup N_e) \vee (n_g \in N_u \cup N_e)\}$
      # cls's with incoherent selector links
      Find $\{cls_{nc}\ \ s.t.\ \ \exists sl_{att1} \subset cls_{nc}\ \ being\ \ sl_{att1} =<< n_c, sel, n_m >, attsl1 > \wedge$
          $\wedge \nexists sl_{att2} \subset cls_{nm}\ \ being\ \ sl_{att2} =<< n_c, sel, n_m >, attsl2 >\}$
      Find $\{cls_{nd}\ \ s.t.\ \ \exists sl_{att3} \subset cls_{nd}\ \ being\ \ sl_{att3} =<< n_m, sel, n_d >, attsl3 > \wedge$
          $\wedge \nexists sl_{att4} \subset cls_{nm}\ \ being\ \ sl_{att4} =<< n_m, sel, n_d >, attsl4 >\}$
      $CLS_{i+1}^{k'} = CLS_i^{k'} - \bigcup_{\forall n_u \in N_u} CLS_{nu} - \bigcup_{\forall n_e \in N_e} CLS_{ne} -$
          $-\{cls_{nb}\} - \{cls_{nc}\} - \{cls_{nd}\}$
      $sg_{i+1}^{k'} =< N_{i+1}^{k'}, CLS_{i+1}^{k'} >$
  until $\left(N_{i+1}^{k'} = N_i^{k'} \wedge CLS_{i+1}^{k'} = CLS_i^{k'}\right)$    # Fixed point condition
  $N^k = N_{i+1}^{k'}, CLS^k = CSL_{i+1}^{k'}, sg^k = sg_{i+1}^{k'}$
  return($sg^k$)
end
```

Figure 18: `Normalize_SG()` function.

```
Materialize_Node()
 Input: $sg^1 =< N^1, CLS^1 >, p \in PTR, sel \in SEL$        # a shape graph, a pointer variable and a selector field
 Output: $sg^k =< N^k, CLS^k >$                              # a shape graph

 Find $n_i \in N^1$ $s.t.$ $\exists pl =< p, n_i >\subset CLS_{ni}$
 Find $n_j \in N^1$ $s.t.$ $\exists sl_{att1} =<< n_i, sel, n_j >, attsl1 >\subset CLS_{ni}$
 # Create a new node $n_m$
 $\forall \mathbf{prop} \in \mathbf{PROP} \Longrightarrow \mathcal{PPM}_{\mathbf{prop}}(\mathbf{n_m}) = \mathcal{PPM}_{\mathbf{prop}}(\mathbf{n_j})$
 Create $N^{k'} = N^1 \cup n_m$
 $\forall n \in N^{k'} \Longrightarrow Create\ \ CLS'_n = \emptyset$
 Find $\{cls_{nj} \subset CLS_{nj}\ \ s.t.\ \exists sl_{att2} \subset cls_{nj}\ \ being\ \ sl_{att2} =<< n_i, sel, n_j >, attsl2 >\} ::= \{cls_{nj}\ \ s.t.\ \ cond.\ A\}$
 forall $cls_{nj} = \{PL_{nj}, SL_{nj}\}$ $s.t.\ \ cond.\ A,$        # Create $CLS'_{nm}$
   Create $PL'_{nm} = SL'_{nm} = \emptyset$
   $\forall pl =< x, n_j >\in PL_{nj} \Longrightarrow Create\ \ pl' =< x, n_m >; PL'_{nm} = PL'_{nm} \cup pl'$
   $\forall sl_{att} =<< n_a, field, n_b >, attsl >\in SL_{nj} \Longrightarrow$
     If $(attsl = \{c\}) \rightarrow$ Create $sl'_{att} =< n_m, field, n_m >, attsl >; SL'_{nm} = SL'_{nm} \cup sl'_{att}$
     If $(attsl = \{o\}) \rightarrow$ Create $sl'_{att} =< n_m, field, n_b >, attsl >; SL'_{nm} = SL'_{nm} \cup sl'_{att}$
     If $(attsl = \{i\} \vee \{s\}) \rightarrow$ Create $sl'_{att} =< n_a, field, n_m >, attsl >; SL'_{nm} = SL'_{nm} \cup sl'_{att}$
     else $\rightarrow$ # cases $\{i, o\}, \{s, o\}, \{i, c\}, \{s, c\}$
       Create $sl'_{att1} =< n_a, field, n_m >, attsl - (o/c) >;$
       Create $sl'_{att2} =< n_m, field, n_b >, attsl - (i/s) >;$
       $SL'_{nm} = SL'_{nm} \cup sl'_{att1} \cup sl'_{att2}$
   Create $cls'_{nm} = \{PL'_{nm}, SL'_{nm}\}$
   $CLS'_{nm} = CLS'_{nm} \cup cls'_{nm}$
 endfor
 Create $CLS'_{nj} = CLS_{nj} - \{cls_{nj}\ \ s.t.\ \ cond.\ A\}$        # Create $CLS'_{nj}$
 forall $cls_{nj} = \{PL_{nj}, SL_{nj}\} \in CLS_{nj}$ $s.t.\ \ \neg cond.\ A,$
   If $(\exists sl_{att6} \subset cls_{nj}\ \ being\ \ sl_{att6} =<< n_j, field, n_j >, attsl6 >::= cls_{nj}\ \ \ s.t.\ \ cond.\ E)$
   Create $T1 = T2 = T2' = T3 = \emptyset$
   $\forall sl_{att} \subset cls_{nj}\ \ s.t.\ \ \neg cond.\ E \Longrightarrow T1 = T1 \cup sl_{att}$
   $\forall sl_{att6} \subset cls_{nj}\ \ s.t.\ \ cond.\ E \Longrightarrow$
     If $(attsl6 \neq \{c\}),$
       If $(c \in attsl6),$
         $T2 = T2 \cup << n_j, field, n_j >, attsl6 - c$
         $T3 = T3 \cup << n_j, field, n_j >, attsl6 - (i/s)$
       else
         If $(\{i/s, o\} \subset attsl6),$
           $T2 = T2 \cup << n_j, field, n_j >, attsl6 - (i/s) > \cup << n_j, field, n_j >, attsl6 - o >$
         else  $T2 = T2 \cup << n_j, field, n_j >, attsl6 >$
     $\forall sl_{att} \in T2\ \ being\ \ sl_{att} =<< n_j, field, n_j >, attsl >\Longrightarrow$
       If $((i/s) \in attsl) \rightarrow$ Create $sl'_{att} =<< n_m, field, n_j >, attsl >$
       else $\rightarrow$ Create $sl'_{att} =<< n_j, field, n_m >, attsl >$
       $T2' = T2' \cup sl'_{att}$
     Create $PL'_{nj} = PL_{nj}; SL'_{nj} = T1 \cup T3$
     for $P = (00...0) : (11..1),$        # P is a binary vector of cardinal(T2) size
       $SL'_{nj} = SL'_{nj} \cup \{P \cdot T2 + \neg P \cdot T2'\}$
       Create $cls'_{nj} = \{PL'_{nj}, SL'_{nj}\}$
       $CLS'_{nj} = CLS'_{nj} \cup cls'_{nj}$
     endfor
 endfor
 ⋮
```

Figure 19: `Materialize_Node()` function (1).

`Materialize_Node()` cont.

```
  ⋮
  forall $n_k \in N^1 \;\; s.t. \;\; n_k \neq n_j$,        # Create $CLS'_{nk}$ being $n_k \neq n_j$
    forall $cls_{nk} = \{PL_{nk}, SL_{nk}\} \in CLS_{nk}$,
      If ($\exists sl_{att3} \subset cls_{nk} \;\; being \;\; sl_{att3} =<< n_k, field, n_j >, attsl3 >::= cls_{nk} \;\;\; s.t. \;\; cond.\ B$),
        Create $sl'_{att3} =<< n_k, field, n_m >, attsl3 >$;
      If ($\exists sl_{att4} \subset cls_{nk} \;\; being \;\; sl_{att4} =<< n_j, field, n_k >, attsl4 > \wedge\, s \notin attsl4 ::= cls_{nk} \;\;\; s.t. \;\; cond.\ C$),
        Create $sl'_{att4} =<< n_m, field, n_k >, attsl4 >$;
      If ($\exists sl_{att5} \subset cls_{nk} \;\; being \;\; sl_{att5} =<< n_j, field, n_k >, attsl5 > \wedge\, s \in attsl5 ::= cls_{nk} \;\;\; s.t. \;\; cond.\ D$),
        Create $sl'_{att5} =<< n_m, field, n_k >, attsl5 - s + i >$;
      Create $T1 = T2 = T2' = T3 = \emptyset$
      $\forall sl_{att} \subset cls_{nk} \;\; s.t. \;\; (\neg cond.\ B \wedge \;\; \neg cond.\ C \wedge \neg cond.\ D) \Longrightarrow T1 = T1 \cup sl_{att}$
      $\forall (sl_{att3} \vee sl_{att4}) \subset cls_{nk} \;\; s.t. \;\; (cond.\ B \vee \;\; cond.\ C) \Longrightarrow T2 = T2 \cup sl_{att3} \cup sl_{att4}; T2' = T2' \cup sl'_{att3} \cup sl'_{att4}$
      $\forall sl_{att5} \subset cls_{nk} \;\; s.t. \;\; cond.\ D \Longrightarrow T3 = T3 \cup sl_{att5} \cup sl'_{att5}$
      Create $PL'_{nk} = PL_{nk}; SL'_{nk} = T1 \cup T3$
      for $P = (00...0) : (11..1)$,        # P is a binary vector of cardinal(T2) size
        $SL'_{nk} = SL'_{nk} \cup \{P \cdot T2 + \neg P \cdot T2'\}$
        Create $cls'_{nk} = \{PL'_{nk}, SL'_{nk}\}$
        $CLS'_{nk} = CLS'_{nk} \cup cls'_{nk}$
      endfor
    endfor
  endfor
  $CLS^{k'} = \bigcup_{\forall n \in N^{k'}} CLS'_n$
  $sg^{k'} =< N^{k'}, CLS^{k'} >$
  $sg^k = Normalize\_SG(sg^{k'})$
  return($sg^k$)
end
```

Figure 20: `Materialize_Node()` function (and 2).

## 4.3 Proof of Correctness

The next theorem provides correctness and termination guarantee for the worklist algorithm proposed in Fig. 5.

**Lemma 4.1** *Given two $RSSG$'s: $RSSG^1$ and $RSSG^2$, such that $RSSG^1 \subseteq RSSG^2$. The transfer functions are monotonic if $\forall s, AS_s(RSSG^1) \subseteq AS_s(RSSG^2)$.*

**Proof:**

■

**Theorem 4.1** *(Worklist Correctness). If transfer functions ensure that any pair of compatible nodes are summarized as well as that any pair of compatible graphs are summarized too, then the worklist algorithm from Fig. 5 yields the least fixed point of the system of dataflow equations from Fig. 4.*

**Proof:**

■

**Corollary 4.1** *The worklist algorithm from Fig. 5 is guaranteed to terminate.*

**Proof:**

■

## 4.4 Analysis refinement: Properties

During the analysis, portions of the heap are summarized into single nodes to avoid unbounded recursive data structures. More specifically, the summarization of nodes takes place during the `Summarize_SG` or the `Join_SG` operations (see Figs. **??** and **??**), being the summarization criterium to join compatible nodes. Obviously, the node summarization operation may suppose some loss of accuracy. By default, our analysis finds two compatible nodes when the set of pointer links associated with them (i.e., the pointer variables pointing to a node) is the same in both nodes (see Fig. 2(a)). Let us recall that in our initial abstract heap representation, the abstract domain for the nodes is defined as $N = \mathcal{P}(PTR) \cup \{null\}$, making the nodes be distinguishabled through the set of pointer variables which point to them. One way to refine the node summarization process in order to avoid aggressive summarizations, consists in extending the abstract domain for the nodes, incorporating more information. For it, we define a set of properties $PROP = \{type, site, touch\}$, where each element $prop \in PROP$ will identify one property that can individually be incorporated to our analysis through specific compilation flags. Here, we describe the general framework to incorporate these (or even new) properties. For each property, we start defining new instrumentation domains:

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

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

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

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

- Let $ID$ be the set of identifiers declared during the preprocessing pass of the analysis. These identifiers are usually defined in pragma statements or some pseudostatements (see the `Touch()` and `Untouch()` functions in Figs. 25 and 26). $P_{touch}$ is the domain for the property $prop = touch$ and is defined as a set that contains a set of identifiers:

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

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

Property Map : $\mathcal{PPM}_{prop}$: $N \longrightarrow P_{prop}$
where, $\forall prop \in PROP$, $P_{prop}$ represents the domain for the corresponding property.

The introduction of the node properties, will affect some of the main operations of our analysis, Specially those that deal with nodes. The changes are depicted in bold face in the corresponding functions of Section 4.2. One of the functions affected, the `Compatible_Node()` function is rewritten in Fig. 21. where we check that two nodes are compatible (and can be summarized) when the set of pointer links is the same in both and when the propeties are equivalent. Precisely, this is done by the auxiliary function `Compatible_Property()` which checks if property $prop \in PROP$ is equivalent in the two nodes $n_1$ and $n_2$.

```
Compatible_Node()
  Input: $n_1, n_2, CLS_{n1}, CLS_{n2}$     # two nodes and their CLS's
  Output: $TRUE/FALSE$

  If ($\forall pl_1 =< x, n_1 >\subset CLS_{n1}, \exists pl_2 =< x, n_2 >\subset CLS_{n2} \wedge$
      $\forall pl_2 =< y, n_2 >\subset CLS_{n2}, \exists pl_1 =< y, n_1 >\subset CLS_{n1}$ ),
         **If ($\forall \mathbf{prop} \in \mathbf{PROP}$, Compatible_Property($\mathbf{n_1, n_2, prop}$) $==$ TRUE),**
              return($TRUE$)
  return($FALSE$)
end
```

Figure 21: Check when two nodes are compatible, incorporating the properties check.

Other auxiliary functions, to deal with properties are `Update_Property()` (that initializes the value of a property in a malloc statement) and `Join_Property()` (that returns the value of a property in two compatible nodes). Both functions are shown in Figs. 22 and 23, respectively.

```
Update_Property()
  Input: $s \in STMT, prop \in PROP$        # a statement $s ::= x = new()$, and a property
  Output: $p_{prop} \in P_{prop}$              # The value of the corresponding property

  Case ($prop$)
      $prop == type$
          $p_{prop} = \mathcal{T}(x)$
          break
      $prop == site$
          $p_{prop} = s$
          break
      $prop == touch$
          $p_{prop} = \emptyset$
          break
  return($p_{prop}$)
end
```

Figure 22: `Update_Property()` function.

## 4.5 Complexity

In this section, we will focus firstly on the computation of the main parameters which will help us to find the complexity of our method. Let us keep in mind that we are going to compute the worst case behavior. One of the parameters of interest, is the maximum number of shape graphs generated by our approach. After a given program statement $s\bullet$, such number of graphs are included in a $RSSG^{s\bullet}$, and it depends on the number of ways of partitioning the live pointer variables at that point. For instance, if the set of live pointer

```
Join_Property()
  Input: $n_1, n_2, prop \in PROP$        # two nodes and a property
  Output: $p_{prop} \in P_{prop}$          # The value of the corresponding property

  $p_{prop} = \mathcal{PPM}_{prop}(n_1) = \mathcal{PPM}_{prop}(n_2)$
  return($p_{prop}$)
end
```

Figure 23: `Join_Property()` function.

variables is {`p1, p2, p3`}, i.e. three live pointer variables, we could find the following shape graphs:

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

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 [**?**],

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

As we are interested in computing the maximum number of shape graphs generated by our approach, we should consider all the possibilities due to different control flow paths, because different paths can establish different alias relationships between pointer variables and let us recall that each shape graph in a $RSSG$ represents a different alias configuration. For instance, a path could generate graphs with just one live pointer variable, another path could generate graphs with two live pointer variables, etc. Assuming that $nv$ represents the maximum number of live pointer variables at any program point, the maximum number of graphs generated at a point should be the sum of all the ways of partitioning $j$ live pointer variables, from $j = 1$ till $j = nv$, i.e., $\sum_{j=1}^{nv} B(j)$. In addition, we should consider the number of properties evaluated in the shape analysis, $np$, as well as the range of the values for each property $p_j$, range that we define as

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

Let us not forget that we are computing the maximum number of shape graphs for a $RSSG$ at a program point $s\bullet$, i.e. for each statement. With all of this, the **maximum number of graphs per statement**, which we name $Ng_s$, could be estimated as we indicate in Eq. 1. An obvious way to compute the **maximum number of graphs** generated for the analyzed code, which we will name $Ng$, would be obtained multiplying $Ng_s$ by the number of statements analyzed in the program, $nstmt$, as we see in Eq. 2.

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

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

There are other interesting parameters that give us more detailed information about how complex the shape graphs are and that are measurable: for instance how many nodes does a graph have and how interconnected these nodes are. About the number of nodes, we are interested in computing an upper bound, i.e. the maximum size of a shape graph. In other words, the **maximum number of nodes per graph**, which we will name $Nn$. It depends on the maximum number of live pointer variables, $nv$, because, in a worst case, when none of the pointers are aliased, then each one could point to a different node. $Nn$ depends too on the number of properties considered, $np$ and the range of the values for each property $p_j$, i.e. $0 : rp_j$, because each value for each property can contribute as a new node. With all of this, $Nn$ can be estimated as we show in Eq. 3.

$$Nn = nv + 2^{\sum_{j=1}^{np} rp_j} \quad (3)$$

About how interconnected the nodes are, we should compute the maximum number of sl's -selector links- and the maximum number of cls's -coexistent links sets-, which are precisely the parameters that encode this information in our approach. We will name the **maximum number of sl's per node**, as $Nsl_{node}$ and the **maximum number of sl's per graph**, as $Nsl$. The former depends on the maximum number of selector or pointer fields declared in the most complex data structure, $nl$. It depends too on the maximum number of nodes, to which any node can be connected through a selector link, i.e. $Nn - 1$. As the links that can coexist in a given node can be incoming from any other node, outgoing to any other node, 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 sl's per graph should be the sum of all the selector links per node when we iteratively incorporate $Nsl_{node}(j)$ for each new node, from $j = 1$ till $Nn$, as we see in Eq. 6.

$$\begin{aligned} Nsl_{node} &= Nsl_{node}(Nn) = nl \cdot (2 \cdot Nn - 1) & (4)\\ Nsl &= \sum_{j=1}^{Nn} Nsl_{node}(j) = \sum_{j=1}^{Nn} nl \cdot (2 \cdot j - 1) = & (5)\\ &= nl \cdot (2 \cdot Nn - 1) \cdot (Nn - 1) & (6) \end{aligned}$$

However, the most important parameter is the maximum number of cls's. A cls contains pointer links and selector links with attributes. As a shape graph represents a concrete alias configuration, the number of pointer links is fixed. The variations come from the selector links with attributes. For instance, for a node, the maximum number of selector links with attributes depends on the combination of the maximum number of selector links that can coexist in the node (excluding the links from/to itself, i.e. $2^{Nsl_{node}-nl}$, see Eq. 4), as well as the number of variations due to the attributes: it is, $5^{nl}$. Let's see this last factor is detail: in a cls there could be five different states representing the attributes for each selector link from/to the same node: i) the selector link does not appear, ii) it is just incoming ($attsl = \{i\}$ or $attsl = \{s\}$), iii) it is just outgoing ($attsl = \{o\}$), iv) it is just cyclic ($attsl = \{c\}$) and v) it is a summary node with the same incoming and outgoing link ($attsl = \{i, o\}$, $attsl = \{i, c\}$, or $attsl = \{s, o\}$, $attsl = \{s, c\}$ for a shared summary node). With all of this, we could compute the **maximum number of cls's for a node**, named $Ncls_{node}$, by Eq. 7. Clearly, the **maximum number of cls's per graph** named $Ncls$, can be computed from Eq. 7 and $Nn$ (the maximum number of nodes) as we see in Eq. 8.

$$\begin{aligned} Ncls_{node} &= \left(2^{Nsl_{node}-nl}\right) \cdot 5^{nl} = \left(2^{2 \cdot nl \cdot (Nn-1)}\right) \cdot 5^{nl} & (7)\\ Ncls &= Ncls_{node} \cdot Nn = \left[\left(2^{2 \cdot nl \cdot (Nn-1)}\right) \cdot 5^{nl}\right] \cdot Nn & (8) \end{aligned}$$

Eq. 7 is a first approximation that gives us a worst case upper bound for the estimation of the maximum number of cls's for a node when there is not available information about the data structures. However, such a number can be greatly reduced when we have some information about the data structures. Till now, we have assumed that all the selector links can be incoming to and outgoing from a node. But, in a cls that represents a real data structure, there is as most, a maximum number of "real" incoming selector links. We will call $nli$ to this important piece of information. For instance, in a singly-linked list $nli = 1$, in a doubly-linked list $nli = 2$, or in a binary tree $nli = 1$. With this information we have to compute all the cls's that are combinations due to the selector links with attributes that are incoming in a node, multiplied by combinations due to the selector links with attributes that can be outgoing from the node. In a node, we know that there could be at most: a) $nl \cdot (Nn - 1)$ selector links from other (different) nodes (cases in which attribute is $\{i\}$ or $\{o\}$), plus b) $nl$ selector links from the same node with attribute $c$, plus c) $nl$ selector links from the same node that represent incoming and outgoing in a summary node (cases in which attributes are $\{i, o\}$ or $\{s, o\}$ or $\{i, c\}$ or $\{s, c\}$). Thus, there could be $nl \cdot (Nn + 1)$ selector links with attributes in a node. From them, at most, only $nli$ would appear as incoming selector links in a cls, therefore, for the computation of the combination of the selector links with attributes 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 with attributes that there could be in a node, we know that in a cls could be from 0 till $nl$ outgoing links. Thus, for the computation of the combination of the selector links with attributes that are outgoing from a node we can do,

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

In other words, a more accurate estimation for the computation of the maximum number of cls's, $Ncls_{node}$, is given by Eq. 9. Again, the maximum number of cls's per graph, named $Ncls$, can be computed from Eq. 9 and the maximum number of nodes, $Nn$, as we see in Eq. 10.

$$Ncls_{node} = \sum_{j=1}^{nli} \binom{nl \cdot (Nn+1)}{j} \cdot \sum_{k=0}^{nl} \binom{nl \cdot (Nn+1)}{k} \tag{9}$$

$$Ncls = Ncls_{node} \cdot Nn \tag{10}$$

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 cls's 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 $O(Nn^4)$.

Table 1: 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 |
| $Ng_s$ | maximum number of graphs per statement $s$ | Eq. 1 |
| $Ng$ | maximum number of graphs | Eq. 2 |
| $Nn$ | maximum number of nodes per graph | Eq. 3 |
| $Nsl_{node}$ | maximum number of sl's per node | Eq. 4 |
| $Nsl$ | maximum number of sl's per graph | Eq. 6 |
| $Ncls_{node}$ | maximum number of cls's per node | Eq. 9 |
| $Ncls$ | maximum number of cls's per graph | Eq. 10 |
| $Npl_{node}$ | maximum number of pl's per node | Eq. 11 |
| $Npl$ | maximum number of pl's per graph | Eq. 12 |

Other parameter of our abstraction, that could be interesting to compute is the **maximum number of pl's per node**, and we will name it as $Npl_{node}$. It depends on the number of live pointer variables, $nv$, and it can be easily computed as we can see in Eq. 11. The **maximum number of pl's per graph**, named $Npl$, is represented in Eq. 12. As we assume that any $RSSG$ will be in normal form, then each pointer variable can appear only once on each graph, therefore $Npl = Npl_{node}$.

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

$$Npl = Npl_{node} = nv \tag{12}$$

Table 1 summarizes the main parameters used in our complexity study, as well as their definitions and their values.

Now, our goal is to estimate the worst theoretical performance of our shape analysis framework. Roughly, the cost of analyzing a pointer statement will depend on the cost of the corresponding transfer function, and more concretely it will depend on the operations that the transfer function invokes. We would like to start summarizing the dominant costs for the main operations that our transfer functions call. These costs can safely be deduced from the algorithms presented in Section 4.2. For the estimation of these dominant costs, we assume a worst case scenario: each shape graph contains the maximum number of nodes ($Nn$), the maximum number of sl's ($Nsl$) and the maximum number of cls's ($Ncls$). Let's see then the costs for the main operations:

- The `Summarize_SG()` operation (see Fig. 16) has a computational cost given by $O(Nn + Nn \cdot Ncls_{node})$, due to the fist and second forall, respectively . We can easily deduce, that the dominant cost for this operation can be estimated as $O(Nn \cdot Ncls_{node} = O(Ncls)$.

- The `Normalize_SG()` operation (see Fig. 18) depends basically on two findings: i) find unreachable nodes, which has a cost of $O(Nn \cdot log(Nn))$ and ii) find cls's with incoherent selector links, which has a cost of $O(Ncls \cdot log(Ncls))$. In other words, the computational cost is dominated by $O(Nn \cdot log(Nn) + Ncls \cdot log(Ncls))$. As we know from Eqs. 3 and 10, $Ncls >> Nn$, therefore, the cost of this operation is dominated by $O(Ncls \cdot log(Ncls))$.

- The `Split_SG()` operation (see Fig. 17) depends on finding a node and then creating a new graph for each cls of that node. When creating the new graphs, the `Normalize_SG()` function is called. Clearly, it presents a cost given by $O(Nn + Ncls_{node} \cdot (Ncls \cdot log(Ncls)))$. Simplifying, The dominant cost of this operation can be expressed as $O(Ncls_{node} \cdot (Ncls \cdot log(Ncls)))$

- The `Materialize_Node()` operation (see Figs. 19 and 20) has a cost of $O(2 \cdot Nn + 2 \cdot Ncls_{node})$ for the two first nodes finding and the creation of the cls's of the new materialized node (the `Create` $CLS'_{nm}$ forall). Next, the `Create` $CLS'_{nj}$ forall has a cost given by $O(Ncls_{node} \cdot Nsl_{node})$, whereas the `Create` $CLS'_{nk}$ forall presents a cost given by $O(Nn \cdot Ncls_{node} \cdot Nsl_{node})$. Finally, a call to the `Normalize_SG()` function will have a cost of $O(Ncls \cdot log(Ncls))$. In summary, the cost of the materialization is given by $O(2 \cdot Nn + 2 \cdot Nncls_{node} + Ncls_{node} \cdot Nsl_{node} + Nn \cdot Ncls_{node} \cdot Nsl_{node} + Ncls \cdot log(Ncls))$. As $Nn \cdot Ncls_{node} = Ncls$, and from Eqs. 4 and 10 we deduce that $Nsl_{node} < log(Ncls)$, we can approximate the dominant cost for this operation as $O(Ncls \cdot log(Ncls))$.

Now that we know the dominant costs of the main operations, we could estimate the costs for the transfer functions. However, we should remark here that the functions presented in Section 4.2 which describe in a simplistic way the transfer functions, are in fact different from our real implementations. In other words, the dominant cost of each transfer function depends on the algorithm implemented. We present here a short indication of these costs. For the estimation of these dominant costs, we have assumed again a worst case scenario: the maximum number of shape graphs included in a $RSSG^{\bullet s}$ is $Ng_s$ (see Eq. 1). In the computation of the dominant costs of our real implementations of the transfer functions we have included the operator $\bigsqcup^{RSSG}$ which roughly has a cost given by $O(Ng_s)$. For instance, the statements `x=null`, `x=new` and `x=y` call to the `Summarize_SG()` operation. In our implementation, the cost for these statements is given by $O(Ng_s \cdot Ncls)$. However, the statements `x->sel=null`, `x->sel=y` and `x=y->sel` call to the `Split_SG()`, `Materialize_Node()` and `Normalize_SG()` operations and, roughly, they present a cost given by $O(Ng_s \cdot Ncls \cdot log(Ncls))$. Clearly, the complexity is dominated by the transfer function of these last statements, so our method has a complexity of $O(Ng_s \cdot Ncls \cdot log(Ncls))$.

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

Summarizing, we find that the complexity of our approach depends on the upper bounds of $Ng_s$ and $Ncls$. From Eq. 10 we know that $Ncls$ has a polynomial behaviour: $O(Nn^3)$ for a singly linked list, $O(Nn^5)$ for a doubly linked list ... Ignoring the properties, from Eq. 3 we know that $Nn = nv + 1$. Therefore, roughly we can approximate an upper bound for the $Ncls$ parameter as $O((nv)^k)$, where $k$ is a constant that depends on the maximum number of links in the structures analyzed, and $nv$ is the maximum number of live pointer variables. On the other hand, from Eq. 1 which represent the theoretical maximum value for $Ng_s$, again ignoring the properties, we can notice that depends on the sum of the numbers of Bell, $\sum_{j=1}^{nv} B(j) < nv \cdot B(nv)$. From [], we know that the asymptotic limit of numbers of Bell is,

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

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

## 4.6 Pseudostatements

We can instrument the analysis providing some useful information from the code. This information is annotated in the source code, by a preprocessing step, in the form of pseudostatements, and later they are abstractly interpreted as normal statements. Currently we support three type of pseudostatements: `force()`, `touch()` and `untouch()`.

The transfer function of the `force()` pseudostatement is described as a function `Force()` in Fig. 24. This kind of pseudostatement extracts semantic information from test conditions in `if` and `while` program flow statements, when these test conditions involve pointers variables. On the branch where the tested expression is null, e.g. `x==null` or `x->sel==null`, the force's transfer function filters out the graphs in which a pointer link of the form $PL =< x, n_i >$ exists, i.e. the variable `x` points to a node, for the first case, or removes the graphs for which the path `x->sel` points to a node, for the second case. On the contrary, on the branch where the tested expression is not null, e.g. `x!=null` or `x->sel!=null`, then the transfer function filters out the graphs in which a pointer link of the form $PL =< x, n_i >$ does not exist, i.e. the variable `x` does not point to a node, or removes the graphs for which the path `x->sel` does not point to a node, respectively. In this way, we allow the analysis to filter out unrealistic memory configurations.

The transfer function of the `touch()` pseudostatement is described as a function `Touch()` in Fig. 25, whereas the transfer function of the `untouch()` psedostatement is described as a function `Untouch()` in Fig. 26. The `touch()` pseudostatement let us annotate the node pointed to by a pointer $x$, with an identifier ($touch_{id} \in ID$ in our function), whereas the `untouch()` pseudostatement removes that identifier from any node of the graph. This kind of annotations is useful when performing some client analysis, for instance a dependence test. In this case, `touch()` pseudostatements are inserted by our client analysis, just after the statements that perform read or write accesses to data or selector fields that potentially may provoke loop carried data dependencies (LCDs). On each pseudostatement $touch_{id}$ codifies the statement id. and the type of access (read/write) performed by the previous statement. When the `touch()` is abstractly interpreted, then the corresponding node is annotated with that information. Later, the data dependence test checks if a node has been actually written and read by statements that could produce LCDs, and in that case a data dependence (and the type of dependence - RAW, WAR or WAW) is reported.

```
Force()
  Input: $sg^1 =< N^1, CLS^1 >$, test_condition        # a shape graph, and a test condition
  Output: $sg^k =< N^k, CLS^k >$                        # a shape graph

  Case (test_condition)
    test_condition  ==  (x==null)
      If ($\exists n_i \in N^1$ $s.t.$ $\exists pl =< x, ni > \subset CLS_{ni}$),
        $sg^k = \emptyset$
      else
        $sg^k = sg^1$
      break
    test_condition  =  (x!=null)
      If ($\exists n_i \in N^1$ $s.t.$ $\exists pl =< x, ni > \subset CLS_{ni}$),
        $sg^k = sg^1$
      else
        $sg^k = \emptyset$
      break
    test_condition  =  (x->sel==null)
      Find $n_i \in N^1$ $s.t.$ $\exists pl =< x, n_i > \subset CLS_{ni}$
      Create $CLS'_{ni} = \emptyset$
      forall $cls_{ni} \in CLS_{ni}$,
        If ($\exists sl_{att} =<< n_i, sel, n_j >, attsl >$ $s.t.$ $n_j = null$,
          $CLS'_{ni} = CLS'_{ni} \cup cls_{ni}$
      endfor
      Create $CLS^{k'} = CLS^1 - CLS_{ni} \cup CLS'_{ni}$; Create $N^{k'} = N^1$
      $sg^{k'} =< N^{k'}, CLS^{k'} >$
      $sg^k$ =Normalize_SG($sg^{k'}$)
      break
    test_condition  =  (x->sel!=null)
      Find $n_i \in N^1$ $s.t.$ $\exists pl =< x, n_i > \subset CLS_{ni}$
      Create $CLS'_{ni} = \emptyset$
      forall $cls_{ni} \in CLS_{ni}$,
        If ($\exists sl_{att} =<< n_i, sel, n_j >, attsl >$ $s.t.$ $n_j \neq null$,
          $CLS'_{ni} = CLS'_{ni} \cup cls_{ni}$
      endfor
      Create $CLS^{k'} = CLS^1 - CLS_{ni} \cup CLS'_{ni}$; Create $N^{k'} = N^1$
      $sg^{k'} =< N^{k'}, CLS^{k'} >$
      $sg^k$ =Normalize_SG($sg^{k'}$)
      break
  return($sg^k$)
end
```

Figure 24: `Force()` function.

```
Touch()
  Input: $sg^1 =< N^1, CLS^1 >, x \in PTR, touch_{id}$          # a shape graph, a pointer and a identifier
  Output: $sg^k =< N^k, CLS^k >$                                 # a shape graph

  Find $n_i \in N^1$  $s.t.$ $\exists pl =< x, n_i >\subset CLS_{ni}$
  $\mathcal{PPM}_{touch}(ni) = \mathcal{PPM}_{touch}(ni) \cup touch_{id}$
  Create $N^k = N^1$; Create $CLS^k = CLS^1$
  Create $sg^k =< N^k, CLS^k >$
  return($sg^k$)
end
```

Figure 25: Touch() function.

```
Untouch()
  Input: $sg^1 =< N^1, CLS^1 >, touch_{id}$          # a shape graph and a identifier
  Output: $sg^k =< N^k, CLS^k >$                      # a shape graph

  Create $List'[N] = \emptyset$; Create $List'[CLS] = \emptyset$
  forall $n_i \in N^1$,
    $\mathcal{PPM}_{tocuh}(n_i) = \mathcal{PPM}_{tocuh}(n_i) - touch_{id}$
    $List'[N] = List'[N] \cup n_i$
    $List'[CLS] = List'[CLS] \cup CLS_{ni}$
  endfor
  $sg^k =$Summarize_SG($List'[N], List'[CLS]$)
  return($sg^k$)
end
```

Figure 26: Untouch() function.

## 5 Interprocedural Analysis

Now, we extend the definition of a program to include the set of functions, $FUN$, declared in that program, and we extend the type of analyzable statements to include the `call()` and `return()` of these functions (see Fig. 27). An important detail is that we distinguish between non-recursive an recursive call sites and recursive and non-recursive return points, respectively. Precisely, the set of call statements defined in non-recursive call sites is called $S_{call\_nrec}$, whereas the set of call statements defined in recursive call sites is called $S_{call\_rec}$. On the other hand, the set of return statements defined at functions return point is called $S_{return}$.

| | |
|---|---|
| programs: | $prog \in P$, $P =< FUN, STMT, PTR, TYPE, SEL >$ |
| functions: | $fun \in FUN$, $FUN =< FUN_{fun}, STMT_{fun}, PTR, TYPE, SEL >$ |
| statements: | $s \in STMT$, $s ::= x = NULL \mid x = malloc() \mid free(x) \mid x = y$ |
| | $\mid x \rightarrow sel = NULL \mid x \rightarrow sel = y \mid x = y \rightarrow sel$ |
| | $\mid x = call() \mid return(y)$ |
| | $FUN_{fun} \subset FUN$, being $foo \in FUN_{fun}$ a callee of $fun$. |
| | $STMT_{fun} \subset STMT$, being $s \in STMT_{fun}$ a stmt. in the body of $fun$. |

Figure 27: Extensions for interprocedural support.

We formulate a context sensitive interprocedural analysis, because we distinguish between different calling context of the same procedure. The analysis at procedure calls must account for the assignment of actual parameters to formal ones and for the change of analysis domain between the caller and the callee. For it, we need to define new instrumentation mapping functions:

| | |
|---|---|
| Local Pointers Map : | $\mathcal{LPM}$: $FUN \longrightarrow PTR$ |
| Actual to Formal Pointers Map: | $\mathcal{AFPM}$: $(S_{call\_nrec} \cup S_{call\_rec}) \times FUN \longrightarrow PTR \times PTR_{fun}$ |
| Returned to Assigned Pointer Map: | $\mathcal{RAPM}$: $(S_{call\_nrec} \cup S_{call\_rec}) \times FUN \longrightarrow (PTR_{fun} \times PTR) \cup \emptyset$ |

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

  $\forall fun \in FUN$, $\mathcal{LPM}(fun) = \{lptr \in PTR$, being $lptr$ a pointer var. declared in the definition or the body of $fun\}$.

Usually, we name $PTR_{fun}$ to that set of formal and local pointer variables associated with function $fun$. On the other hand, we will name $GLB$ to the set of global pointers, $GLB \subset PTR$.

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

  $\forall s \in (S_{call\_nrec} \cup S_{call\_rec})$, being $fun \in FUN$ called by $s$, $\mathcal{AFPM}(s, fun) = \{< aptr, fptr >$, where $aptr \in PTR$ an actual parameter in statement $s$, and $fptr \in PTR_{fun}$ a formal parameter in $fun\}$.

  Sometimes, we just need the set of actual pointer parameters ($aptr$) for a call statement $s$. We will name $APTR_s$ to that set. It can easily be deduced from $\mathcal{AFPM}(s, fun)$.

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

  $\forall s \in (S_{call\_nrec} \cup S_{call\_rec})$, being $fun \in FUN$ called by $s$, $\mathcal{RAPM}(s, fun) =< retptr, assptr >$, where $retptr \in PTR_{fun}$ the pointer returned at the exit point of $fun \wedge assptr \in PTR$ the pointer assigned at statement $s$. In the case that the function does not return a pointer, then this function gives $\emptyset$.

Now, we need to include the new interprocedural dataflow equations that we show in Fig. 28 to augment the intraprocedural Eqs. from Fig. 4. Basically, we present two different equations for the ENTRY/EXIT dataflow transfers from the caller to the callee and from the callee to the caller. We distinguish between non-recursive and recursive calls and returns. In these new equations, we assume that $fun$ is the function called by $s$, $se_{fun}$ the entry point at $fun$ and $sr_{fun}$ the return point of $fun$. Equations [ENTRY$_{nrec}$] and [ENTRY$_{rec}$] perform the transfer from the caller to the callee in the case of a non-recursive or a recursive call, respectively; Equations [EXIT$_{nrec}$] and [EXIT$_{rec}$] transfer the analysis back to the caller.

To simplify the formal definitions of the ENTRY/EXIT transfer functions, we use the functions `CTS`$_{nrec}$(), `CTS`$_{rec}$(), `RTC`$_{nrec}$(), `RTC`$_{rec}$() (see Figs. **??**) to describe the transformations that take place in our abstract heap when the analysis flow from the caller to the callee and from the callee to the caller. But fist, let's see how to augment our abstract heap to incorporate the recursive flow links.

[ENTRY$_{nrec}$]: $RSSG^{\bullet se_{fun}} = IN_{s \in S_{call\_nrec}}(RSSG^{\bullet s})$, where
$IN_{s \in S_{call\_nrec}}(RSSG^{\bullet s}) = \bigsqcup_{sg^i \in RSSG^{\bullet s}}^{RSSG} CTS_{nrec}(sg^i, PTR_{fun}, \mathcal{AFPM}(s, fun))$

[ENTRY$_{rec}$]: $RSSG^{\bullet se_{fun}} = IN_{s \in S_{call\_rec}}(RSSG^{\bullet s})$, where
$IN_{s \in S_{call\_rec}}(RSSG^{\bullet s}) = \bigsqcup_{sg^i \in RSSG^{\bullet s}}^{RSSG} CTS_{rec}(sg^i, PTR_{fun}, \mathcal{AFPM}(s, fun))$

[EXIT$_{nrec}$]: $RSSG^{s \bullet} = OUT_{s \in S_{call\_nrec}}(RSSG^{\bullet sr_{fun}})$, where
$OUT_{s \in S_{call\_nrec}}(RSSG^{\bullet sr_{fun}}) = \bigsqcup_{sg^i \in RSSG^{\bullet sr_{fun}}}^{RSSG} RTC_{nrec}(sg^i, PTR_{fun}, \mathcal{AFPM}(s, fun), \mathcal{RAPM}(s, fun))$

[EXIT$_{rec}$]: $RSSG^{s \bullet} = OUT_{s \in S_{call\_rec}}(RSSG^{\bullet sr_{fun}})$, where
$OUT_{s \in S_{call\_rec}}(RSSG^{\bullet sr_{fun}}) = \bigsqcup_{sg^i \in RSSG^{\bullet sr_{fun}}}^{RSSG} RTC_{rec}(sg^i, PTR_{fun}, \mathcal{AFPM}(s, fun), \mathcal{RAPM}(s, fun))$

Figure 28: Dataflow equations for interprocedural support.

## 5.1 Recursive Flow Links

To provide interprocedural support, especially for the case of recursive functions, we need that our heap abstraction maintains the state of formal pointer parameters and local pointers (from now on, the pointers in $PTR_{fun}$) in a sequence of recursive calls until the fixed point is reached. During program execution, at runtime, the *Activation Record Stack* (ARS) provides explicit information about the state of these variables for every call. We chose to abstract that information in our concrete domain, by augmenting the PLc and SLc sets respectively with new sets that contains the so named *concrete recursive flow links*. These recursive flow links will let us easily to trace the path of formal and local pointers in a sequence of recursive calls. For it, we include two new partial functions, $\mathcal{RFPM}^c$ and $\mathcal{RFSM}^c$ that trace the locations to where each formal and local pointer of a function call, was pointing to in the previous pending calls in a stack of recursive calls. They are defined as follows:

| | |
|---|---|
| Rec. Flow Pointer Map (in the concrete domain): | $\mathcal{RFPM}^c$: $PTR_{fun} \longrightarrow L$ |
| Rec. Flow Selector Map (in the concrete domain): | $\mathcal{RFSM}^c$: $L \times PTR_{fun} \longrightarrow (L \cup null)$ |

- $\mathcal{RFPM}^c$ maps a formal or local pointer variable $x \in PTR_{fun}$ to the location $l$ pointed to by $x$ in the immediately previous pending call (previous context):

  $\forall x \in PTR_{fun}, \exists l \in L \mid \mathcal{PRFM}^c(x) = l \;\; s.t. \;\; \mathcal{PM}^c(x) = l$ in the immediately previous pending call.

  Usually, we use the tuple $rfplc =< x_{rfptr}, l >$, which we name *concrete recursive flow pointer link*, to represent this binary relation. The set of all concrete recursive flow pointer links is named $RFPLc$.

- $\mathcal{RFSM}^c$ models the path (between locations $l_1$ and $l_2$) tracked for a formal or local pointer $x \in PTR_{fun}$ through two consecutive previous pending calls. Let's assume that we name $pc_l$ to a pending call and $pc_{t-1}$ to the consecutive previous to that call:

$\forall l_2 \in L \;\; s.t. \;\; \mathcal{PM}^c(x) = l_2$ in a previous pending call $pc_t$, $\exists l_1 \in (L \cup null) \;\; s.t. \;\; \mathcal{PM}^c(x) = l_1$ in the consecutive previous to that pending call $pc_{t-1}$, $\;|\; \mathcal{RFSM}^c(l_2, x) = l_1$.

We use a tuple $rfslc =< l_2, x_{rfsel}, l_1 >$, which we name *concrete recursive flow selector link*, to represent this relation. The set of all concrete recursive flow selector links is called $RFSLc$.

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

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

| | |
|---|---|
| Rec. Flow Pointer Map (in the abstract domain): | $\mathcal{RFPM}^a$: $PTR_{fun} \longrightarrow N$ |
| Rec. Flow Selector Map (in the abstract domain): | $\mathcal{RFSM}^a$: $N \times PTR_{fun} \longrightarrow N$ |

- $\mathcal{RFPM}^a$ maps a formal or local pointer variable $x \in PTR_{fun}$ to the node $n$ pointed to by $x$ in the inmediately previous pending call (previous context):

  $\forall x \in PTR_{fun}$, $\exists n \in n \;\; | \;\; \mathcal{RFPM}^a(x) = n \;\; s.t. \;\; \mathcal{PM}^a(x) = n$ in the immediately previous pending call.

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

- $\mathcal{RFSM}^a$ models the path (between nodes $n_1$ and $n_2$) tracked for a formal or local pointer $x \in PTR_{fun}$ through two or more consecutive previous pending calls. Let's assume that we name $pc_t$ to a pending call and $pc_{t-1}$ to the consecutive previous to that call:

  $\forall n_2 \in N \;\; s.t. \;\; \mathcal{PM}^a(x) = n_2$ in a previous pending call $pc_t$, $\exists n_1 \in N \;\; s.t. \;\; \mathcal{PM}^a(x) = n_1$ in the consecutive previous to that pending call $pc_{t-1}$, $\;|\; \mathcal{RFSM}^a(n_2, x) = n_1$.

  We use a tuple $rfsl =< n_2, x_{rfsel}, n_1 >$, which we name *recursive flow selector link*, to represent this relation. The set of all recursive flow selector links is called $RFSL$. We should note that in the case that $n_2 = n_1$ in the $rfsl$, , then more than two consecutive pending calls are represented by this relation: in this case, all the pending calls for which $\mathcal{PM}^a(x) = n_1 = n_2$ are represented by just one recursive flow selector link.

It must be clear that $x_{rfptr}$ and $x_{rfsel}$ are symbolic names to represent the state of variable $x$ (where it points to) in previous pending calls.

We extend the sets, $PL \cup RFPL$ and $SL \cup RFSL$ to augment the domain of the selector links with attributes: $SL_{att} = (SL \cup RFSL) \times ATTSL$, and the Coexistent Links Set abstraction: $\mathcal{CLM}$: $N \longrightarrow \mathcal{P}(PL \cup RFPL) \times \mathcal{P}(SL_{att})$. In other words, now a coexistent links set for anode $n$, $cls_n$, is defined as follows:

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

where:

$$\begin{aligned} PL_n &= \{pl \in PL \;\; s.t. \;\; pl =< x, n >\} \cup \{rfpl \in RFPL \;\; s.t. \;\; rfpl =< x_{rfptr}, n >\} \\ SL_n &= \{sl_{att} \in SL_{att} \;\; s.t. \;\; sl_{att} =<< n_1, sel, n_2 >, attsl > \vee \\ &\vee \;\; sl_{att} =<< n_1, x_{rfsel}, n_2 >, attsl >, \; being \; (n_1 = n \vee n_2 = n)\} \end{aligned}$$

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

We present in Fig. 29 the extended worklist algorithm for solving the dataflow equations presented in Fig. 4 and Fig. 28. The input of our worklist algorithm is a program $P$ with functions, or a function $FUN$ with its corresponding functions, and an input $RSSG^{in}$. The initial $RSSG^{in} = \emptyset$. The output of the algorithm is the $RSSG^{out}$ resultant at the exit program or function point. Without loss of generality we assume that there is only one return point on each function. We could mention that the algorithm also computes the resultant $RSSG^{s\bullet}$ at each program point. Our code processes the worklist using the main loop defined in lines 4-30. We can see that the algorithm is sensitive to the type of statement being processed (line 7). If $s \in S_{call\_nrec}$, i.e., it is a non-recursive call (lines 8-12) then the algorithm propagates the resultant RSSG, after the [ENTRY$_{nrec}$] transformation to the entry point of the caller ($se_{fun}$, line 10), and later, a new instance of the worklist algorithm is invoked to process the statements of the body of the called function (line 11). On the other hand, if $s \in S_{call\_rec}$, i.e., it is a recursive call (lines 13-17), then the algorithm propagates again the resultant RSSG, after the [ENTRY$_{rec}$] transformation to the entry point of the caller ($se_{fun}$, line 15), and next a different worklist algorithm, `Worklist_rec`, shown in Fig. 30, is invoked to process the statements of the body of the recursive function (line 16). In the case that $s \in S_{return}$ (lines 18-22), then the algorithm propagates the resultant RSSG, after the [EXIT$_{nrec}$] transformation to the exit point of the callee, obtaining $RSSG^{out}$ (line 20). If $s$ is not a call or a return statement (lines 23-25), then just the corresponding transfer function is applied (line 24). Once the statement is processed, if the resultant $RSSG^{s\bullet}$ has changed, then the algorithm adds the successors of the statement under consideration (`succ(s)`) to the worklist (lines 26-29). The `Worklist_rec` algorithm (Fig. 30) processes the non-recursive call statements (lines 8-12) and the statements which are not a call or a return (lines 22-24) in similar way. Only in the case that statement $s$ is a recursive call (lines 13-16) or a return (a recursive return, in fact, lines 17-21), then it propagates the resultant output graphs $RSSG^{s\bullet}$ (after the [IN/OUT] transformation) to the entry points of the callee function (line 15) or the return point of the caller sites (line 20), respectively.

```
Worklist()
  Input: $P =< FUN, STMT, PTR, TYPE, SEL > \mid$     # A program or a non-recursive fun and an input $RSSG$
          $FUN =< FUN_{fun}, STMT_{fun}, PTR, TYPE, SEL >, RSSG^{in}$
  Output: $RSSG^{out}$                                # The $RSSG$ at the exit program point

 1:  Create $W = STMT$
 2:  $RSSG^{\bullet se} = RSSG^{in}$
 3:  $\forall s \in STMT \rightarrow RSSG^{s\bullet} = \emptyset$
 4:  repeat
 5:      Remove $s$ from $W$ in lexicographic order
 6:      $RSSG^{\bullet s} = \bigsqcup^{RSSG}_{s' \in pred(s)} RSSG^{s'\bullet}$
 7:      Case ($s$),
 8:          $s \in S_{call\_nrec}$
 9:              Let $fun \in FUN$, called by $s$
10:              $RSSG^{\bullet se_{fun}} = IN_{s \in S_{call\_nrec}}(RSSG^{\bullet s})$
11:              $RSSG^{s\bullet}$ =Worklist($< FUN_{fun}, STMT_{fun}, PTR, TYPE, SEL >, RSSG^{\bullet se_{fun}}$)
12:              break
13:          $s \in S_{call\_rec}$
14:              Let $fun \in FUN$, called by $s$
15:              $RSSG^{\bullet se_{fun}} = IN_{s \in S_{call\_rec}}(RSSG^{\bullet s})$
16:              $RSSG^{s\bullet}$ =Worklist_rec($< FUN_{fun}, STMT_{fun}, PTR, TYPE, SEL >, RSSG^{\bullet se_{fun}}$)
17:              break
18:          $s \in S_{return}$
19:              Let $s' \in S_{call\_nrec}$
20:              $RSSG^{out} = RSSG^{s\bullet} = OUT_{s' \in S_{call\_nrec}}(RSSG^{\bullet s})$
21:              $succ(s) = \emptyset$
22:              break
23:          $default$
24:              $RSSG^{s\bullet} = AS_s(RSSG^{\bullet s})$
25:              break
26:      If ($RSSG^{s\bullet}$ has changed),
27:          forall $s' \in succ(s)$,
28:              $W = W \cup s'$
29:          endfor
30:  until ($W = \emptyset$)
31:  return($RSSG^{out}$)
end
```

Figure 29: The extended worklist algorithm for interprocedural support. It computes the $RSSG^{s\bullet}$ at each program point.

```
Worklist_rec()
  Input: $FUN =< FUN_{fun}, STMT_{fun}, PTR, TYPE, SEL >$, $RSSG^{in}$    # A rec. $fun \in FUN$ and an input $RSSG$
  Output: $RSSG^{out}$                                  # The $RSSG$ at the exit program point

 1:  Create $W = STMT_{fun}$
 2:  $RSSG^{\bullet se_{fun}} = RSSG^{in}$
 3:  $\forall s \in STMT_{fun} \rightarrow RSSG^{s\bullet} = \emptyset$
 4:  repeat
 5:      Remove $s$ from $W$ in lexicographic order
 6:      $RSSG^{\bullet s} = \bigsqcup_{s' \in pred(s)}^{RSSG} RSSG^{s'\bullet}$
 7:      Case ($s$),
 8:          $s \in S_{call\_nrec}$
 9:              Let $foo \in FUN_{fun}$, called by $s$
10:              $RSSG^{\bullet se_{foo}} = IN_{s \in S_{call\_nrec}}(RSSG^{\bullet s})$
11:              $RSSG^{s\bullet}$ =Worklist($< FUN_{foo}, STMT_{foo}, PTR, TYPE, SEL >, RSSG^{\bullet se_{foo}}$)
12:              break
13:          $s \in S_{call\_rec}$
14:              $RSSG^{\bullet se_{fun}} = IN_{s \in S_{call\_rec}}(RSSG^{\bullet s})$
15:              $succ(s) = se_{fun}$
16:              break
17:          $s \in S_{return}$
18:              Let $\{s' \in S_{call\_rec} \subset STMT_{fun}\}$    # the recursive call sites at $fun$
19:              $RSSG^{out} = RSSG^{s\bullet} = \bigsqcup_{s' \in S_{call\_rec}} OUT_{s'}(RSSG^{\bullet s})$
20:              $succ(s) = \{succ(s') \ \forall s' \in S_{call\_rec} \subset STMT_{fun}\}$
21:              break
22:          $default$
23:              $RSSG^{s\bullet} = AS_s(RSSG^{\bullet s})$
24:              break
25:      If ($RSSG^{s\bullet}$ has changed),
26:          forall $s' \in succ(s)$,
27:              $W = W \cup s'$
28:          endfor
29:  until ($W = \emptyset$)
30:  return($RSSG^{out}$)
end
```

Figure 30: The Worklist_rec algorithm for recursive support. It computes the $RSSG^{s\bullet}$ at each statement function point.

```
CTS$_{nrec}$()
  Input: $sg^1 =< N^1, CLS^1 >, PTR_{fun}, \mathcal{AFPM}(s, fun)$        # a shape graph, formal and local pointers for $fun$
                                                  # and the set of pairs $< aptr, fptr >$ of the corresponding call site
  Output: $RSSG^k$                                # a reduced set of shape graphs

  $RSSG^2 = sg^1$
  forall $x \in APTR_s$                           # $APTR_s$ is the set of actual pointers in the call stmt. $s$
    Find the pair $< aptr, fptr > \in \mathcal{AFPM}(s, fun)$ s.t. $x = aptr$
    $RSSG^3 = \bigsqcup_{\forall sg' \in RSSG^2}^{RSSG} XY(sg', fptr, aptr)$      # $fptr = aptr$
    If $(aptr \notin GLB)$,
      $RSSG^4 = \bigsqcup_{\forall sg'' \in RSSG^3}^{RSSG} XNull(sg'', aptr)$  # $aptr = null$
    else $\rightarrow RSSG^4 = RSSG^3$
    $RSSG^2 = RSSG^4$
  endfor
  If $(\exists s' \in STM_{fun}$ s.t. $s' \in S_{call\_rec})$,        # The case when $fun$ will include a recursive call site
    forall $x \in PTR_{fun}$ s.t. $x \neq assptr$,
      forall $sg^i =< N^i, CLS^i > \in RSSG^2$,
        forall $n_j \in N^i$,                     # Initialize $x_{rfsel}$ for all nodes in all graphs
          Create $sl'_{att} =<< n_j, x_{rfsel}, null >, attsl' = \{o\}$
          $\forall cls_{nj} = \{PL_{nj}, SL_{nj}\} \in CLS_{nj}\ (being\ \ CLS_{nj} \subset CLS^i) \Longrightarrow SL_{nj} = SL_{nj} \cup sl'_{att}$
        endfor
      endfor
    endfor
  $RSSG^k = RSSG^2$
  return($RSSG^k$)
end
```

Figure 31: The CTS$_{nrec}$() function.

```
CTS$_{rec}$()
  Input: $sg^1 =< N^1, CLS^1 >, PTR_{fun}, \mathcal{AFPM}(s, fun)$        # a shape graph, formal and local pointers for $fun$
                                                                          # and the set of pairs $< aptr, fptr >$ of the corresponding call site
  Output: $RSSG^k$                                                        # a reduced set of shape graphs

  $RSSG^2 = sg^1$
  forall $x \in PTR_{fun}$ s.t. $(x \notin APTR_s \wedge x \neq assptr)$,   # $APTR_s$ is the set of actual pointers in the call stmt. $s$
      $RSSG^3 = \bigsqcup^{RSSG}_{\forall sg' \in RSSG^2} XSelY(sg', x, x_{rfsel}, x_{rfptr})$      # $x- > x_{rfsel} = x_{rfptr}$
      $RSSG^4 = \bigsqcup^{RSSG}_{\forall sg'' \in RSSG^3} XY(sg'', x, x_{rfptr}, x)$    # $x_{rfptr} = x$
      $RSSG^5 = \bigsqcup^{RSSG}_{\forall sg''' \in RSSG^4} XNull(sg''', x)$          # $x = null$
  endfor
  forall $x \in APTR_s$
      Find the pair $< aptr, fptr > \in \mathcal{AFPM}(s, fun)$ s.t. $x = aptr$
      $RSSG^3 = \bigsqcup^{RSSG}_{\forall sg' \in RSSG^2} XY(sg', fptr, aptr)$      # $fptr = aptr$
      If $(aptr \notin GLB)$,
          $RSSG^4 = \bigsqcup^{RSSG}_{\forall sg'' \in RSSG^3} XNull(sg'', aptr)$  # $aptr = null$
      else $\rightarrow RSSG^4 = RSSG^3$
      $RSSG^2 = RSSG^4$
  endfor
  $RSSG^k = RSSG^2$
  return($RSSG^k$)
end
```

Figure 32: The CTS$_{rec}$() function.

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

  $RSSG^1 = XY(sg^1, assptr, retptr)$   # $assptr = retptr$
  $RSSG^2 = RSSG^1$
  forall $x \in APTR_s$                 # $APTR_s$ is the set of actual pointers in the call stmt. $s$
    Find the pair $< aptr, fptr > \in \mathcal{AFPM}(s, fun)$ s.t. $x = aptr$
    $RSSG^3 = \bigsqcup^{RSSG}_{\forall sg' \in RSSG^2} XY(sg', aptr, fptr)$   # $aptr = fptr$
    $RSSG^2 = RSSG^3$
  endfor
  forall $x \in PTR_{fun}$,
    $RSSG^3 = \bigsqcup^{RSSG}_{\forall sg'' \in RSSG^2} XNull(sg'', x)$   # $x = null$
    $RSSG^4 = \bigsqcup^{RSSG}_{\forall sg''' \in RSSG^3} XNull(sg''', x_{rfptr})$   # $x_{rfptr} = null$
    $RSSG^5 = \emptyset$
    forall $sg^i =< N^i, CLS^i > \in RSSG^4$,
      forall $n_j \in N^i$,             # Remove $x_{rfsel}$ for all nodes in all graphs
        forall $cls_{nj} = \{PL_{nj}, SL_{nj}\} \in CLS_{nj}$ $(being\ CLS_{nj} \subset CLS^i)$,
          Find $sl_{att1} \subset cls_{nj}\ being\ sl_{att1} =<< n_k, x_{rfsel}, n_p >, attsl1 >$
          $SL_{nj} = SL_{nj} - sl_{att1}$
        endfor
      endfor
      $sg^{i'}$ =Normalize_SG($sg^i$)
      $RSSG^5 = RSSG^5 \cup sg^{i'}$
    endfor
    $RSSG^2 = RSSG^5$
  endfor
  $RSSG^k = RSSG^2$
  return($RSSG^k$)
end
```

Figure 33: The RTC$_{nrec}$() function.

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

  $RSSG^1 = XY(sg^1, assptr, retptr)$     # $assptr = retptr$
  $RSSG^2 = RSSG^1$
  forall $x \in APTR_s$                   # $APTR_s$ is the set of actual pointers in the call stmt. $s$
    Find the pair $< aptr, fptr >\in \mathcal{AFPM}(s, fun)$ s.t. $x = aptr$
    $RSSG^3 = \bigsqcup_{\forall sg' \in RSSG^2}^{RSSG} XY(sg', aptr, fptr)$    # $aptr = fptr$
    $RSSG^2 = RSSG^3$
  endfor
  forall $x \in PTR_{fun}$ s.t. $(x \notin APTR_s \wedge x \neq assptr)$,
    $RSSG^4 = \bigsqcup_{\forall sg'' \in RSSG^2}^{RSSG} XY(sg'', x, x_{rfptr})$          # $x = x_{rfptr}$
    $RSSG^5 = \bigsqcup_{\forall sg''' \in RSSG^4}^{RSSG} XYSel(sg''', x_{rfptr}, x, x_{rfsel})$   # $x_{rfptr} = x- > x_{rfsel}$
    $RSSG^6 = \bigsqcup_{\forall sg'''' \in RSSG^5}^{RSSG} XSelNull(sg'''', x, x_{rfsel})$      # $x- > x_{rfsel} = null$
    $RSSG^2 = RSSG^6$
  endfor
  $RSSG^k = RSSG^2$
  return($RSSG^k$)
end
```

Figure 34: The RTC$_{rec}$() function.

**Overview of the tests**

We have considered six programs for our tests. The first four are synthetic codes representative of typical recursive data structures found in pointer-based codes. For the last two tests, we have designed a small program that computes the product of a sparse matrix by a sparse vector. Sparse structures are usually built with pointers to avoid wasting storage capacity with many empty values.

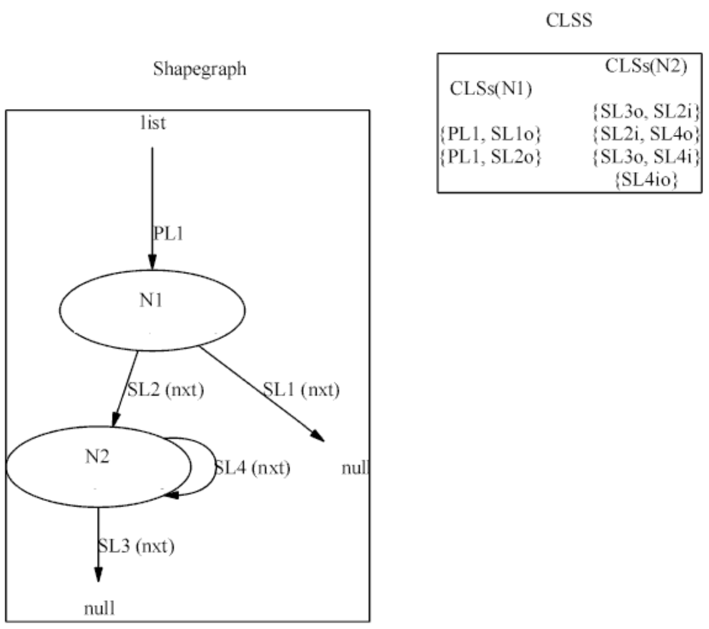
Programs are preprocessed by a custom pass created over Cetus [4], an extensible Java infrastructure for source-to-source transformations. Basically, this pass translates a C input program into a format recognizable by the shape analyzer. When analysing a program, we do not need to consider all statements. Our technique only cares about control flow statements and pointer access statements, which is what the shape analyzer needs to obtain the graphs that describe the shape of memory configurations in the heap. In the codes shown below for the tests, we show the abridged version as analyzed by the shape analyzer. Therefore, the statements shown are exactly the statements analyzed.

Since shape analysis is a conservative technique by nature, it must account for all possible flow paths in the program. We do not pay attention to conditions in branching statements, but consider all possibilities, i.e., branch taken and branch not taken. That is why branches and loops do not show the conditions in the code for the tests. However, when a pointer condition is known, it is valuable for discarding configurations rendered impossible by the condition. *Force directives* are used in such cases to enforce pointer conditions at certain points in the program. They are derived from the conditions specified at control flow statements. For example, when entering a `while(p!=NULL)` loop, we can enforce the analysis to consider `p!=NULL` inside the loop and `p==NULL` just outside the loop. Force directives make the analysis more precise and faster, because it can rule out unnecessarily conservative memory configurations. Force directives are added with pragma directives. There is work in progress to add a source-to-source translation pass based on Cetus to automatically add force directives, but at this point they are added by the programmer.

In the codes below, you will also notice several *nullification statements*. Pointers can be nullified as long as they are *dead*, i.e., there is no use before a definition following the flow path from a point in the program. By nullifying pointers early, we make the analysis faster as it suffers from exponential complexity with respect to the number of non-null live pointer variables. There would be a prior *dead variable nullification* pass to condition the code in this manner in an automated basis, but at this point pointer nullification is done by the programmer.

Next we describe each test with the code analyzed and the graph resulting from its analysis, as displayed by our visualization companion tool. In the graphs, CLSs for the nodes are displayed unordered, i.e., the order in which CLSs appear does not have to match the order in which they were calculated by the analyzer. Tests are run in *multi-graph mode*, meaning that there may be several graphs per statement during the analysis, to achieve precision at nodes pointed to by pointers. However, we only show the final graph, obtained as the joining of all available graphs resulting at the end of the analysis. No properties are considered for summarization.

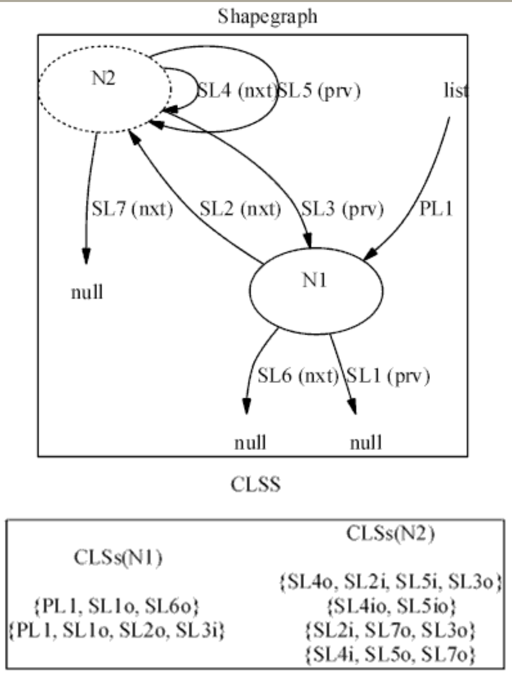
**Test 1: singly-linked list**

| | |
|---|---|
| **Code:** this test first creates a singly-linked list (stmts. 1-6), then traverses it (stmts. 11-15). Nullification statements and force directives are inserted where appropriate. | **Graph:** it captures a singly-linked list of length greater or equal to 1 element. `N1` represents the first element in the list. From it, the `nxt` selector can lead to `null` for a 1-element list (with `CLS(N1)={PL1,SL1o}`), or it can lead to the second element (`CLS(N1)={PL1,SL2o}` for `N1` and `CLS(N2)` contaning `SL2i` for `N2`). `N2` is a summary node that represents all possible locations in the list that are not pointed to by pointers. `CLSs(N2)` describe the four possibilities of connectivity for such locations: `{SL3o,SL2i}` represents the second element in a 2-element list; `{SL2i, SL4o}` represents the second element in a list longer than 2 elements; `{SL3o,SL4i}` captures the last element in a list longer than 2 elements; finally `{SL4io}={SL4i,SL4o}` stands for all intermediate locations. |
| `1   list = malloc();`<br>`2   p = list;`<br>`3   while(){`<br>`4       q = malloc();`<br>`5       p->nxt = q;`<br>`6       p = q;`<br>`    }`<br>`7   Force(list != NULL)`<br>`8   p->nxt = NULL;`<br>`9   q = NULL;`<br>`10  p = NULL;`<br>`11  p = list;`<br>`12  while(){`<br>`13      q = p -> nxt;`<br>`14      p = q;`<br>`    }`<br>`15  Force(p = NULL)`<br>`16  q = NULL;`<br>`17  p = NULL;` |  CLSS<br>Shapegraph<br>list<br>PL1<br>N1<br>SL2 (nxt)<br>SL1 (nxt)<br>N2<br>SL4 (nxt)<br>nul<br>SL3 (nxt)<br>null<br>CLSs(N2)<br>CLSs(N1)<br>{SL3o, SL2i}<br>{PL1, SL1o}<br>{SL2i, SL4o}<br>{PL1, SL2o}<br>{SL3o, SL4i}<br>{SL4io} |

**Test 2: doubly-linked list**

| | |
|---|---|
| **Code:** this is basically the same as test1, but the list is doubly-linked. | **Graph:** this graph captures a doubly-linked list. `N1` is the entry element for the list, pointed to by the `list` pointer. `N2` represents all possible locations beyond the first element. It is drawn in dotted line to indicate that locations represented can be reachable more than once through *different* selectors. This is certainly true in a doubly-linked list, as elements in the middle are referenced through the `nxt` selector from the previous element, and through the `prv` selector from the next element. A location cannot be reached through the same selector more than once, thus preventing the existence of cycles other than those produced by the `N2.nxt-N2.prv` sequence. Note that most shape analysis techniques have troubles capturing doubly-linked structures. |

```
list = malloc();
list->prv = NULL;
p = list;
while(){
     q = malloc();
     p->nxt = q;
     q->prv = p;
     p = q;
}
Force(list != NULL)
p->nxt = NULL;
q = NULL;
p = NULL;
p = list;
while(){
     q = p -> nxt;
     p = q;
}
Force(p = NULL)
q = NULL;
p = NULL;
```

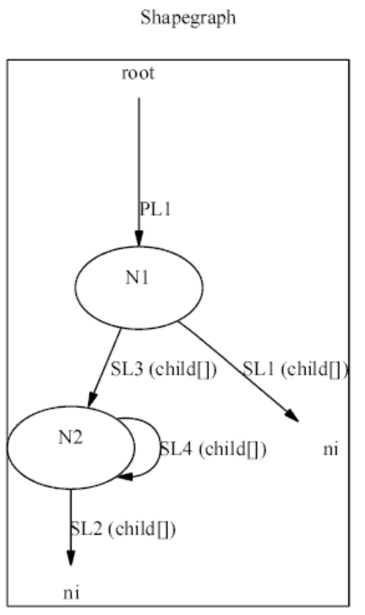


**Test 3: n-ary tree**

**Code:** this test creates an array-based n-ary tree. Each location in the program contains a pointer array, whose elements can points to other locations. The tree is traversed during its creation, as each new leaf is added starting from the root. Statements 6 and 17 indicate that the array index has been written, which makes the analyzer *forget* the previous value.

**Graph:** this graph, as simple as it may seem, represents an array-based n-ary tree. This graph features *multi-selectors* (recognizable by the "`[]`" suffix), which are selectors that can point to several different locations at the same time, unlike regular selectors. `N1` is the root for the tree. `N2` is a summary node for the rest of elements in the tree (intermediate elements and the leaves). `CLS(n1)={PL1,SL1o,SL3o}` tells that the first element can link through the `child[]` multi-selector to other elements (represented by `N2`) and also have uninitialized links (reaching `ni`, meaning non-initialized). `CLS(n2)={SL2o,SL4io}={SL2o,SL4i,SL4o}` represents locations in the middle of the tree which are linked from *just one* intermediate element located upper in the tree (`SL4i`), and that links to other lower elements (`SL4o`) and also may have uninitialized links in its multi-selector (`SL2o`). What is important here is that every location in the tree cannot be reached more than once by following the `child[]` multi-selector, because nodes are not in dotted line. Therefore children do not link back to any ancestor nor are they shared for different parents, so the tree shape is correctly captured. Note also that current shape analysis techniques do not support pointer arrays explicitly.

```
1    root = malloc();
2    while(){
3        p = root;
4        while(){
5            Force(p != NULL)
6            i = ...;
7            if(){
8                Force(p->child[i] != NULL)
9                q = p -> child[i];
10               p = q;
11               q = NULL;
             }else{
             }
         }
12       Force(p->child[i] = NULL)
13       x = malloc();
14       p->child[i] = x;
15       x = NULL;
     }
16   p = NULL;
17   i = ...;
```



CLSS

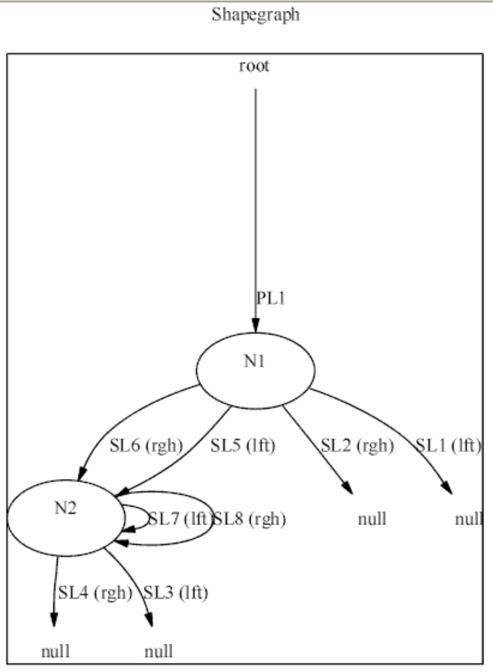
| CLSs(N1) | CLSs(N2) |
|---|---|
| | {SL2o, SL3i} |
| | {SL2o, SL4i} |
| {PL1, SL1o} | {SL2o, SL3i, SL4o} |
| {PL1, SL1o, SL3o} | {SL2o, SL4io} |

**Test 4: binary tree**

**Code:** this test creates a binary tree. Each location in the program contains two selectors (lft and rgh) that can point to 2 children. The tree is traversed during its creation, as each new leaf is added starting from the root.

**Graph:** this graph represents a binary tree. `N1` represents the root element, pointed by the `root` pointer. `N2` represents all intermediate locations in the tree and the leaves. CLSs for `N2` are many, to correctly capture all possibilities: second-level element as left child of root with right and left children (9th `CLS(N2)={SL7o,SL8o,SL5i}`), intermediate-level element as right child of parent with right and left children (last `CLS(N2)={SL7o,SL8io}`), leaf as left child of parent (3rd `CLS(N2)={SL7i,SL4o,SL3o}`), etc.
Again, what is important here is that no node is reached through `SL7i` and `SL8i` in the same CLS (both a left and right child at the same time), `N2` is not in dotted lines (children do not link back to ancestors), and that no SL is shared in any CLS (for example, a left child for two or more parents). Thus the binary tree shape characteristics are accurately captured in the graph.

```
root = malloc();
root->lft = NULL;
root->rgh = NULL;
while(){
    p = root;
    while(){
        Force(p != NULL)
        if(){
            q = p -> lft;
            p = q;
            q = NULL;
        }else{
            q = p -> rgh;
            p = q;
            q = NULL;
        }
    }
    Force(p != NULL)
    x = malloc();
    x->lft = NULL;
    x->rgh = NULL;
    if(){
        Force(p->lft = NULL)
        p->lft = x;
    }else{
        Force(p->rgh = NULL)
        p->rgh = x;
    }
    x = NULL;
}
p = NULL;
```



CLSS

| CLSs(N1) | CLSs(N2) |
|---|---|
| {PL1, SL1o, SL2o} | {SL4o, SL3o, SL5i} |
| {PL1, SL2o, SL5o} | {SL6i, SL4o, SL3o} |
| {PL1, SL1o, SL6o} | {SL7i, SL4o, SL3o} |
| {PL1, SL6o, SL5o} | {SL7o, SL4o, SL5i} |
| | {SL6i, SL7o, SL4o} |
| | {SL8i, SL4o, SL3o} |
| | {SL8o, SL3o, SL5i} |
| | {SL6i, SL8o, SL3o} |
| | {SL7o, SL8o, SL5i} |
| | {SL6i, SL7o, SL8o} |
| | {SL7io, SL4o} |
| | {SL7o, SL8i, SL4o} |
| | {SL7i, SL8o, SL3o} |
| | {SL8io, SL3o} |
| | {SL7io, SL8o} |
| | {SL7o, SL8io} |

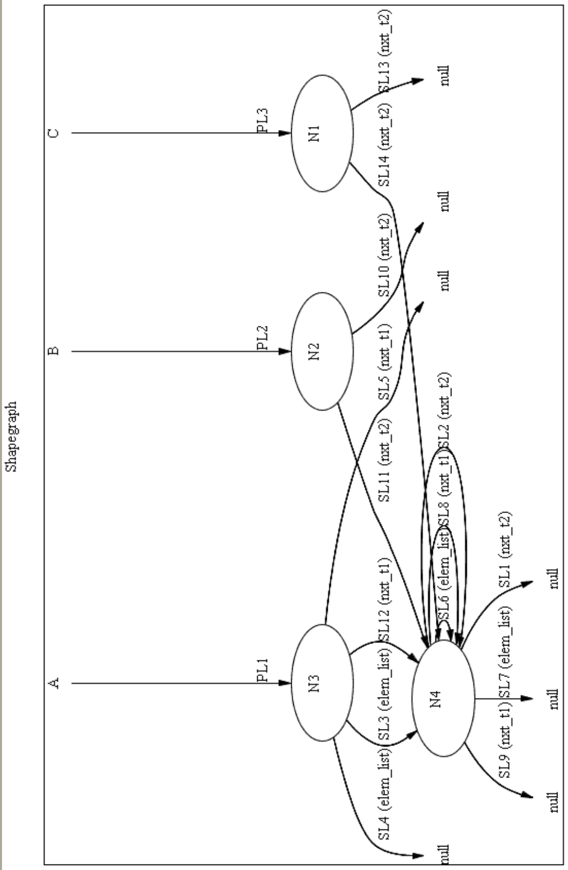
## Test 5: Sparse matrix by sparse vector based on singly-linked lists

**Code:** this test takes a real working program that computes the product of a sparse matrix by a sparse vector. The matrix is constructed as a list of singly-linked header elements of type `t1`, that link through selector `nxt_t1`. Each header element links to a list of singly-linked elements of type `t2`, that link through selector `nxt_t2`. The vectors are built as singly-linked lists of elements of type `t2` The analyzer is fed with the code below. The entry point for the analysis is statement 83, the call to main()at statement 1. First the input matrix `A` is created (stmts. 2-31), then the input vector `B` is created (stmts. 32-47). Finally the output vector `C` is created as `A` and `B` are traversed (stmts. 48-82). Structure navigation statements that read and write on the same location are decomposed using temporal variables (`_tmpx`). For example, statements 74-76 show how the navigation pointer for the header list of the matrix, `auxHA`, is updated using a temporal variable in the loop that computes the product (stmts. 50-76).

**Graph:** this graph captures the 3 structures used in this test: `A`, the input matrix; `B`, the input vector; and `C` the output vector. As we use no properties all locations that are not directly accessed by pointer are summarized in node `N4`. The node is drawn in solid line. This means that every location represented by `N4` links to other different location, i.e., there are no locations which are linked twice or more from other locations. Therefore, although `N4` serves as summary nodes for all intermediate elements in the 3 structures, `CLSs(N4)` assure that the structures are disjoint. This includes the fact that rows *hanging* from the header list in the matrix are not shared either, otherwise there would be a `CLS(N4)` with `SL3is` (shared incoming `SL3`). The main characteristics of the heap for this program are captured in the graph: 3 disjoint structures based on acyclic singly-linked lists.

```
1   main(){
2       auxH = NULL;
3       while(){
4           newH = malloc();
5           if(){
6               Force(auxH != NULL)
7               auxH->nxt_t1 = newH;
            }else{
8               Force(auxH = NULL)
9               A = newH;
            }
10          auxH = newH;
11          auxE = NULL;
12          while(){
13              if(){
14                  newE = malloc();
15                  if(){
16                      Force(auxE!=NULL)
17                      auxE->nxt_t2=newE;
                    }else{
18                      Force(auxE=NULL)
19                      anchor = newE;
                    }
20                  auxE = newE;
                }else{
                }
            }
21          auxE = NULL;
22          if(){
23              Force(newE != NULL)
24              newE->nxt_t2 = NULL;
            }else{
25              Force(newE = NULL)
            }
26          newE = NULL;
27          auxH->elem_list = anchor;
28          anchor = NULL;
        }
29      newH->nxt_t1 = NULL;
30      newH = NULL;
31      auxH = NULL;
32      B = NULL;
33      lastE = NULL;
34      while(){
```

```
35          if(){
36              newE = malloc();
37              if(){
38                  Force(B = NULL)
39                  B = newE;
                }else{
40                  Force(B != NULL)
41                  lastE->nxt_t2 = newE;
                }
42              lastE = newE;
43              newE = NULL;
            }else{
            }
        }
44      lastE->nxt_t2 = NULL;
45      lastE = NULL;
46      auxHA = A;
47      auxHC = NULL;
48      C = NULL;
49      lastE = NULL;
50      while(){
51          Force(auxHA != NULL)
52          auxEB = B;
53          while(){
54              Force(auxEB != NULL)
55              auxEA = auxHA->elem_list;
56              while(){
57                  _tmp1 = auxEA->nxt_t2;
58                  auxEA = _tmp1;
59                  _tmp1 = NULL;
                }
60              auxEA = NULL;
61              _tmp2 = auxEB -> nxt_t2;
62              auxEB = _tmp2;
63              _tmp2 = NULL;
            }
64          auxEB = NULL;
65          if(){
66              newE = malloc();
67              if(){
68                  Force(C = NULL)
69                  C = newE;
                }else{
70                  Force(C != NULL)
71                  lastE->nxt_t2 = newE;
                }
72              lastE = newE;
73              newE = NULL;
            }else{
            }
74          _tmp3 = auxHA -> nxt_t1;
75          auxHA = _tmp3;
76          _tmp3 = NULL;
        }
77      if(){
78          Force(lastE != NULL)
79          lastE->nxt_t2 = NULL;
        }else{
80          Force(lastE = NULL)
        }
81      lastE = NULL;
82      auxHA = NULL;
    }
83  main();
```



CLSS

CLSs(N1)
{PL3, SL13o}
{PL3, SL14o}

CLSs(N2)
{PL2, SL10o}
{PL2, SL11o}

CLSs(N3)
{PL1, SL5o, SL3o}
{PL1, SL5o, SL4o}
{PL1, SL3o, SL12o}
{PL1, SL4o, SL12o}

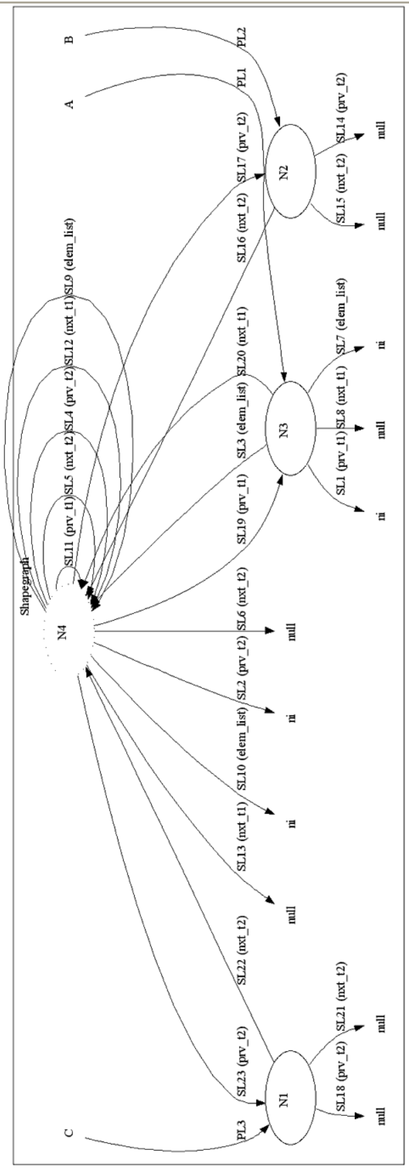
CLSs(N4)
{SL3i, SL1o}
{SL2i, SL1o}
{SL2o, SL3i}
{SL2io}
{SL6i, SL1o}
{SL6i, SL2o}
{SL6o, SL8i, SL9o}
{SL8i, SL9o, SL7o}
{SL6o, SL8io}
{SL8io, SL7o}
{SL11i, SL1o}
{SL2o, SL11i}
{SL6o, SL9o, SL12i}
{SL9o, SL7o, SL12i}
{SL6o, SL8o, SL12i}
{SL8o, SL7o, SL12i}
{SL14i, SL1o}
{SL14i, SL2o}

**Test 6: Sparse matrix by sparse vector based on doubly-linked lists**

**Code:** this test is basically the same as test 5, but all lists are doubly-linked. You will also notice some special statements (stmts. 68, 69, 74 and 90) related to the *touch* property. This statements are used to draw information about how the structures are traversed. However, all presented tests are run without properties, as stated above. Therefore touch statements are ignored in this test.

**Graph:** this graph is the double-linked counterpart for that of test 5. Here, locations represented by `N4` can be reachable more than once, therefore the node is drawn in dotted line. Let us check the structures characteristics by observing available CLSs for `N4`. The 4th `CLS(N4)={SL4io,SL5io}`, tells that structures of type `t2` are based on doubly-linked lists, while the 9th `CLS(N4)={SL11io,SL12io,SL9o}`, tells that structures of type `t1` are also based on doubly-linked lists. There are no shared SLs in any CLS, so elements are not reached twice from the same selector. In particular, hanging lists from the header list in `A`, are not shared through the `elem_list` selector. To sum up, this graph represents 3 disjoint heap structures based on doubly-linked lists that contain no cycles other than the `nxt-prv` cycle inherent to doubly-linked lists.

```
1    main(){
2       auxH = NULL;
3       while(){
4            newH = malloc();
5            if(){
6                Force(auxH != NULL)
7                newH->prv_t1 = auxH;
8                auxH->nxt_t1 = newH;
            }else{
9                Force(auxH = NULL)
10                A = newH;
            }
11          auxH = newH;
12          auxE = NULL;
13          while(){
14              if(){
15                  newE = malloc();
16                  if(){
17                      Force(auxH->elem_list=NULL)
18                      auxH->elem_list = newE;
                    }else{
                    }
19                  if(){
20                      Force(auxE != NULL)
21                      newE->prv_t2 = auxE;
22                      auxE->nxt_t2 = newE;
                    }else{
23                      Force(auxE = NULL)
24                      auxH->elem_list = newE;
                    }
25                  auxE = newE;
                }else{
                }
            }
26          auxE = NULL;
27          if(){
28              Force(newE != NULL)
29              newE->nxt_t2 = NULL;
            }else{
30              Force(newE = NULL)
            }
31           newE = NULL;
        }
32      newH->nxt_t1 = NULL;
33      newH = NULL;
34      auxH = NULL;
35      B = NULL;
36      lastE = NULL;
37      while(){
38          if(){
39              newE = malloc();
40              if(){
41                  Force(B = NULL)
42                  B = newE;
43                  newE->prv_t2 = NULL;
                }else{
44                  Force(B != NULL)
45                  lastE->nxt_t2 = newE;
46                  newE->prv_t2 = lastE;
                }
47              lastE = newE;
48              newE = NULL;
            }else{
            }
        }
49      lastE->nxt_t2 = NULL;
50      lastE = NULL;
51      auxHA = A;
52      auxHC = NULL;
53      C = NULL;
54      lastE = NULL;
55      while(){
56          Force(auxHA != NULL)
57          auxEB = B;
58          while(){
59              Force(auxEB != NULL)
60              auxEA = auxHA -> elem_list;
61              while(){
62                  Force(auxEA != NULL)
63                   _tmp1 = auxEA -> nxt_t2;
64                   auxEA = _tmp1;
65                   _tmp1 = NULL;
                }
66              if(){
67                  Force(auxEA != NULL)
                }else{
                }
68              Touch(auxEA, Read68)
69              Touch(auxEB, Read69)
70              auxEA = NULL;
71              _tmp2 = auxEB -> nxt_t2;
72              auxEB = _tmp2;
73              _tmp2 = NULL;
            }
74          UnTouch(Read69)
75          auxEB = NULL;
76          if(){
77              newE = malloc();
78              if(){
79                  Force(C = NULL)
80                  C = newE;
81                  newE->prv_t2 = NULL;
                }else{
82                  Force(C != NULL)
83                  lastE->nxt_t2 = newE;
84                  newE->prv_t2 = lastE;
                }
85              lastE = newE;
86              newE = NULL;
            }else{
            }
87          _tmp3 = auxHA -> nxt_t1;
88          auxHA = _tmp3;
89          _tmp3 = NULL;
        }
90      UnTouch(Read68)
91      if(){
92          Force(lastE != NULL)
93          lastE->nxt_t2 = NULL;
        }else{
94          Force(lastE = NULL)
        }
95      lastE = NULL;
96      auxHA = NULL;
    }
97  main();
```



CLSS

CLSs(N4)

{SL2o, SL3i, SL6o}
{SL4o, SL5i, SL6o}
{SL2o, SL3i, SL4i, SL5o}
{SL4io, SL5io}
{SL2o, SL6o, SL9i}
{SL2o, SL4i, SL5o, SL9i}
{SL11o, SL12i, SL13o, SL9o}
{SL11o, SL10o, SL12i, SL13o}
{SL11io, SL12io, SL9o}
{SL11io, SL10o, SL12io}
{SL17o, SL6o, SL16i}
{SL4i, SL17o, SL5o, SL16i}
{SL19o, SL20i, SL13o, SL9o}
{SL19o, SL10o, SL20i, SL13o}
{SL19o, SL11i, SL12o, SL20i, SL9o}
{SL19o, SL11i, SL10o, SL12o, SL20i}
{SL23o, SL22i, SL6o}
{SL4i, SL23o, SL22i, SL5o}

CLSs(N1)

{PL3, SL21o, SL18o}
{PL3, SL18o, SL23i, SL22o}

CLSs(N2)

{PL2, SL15o, SL14o}
{PL2, SL17i, SL16o, SL14o}

CLSs(N3)

{PL1, SL3o, SL1o, SL8o}
{PL1, SL7o, SL1o, SL8o}
{PL1, SL19i, SL3o, SL1o, SL20o}
{PL1, SL19i, SL7o, SL1o, SL20o}

**Results**

| Data structure | # stmts | Time | # graphs | Nodes, links & CLSs per graph |
|---|---|---|---|---|
| Singly-linked list | 17 | 0.47 sec | 62 | 2.51 (4) / 3.64 (7) / 4.75 (13) |
| Doubly-linked list | 19 | 0.52 sec | 74 | 2.59 (4) / 6.90 (13) / 4.55 (13) |
| N-ary tree | 17 | 0.62 sec | 372 | 2.61 (4) / 6.39 (12) / 9.38 (22) |
| Binary tree | 25 | 2.02 sec | 435 | 2.73 (4) / 10.58 (20) / 23.84 (65) |
| Matrix-vector(s) | 83 | 1.14 min | 2477 | 7.56 (12) / 26.10 (40) / 29.34 (50) |
| Matrix-vector(d) | 97 | 1.55 min | 2931 | 7.60 (12) / 30.95 (48) / 30.37 (50) |

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

Table I describes the structures tested and displays some metrics for the analysis performed. The first column identifies each test, while the second column holds the number of analyzed statements. The third column shows times for the tests. Only the time for the actual shape analysis is shown (no parsing or preprocessing), as measured in a Pentium IV 2.4 GHz with 1 GB RAM, with the `time()` command in a Fedora Core 3 Linux OS. We think that times are very reasonable for such a detailed analysis. Within the first four examples of synthetic codes, the highest time is that of the binary tree analysis, probably due to its more complex CFG. It should be noted that more possible flow paths make the analysis more costly, as it has to consider all possibilities conservatively. On the other hand, the first three examples run in less than a second. The matrix by vector product takes longer, clocking at more than 1 minute, which is only reasonable considering there are quite some more statements to analyze than in previous tests.

The fourth column indicates the total number of graphs generated for each test. The numbers range from a few dozens to a few thousands, accounting for higher number of analyzed statements and/or higher complexity of the structure. Memory use is quite reasonable, staying below 17 MB in the worst case (`matrix-vector(d)`). This is very encouraging considering the big penalty in memory use found in related work. Also remember that all tests are run in multi-graph mode, meaning that several graphs can be used per statement in order to correctly capture memory configurations arising in the program. Therefore these runnings represent the most costly analysis case for our tool.

Next columns show the total number of nodes, links and CLSs per graph, as average values with the maximum in brackets. The number of nodes per graph is essentially constant in the first four tests, as it depends mostly on the number of simultaneously live pointers, which is usually one for the structure handle and two for navigating it. The matrix by vector test has three times more nodes because there are three different structures, instead of one. The number of links depends on the amount of different links that each element has. Typically each element in a recursive data structure does not have more than two links. Finally, CLSs are the elements where most of the complexity reside: they describe how nodes and links can combine to create all possible memory configurations arising in the program. The highest maximum is for the binary tree among all tests, but the maximum average is attained in the matrix by vector program based on doubly-linked lists.

To sum up, we can say that the shape analyzer can effectively analyze common data structures for pointer-based codes. Generated graphs accurately capture heap structures. Furthermore, we think that such graphs can be obtained in manageable times, specially for such a complex technique. Let us not forget that we are performing fixed-point abstract interpretation of pointer and flow statements to create and modify very detailed graphs.

Despite this encouraging results, it is clear that this is a costly technique which is not likely to succeed if used for whole program analysis. Instead it would be better used within a client analysis module that would focus on *local analysis*.

In this regard, we 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 graph that represents the data structure in the program. With this approach we avoid to analyze irrelevant statements that slow down the shape analysis.

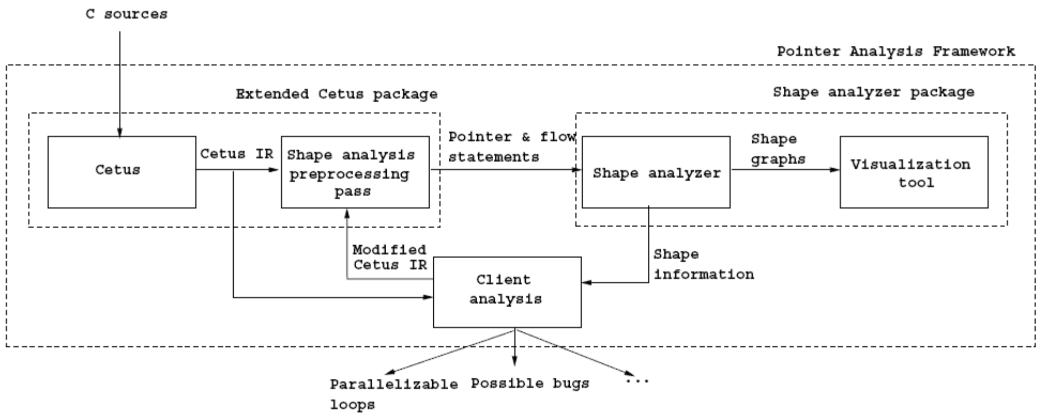
We have tried this approach on the matrix by vector examples. Let us revisit them now, having pruned all traversal statements that are not involved in the output vector creation (stmts. 51-64 and 74-76 for test 5, and stmts. 59-75 and 87-89 for test 6). The new values for the tests are shown in table II, where the original values for the unprocessed versions are also displayed for reference.

| Data structure | # stmts | Time | # graphs | Nodes, links & CLSs per graph |
|---|---|---|---|---|
| Matrix-vector(o,s) | 83 | 1.14 min | 2477 | 7.56 (12) / 26.10 (40) / 29.34 (50) |
| Matrix-vector(p,s) | 66 | 7.52 sec | 772 | 5.69 (10) / 19.28 (36) / 19.91 (48) |
| Matrix-vector(o,d) | 97 | 1.55 min | 2931 | 7.60 (12) / 30.95 (48) / 30.37 (50) |
| Matrix-vector(p,d) | 77 | 9.22 sec | 823 | 5.45 (10) / 21.29 (42) / 19.68 (48) |

**Table II**. The matrix by vector product analyzed in original (o) and pruned (p) forms, based in singly-linked (s) or doubly-linked (d) lists.

The results prove that def-use driven shape analysis works much better, as the analysis time has been reduced dramatically. Pruned tests produce the same output graphs than their original counterparts, thus capturing memory configuration without any loss in precision. This example motivates us to tightly integrate shape analysis within client analysis that focus on the statements of interest.

In this sense, we have already started work toward using the shape analyzer as a base tool for a pointer analysis framework [1], that combines several pointer analysis techniques, existent and new, for optimizations related to parallelism and locality. This way, shape information could be used by client analysis modules to derive information about safely parallelizable loops, possible bugs, etc. Next figure gives an overview of such a framework.



## References

**1. Towards a Versatile Pointer Analysis Framework**,
R. Castillo, A. Tineo, F. Corbera, A. Navarro, R. Asenjo and E.L. Zapata,
In European Conference on Parallel Computing (EURO-PAR) 2006, 29th August - 1st September 2006 (submitted).

**2. Shape Analysis for Dynamic Data Structures based on Coexistent Links Sets**,
A. Tineo, F. Corbera, A. Navarro, R. Asenjo and E.L. Zapata,
In 12th Workshop on Compilers for Parallel Computers, CPC'06, 9-11 January 2006, A Coruña, Spain.

**3. A New Strategy for Shape Analysis Based on Coexistent Links Sets**,
A. Tineo, F. Corbera, A. Navarro, R. Asenjo and E.L. Zapata,
In Parallel Computing 2005 (ParCo'05). 13-16 September 2005, Malaga, Spain.

**4. Cetus - An Extensible Compiler Infrastructure for Source-to-Source Transformation**,
Sang-Ik Lee, Troy A. Johnson, and Rudolf Eigenmann,
16th International Workshop on Languages and Compilers for Parallel Computing (LCPC), pages 539-553, October 2003.