Experimental results in shape analysis

Technical report

Rosa Castillo, Adrian Tineo, Francisco Corbera,
Angeles Navarro, Rafael Asenjo, Emilio L. Zapata
Dpt. of Computer Architecture, University of Málaga

{rosa,tineo,corbera,angeles,asenjo,ezapata}@ac.uma.es

Last update: 9th-March-06

Table of contents:

Foreword

This page documents the results of some experiments conducted for our shape analyzer based on CLSs, as found in [1] and [2]. The rest of this document assumes you are familiar with the basics of the technique. See the References section for appropriate background.

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.

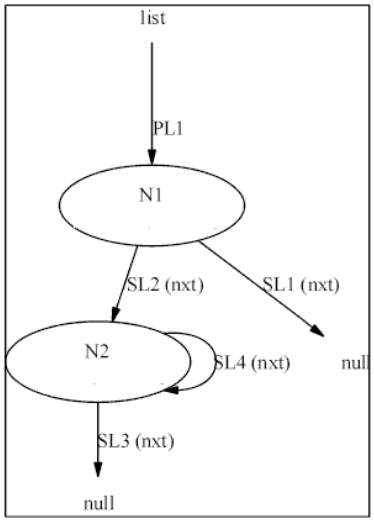
Test 1: singly-linked list

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();
2   p = list;
3   while(){
4       q = malloc();
5       p->nxt = q;
6       p = q;
    }
7   Force(list != NULL)
8   p->nxt = NULL;
9   q = NULL;
10  p = NULL;
11  p = list;
12  while(){
13      q = p -> nxt;
14      p = q;
    }
15  Force(p = NULL)
16  q = NULL;
17  p = NULL;
```

Shapegraph



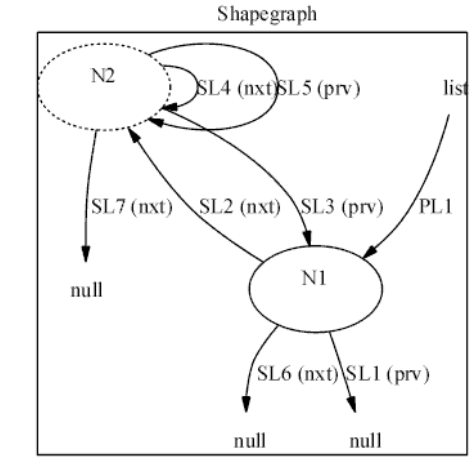
CLSS

CLSs(N1)	CLSs(N2)
	{SL3o, SL2i}
{PL1, SL1o}	{SL2i, SL4o}
{PL1, SL2o}	{SL3o, SL4i}
	{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.

```
1   list = malloc();
2   list->prv = NULL;
3   p = list;
4   while(){
5       q = malloc();
6       p->nxt = q;
7       q->prv = p;
8       p = q;
    }
9   Force(list != NULL)
10  p->nxt = NULL;
11  q = NULL;
12  p = NULL;
13  p = list;
14  while(){
15      q = p -> nxt;
16      p = q;
    }
17  Force(p = NULL)
18  q = NULL;
19  p = NULL;
```



CLSS

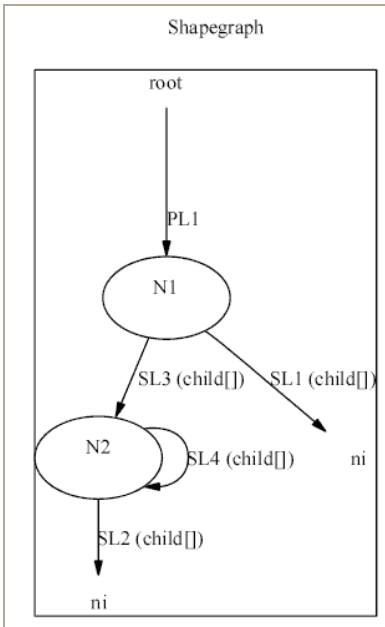
CLSs(N1)	CLSs(N2)
{PL1, SL1o, SL6o}	{SL4o, SL2i, SL5i, SL3o}
{PL1, SL1o, SL2o, SL3i}	{SL4io, SL5io}
	{SL2i, SL7o, SL3o}
	{SL4i, SL5o, SL7o}

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

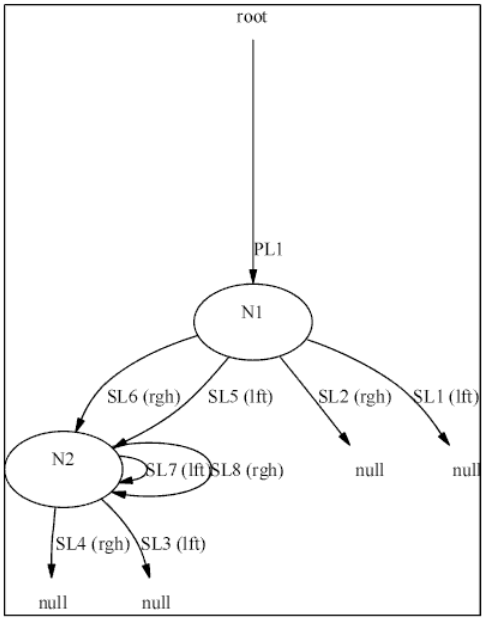
Test 4: binary tree

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.

```
1    root = malloc();
2    root->lft = NULL;
3    root->rgh = NULL;
4    while(){
5        p = root;
6        while(){
7            Force(p != NULL)
8            if(){
9                q = p -> lft;
10               p = q;
11               q = NULL;
             }else{
12               q = p -> rgh;
13               p = q;
14               q = NULL;
             }
         }
15       Force(p != NULL)
16       x = malloc();
17       x->lft = NULL;
18       x->rgh = NULL;
19       if(){
20           Force(p->lft = NULL)
21           p->lft = x;
         }else{
22           Force(p->rgh = NULL)
23           p->rgh = x;
         }
24       x = NULL;
     }
25   p = NULL;
```

Shapegraph



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}

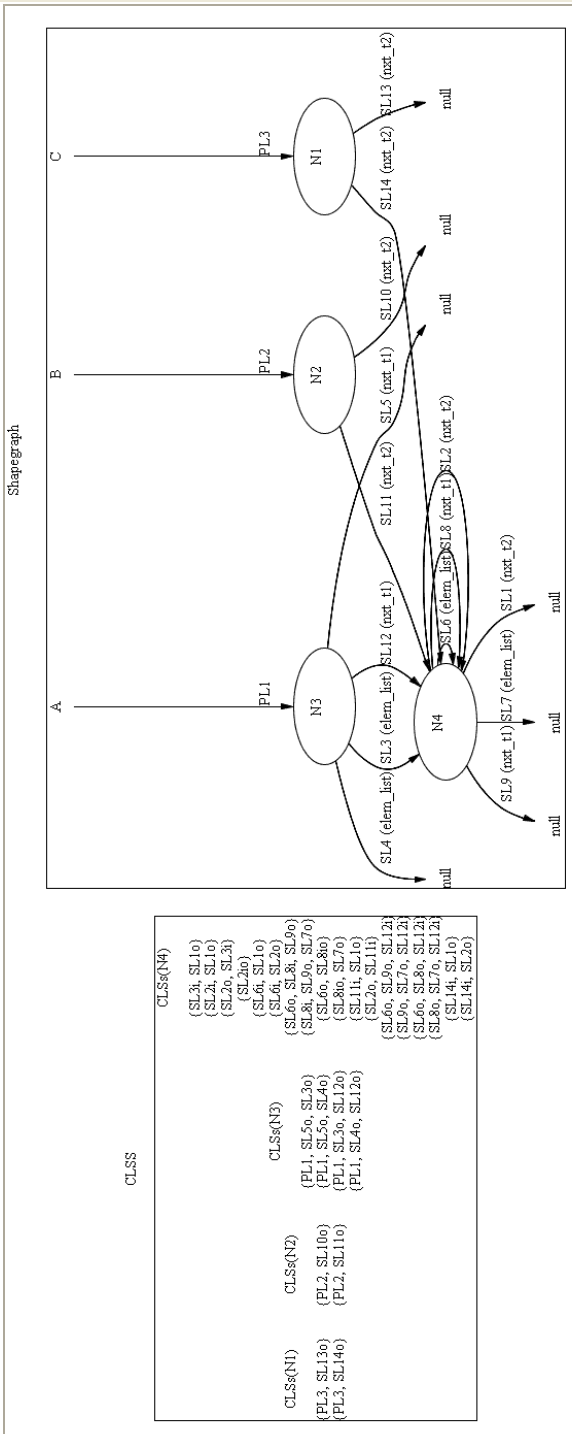
Test 5: Sparse matrix by sparse vector based on singly-linked lists

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();
```



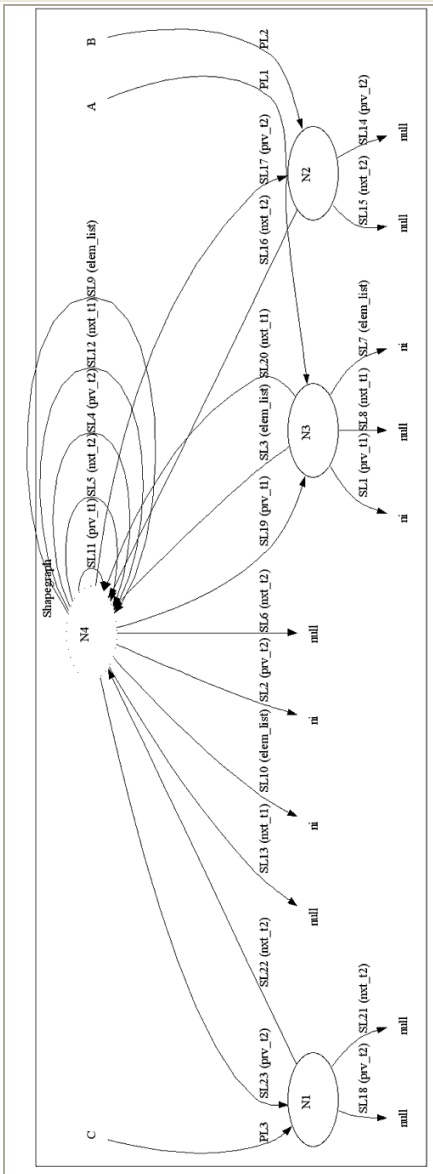
Test 6: Sparse matrix by sparse vector based on doubly-linked lists

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}
{SL4o, SL5io}
{SL2o, SL6o, SL9i}
{SL2o, SL4i, SL5o, SL9i}
{SL11o, SL12i, SL13o, SL9o}
{SL11o, SL10o, SL12i, SL13o}
{SL11io, SL12o, 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, SL6o}

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.

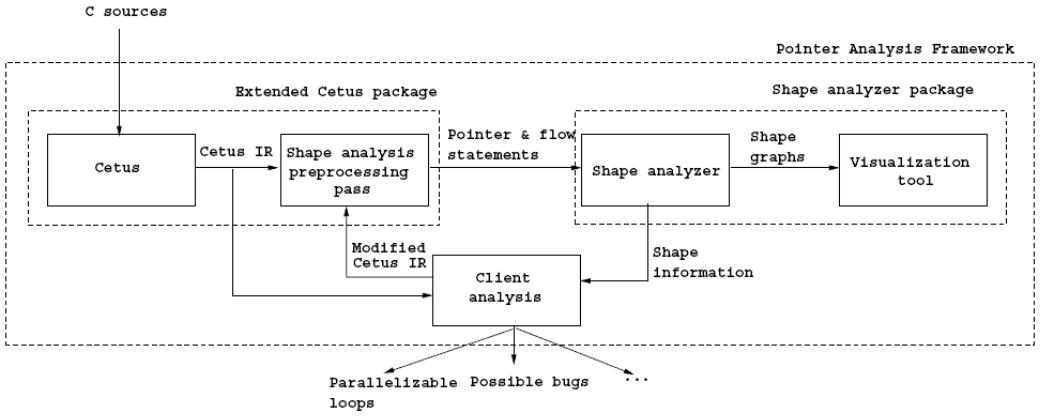
We have tried this approach on the matrix by vector examples. Let us revisit them now, having pruned all traversal statements that are not involved in the output vector creation (stmts. 51-64 and 74-76 for test 5, and stmts. 59-75 and 87-89 for test 6). The new values for the tests are shown in table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