A new Strategy for Shape Analysis based on Coexistent Links Sets *

A. Tineo[a], F. Corbera[a], A. Navarro[a], R. Asenjo[a], and E.L. Zapata[a]

[a]Dpt. of Computer Architecture, University of Málaga,
Complejo Tecnológico, Campus de Teatinos, E-29071. Málaga, Spain.
{tineo,corbera,angeles,asenjo,ezapata}@ac.uma.es

The analysis of dynamic heap-based data structures is difficult due to the alias problem. Shape analysis tries to gather information conservatively about these structures at compile time. In the context of parallelizing compilers, information about how memory locations are arranged in the heap at runtime is essential for data dependence analysis. With proper shape information we can reveal parallelism for heap-based structures, which are typically ignored by compilers. Existing shape analysis techniques face a dilemma: either they are too costly to be useful for real compilers or they are too imprecise to be useful for real programs. In this work, we present a new strategy for shape analysis based on a compact representation for the shape of data structures. This is done by using *Coexistent Links Sets* for nodes in a graph. The technique is simple to implement and very precise at the core level. Further precision-vs-cost balance can be tuned with the use of extensible properties.

1. Introduction

Static knowledge of memory references in a program is a must for compilers, if they are to provide optimizations related to parallelism in an automated basis. Such knowledge is not easy to gather due to the existence of aliases. Arrays, pointers and pointer-based dynamic data structures introduce aliases in programs. Parallelizing compilers have obtained a reasonable degree of success when dealing with array aliases and stack-directed pointers. However, heap-directed pointers and the structures they dereference are a whole different ground that still needs significant work.

The problem of characterizing dynamically allocated memory locations in a program can be approached in several ways. We believe that, in order to provide accurate information for real-life programs, some sort of abstraction in the form of a bounded graph must be performed. The kind of analysis that represents the heap as a storage shape graph is known as *shape analysis*. Its main goal is to capture the *shape* of memory configurations that are accessible through heap-directed pointers in a program.

Information about the shape of dynamic data structures is useful for parallelizing compiler transformations over the input program. Maybe the most obvious application is data dependence detection, needed for instruction scheduling, data-cache optimization, loop transformation and automatic vectorization and parallelization. Another interesting application comes from the use of the shape information for debugging analysis of the program. The shape abstraction can provide information about incorrect pointer usage that can lead to mistakes difficult to track.

We present in this work a new strategy for shape analysis based on what we call *Coexistent Links Sets* (CLSs). CLSs codify possibilities of connectivity between memory locations in a

*This work was supported in part by the Ministry of Education of Spain under contract TIC2003-06623.

neat and compact way. This is done by using graphs that represent the possible states of the memory configuration at a program point. Information is kept as a combination of possible reaching and leaving links over the memory locations. CLSs provide a rich description of the data structure with little storage requirements.

The goal of this paper is to present a new technique to achieve a shape abstraction of dynamic data structures. We think that it will provide more precision than existing techniques while at the same time keeping the storage and computation cost at a reasonable level. CLSs are the key instruments for the development of this technique. The remainder of this paper is organized as follows: Section 2 introduces the basics for our shape analysis technique; Section 3 describes CLSs in greater detail; Section 4 explains our criteria for summarization in graphs; Section 5 describes how the shape analyzer works by example; Section 6 comments some related work; and finally Section 7 concludes with the main contributions and ideas for future work.

2. Shape analysis basics

Our approach to shape analysis is based on graphs. A program dealing with dynamic data structures performs allocation of *memory locations* and establishes links between them. We represent memory locations in the program as nodes in graphs. Nodes can be referenced by pointer variables through *pointer links* (PLs). Additionally, *selector links* (SLs) are used to link nodes with other nodes.

The size of dynamic data structures is undecidable at compile time. Therefore, we must provide some mechanism to sum up the possible memory configurations in a finite, *bounded shape graph.* To achieve this, we can summon nodes that can actually represent several memory locations that are similar. We call this kind of nodes, *summary nodes* and the process of merging similar nodes, *summarization.* Very often, however, some of those locations are accessed later in the program and become so-called *singular* locations, which are somewhat different to other locations in the structure. It would be desirable to provide a mechanism to *invert* the summarization process, i.e., we would like to be able to *focus* over a singular node extracted out of a summary node. This can be achieved with the *materialization operation.* Depending on the case, we will be able to recover the information as we had it or instead, a conservative and less precise node will be materialized.

In our approach, there is a single graph associated to every statement in the program, which represents the possible memory configuration states at that point in the program. The shape analyzer works as an iterative data-flow analysis, by symbolically executing the statements in the source program, a process called *abstract interpretation.* For example, pointer statements receive an input graph (SG_{in}) and modify it to produce an output graph (SG_{out}). The rules for such transformation are determined by the statement *abstract semantics.*

Our technique only cares about statements that involve operations through pointers (pointer statements) and control flow decisions (loop and branch statements). Fig. 1 sketches out how the analysis operates in the presence of these kind of statements in a general, descriptive way.

In order to simplify the analysis, pointer statements are normalized to *simple* pointer statements, i.e., those that contain only simple access paths, or 1-level indirection. The simple pointer statements are (in C syntax):

```
ptr=NULL;         ptr=malloc(...);  ptr1=ptr2;
ptr1->sel=ptr2;   ptr1=ptr2->sel;   ptr->sel=NULL;
```

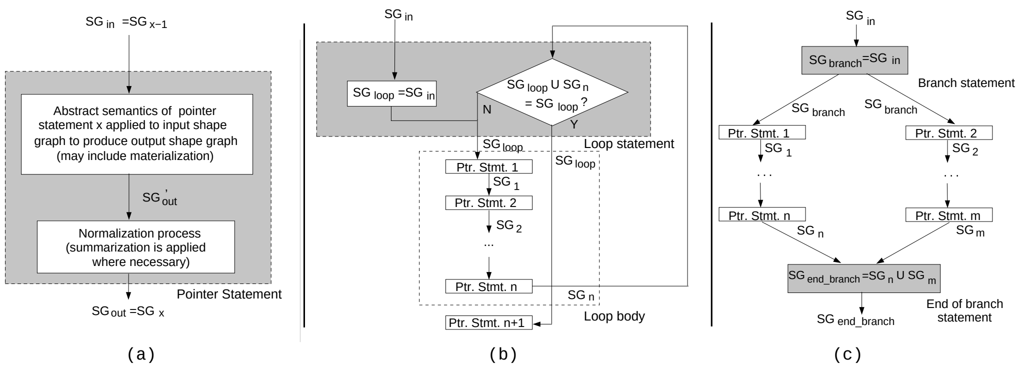


Figure 1. Analysis operation in presence of (a) pointer statements, (b) loop statements and (c) branch statements.

At loop bodies the analysis must iterate over the statements of the loop until the graphs for each statement change no more, i.e., until the analysis achieves a *fixed-point*. This way we ensure that the graph for each statement holds all possible memory configurations at that point of the program at runtime. When a fixed-point is reached for all statements in the program, the analysis terminates. Termination of the algorithm is assured by the summarization process that automatically occurs whenever nodes are similar. The graphs cannot grow out of control and eventually the graphs will be bounded and stationary.

At join points in the CFG, such as loops and branching statements, *incompatible* memory configurations occur. An operation to join graphs, (*graph union*) while conservatively keeping all possibilities is needed. For instance, fig. 2(b) shows the two possible memory configurations (MC1 and MC2) at the end of the program in fig. 2(a). In the next section we will see how these two MCs can be represented in one graph.

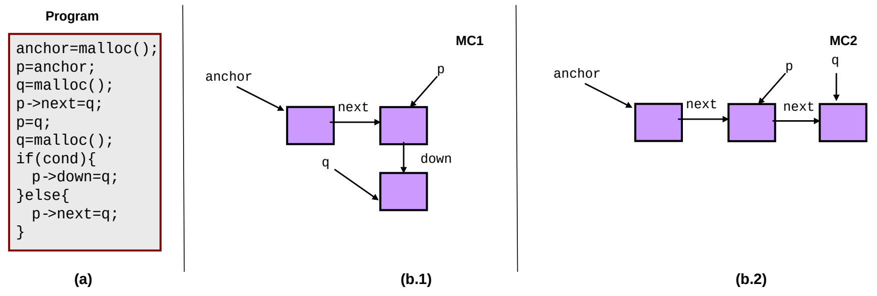

Program
```
anchor=malloc();
p=anchor;
q=malloc();
p->next=q;
p=q;
q=malloc();
if(cond){
  p->down=q;
}else{
  p->next=q;
}
```

Figure 2. There are two memory configurations for program (a), namely (b.1) and (b.2).

3. Coexistent Links Set (CLS)

Our main contribution to shape analysis in this paper is the introduction of *Coexistent Links Sets* (CLSs). To better help us describe what a CLS is, let us consider an example. Fig. 3 shows the graph that would represent the memory configurations shown in fig. 2.

Let us assume at this point that we need three nodes (n1, n2 and n3) for the graph that captures MC1 and MC2 from fig. 2. In Section 4 we describe in detail why this is so. We could expose a hierarchical view of the graph: the base elements are nodes, pointers and selectors; pointers and nodes can be combined to create pointer links (PLs); nodes and selectors can be combined to create selector links (SLs); finally pointer links and selector links can be combined to create coexistent links sets (CLSs).

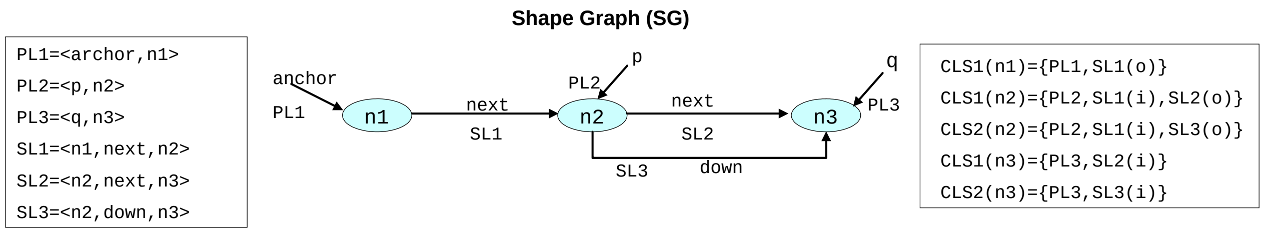


Figure 3. Shape graph for example of fig. 2. CLSs describe the shape abstraction.

CLSs allow us to express possibilities of connectivity between nodes, i.e., they describe the links that may reach and leave a node in the memory configuration abstraction. In the example of fig. 3, CLS1(`n1`) is telling us that `n1` supports `PL1=<anchor,n1>` and `SL1=<n1,next,n2>`. No other combination of links is possible for this node: `n1` can only be reached through the pointer `anchor` and connects undoubtedly to `n2` through its `next` selector. For `n2`, there are two possibilities. In any of them, `PL2=<p,n2>` reaches the node (i.e., `p` points to `n2` in any case) and the node is reached from `n1` through selector `next` (SL1). However, `n2` can follow to `n3` through the `next` selector or through the `down` selector, yielding two possibilities of connectivity for this node: CLS1(`n2`) and CLS2(`n2`). CLSs for `n3` work in a similar way: `n3` always supports `PL3=<q,n3>` and can be reached from `n2` by following either SL2 or SL3.

SLs are stand-alone entities in the graph. They represent links between nodes through selectors. However, when used in the context of a CLS, they are complemented with *attributes* to correctly describe the memory configurations occurring in the program. In particular, we consider two different attributes for SLs in CLSs: the *one-way* attribute and the *share* attribute.

The *one-way* attribute gives information about the origin and destination of a SL for the current node in a given CLS. It takes one of three values: 1) *incoming* `(i)`, when the SL is a *reaching* link for the current node; 2) *outgoing* `(o)`, when the SL is a *leaving* link for the current node; 3) and *cyclic* `(c)`, when the SL is at the same time a reaching and leaving link, so it is a *cyclic* link, like those of self-referenced memory locations.

The *share* attribute indicates when a node is reachable more than once through the same SL for a given CLS. It can take one of two values: true or false, i.e., a SL is either shared or not for a node in a given CLS. In our notation, a SL is not shared unless labelled with `(s)`.

The *one-way* attribute is really only necessary for self-SLs (selector links of the form `<ni,sel,ni>`). For non-self-SLs, the origin and destination are implied in the CLSs that make use of such SLs. However, from a formulation point of view, it is simpler to considerate this attribute for all SLs used in CLSs. The *share* attribute is necessary in every case to allow for better materialization.

4. Summarization criteria

In the previous example we asked the reader to assume that three nodes were needed for the shape abstraction representation. We now explain the summarization criteria for our shape analysis technique. This criteria determines to a great extent the behavior and precision of the analysis. First, we define a basic, fixed mechanism about what nodes should be kept separate and what nodes *could* be merged; second, we add configurable *properties* to fine-tune summarization decisions.

The basic criterion for summarizing nodes dictates that nodes that are pointed by the same (possibly empty) set of pointers are merged together, i.e., if P(`n1`)=P(`n2`) then `n2` merges into `n1`. Nodes that are pointed by pointers always remain singular nodes, i.e., they represent only one memory location for a given memory configuration. On the contrary, nodes that are not pointed by pointers will be merged (according to this basic criterion) into a unique summary node, that may be representing more than one actual memory location. This basic idea allows us to achieve precision on the *entry points* to the data structures, where it is more likely to be needed. This is the criterion applied in fig. 3.

However, merging all locations that are not directly accessed by pointers in a single node can be very imprecise as soon as the data structures are a little complex, and this is certainly the case for real programs. *Node properties* can be used to prevent *too much* summarization in such cases. A compatibility function must be defined for every property. If two nodes are *compatible* with respect to all available properties and conform to the basic criterion (share the same pointers), then they will be merged. Otherwise they will be kept separate. This way there could be several summary nodes, while in the case of using only the basic criterion, there would be just one summary node for the whole graph.

Properties are configurable, in the sense that they can be *turned on* and *off* at will, depending on the requirements of the analysis. It is clear to see that keeping a lot of properties will yield bigger graphs, with more nodes, and this will have an impact in the analysis cost. On the other hand, properties are absolutely needed to carry precision for analysis of complex structures.

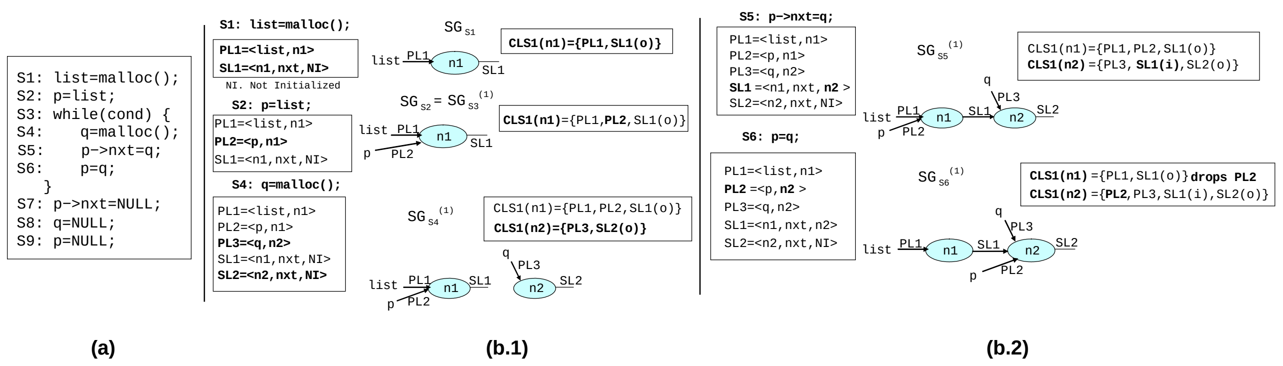
We could define here a large set of properties that could be used with our technique but instead we will just describe the *touch* property, which is a key instrument for data dependence analysis. While CLSs on their own capture *spatial* or *topological* information, the *touch* property is used to capture *temporal* information. During a typical structure traversal, locations are accessed for reading or writing. From a data dependence analysis point of view, it is of key importance to be able to discriminate, in the course of the traversal, between accessed (we say *touched*) locations in a structure and locations that have not been yet accessed (*untouched*). The touch property is annotated in nodes to avoid merging visited nodes with unvisited ones. Other properties can be effortlessly added, to add information about data type, structure connection, allocation site, etc., similarly to [2].

5. Shape analysis operations

In this section we illustrate how the shape analyzer works by example. First, we analyze a program that creates a single-linked list. It should be noted that the purpose of this section is not to explain in full detail how the analysis operates in each step during abstract interpretation of the source program, but rather focus on certain key aspects that are needed to understand the process.

In fig. 4(a) we present the program used to create the list. In fig. 4(b.1) and 4(b.2) we present the first steps of the algorithm. Input graph is empty. Abstract interpretation of statement `S1` produces the creation of node `n1` and the corresponding PLs, SLs and CLSs. Notice that the analysis can track uninitialized selectors. This is useful for code correctness checking. Upon entering a loop for the first time (`S3` in this example), input graph is preserved. Then, in `S4`, a new element is created and PLs, SLs and CLSs are updated accordingly. Changes with respect to previous graph appear in bold. Graphs for statements inside a loop are labelled with a superscript showing the number of the symbolic iteration.

Let us now consider an interesting situation that appears later in the analysis. Upon interpreting `S5` for the third time (third iteration), we obtain the shape graph shown in fig. 5 (a). At this point the list can be two (`n6` and `n3`), three (`n1`, `n4` and `n3`) or four elements (`n1`, `n2`,



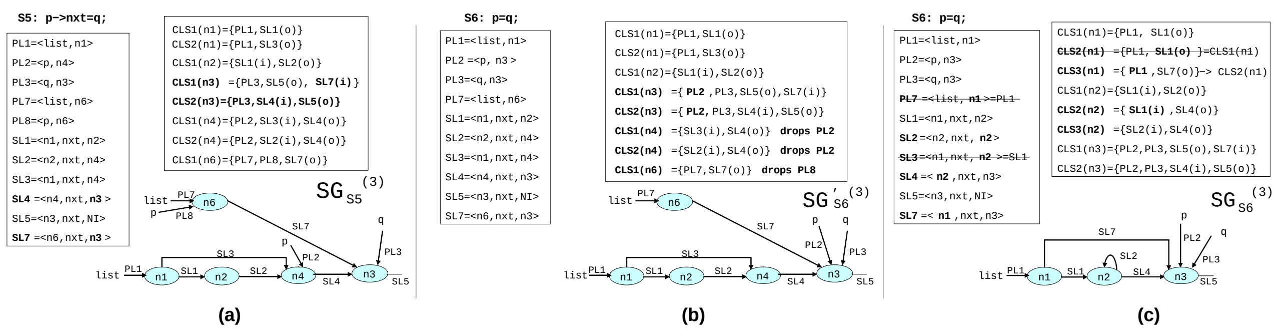
```
S1: list=malloc();
S2: p=list;
S3: while(cond) {
S4:     q=malloc();
S5:     p->nxt=q;
S6:     p=q;
    }
S7: p->nxt=NULL;
S8: q=NULL;
S9: p=NULL;
```

```
S1: list=malloc();
PL1=<list,n1>
SL1=<n1,nxt,NI>
NI. Not Initialized
CLS1(n1)={PL1,SL1(o)}

S2: p=list;
PL1=<list,n1>
PL2=<p,n1>
SL1=<n1,nxt,NI>
CLS1(n1)={PL1,PL2,SL1(o)}

S4: q=malloc();
PL1=<list,n1>
PL2=<p,n1>
PL3=<q,n2>
SL1=<n1,nxt,NI>
SL2=<n2,nxt,NI>
CLS1(n1)={PL1,PL2,SL1(o)}
CLS1(n2)={PL3,SL2(o)}

S5: p->nxt=q;
PL1=<list,n1>
PL2=<p,n1>
PL3=<q,n2>
SL1 =<n1,nxt, n2 >
SL2=<n2,nxt,NI>
CLS1(n1)={PL1,PL2,SL1(o)}
CLS1(n2) ={PL3, SL1(i),SL2(o)}

S6: p=q;
PL1=<list,n1>
PL2 =<p, n2 >
PL3=<q,n2>
SL1=<n1,nxt,n2>
SL2=<n2,nxt,NI>
CLS1(n1) ={PL1,SL1(o)}drops PL2
CLS1(n2) ={PL2,PL3,SL1(i),SL2(o)}
```

Figure 4. (a) Single-linked list creation program; (b.1) and (b.2) First steps of abstract interpretation

n4 and n3) long. Two nodes exist for the first element of the list to account for incompatible configurations.

After applying the abstract semantics of S6 in fig. 5(b), we find that nodes n1-n6 and n2-n4 now meet the summarization basic criterion, i.e., they are pointed by the same (possibly empty) set of pointers (P(n1)=P(n6)={list}, P(n2)=P(n4)={∅}), so they must be merged. For this example we do not consider properties for summarization. The process is completed in fig. 5(c), where the shape graph has been *normalized* to conform to the selected rules of summarization. The normalization process involves five steps: 1) find nodes that meet the current summarization criteria (n1-n6 and n2-n4); 2) substitute all references of the nodes to be merged by one of them in the lists of PLs and SLs (substitute n6 for n1 in PL7 and SL7 and substitute n4 for n2 in SL2, SL3 and SL4); 3) delete repeated PLs and SLs (eliminate PL7 that equals PL1 and SL3 that equals SL1); 4) update nodes, PLs, and SLs in the list of CLSs accordingly (substitute n6 for n1, n4 for n2, PL7 for PL1 and SL3 for SL1) and 5) eliminate duplicates in CLSs (CLS2(n1) that equals CLS1(n1)) and rename to avoid numbering gaps (CLS3(n1) becomes CLS2(n1)).

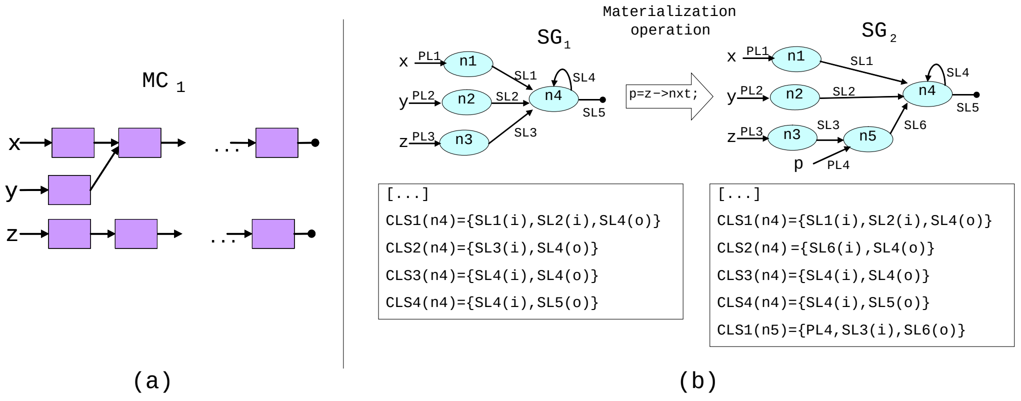


```
S5: p->nxt=q;
PL1=<list,n1>
PL2=<p,n4>
PL3=<q,n3>
PL7=<list,n6>
PL8=<p,n6>
SL1=<n1,nxt,n2>
SL2=<n2,nxt,n4>
SL3=<n1,nxt,n4>
SL4 =<n4,nxt,n3 >
SL5=<n3,nxt,NI>
SL7 =<n6,nxt,n3 >

CLS1(n1)={PL1,SL1(o)}
CLS2(n1)={PL1,SL3(o)}
CLS1(n2)={SL1(i),SL2(o)}
CLS1(n3) ={PL3,SL5(o), SL7(i)}
CLS2(n3)={PL3,SL4(i),SL5(o)}
CLS1(n4)={PL2,SL3(i),SL4(o)}
CLS2(n4)={PL2,SL2(i),SL4(o)}
CLS1(n6)={PL7,PL8,SL7(o)}

S6: p=q;
PL1=<list,n1>
PL2 =<p, n3 >
PL3=<q,n3>
PL7=<list,n6>
SL1=<n1,nxt,n2>
SL2=<n2,nxt,n4>
SL3=<n1,nxt,n4>
SL4=<n4,nxt,n3>
SL5=<n3,nxt,NI>
SL7=<n6,nxt,n3>

CLS1(n1)={PL1,SL1(o)}
CLS2(n1)={PL1,SL3(o)}
CLS1(n2)={SL1(i),SL2(o)}
CLS1(n3) ={ PL2 ,PL3,SL5(o),SL7(i)}
CLS2(n3) ={ PL2,PL3,SL4(i),SL5(o)}
CLS1(n4) ={SL3(i),SL4(o)}  drops PL2
CLS2(n4) ={SL2(i),SL4(o)}  drops PL2
CLS1(n6) ={PL7,SL7(o)}  drops PL8

S6: p=q;
PL1=<list,n1>
PL2=<p,n3>
PL3=<q,n3>
~~PL7 =<list, n1 >=PL1~~
SL1=<n1,nxt,n2>
SL2 =<n2,nxt, n2 >
~~SL3 =<n1,nxt, n2 >=SL1~~
SL4 =< n2 ,nxt,n3>
SL5=<n3,nxt,NI>
SL7 =< n1 ,nxt,n3>

CLS1(n1)={PL1, SL1(o)}
~~CLS2(n1) ={PL1, SL1(o) }=CLS1(n1)~~
CLS3(n1) ={ PL1 ,SL7(o)}-> CLS2(n1)
CLS1(n2)={SL1(i),SL2(o)}
CLS2(n2) ={ SL1(i) ,SL4(o)}
CLS3(n2) ={SL2(i),SL4(o)}
CLS1(n3)={PL2,PL3,SL5(o),SL7(i)}
CLS2(n3)={PL2,PL3,SL4(i),SL5(o)}
```

Figure 5. Node summarization operation. (a) Input, (b) working and (c) normalized graphs

Node summarization solves the issue of bounding the graphs. Additionally, the graph union operation is used to reconcile different graphs that reach a join point in the CFG. A typical case is the head of a loop, where the graph coming from the last statement in the loop merges with the graph from the previous iteration. Graph union is performed by adding all the information of both graphs into a *working* graph. This graph is then normalized as described in the summarization operation, so that redundancies are removed. The result is a graph that conservatively captures all possibilities for the memory configurations at that point in the analysis.

Then we need to decide wether to enter the loop again or exit it. This is determined by the fixed-point test. If the result of the last iteration, $SG^{(i)}_{loop}$, contains the same information than the previous iteration, $SG^{(i-1)}_{loop}$, then we have reached a fixed-point. In that case, we can leave the loop because we have conservatively registered all possible memory configurations that originate from it. Otherwise, we must keep iterating until the information of the graphs at the head of the loop are equivalent. This state is ensured by the existence of summarization. The graphs cannot grow endlessly nor can we enter a *deadlock* situation.



```
[...]
CLS1(n4)={SL1(i),SL2(i),SL4(o)}
CLS2(n4)={SL3(i),SL4(o)}
CLS3(n4)={SL4(i),SL4(o)}
CLS4(n4)={SL4(i),SL5(o)}
```

```
[...]
CLS1(n4)={SL1(i),SL2(i),SL4(o)}
CLS2(n4)={SL6(i),SL4(o)}
CLS3(n4)={SL4(i),SL4(o)}
CLS4(n4)={SL4(i),SL5(o)}
CLS1(n5)={PL4,SL3(i),SL6(o)}
```

Figure 6. (a) Memory configuration of shared and non-shared lists; (b) Materialization ex.

Node summarization and graph union are necessary for the analysis to work. However, they only provide ways to merge information. Conversely, the materialization operation *separates* information, which is the key to achieve precision in shape analysis, as shown in the following example. Fig. 6(a) depicts a memory configuration for three single-linked lists: lists `x` and `y` share memory locations from the second element onwards, while the `z` list is independent. The shape graph that matches this configuration appears in fig. 6(b). It represents all memory locations that are not directly accessed by stack pointers in a summary node (`n4`). CLSs for `n4` are also shown. They indicate possibilities of connectivity of `n4` with the rest of the nodes in the graph. In particular, CLS1(`n4`) is telling that, when `n4` is reached from `n1`, it is undoubtedly reached from `n2` as well, but not from `n3`. On the other hand, CLS2(`n4`) is telling that when `n4` is reached from `n3`, it is not reached from `n1` or `n2`. Therefore, CLSs are accurately capturing the memory configuration considered for this example. More importantly, since we have all the information about possible connections of links over `n4`, we are able to materialize a new node (`n5`) by traversing the `z` list (`p=z->nxt`), free of links from `n1` and `n2`. No other shape analysis that we know of would be able to discard those *false* links in the materialized node, given such a high level of *compression* for the summary node. This is achieved with very little storage requirements.

6. Related work

In the past decades pointer analysis has attracted a great deal of attention. A lot of studies have focused on stack-pointer analysis while others, more related to our work, have focused on heap-pointer analysis. Both fields require different techniques of analysis. In the context of heap analysis, some authors have proposed approaches that are based on encoding access-paths in path matrices [3] or limited path expressions [5]. Such approaches do not consider a graph representation of the heap. Other authors ([4], [1], [6], [7]) have considered shape abstraction expressed as graphs, just like us.

Some early shape analysis techniques started with coarse characterization of the shape of the data structures as a matching process with pre-defined shapes, namely tree, DAG (direct acyclic graph) or cycle, like in [3]. In the case of cyclic structures nearly all precision is lost. However, Hwang et al. [5] have achieved some success applying his shape analysis abstraction to dependence detection in programs with cyclic structures traversed in an acyclic fashion.

Sagiv et al. [7] present a graph-based shape analysis framework that sets the foundations for our approach to the shape analysis problem. Their use of abstract interpretation/abstract semantics, along with materialization, were taken in as the basics for the development of our first shape analysis framework [2]. Corbera et. al augmented the analysis precision by adding several graphs per statement and introducing the concept of properties in nodes to be able to have separate summary graphs. Later, a powerful loop-carried dependence test was developed over this framework to provide good results for real-life programs that deal with complex pointer-based structures [8]. We propose now a new shape analysis technique based on CLSs, with one graph per statement, that aims to surpass previous works.

7. Conclusions and future work

In this paper we have presented a new strategy for the static analysis of dynamically allocated data structures in pointer-based programs, like those easily found in C or C++. We have presented the key concept of Coexistent Links Sets, that codify in a neat and compact way possibilities of connectivity between memory locations. The sempiternal trade-off between precision and cost, that is so decisive in shape analysis, can be fine-tuned with the help of configurable and extensible properties. We are currently working in a compiler framework that conforms to the CLS description, in order to provide experimental results that can demonstrate the advantages of our approach.

References

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

[2] F. Corbera, R. Asenjo, and E.L. Zapata. A framework to capture dynamic data structures in pointer-based codes. *Transactions on Parallel and Distributed System*, 15(2):151–166, 2004.

[3] Rakesh Ghiya and Laurie J. Hendren. Is it a tree, a DAG, or a cyclic graph? A shape analysis for heap-directed pointers in C. In *Conference Record of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, St. Petersburg, Florida, January 1996.

[4] S. Horwitz, P. Pfeiffer, and T. Reps. Dependence analysis for pointer variables. *ACM SIGPLAN Notices*, 1989.

[5] Y. S. Hwang and J. Saltz. Identifying parallelism in programs with cyclic graphs. *Journal of Parallel and Distributed Computing*, 63(3):337–355, 2003.

[6] J. Plevyak, A. Chien, and V. Karamcheti. Analysis of dynamic structures for efficient parallel execution. In *Int'l Workshop on Languages and Compilers for Parallel Computing (LCPC'93)*, 1993.

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

[8] A. Tineo, F. Corbera, A. Navarro, R. Asenjo, and E.L. Zapata. A novel approach for detecting heap-based loop-carried dependences. In *The 2005 International Conference on Parallel Processing (ICCP'05)*, June 2005.