Shape Analysis for Dynamic Data Structures based on Coexistent Links Sets *

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

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

Abstract. 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 approaches 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 describe a new technique 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.

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

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 describe in this work a shape analysis algorithm based on *Coexistent Links Sets* (CLSs). CLSs codify possibilities of connectivity between memory locations in a 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.

A shape analyzer tool conforming to the CLS description is in the works. It is written in Java, a modern language where advanced software engineering techniques can be used to create a simple yet extensible tool. The focus on the development is set on simplicity and scalability, building over a solid core of basic operations so that further functionality can be easily added afterwards. Performance is also considered as a key aspect in the development of the tool. In that respect, we provide straightforward mechanisms to control performance behavior just by exchanging objects that implement certain interfaces.

The goal of this paper is to describe how our shape analysis technique achieves graph abstractions of dynamic data structures. We think that we can 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 offers some insight into key aspects of the development of the shape analyzer tool; Section 7 comments some related work; and finally Section 8 concludes with the main contributions and ideas for future work.

2 Shape analysis basics

A program dealing with dynamic data structures performs runtime allocation of memory pieces, that we call *memory locations*. Those locations are accessed through *pointers* and are linked together through *selectors*, creating *recursive data structures*, such as lists or trees. A *memory configuration* is the memory arrangement of the program at a given point during the its execution. Our approach to shape analysis is based on graphs. Memory configurations arising in the program are translated into graph abstractions.

Next, we introduce the elements that make up our shape graphs. They are depicted in a hierarchical view in fig. 1. At the lower level we have: *pointers*, used

as access points to the structures; *nodes*, used to represent memory locations; and *selectors*, used to join nodes. Combining these base elements together, we can create *pointer links*(PLs), which are links between pointers and nodes, and *selector links* (SLs), which are links between nodes through a selector. Finally, PLs and SLs can be combined together to form *Coexistent Links Sets* (CLSs), that describe combinations of links that may exist simultaneously over a node. CLSs will described in greater detail in Section 3.

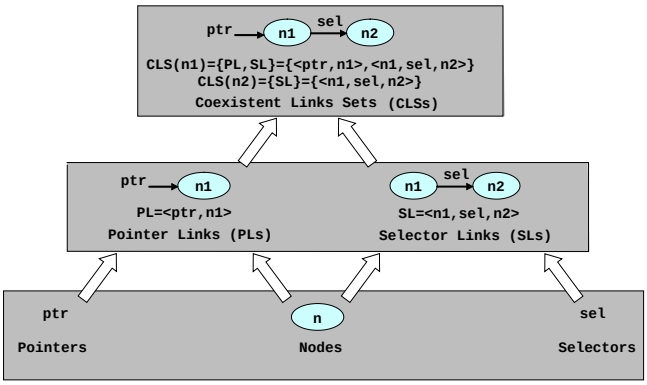

Fig. 1. Hierarchical view of shape graph elements. Elements from a lower level combine to form elements in the upper level.

During the course of a program execution, memory configurations change over time. Unfortunately, the number of memory configurations that a program can produce is usually infinite. In general, the size and *shape* of dynamic data structures are undecidable at compile time. Since shape analysis is conservative in nature, we must provide means to gather all possibilities of memory configurations in a compressed form.

We capture memory configurations arising in the program as finite, *bounded* shape graphs. In our approach, there is a single graph associated to every statement in the program, which represents all possible memory configuration states that may reach the statement at runtime. 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 *somehow* different (see Section 4) to every other location 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.

The shape analyzer works as an iterative data-flow analysis algorithm, 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 cares only about statements that involve operations through pointers (pointer statements) and control flow decisions (loop and branch statements). Fig. 2 sketches out how the analysis operates in the presence of these kind of statements in a general, descriptive way.

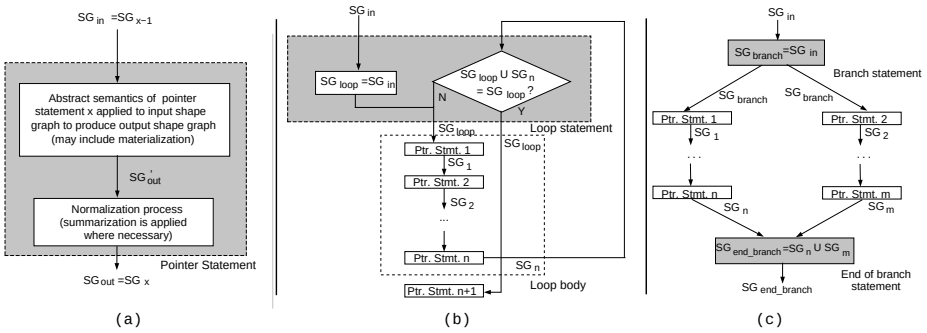

Fig. 2. Analysis operation in presence of (a) pointer statements, (b) loop statements and (c) branch statements.

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;
```

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.

At join points in the CFG, such as loops and branching statements, *incompatible* or mutually exclusive memory configurations occur. An operation to join graphs, (*graph union*) while conservatively keeping all possibilities is needed. For instance, fig. 3(b) shows the two possible memory configurations (`MC1` and `MC2`) at the end of the program in fig. 3(a). In the next section we will see how these two MCs can be represented in one graph.

To sum up, our shape analysis is an iterative data-flow analysis technique that represents memory locations in a program as nodes in a graph, where we can perform three basic operations: node summarization, node materialization and graph union.

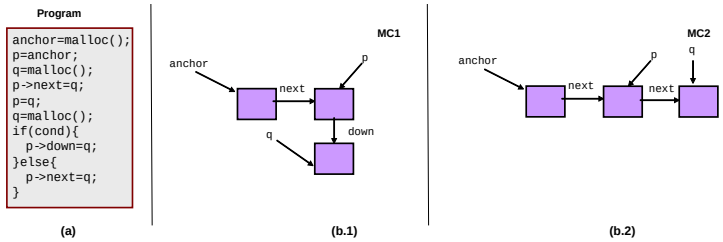

Fig. 3. There are two memory configurations for program (a), namely (b.1) and (b.2).

3 Coexistent Links Set (CLS)

Coexistent Links Sets (CLSs) are the key elements in our shape analysis technique. They capture the possible shapes of data structures by keeping combinations of links that may exist simultaneously over nodes in the graph. Let us consider an example to introduce the concept of CLS. Fig. 4 shows the graph that would represent the memory configurations shown in fig. 3. Let us assume at this point that we need three nodes (`n1`, `n2` and `n3`) for this graph. In Section 4 we describe in detail why this is so.

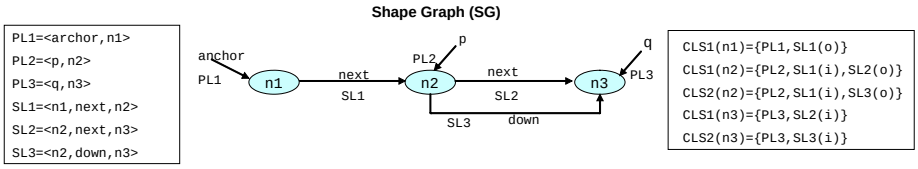

Fig. 4. Shape graph for example of fig. 3. CLSs describe the shape abstraction.

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. 4, 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 oc-

curring 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.

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)`.

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 [9], i.e., if P(`n1`) = P(`n2`) then `n2` merges into `n1`, where P(`n`) indicates the set of pointers over node `n`. 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. 4.

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. 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 [2], 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. Fig. 5 shows how

the *touch* property would affect node summarization in the case of a typical list traversal: the `p` pointer is used to traverse the structure and some accesses occur in every iteration of the traversing loop. At a point during the traversal, elements already visited have been accessed or *touched* in a certain way (`n1(t)`, `n2(t)` and `n3(t)`), while the elements that are yet to be visited (`n4`) are not accessed or *untouched*.

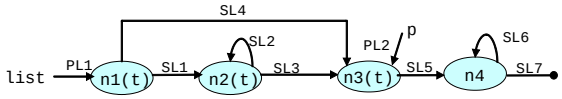

Fig. 5. The touch property applied to a typical list traversal.

The touch property is annotated in nodes to avoid merging visited nodes with unvisited ones. This way `n2(t)` and `n4` do not merge even though they are pointed by the same set of pointers (the empty set). From a data dependence analysis point of view, it is of key importance to be able to discriminate between accessed and not accessed locations in the course of loop traversals. Other properties can be effortlessly added, to include information about data type, structure connection, allocation site, etc.

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. This is a very simple case of a dynamic data structure creation but will suffice to explain the basics of *abstract interpretation*, *node summarization*, *graph union* and *fixed-point test* in the context of our analysis.

In fig. 6 we present the program used to create the list and the first steps of the analysis. Input graph is empty. Every statement appears next to the graph it produces as output, which in turn serves as input to the following statement. The abstract semantics of the statement determines how it modifies the graphs. A high-level overview of every pointer statement abstract semantics is shown in table 1.

Element creation statement `S1: list=malloc()` produces node `n1` and the corresponding PLs, SLs and CLSs. Notice that the analysis can track uninitialized selectors. This is useful for code correctness checking. Control flow statements, like `S3`, work differently. They do not enforce changes in graphs but control the statements that carry on the analysis. When entering a loop for the first time, input graph is passed unmodified to the body loop, as sketched in fig. 2. In following iterations, `S3` will control the termination condition through the fixed-point test.

The example continues entering the loop and statement `S4` creates a new element. Changes with respect to previous graph appear in bold. Graphs for

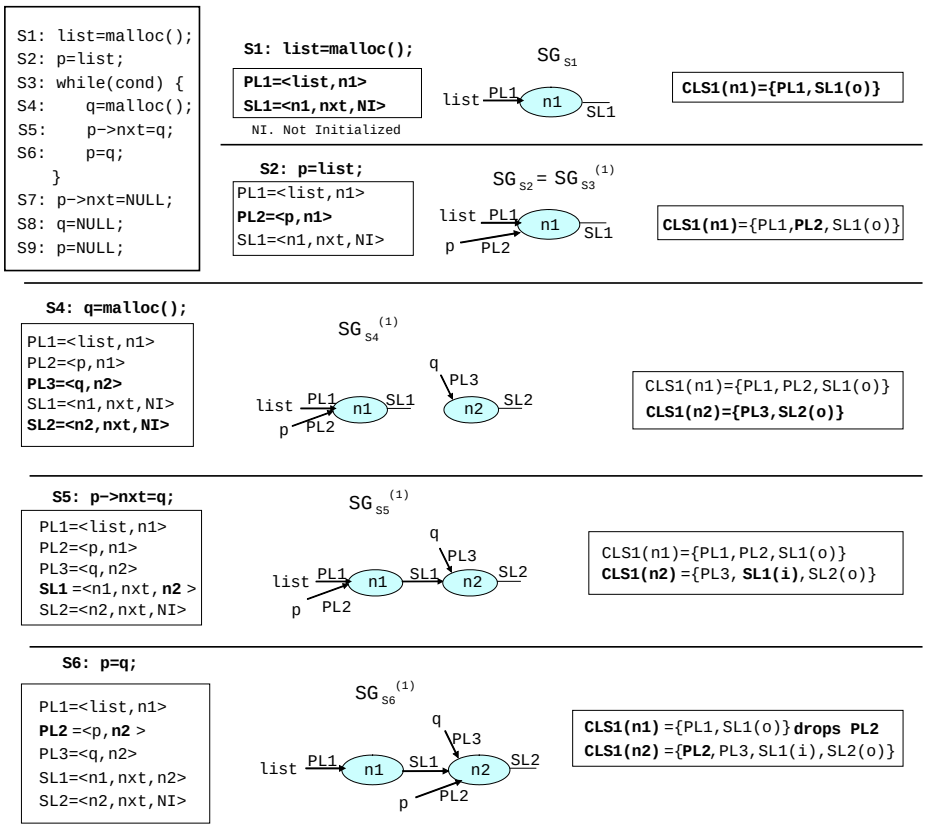

Fig. 6. Single-linked list creation program and first steps of abstract interpretation.

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. 7 (a). At this point the list can be two (`n6` and `n3`), three (`n1`, `n4` and `n3`) or four elements (`n1`, `n2`, `n4` and `n3`) long. Two nodes exist for the first element of the list to account for incompatible memory configurations.

Statement `S6: p=q` is a pointer alias statement, whose abstract semantics is described in table 1. As a previous step, this statement forces the execution of $\texttt{S6}_{prev}$`: p=NULL`, a pointer nullification statement. Steps 1 and 2 for $\texttt{S6}_{prev}$ (from table 1 again) have been applied in fig. 7(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`)={$\emptyset$}), so they must be merged (we do not consider properties for summarization in this example). This process is carried out by the *normalization* function, described in table 2. The graph is also cleaned from duplicated PLs, SLs and CLSs. The result is shown in fig. 7(c). Finally, the process triggered by `S6` ends by pointing

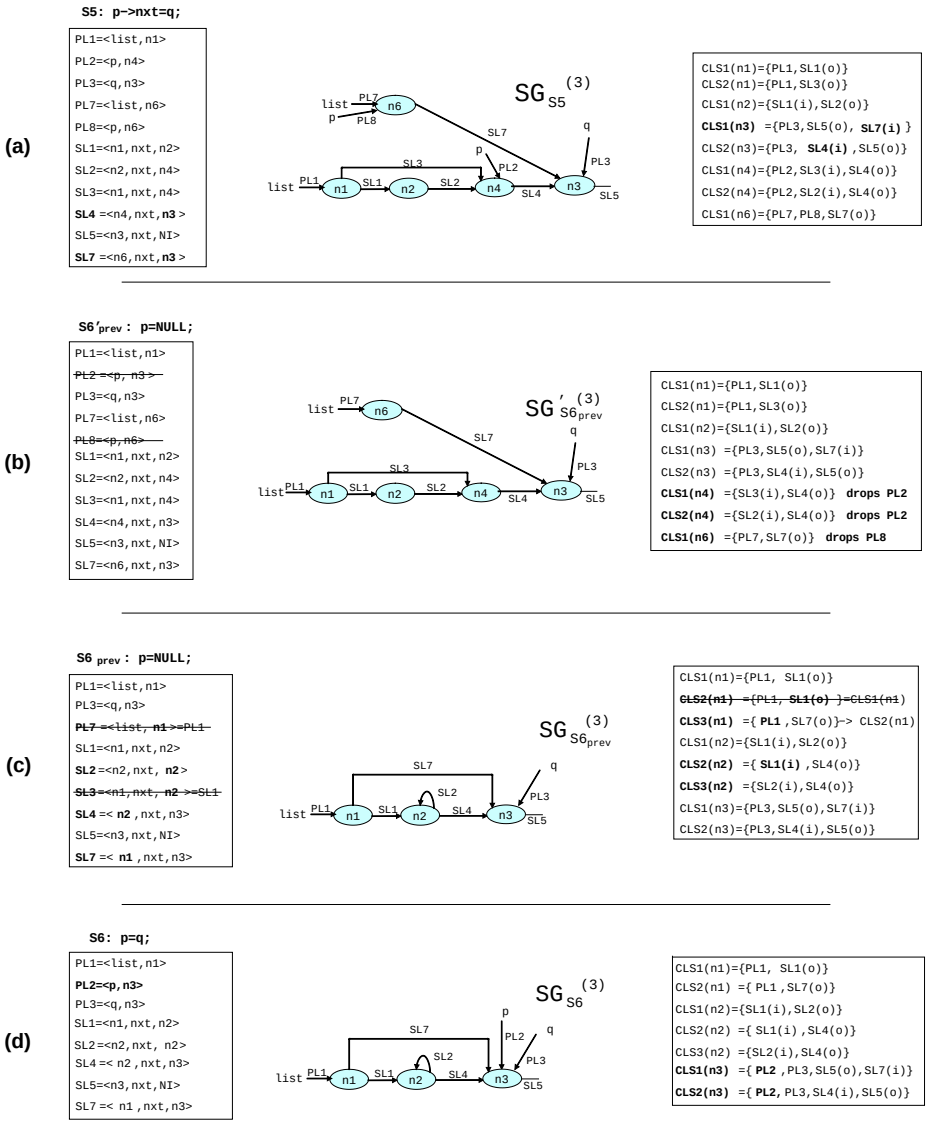

Fig. 7. Node summarization. (a) Input graph, (b) nullification, (c) normalization, and (d) pointer aliasing.

p to the node pointed by q, as shown in fig. 7(d). A new PL (PL2) is added and CLSs(n3) are updated accordingly.

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. The graph union operation is described in table 2. It adds all the information of both graphs into a *working* graph. This graph is then normalized, so that compatible nodes are summarized and redundancies are removed. The result is a graph that conservatively captures all possibilities for the memory configurations until 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, $SG_{loop}^{(i)}$, contains the same information than the previous iteration, $SG_{loop}^{(i-1)}$, 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.

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. 8(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. 8(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.

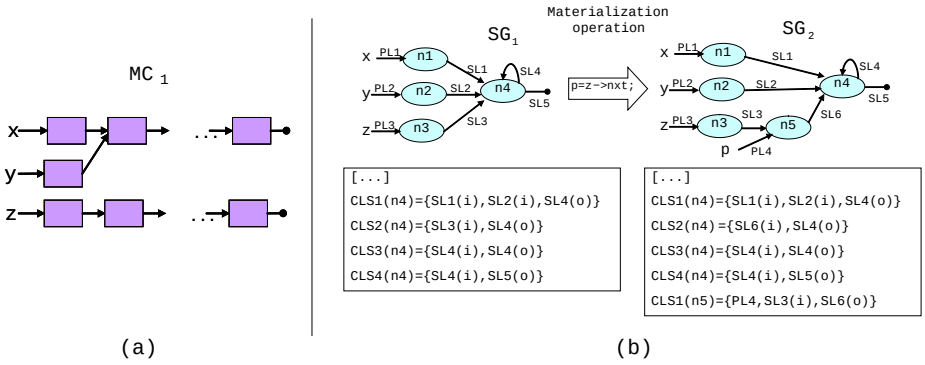

Fig. 8. (a) Memory configuration of shared and non-shared lists; (b) Materialization example.

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`) with the structure traversing statement `p=z->nxt`, whose abstract semantics is described in table 1. This new node is 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.

`ptr = malloc(...);` Element creation statement. 1. Create new node `n1` 2. Create `PL=<ptr,n1>` 3. Create SLs from `n1` to the non initialized node (NI): `SL1=<n1,sel1,NI>`, `SL2=<n1,sel2,NI>`, ... 4. Create `CLS(n1)`={`PL,SL1,SL2,...`}	`ptr = NULL;` Pointer nullification statement. 1. Remove every `PL=<ptr,ni>`, where `ni` is any node 2. Update `CLSs(ni)` dropping removed PLs 3. Normalize graph*
`ptr1->sel = ptr2;` Element linking statement. 1. Split graph by `ptr1->sel`* and apply steps 2-6 to all resulting graphs 2. Materialize new node `nm` by `ptr1->sel`* 3. Remove `SL=<n1,sel,nm>`, where `n1` is the node pointed by `ptr1`, and update `CLSs(n1)` and `CLSs(nm)` accordingly 4. Add `SL=<n1,sel,n2>`, for all nodes `n2` pointed by `ptr2` 5. Update `CLSs(n1)` and `CLSs(n2)` adding new SLs 6. Normalize graph* 7. Join resulting graphs*	`ptr->sel = NULL;` Selector nullification statement. 1. Split graph by `ptr->sel`* and apply steps 2-5 to all resulting graphs 2. Materialize new node `nm` by `ptr->sel`* 3. Remove `SL=<n1,sel,nm>`, where `n1` is the node pointed by `ptr` and update `CLSs(n1)` and `CLSs(nm)` accordingly 4. Add `SL=<n1,sel,NULL>` and update `CLSs(n1)` accordingly 5. Normalize graph* 6. Join resulting graphs*
`ptr1 = ptr2->sel;` Structure traversing statement. 1. Apply `ptr1 = NULL;` 2. Split graph by `ptr2->sel`*. Apply steps 2-6 to all resulting graphs 3. Materialize new node `nm` by `ptr2->sel`* 4. Add `PL=<ptr1,nm>` 5. Update `CLSs(nm)` to include new PL 6. Normalize graph* 7. Join resulting graphs*	`ptr1 = ptr2;` Pointer alias statement. 1. Apply `ptr1 = NULL;` 2. For every `PL=<ptr2,ni>` (where `ni` is any node), create `PL=<ptr1,ni>` 3. Update `CLSs(ni)` adding new PLs

Table 1. Overview of abstract semantics for pointer statements. Refer to table 2 for description of functions marked with *.

Normalize graph. Summarize compatible nodes and remove redundant information.

1. Find compatible nodes and merge them, calculating merged values for properties
2. Remove duplicated PLs and SLs
3. Update CLSs exchanging nodes, PLs and SLs
4. Remove duplicated CLSs

Join graphs. Add information from input graphs `SG1` and `SG` into the working graph `SG3`.

1. Add all PLs from `SG1` and `SG2` to `SG3`
2. Add all SLs from `SG1` and `SG2` to `SG3`
3. Add all CLSs from `SG1` and `SG2` to `SG3`
4. Normalize `SG3` and return it to calling function

Split graph by `ptr->sel`. Split a graph into several graphs so that each of them points to just one node by following `ptr->sel`.

1. Apply steps 2-6 to all nodes `n1` pointed by `ptr`
2. Apply steps 3-6 to all nodes `n2` belonging to a `SL=<n1,sel,n2>`
3. Create a copy of the input graph
4. Remove `CLSs(n1)` containing `PL=<ptr2,n1>`, where `ptr2!=ptr1`
5. Remove `CLSs(n1)` and `CLSs(n2)` containing `SL=<n1,sel,n3>`, where `n3!=n2`
6. Remove unused elements
7. Return all generated graphs

Materialize by `ptr->sel`. Creates a new node `nm` out of the summary node reached through `ptr->sel`.

1. Find node `n1` pointed by `ptr` and node `n2` found in `SL=<n1,sel,n2>`
2. Create node `nm` as a copy of `n2`, with the same properties (if any)
3. Make `nm` inherit all PLs, SLs and CLSs from `n2`
4. Remove `SL=<n1,sel,n2>` and drop it from `CLSs(n1)` and `CLSs(n2)`
5. Add `SL=<n1,sel,nm>` to `CLSs(n1)` and `CLSs(nm)`
6. Make `nm` a singular node (only represents one memory location)
7. Remove unused SLs and CLSs until graph is consistent

Table 2. Overview of basic functions for the shape analyzer.

6 Tool design

At the time of writing this paper, we are finishing the first working version of the shape analyzer. Still much work is needed in order to make it a full application. Ultimately, this tool will be able to take any pointer-based C program as input and produce the shape graphs that capture memory configurations that occur at runtime.

We believe the technique described in this article is fairly simple, at least for a graph-based shape analysis technique. The development of the tool revolves around a *core* that includes all the basic graph operations and abstract semantics of statements. Part of this core functionality is roughly described in tables 1 and 2. The core is basically completed at 3000 lines of Java code.

Java is the language of choice for our implementation of the CLS-based shape analyzer. The object oriented approach seems tailored for the development of the tool. Development in Java is easy, due to the existence of powerful and useful GUI's, debugging aids, the Java foundation classes, javadoc, etc. We are actively using all new advantages of Java 1.5 for the development of the tool, such as

generics or the for-each loop. Our intention is to produce a tight, clear and well-organized code that can be easily extended from the core for the development of future client applications, such as data dependence tests. Java provides the means for this kind of development.

The use of standard Java classes is a big help for the development of the tool. However, one could argue that the object abstraction penalty could eventually degradate the performance of the shape analyzer. It is in the philosophy of this technique to be able to adjust to real world applications, so bigger and more complex codes can be analyzed. In order to mitigate possible performance degradations caused by the use of standard Java classes, the whole tool has been built relying on abstract collections for the elements of the graphs (nodes, PLs, SLs, CLSs, properties,...). This way we can use familiar collection classes such as `ArrayList` or `HashMap`, but if we need to, we can easily switch to other custom-made classes that offer the same interface but better performance for certain key operations.

An interesting feature about properties is that they do not belong to the core of the method. The shape analyzer can work without properties, using just the basic criterion for summarization decisions. This is an advantage for the development of the shape analyzer tool, because we can focus on basic graph operations providing just general support for properties. At the same time, it allows for easy inclusion of new properties at a later stage. This is specially useful because at this point we might not be fully aware of requirements for specific programs in terms of shape graph abstractions. The analysis of new codes will surely suggest new properties. The design of the shape analyzer allows for the straightforward inclusion of properties over the core functionality.

In the description of the technique, we have stated that there is only one graph per statement. From an storage point of view, we can shrink even more: we can keep just one graph flowing for the whole program, except for loop statements that need to keep the latest result for fixed-point comparisons. Conversely, we can easily adjust the tool to keep all graphs generated, so we can easily *freeze* memory configurations at points of interest. This is useful for general program analysis, as a learning or debugging tool for example.

A side issue to consider is the front-end needed to translate input C programs into the internal representation needed by the shape analyzer. In this process, we filter the program to keep only pointer and control flow statements and simplify complex pointer expresions, among other tasks. For this purpose, we use and extend *Cetus* [7], [6]. Cetus is a compiler infrastructure specially aimed towards the development of compilation passes of high-level nature. It can parse C and C++ input programs, and soon Java will be supported as well. It is written in Java and its source code is publicly available under a non-restrictive license. Creating our own translation pass over Cetus, we can translate input C programs into our intermediate representation in an automated basis, which brings us closer to the final goal of automatically analyze pointer based applications.

7 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], [8], [9]) 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. [9] 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 [10]. We propose now a new shape analysis technique based on CLSs, with one graph per statement, that aims to surpass previous works.

8 Conclusions and future work

In this paper we have described a technique for the static analysis of dynamically allocated data structures in pointer-based programs, like those easily found in C or C++. This technique is based on the key concept of Coexistent Links Sets, that codify in a neat and compact way possibilities of connectivity between memory locations.

A working implementation of the shape analyzer is nearly completed with focus on simplicity and scalability. 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. Underlying data structures can be easily exchanged if performance bottlenecks appear.

In the near future, we plan to analyze some benchmark programs to provide experimental results for the shape analyzer. Soon after that, we plan to use this tool as the base for some client analysis, such as data dependence detection. Further down the road, we expect to tackle the issue of automatic parallelization of pointer-based programs, by using information from client analysis for thread creation modules. This way we will have completed an automatic end-to-end

pointer analysis framework with parallel code generation for programs that rely heavily on dynamic data structures.

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. Troy A. Johnson, Sang-Ik Lee, Long Fei, Ayon Basumallik, Gautam Upadhyaya, Rudolf Eigenmann, and Samuel P. Midkiff. Experiences in using Cetus for source-to-source transformations. In *The 17th International Workshop on Languages and Compilers for Parallel Computing (LCPC '04)*, West Lafayette, Indiana, USA, September 2004.
7. Sang-Ik Lee, Troy A. Johnson, and Rudolf Eigenmann. Cetus - an extensible compiler infrastructure for source-to-source transformation. In *The 16th International Workshop on Languages and Compilers for Parallel Computing (LCPC '03)*, pages 539–553, College Station, Texas, USA, October 2003.
8. 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.
9. 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.
10. 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.