Parallelizing irregular C codes assisted by interprocedural shape analysis *

R. Asenjo R. Castillo F. Corbera A. Navarro A. Tineo
E.L. Zapata
Dpt. of Computer Architecture, University of Málaga,
Complejo Tecnologico, Campus de Teatinos, E-29071. Málaga, Spain.
{asenjo,rosa,corbera,angeles,tineo,ezapata}@ac.uma.es

Abstract

In the new multicore architecture arena, the problem of improving the performance of a code is more in the software side than in the hardware one. However, optimizing irregular dynamic data structure based codes for such architectures is not easy, either by hand or compiler assisted. Regarding this last approach, shape analysis is a static technique that achieves abstraction of dynamic memory and can help to disambiguate, quite accurately, memory references in programs that create and traverse recursive data structures. This kind of analysis has promising applicability for accurate data dependence tests in loops or recursive functions that traverse dynamic data structures. However, support for interprocedural programs in shape analysis is still a challenge, especially in the presence of recursive functions. In this work we present a novel fully context-sensitive interprocedural shape analysis algorithm that supports recursion and can be used to uncover parallelism. Our approach is based on three key ideas: i) intraprocedural support based on "Coexistent Links Sets" to precisely describe the memory configurations during the abstract interpretation of the C code; ii) interprocedural support based on "Recursive Flow Links" to trace the state of pointers in previous calls; and iii) annotations of the read/written heap locations during the program analysis. We present preliminary experiments that reveal that our technique compares favorably with related work, and obtains precise memory abstractions in a variety of recursive programs that create and manipulate dynamic data structures. We have also implemented a data dependence test over our interprocedural shape analysis. With this test we have obtained promising results, automatically detecting parallelism in three C codes, which have been successfully parallelized.

*This work was supported in part by the Ministry of Education of Spain under contract TIN2006-01078 and by the Project HPC-EUROPA(RII3-CT-2003-506079), with the support of the European Community - Research Infrastructure Action under the FP6 *"Structuring the European Research Area"* Programme.

1. Motivation

The problem we want to solve is the automatic parallelization of codes based on heap-stored dynamic/recursive data structures. This is a very tough and still unsolved problem. Finding its solution would have a big impact since dynamic data structures are widely used in many irregular codes and multiprocessor/multicore/multithread architectures are very common nowadays.

A basic step in the automatic parallelization process is the detection of parallel loops or parallel function calls using a data dependence test. When dealing with codes based on dynamic data structures, the data dependence test needs information about the properties of the data structures traversed in the loops or in function bodies. This leads us to shape analysis techniques which are able to capture, at compile time, dynamically allocated data structures, i.e., those allocated at runtime and accessed through heap-directed pointers. Actually, we have implemented a loop-carried data dependence test [15] based on a previously developed shape analysis algorithm [4]. The main idea in this test is to carry out the abstract interpretation of the statements of the analyzed loop, abstracting the accessed heap locations with nodes of shape graphs and annotating these nodes with read/write information.

Let us illustrate the main idea behind our dependence test. The code in fig. 1 creates a singly-linked list and then traverses it, writing in a list-element the data field of the previous one. Our test symbolically executes the code abstracting the data structures by shape graphs, sg. In the example, $sg1$ is the abstraction of the list created at statement `S1`. Using abstract interpretation, the abstract semantics of each statement updates the sg of the previous statement. Since there is a potential loop-carried data dependence, LCD, between `S3` and `S4`, the abstract semantics of these statements will annotate which memory locations are read and/or written. In this example, in $sg5$ at the second symbolic execution of statement `S2`, we detect that a given location has been written in an iteration and read in the next one. Thus,

a RAW LCD is detected. More details can be found in [15].

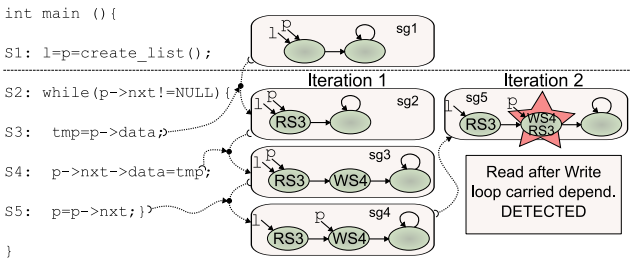

Figure 1. Data dependence test simple example.

Such a test requires a precise description of the heap locations and its interconnection topology to successfully detect parallel loops. Till now, the most accurate way of characterizing dynamic data structures consider shape abstractions expressed as graphs to model the heap. For this reason, we adopted this approach developing a store-based shape analysis based on "Reference Shape Graphs", RSG [4] to be the kernel of our test.

This RSG shape analysis improved previous works, such as [16] or [18], in terms of precision and analysis time. However it had a main drawback: the lack of interprocedural support forced us to do inlining and to manually translate recursive calls into loops. Interprocedural support in shape analysis is a challenge, especially in the presence of recursive functions. Some recent works ([6], [13], [1], [8] and [17]) have faced this issue, albeit they are targeted towards program verification, where the goal of the analysis is to infer a shape invariant of a data structure, and to prove it is preserved at the end of any possible run of a function/program. On the contrary, our goal is to capture temporal relationships between memory accesses, using this information for dependence analysis and subsequent automatic parallelization. The present work builds upon previous experience to provide a brand new shape analysis technique with support for recursion, but directed towards the development of a data dependence test.

In order to improve the precision and performance of the shape analysis and because our previous RSG approach could not be easily extended to deal with interprocedural support, first, the base intraprocedural shape analysis technique has been completely redesigned: based on the key new idea of *coexistent links sets*, we explicitly encode the possible links that relate a heap element to its neighbors (section 2). Then, we have extended upon the base design to add interprocedural support to the analysis (section 3). The key point is that we have extracted the information provided by the *Activation Record Stack* (ARS), which explicitly holds the information of interprocedural nature during a program execution, in a way that is easily abstracted and manipulated in the shape graph domain. The strategy to achieve this involves 2 main aspects. Firstly, the shape graph representation is augmented with a new kind of link, the *recursive flow links*. They are used to leave a *trace* so that the state of pointers in the context for previous calls can be recovered. Secondly, we define some *context change rules* that dictate how the shape abstractions are transformed when entering to a function or returning to a call site. All this means that we can still use our base (intraprocedural) shape analysis with minor extensions to also deal with recursive codes. All our algorithms have been implemented in Java and integrated into the Cetus compiler infrastructure [11].

1.1. Contributions

In this paper, we contribute with a newly designed, fully context-sensitive interprocedural shape analysis technique that supports recursion. We have implemented a prototype of our technique and we have run some experiments (section 4). We have compared the results and performance of our algorithm with those of a landmark related work [17] (section 5 discusses this and other related works). Besides, we have implemented a data dependence test based on this new shape analysis and the preliminary experiments with this test are encouraging. The main parallel loops automatically detected by our data dependence test have been manually parallelized obtaining good speed-ups. This is, the preliminary results show that our algorithm correctly and accurately captures dynamic data structures, that the data dependence test is able to identify the parallel loops that traverse these data structures, and that the parallel execution of these loops reports good speed-ups.

2. Intraprocedural shape analysis

Our approach to shape analysis is based on constructing bounded *shape graphs*, that capture the way data structures are arranged in the heap. The analysis works by symbolically executing statements in the analyzed program, a process called *abstract interpretation*, until a *fixed-point* is reached. Termination of the analysis is guaranteed by the existence of *summarization mechanisms*. These mechanisms are responsible for bounding the abstractions and preventing them from changing endlessly.

Shape graphs are updated according to the *abstract semantics* of each statement, which describes the effect of the statement over a graph. The result of our analysis is the set of shape graphs that describe the state of the heap for every statement and by following any possible flow path in the program. These results are always conservative, meaning that a super-set for all possible shape graphs that represent

the program heap, is constructed. All this can be illustrated by the example of fig. 2, where we can see how the statements of the code which builds a single linked list are symbolically executed until a fixed point is reached. Next, we describe the main characteristics of our shape abstractions.

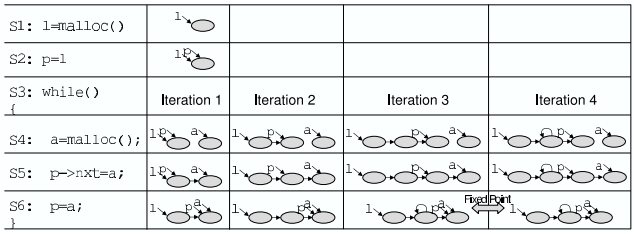

Figure 2. Building a RSG for each statement of an example code.

2.1. The base shape abstractions. Coexistent Links Sets (CLS)

The heap information present at runtime belongs to the *concrete domain* and is described as *memory configurations*, mc. That information is abstracted for its analysis in the *abstract domain*, in the form of *shape graphs*, sg.

A memory configuration is described with several sets of elements. First we have the set of *memory locations* $l \in L$, i.e., the pieces of memory dynamically allocated in the program. *Concrete pointer links*, $plc = \langle x, l \rangle \in PLc$ are the links that are established between pointers, $x \in PTR$, and memory locations, l, while *concrete selector links*, $slc = \langle l_1, sel, l_2 \rangle \in SLc$, establish a path from different memory locations, l_1 and l_2, through pointer fields, also called selectors ($sel \in SEL$). The upper side of fig. 3 shows a singly-linked list of 4 elements in the concrete domain.

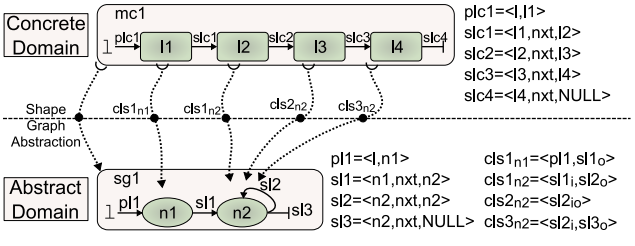

Figure 3. A singly-linked list in both domains.

A shape graph, sg, is defined analogously to a memory configuration. In this case, however, the base element is a node, n, which represents one or several memory locations. The set of all the nodes, N, includes an special NULL node. In a shape graph, the number of nodes is bounded by the *summarization* policy. The simplest summarization policy states that locations that are not pointed to by pointers are abstracted by a single summary node. For instance, in fig. 2, Iteration 3, we can see that the abstract interpretation of statement S6 leads to the summarization of the two middle nodes of the graph resulting from S5. The PL and SL sets are now the *abstract pointer links* set and *abstract selector links* set respectively, with the same information of PLc and SLc but changing locations, l, for nodes, n. Finally, we include the set of pointer links and selector links that enter or leave a memory location represented by a certain node. We call this set, the *coexistent links set* for node n, cls_n, as it expresses the links that may exist *simultaneously* over a memory location contained in n.

All this will be better understood by having a look at the abstract representation of the singly-linked list in fig. 3. It shows the shape graph abstraction, $sg1$, for the memory configuration, $mc1$. Node $n1$ abstracts memory location $l1$, which is pointed to by pointer `l`. Memory locations $l2$, $l3$ and $l4$ are not pointed to by pointers and thus may be summarized by a single node, $n2$. Selector links in the graph domain are updated to match their nodes and therefore may also summarize actual selector links in the concrete domain. Despite this reduction in the number of matching elements, the shape graph $sg1$ contains, within the coexistent links sets, *all* the information present in the memory configuration, $mc1$ (although, as a result of being a conservative abstraction, $sg1$ represents a list of 4 or more list elements). Note that there may be more than one coexistent links set cls_n for a node $n \in N$ (e.g., $cls1_{n2}$, $cls2_{n2}$ and $cls3_{n2}$). Since a node can represent several memory locations, its coexistent links sets must contain all the possibilities of links existing in those memory locations. In this example, $cls1_{n2}$ indicates the links for a node that is reached through $sl1$ (the '`i`' indicates that $sl1$ is an *incoming* link for the location represented by $n2$). From there, you can reach another node by following the $sl2$ selector (the '`o`' indicates it is an *outgoing* link), i.e., $cls1_{n2}$ is capturing the links for the $l2$ location in the concrete domain. Likewise, $cls2_{n2}$ indicates that $n2$ can also be reached through $sl2$ (incoming), and leave to another location through $sl2$ (outgoing), which

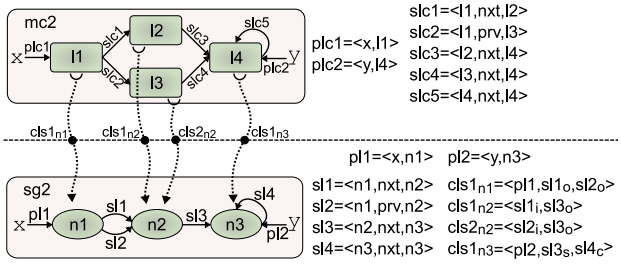

Figure 4. Shared and cyclic in node n3.

corresponds to the links for $l3$. Finally, $cls3_{n2}$ is indicating the links for $l4$.

As we have seen in this previous example, the expressiveness of the selector links is improved thanks to the set of attributes, $ATT = \{i, o, c, s\}$. Each element $att \in ATT$ codifies information about the direction and nature of a selector link when it is related to a node. Intuitively, $att = i$ stands for an input link, $att = o$ for an output link, $att = c$ for a cyclic link, and $att = s$ for a shared one. From the set ATT we define a new domain $ATTSL = \mathcal{P}(ATT)$, where each element of this new domain $attsl \in ATTSL$ represents a possible combination of attributes that describe the characteristics of a selector link when it is associated to a node. Besides, from $ATTSL$ and the set of all selector links, SL, we define the domain $SL_{att} = SL \times ATTSL$. An element sl_{att} in this domain, which we call a *selector link with attributes*, is represented as a tuple $sl_{att} = \langle sl, attsl \rangle$, being $sl \in SL$ and $attsl \in ATTSL$. For example, in fig. 3, $sl2_{io} = \langle \langle n2, nxt, n2 \rangle, \{i, o\} \rangle$.

We present a different example in fig. 4 in order to illustrate the expressiveness of attributes shared, s, and cyclic, c. In that figure, locations $l2$ and $l3$ are summarized in the node $n2$. Concrete $slc1$ and $slc2$ translate to $sl1$ and $sl2$ respectively since they follow different selectors (nxt and prv). Note that $sl1$ and $sl2$ are in different cls's so they can *not* coexist, which precisely captures the fact that following nxt or prv from $n1$ leads to different locations. However, $slc3$ and $slc4$ (both using nxt) are mapped into $sl3$. That way, $sl3_s$ in $cls1_{n3}$ points out that you can point to a location represented by node $n3$ from more than one different locations represented in node $n2$ by following the same selector (nxt). On the other hand, $sl4_c$ in $cls1_{n3}$ expresses that the location $l4$ represented in $n3$ is pointing to itself. Please, note the difference with $sl2_{io}$ in $cls2_{n2}$ in fig. 3, which indicates that one location represented in node $n2$ is pointing to a *different* location represented in the same node.

The key feature of our model is the ability to maintain the connectivity and aliasing information that can coexist in an abstract node, even when the node represents different memory locations with different connection patterns. This is achieved through the coexistent links set abstraction. A coexistent links set, cls_n, codifies a possible aliasing and connectivity pattern for a node n, and it is defined as $cls_n = \{PL_n, SL_n\}$, where PL_n is a set of pointer links pointing to n, and $SL_n = \{sl_{att} \in SL_{att}\ s.t.\ sl_{att} = \langle \langle n_1, sel, n_2 \rangle, attsl \rangle,$ being $(n_1 = n \vee n_2 = n)\}$, is a set of sl's with attributes that contain n. The set of all the cls_n associated to a node n is called CLS_n. In addition, for all the nodes n defined in our abstract heap, we can create the set $CLS = \{CLS_n, \forall n \in N\}$.

2.2. Shape Graphs (SG)

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

We restrict this abstract domain by defining a *normal form* of the shape graphs. We will need two auxiliary functions: (i) `Cmpt_N()` which returns TRUE if two nodes can be summarized into a single one (they are compatible); and (ii) `Path`(n_i, n_j, CLS^k) which returns TRUE if there is a path between node n_i and n_j using the selector links in CLS^k. We say that a shape graph $sg^i = \langle N^i, CLS^i \rangle$ is in normal form if:

1. It has no compatible nodes: $\nexists n_1, n_2 \in N^i \ \ s.t.$ `Cmpt_N`$(n_1, n_2, CLS_{n1}, CLS_{n2}) = TRUE$
2. It has no unreachable nodes: $(\forall n_1 \in N^i, \exists pl_1 = \langle x, n_1 \rangle \subset CLS_{n1}) \vee (\exists n_2 \in N^i \ \ s.t. \ \ \exists pl_2 = \langle x, n_2 \rangle \subset CLS_{n2} \wedge path(n_2, n_1, CLS^i) = TRUE)$
3. A pointer variable unambiguously points to one node: $\forall n_1, n_2 \in N^i \ \ s.t. \ \ n_1 \neq n_2$, If $\exists pl_1 = \langle x, n_1 \rangle \subset CLS_{n1} \Longrightarrow \nexists pl_2 = \langle x, n_2 \rangle \subset CLS_{n2}$
4. The selector links of connected nodes, are coherent: $\forall n_1, n_2 \in N^i \ \ s.t. \ \ n_1 \neq n_2$, If $\exists sl_{att} = \langle \langle n_1, sel_k, n_2 \rangle, attsl \rangle \subset CLS_{n1} \implies \exists sl_{att} = \langle \langle n_1, sel_k, n_2 \rangle, attsl' \rangle \subset CLS_{n2}$

Regarding the `Cmpt_N()` function, the basic compatibility criterion states that two nodes can be summarized if they are pointed to by the same set of pointers. This includes that nodes not pointed to by pointers ($PL_n = \emptyset$) are compatible. This criterion can be extended by adding parametric properties to the nodes and compatibility of properties functions. For instance, the data dependence test is based on the *"DepTouch"* property which annotates the nodes with information about the statements that have read and/or written the locations abstracted in these nodes. Nodes with different *DepTouch* information will not be summarized. See [5] for more details.

2.3. Reduced Set of Shape Graphs (RSSG)

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

Again, we impose a restriction in this set of graphs, and it is that the set is in normal form. We say that a reduced set of shape graphs, $RSSG^s = \{sg^i\}$ is in normal form if it has no compatible shape graphs: $\nexists sg^1, sg^2 \in RSSG^s \;\; s.t. \;\; \texttt{Cmpt\_SG}(sg^1, sg^2) = TRUE$. The auxiliary function $\texttt{Cmpt\_SG}(sg^1, sg^2)$ is formally described in [5], but for the purpose of this paper, it will suffice to state that two sg's are compatible if both sg's have the same alias relationship between pointers and the nodes pointed to by pointers in both sg's are compatible. In that case, these two compatible sg's are joined into a single one which captures the memory configuration represented by the two original sg's. For example, in fig. 5, the symbolic execution of each statement translates in a $RSSG$ capturing the heap after the execution of each one of them. The firsts $RSSG$'s have just one sg, but at the join point of the CFG we have 3 possible memory configurations represented by $sg4$, $sg5$ and $sg6$. Since, $sg4$ and $sg5$ are compatible, we can join them and after the abstract interpretation of statement 7 we obtain $sg7$. Please note that unlike flow-sensitive points-to analysis, this $sg7$ accurately describes both memory configurations thanks to the cls's (not shown for simplicity in the figure). However, $sg6$ is not compatible and after statement 7 gives $sg8$. Both $sg7$ and $sg8$ are in the $RSSG^7$ set. The operator $\bigsqcup^{SG}$ represents the join operation in the SG domain.

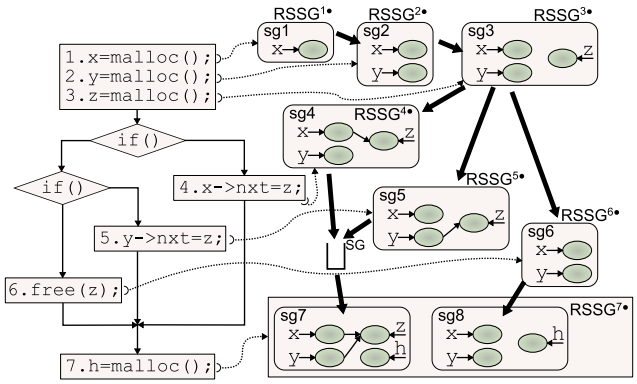

Figure 5. Reduced Set of Shape Graphs in statement 7.

2.4. Abstract Semantics

We formulate our analysis as a dataflow analysis that computes a reduced set of shape graphs at each program point. For each statement in the program, $s \in STMT$, we define two program points: $\bullet s$ is the program point before s, and $s\bullet$ is the program point after s. Therefore, the result of the analysis for that statement is a reduced set of shape graphs, $RSSG^{\bullet s}$ before s, and $RSSG^{s\bullet}$ after that. Let $pred()$ map statements to their predecessor statements in the control flow. Fig. 6 shows the dataflow equations.

[JOIN]: $RSSG^{\bullet s} = \bigsqcup^{RSSG}_{s' \in pred(s)} RSSG^{s'\bullet}$
[TRANSF]: $RSSG^{s\bullet} = AS_s(RSSG^{\bullet s})$, where

$AS_{s::=x=null}(RSSG^{\bullet s}) = \bigsqcup^{RSSG}_{sg^i \in RSSG^{\bullet s}} XNull(sg^i, x)$
$AS_{s::=x=malloc()}(RSSG^{\bullet s}) = \bigsqcup^{RSSG}_{sg^i \in RSSG^{\bullet s}} XNew(sg^i, x)$
$AS_{s::=free(x)}(RSSG^{\bullet s}) = \bigsqcup^{RSSG}_{sg^i \in RSSG^{\bullet s}} XFree(sg^i, x)$
$AS_{s::=x=y}(RSSG^{\bullet s}) = \bigsqcup^{RSSG}_{sg^i \in RSSG^{\bullet s}} XY(sg^i, x, y)$
$AS_{s::=x\to sel=null}(RSSG^{\bullet s}) = \bigsqcup^{RSSG}_{sg^i \in RSSG^{\bullet s}} XselNull(sg^i, x, sel)$
$AS_{s::=x\to sel=y}(RSSG^{\bullet s}) = \bigsqcup^{RSSG}_{sg^i \in RSSG^{\bullet s}} XselY(sg^i, x, sel, y)$
$AS_{s::=x=y\to sel}(RSSG^{\bullet s}) = \bigsqcup^{RSSG}_{sg^i \in RSSG^{\bullet s}} XYsel(sg^i, x, y, sel)$

Figure 6. Dataflow equations.

```
Worklist()
  Input: P = ⟨STMT, PTR, SEL⟩, RSSG^in
  Output: RSSG^out

1:  Create W = STMT
2:  RSSG^{•se} = RSSG^in
3:  ∀s ∈ STMT → RSSG^{s•} = ∅
4:  repeat
5:      Remove s from W in lexicographic order
6:      RSSG^{•s} = ⊔^RSSG_{s'∈pred(s)} RSSG^{s'•}
7:      RSSG^{s•} = AS_s(RSSG^{•s})
8:      If (RSSG^{s•} has changed),
9:          forall s' ∈ succ(s),
10:             W = W ∪ s'
11:         endfor
12: until (W = ∅)
13: RSSG^out = RSSG^{sr•}
14: return(RSSG^out)
```

Figure 7. Intraprocedural worklist algorithm.

We model the analysis of individual statements computing a transfer function for each one. To simplify the formal definitions of the transfer functions we use the functions `XNull()`, `XNew()`, `XFree()`, `XY()`, `XselNull()`, `XselY()` and `XYsel()` to describe the transformations that take place in the abstract heap when a simple statement s is interpreted. The operator $\bigsqcup^{RSSG}$ represents the join operation in the $RSSG$ domain.

Basically, the transfer functions for the `x=null`, `x=malloc()`, `free(x)` and `x=y` statements, take each shape graph from the input set $RSSG^{\bullet s}$, transform it according to the statement semantics, and later join all the transformed graphs to build the output set $RSSG^{s\bullet}$. An example of such behavior can be seen for statement 7 in fig. 5, where the input shape graph set for the statement is formed by two graphs: $sg6$ and the joining of compatible

graphs $sg4$ and $sg5$. Both graphs are transformed according to the abstract semantics of `XNew()`, producing $sg7$ and $sg8$. These graphs are not compatible according to the `Cmpt_SG()` function, hence both form $RSSG^{7\bullet}$.

On the other hand, the transfer functions for the `x->sel=null`, `x->sel=y` and `x=y->sel` statements, involve more steps because intermediate graphs can be generated by following the `x->sel` or `y->sel` path. Consider the example shown in fig. 8. The input shape graph set is made of a single graph, $sg1$ within $RSSG^{\bullet s}$, which represents a singly-linked list of one or more elements (sl's and cls's are shown in the figure). This graph is first split by `y->nxt`. Since there are two possible ways to follow from $n1$ (the node pointed by `y`) through selector `nxt`, two graphs are generated: sgA (1-element list) and sgB (2 or more element list). In the next step, a new node is *materialized* to focus the part of the list where the access is taking place. This operation only makes sense for sgB, producing sgD. Then follow the transformations corresponding to the abstract semantics of the current operation, in this example, the `x=y->nxt` statement, which invokes the `XYSel()` function (as shown in fig. 6). At the last step, all graphs are normalized and compatible graphs are joined forming the output shape graph set, $RSSG^{s\bullet}$. For this case, sgE and sgF are not compatible, and cannot be joined together. Please, see [5] for a detailed description of all these abstract semantic functions. Finally, note that pointer arithmetic is not currently supported.

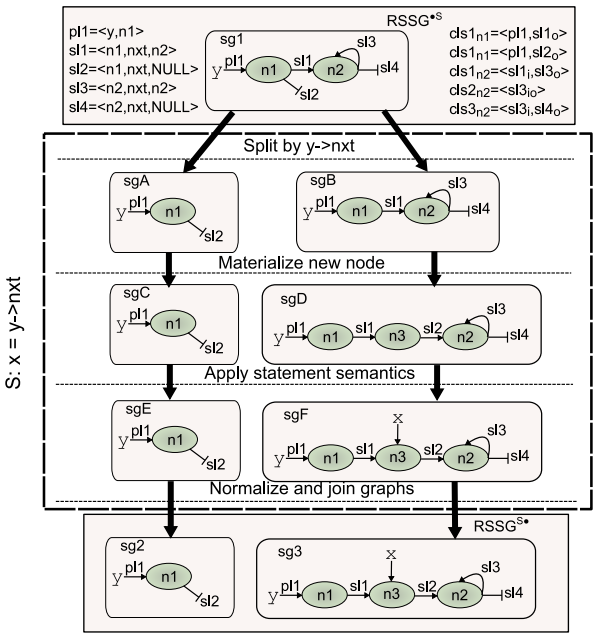

Figure 8. Steps for `x=y->sel` statement.

We present in fig. 7 a worklist algorithm for solving the dataflow equations presented in fig. 6. The input of our worklist algorithm is a program P and an initial $RSSG^{in} = \emptyset$, whereas the output is the $RSSG^{out}$ resultant at the return program point, assuming that the return point is statement $sr \in STMT$. This algorithm also computes the resultant $RSSG^{s\bullet}$ at each program point. Lines 1-3 perform the initialization, where the $RSSG$ at the input of the program entry point (in our case statement $se \in STMT$) is initialized with $RSSG^{in}$. Next, the algorithm processes the worklist using the loop defined in lines 4-12. At each iteration, it removes, in program lexicographic order, a statement from the worklist, computes the join of the $RSSG$'s from the predecessors as the statement input, and then it applies the corresponding transfer function. If the resultant $RSSG$ has changed, the algorithm adds the successors of the statement under consideration ($succ(s)$) to the worklist (line 10).

2.5. Pseudostatements

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

A preprocessing pass extracts semantic information from test conditions in `if` and `while` program flow statements, when these test conditions involve pointers variables. Then, `Force` pseudostatements (also called *assume* statements) are introduced to communicate that information to the analysis. On the branch where the tested expression is null, e.g. `z==null`, the force's transfer function filters out the graphs in which a pointer link of the form $pl =< z, n_i >$ exists, i.e. the variable `z` points to a node. Statement 16 in fig. 9 is an example of such a statement. On the contrary, on the branch where the tested expression is not null, e.g. `z!=null`, then the transfer function filters out the graphs in which a pointer link of the form $pl =< z, n_i >$ does *not* exist, i.e. the variable `z` does not point to a node. An example for this `Force` is statement 12 in fig. 9. In this way, we allow the analysis to filter out unrealistic memory configurations.

The `DepTouch` pseudostatement lets us annotate the node pointed to by a pointer x, with an identifier, whereas the `DepUntouch` pseudostatement removes that identifier from any node of the graph. This kind of annotations is useful for dependence analysis. `DepTouch` pseudostatements are inserted by our preprocessing pass, just after the statements that perform read or write accesses to data or selector fields that potentially may provoke loop carried data dependencies (LCDs). Each `DepTouch` codifies a statement and its type of access (read(`R`)/write(`W`)). When it is abstractly interpreted, then the corresponding node is an-

notated with that information. Later, the data dependence test checks if a node has been actually written and read by statements that could produce LCDs, and in that case a data dependence (and the type of dependence - RAW, WAR or WAW) can be reported. In the example of fig. 1, statement `S3` produces the annotation of the node pointed by `p` with the `RS3` tag, whereas `S4` annotates with `WS4` (`DepTouch` pseudostatements are not shown for simplicity in the code). That information is then used to detect a RAW LCD.

3. Extensions for interprocedural analysis

```
   struct node{
       int data;
       struct node *nxt;
   } *r;
 1: struct node *create_list(int size){ ...  }

 9: struct node *rev(struct node *x){
       struct node *y, *z;
10:    z=x->nxt;
11:    if(z!=NULL){
12:       #pragma SAP.force(z!=NULL)
13:       y=rev(z);
14:       x->nxt=NULL;
15:       z->nxt = x;
       }else{
16:       #pragma SAP.force(z==NULL)
17:       y=x;
       }
18:    return y;
   }
19:int main(int argc, char *argv[]){
       struct node *l;
20:    int size = size_from_args(argv);
21:    l = create_list(size);
22:    r = rev(l);
23:    return 1;
   }
```

Figure 9. Example program.

An *interprocedural* shape analysis technique must also be able to deal with function calls and return sites. The main challenge is encountered in the presence of recursive functions, where heap abstractions need to maintain the state of pointer formal parameters and local pointers in a sequence of recursive calls. At runtime, the *Activation Record Stack* (ARS) provides explicit information about the state of variables for every call. In our approach, we abstract the information of the ARS by using (i) a new kind of link over the base shape graph representation, and (ii) some context change rules that dictate how the graphs and these new links are modified at function calls or return sites.

From now on, we will use a simple example to illustrate the main features of our interprocedural shape analysis strategy. Fig. 9 shows the abridged C code for `reverse`, a program that creates a singly-linked list and then reverses it with a recursive function, `rev()`. Let us assume now that the memory configuration for the 4-element list of fig. 3 is used as input for the `rev()` function. The list is then traversed in a sequence of recursive calls. The memory configuration that results at the 4th invocation of the `rev()` function (line 9) is shown in fig. 10(a), where the *Activation Record Stack* (ARS) has been included to maintain the pointer links for `x` in previous calls of `rev()`. The information within the ARS is required when we go back to the return site of an enclosing call, so that we know the destination of the corresponding pointer formal parameters and local pointers.

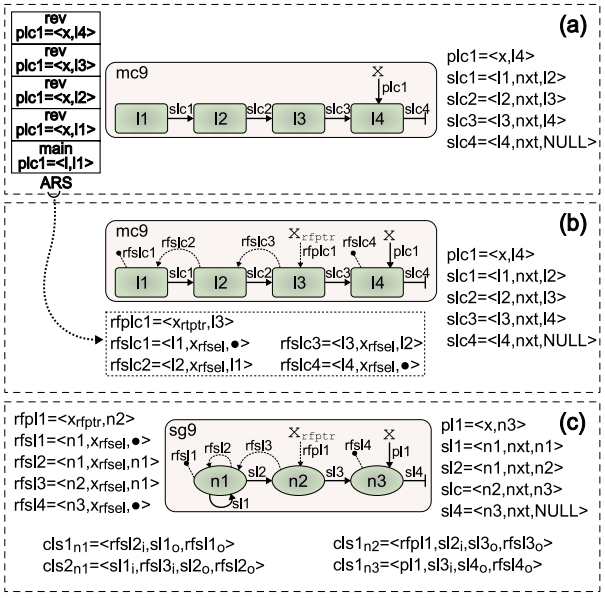

Figure 10. A 4-element list after the 4th invocation to `rev()`: (a) with ARS, (b) with recursive flow links, and (c) its shape graph.

3.1. Recursive flow links

We introduce a new element to extend the base shape graph abstraction, as described so far, to include the information that we need from the ARS. This is done with two kinds of recursive flow links: *recursive flow pointer links* (`rfpl`) and *recursive flow selector links* (`rfsl`). They are defined very similarly to regular pointer links and selector links, only based on *recursive flow pointers* (`rfptr`) and *recursive flow selectors* (`rfsel`) respectively. Recursive flow pointers and recursive flow selectors are introduced for every pointer variable that needs to be traced along the in-

terprocedural control flow (in the worst case, all the pointer formal parameters and local pointers). By convention, they are named after the pointer variables that they trace, with the subscript *rfptr* or *rfsel*, respectively.

Recursive flow links do not represent actual links existing in the program data structure but rather *trace* the positioning of pointer formal parameters and local pointers along the recursive, interprocedural control flow. A *recursive flow pointer link* points to the same memory location/node where the traced pointer was pointing to in the *immediately previous* call in a stack of recursive calls, while a *recursive flow selector link* points to the locations/nodes *beyond* the immediately previous activation record.

Fig. 10(b) shows the memory configuration from (a), exchanging the ARS for the needed recursive flow links, which are shown in dashed lines. In the *rfplc* and *rfslc*, the last "c" stands for "concrete". The location/node denoted by • is representing the `NULL` location for the recursive flow path. Following the trace through a recursive flow selector link with • as destination would not correspond to any activation record in the succession of recursive calls, and therefore would not render any realistic memory configuration. Fig. 10(c) shows the abstraction, in the abstract domain, of the memory configuration in (b). Here memory locations $l1$ and $l2$ are summarized by $n1$. The selector links are updated accordingly as shown in the figure.

3.2. Context change rules

We now consider a program to include the set of functions, F, declared in that program, and we extend the type of analyzable statements to include the `call()` and `return()` of these functions. Function pointers are not supported. An important detail is that we distinguish between non-recursive and recursive call sites and between recursive and non-recursive return points. We formulate a context sensitive interprocedural analysis, because we distinguish between different calling context of the same procedure. The analysis at procedure calls must account for the assignment of actuals to formals and for the change of analysis domain between the caller and the callee. For it, shape graphs are transformed into the appropriate context while flowing in and out of functions by the *context change rules*, namely the *call-to-start* (`CTS`) rule, and the *return-to-call* (`RTC`) rule.

The call-to-start rule determines how the recursive flow links in the shape graphs are transformed from a function call to the context inside the function. On the other hand, the return-to-call rule transforms the heap abstraction returned by a function to the appropriate context at the calling site. Each of these rules has a recursive ($\texttt{CTS}_r$ and $\texttt{RTC}_r$) and non-recursive ($\texttt{CTS}_{nr}$ and $\texttt{RTC}_{nr}$) version. Due to space constraints, only a brief description for the $\texttt{CTS}_r$ and $\texttt{RTC}_r$ rules will be covered here (see [5] for more details). $\texttt{CTS}_r$ finds the node n_i pointed to by each formal or local pointer x, and the node n_j pointed to by the same pointer in the previous pending call (x_{rfptr}). Then it creates the new recursive flow pointers and links for the current context, accordingly updating the cls's. In additions, all actual parameters are mapped into the corresponding formal ones. $\texttt{RTC}_r$ works by assigning local pointers to the nodes where they were pointing to in the previous context. For that, the matching of returned pointer and assigned pointer at call site, as well as the matching between formals and actuals, are used. The rest of local pointers are reassigned according to the existing recursive flow links.

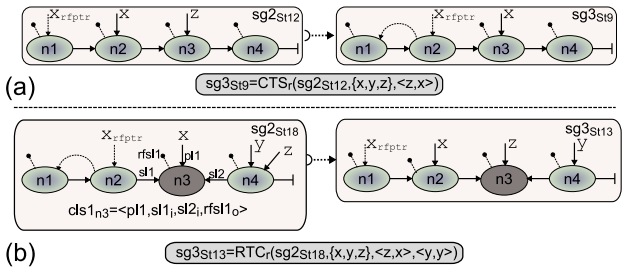

Figure 11. Example of the (a)$\texttt{CTS}_r$ and (b)$\texttt{RTC}_r$ rules.

These context change rules are better understood by example. Fig. 11(a) shows how the $\texttt{CTS}_r$ rule acts upon shape graph $\texttt{sg2}_{St12}$, the abstraction of a 4-element singly-linked list that reaches the third call to `rev()`, by taking the `if` branch in the code of fig. 9. The result of applying the $\texttt{CTS}_r$ rule is named $sg3_{St9}$. Null assigned or uninitialized pointers are not shown. Links are only displayed graphically, and coexistent links sets are left out, to simplify the presentation. Note that the location pointed to by `z` in $sg2_{St12}$ is now pointed to by `x` in $sg3_{St9}$. The $\texttt{x}_{rfptr}$ pointer is updated to point to $n2$, node pointed to by `x` in the previous context, and the recursive flow selector from $n2$ to $n1$ in $sg3_{St9}$ leaves the trace to the node pointed to by $\texttt{x}_{rfptr}$ in the previous context. Local pointer `y` is not defined before the recursive call so no tracing is necessary.

Fig. 11(b) shows how the $\texttt{RTC}_r$ works on $\texttt{sg2}_{St18}$, the graph resulting from the third call to `rev()`. The result is $\texttt{sg3}_{St13}$, which exchanges actual parameter `z` for its matching formal parameter `x`. Pointer `y` is assigned over itself, so it causes no change. Pointer `x` is reassigned to the node pointed to by $\texttt{x}_{rfptr}$, which is updated by following its recursive flow selector link.

An interesting fact about the use of coexistent links sets in our abstraction can also be observed in fig. 11(b): `n3` is pointed from `n2` and `n4`, both links coexisting in its cls (shown for $\texttt{sg2}_{St18}$). We say `n3` is *shared*, because it can

be directly reached from more than one node. This is highlighted graphically by shading the oval depicting the node. The cls information allows to know that, not only can `n3` be accessed by following two different paths, but we now exactly what paths. This sharing pattern is then undone as the list continues to be reversed, preserving the *listness* of the structure at the end of the analysis.

The context change rules require that the program is preprocessed so that pointer formal parameters are not modified inside the procedure. This restriction makes the formulation simpler and does not involve any loss of generality, as it is always possible to rewrite a function to comply with this condition by using additional local pointer variables, whenever pointer parameters are passed *by value*. When pointer parameters are passed *by reference*, they are treated just like global pointers. The technique also assumes there is only one return point in the function, at the end of its body. Again, this involves no loss in generality as any function are easily rewritten to comply with this condition by our preprocessing compiler pass.

3.3. Reuse of function summaries

The technique described so far is enhanced by using a tabulation algorithm (see [5]) that records function summaries for reuse under equivalent calling contexts. Every time a non-recursive call statement is encountered the shape graphs in the current shape graph set are split according to the *reachability* of actual parameters in the call and global pointers. This way, for each incoming shape graph we obtain two graphs: (i) the *reachable graph*, which abstracts the part of the heap accessible through the function call actual parameters and global pointers, by following any pointer-chasing path from them; and (ii) the *unreachable graph*, which abstracts the part of the heap that is not accessible inside the function called, or equivalently, the part of the heap accessible through the rest of pointers in the program. This splitting process is performed within the tabulation algorithm, which checks if any of the incoming reachable graphs has been previously analyzed. If it has, then the previously obtained output is returned. Otherwise, it is analyzed and the input-output pair is stored for future reuse. In any case, the unreachable part of the graph is finally joined with the resulting shape graphs to yield the total effect of the function call.

The *tabulation* algorithm in conjunction with the *reachability splitting* mechanism allow to reuse the computed effects for functions in the case of graphs whose reachable elements match a previously analyzed graph. However, not all input shape graphs can be split. Whenever a shape graph represents memory locations that are found both in the reachable and non-reachable graphs, then the graph cannot be safely split, because it could not be reconstructed by a simple join graph operation. This is similar to the concept of *cutpoint* presented in [17], but somewhat more restrictive. In the case of such a *cutpoint*, the analysis must proceed with the whole shape graph and will be less likely to reuse function summaries.

4. Experimental results

We have implemented the algorithms presented in this paper within our optimizing-compiler framework [3]. We focus on the analysis of C sequential programs. All the needed preprocessing passes are performed with custom-made passes built upon Cetus [11], a versatile source-to-source compiler framework.

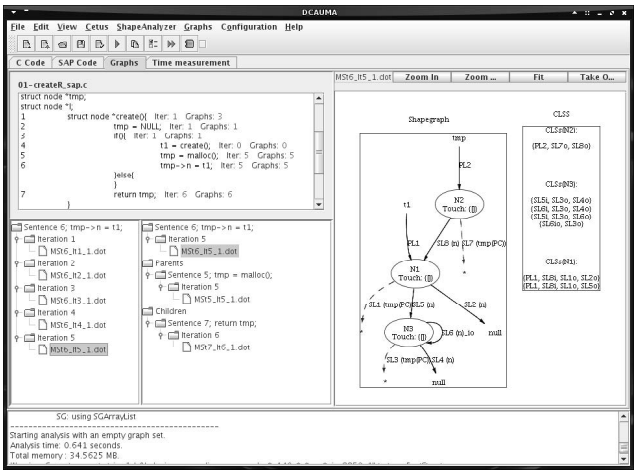

Figure 12. Graphical User Interface for Shape Analysis.

Program	Time (sec.)	Space (MB)
1-Singly-linked list	0.06	1.9
2-Doubly-linked list	0.08	1.9
3-N-ary tree	0.14	1.9
4-Binary tree	0.52	1.9
5-Matrix-vector(s)	0.33	1.9
6-Matrix-vector(d)	0.39	1.9
7-Matrix-Matrix(s)	3.51	4.9
8-Matrix-Matrix(d)	4.09	4.9
9-em3d	22.98	4.9

Figure 13. Analysis time and required memory for the analysis of some pointer based programs. Programs marked with (s) have their structures based on singly-linked lists, while those marked with (d) are based on doubly-linked lists. The testing platform is a 3GHz Pentium 4 with 1GB RAM.

We have also implemented a GUI to enable a friendly use of our shape analyzer tool. In fig. 12 we can see one of

the available windows in which the "Graphs" tab is selected. In that tab we have got the analyzed code with each statement annotated with information regarding the number of times that statement have been symbolically executed and the number of sg's associated with it. We also provide the links to each graph and information about the parents and children of each one of them, as well as the graphical view of the graphs and its cls's. There is also a "SAP code" tab (SAP stands for Shape Analyzer Preprocessing). This tab is very useful to compare the original C code and the preprocessed version resulting from the Cetus compiler pass (as we said, this preprocessing takes care of the insertion of force statements and other transformations and simplification that have to be performed to optimize the shape analysis).

We have conducted three kind of experiments. In a first place, fig. 13 shows a group of programs based on dynamic data structures. These programs were analyzed as a test for the ability of the technique to capture several types of dynamic data structures. These structures include singly-linked lists, doubly-linked lists, binary trees, n-ary trees, and sparse matrices or sparse vectors built based on singly- or doubly-linked lists. The codes in fig. 13 do not include recursive functions. Programs 1 to 4 in fig. 13 create and traverse their corresponding data structures. Programs 5 to 8 in the same figure, implement the product of sparse matrix by sparse vector, or the product of two sparse matrices. Program 9 is the `em3d` from the Olden suit [2]. All the structures tested were captured accurately.

The second kind of experiment focuses on the analysis of some basic recursive algorithms based on singly-linked lists or binary trees. The motivation for these tests is two-fold: (i) to test the mechanism for recursive analysis, and (ii) to compare results and performance with significant related work. In particular, we have considered the codes tested in [17], forming a small library of recursive algorithms that operate on singly-linked lists (create list, find element, insert element, delete element, append element, reverse list, splice list) and binary trees (create tree, insert element, find element, find tree height, splice left subtree, rotate tree). Both [17] and our method are able to determine that structure is preserved after the call to the recursive functions. This means that for the list tests, if the function is called with an acyclic singly-linked list, then the output is also an acyclic singly-linked list, i.e., no cycles have been introduced in the list. For the tree tests, if the input is an unshared binary tree (no child has 2 parents), then that shape is preserved at the output. The analysis times for all the codes, in a Pentium M 1.6 GHz with 224 MB, are less than 1 sec. for the list codes and less than 32 sec. for the tree codes. The memory consumed fits in a block of 1.9 MB for all the list tests. For the tree tests it never goes above 4 MB. Published time and memory requirements in [17] are much larger in a similar platform (Pentium M 1.5GHz, 1GB): regarding their times, they range from 9.3 to 46 sec. for the list codes and from 14.3 to 105 sec. for the tree codes. More details about these experiments can be found in [5].

To further test our technique regarding recursive analysis, we considered `power` (nested linked lists) and `treeadd` (binary tree) from the Olden suite. We were able to accurately capture the structures recursively created and traversed in these benchmaks, in 8 sec/13 MB and 70 sec/11 MB respectively in a 3Ghz Pentium 4 with 1GB RAM.

4.1. Data dependence test and parallel execution results

Finally, we have also conducted some preliminary experiments of our data dependence test [15], now built on top of our new interprocedural shape analysis algorithm. Using the node property "DepTouch" and symbolically executing the loops of a code, we can annotate which memory locations are read or written and detect loop-carried data dependences (LCD). This test have been applied to the sparse matrix-vector, the sparse matrix-matrix and em3d codes of figure 13. The matrix codes were based on doubly-linked lists to store the sparse matrices and vectors.

Program	# stmts.	A. Time	Space	Speed-Up
MxV(d)	107	2:08	19.1	7.57
MxM(d)	143	9:26	7.02	7.72
em3d	224	0:34	5.16	2.02

Table 1. Analysis time (min:sec) and required memory (MB) for the data dependence test and speed-up results. The data dependence test was conducted in a 3GHz Pentium 4 with 1GB RAM. The parallel codes are executed in 8 UltraSparc III 900MHz.

The test successfully reports no data dependences and that the outer loops of the two sparse codes and the `compute_nodes()` routine of the em3d can be safely parallelized. Regarding the analysis time, this test needs 2 min. 8 sec. for the matrix-vector and 9 min. 26 sec. for the matrix-matrix. Clearly, keeping track of the accessed locations has a big impact in the analysis time. However, the data dependence times we previously reported using our old RSG shape analysis [15] were 1 min. 47 sec. for the matrix-vector and 94 min. for the matrix-matrix in the same Pentium 4. More experiments have to be carried out, and there are several issues to consider regarding the comparison of both implementations (for instance the RSG version was implemented in C and the new one based on CLSs has been done in Java). However, we can anticipate that for simpler codes like the matrix-vector one, the new underlying shape analysis does not improve the previous one, but

for more complex codes like the matrix-matrix (with four nested loops and several conditional branches inside them) we have observed that the CLS are able to capture the complexity of the analyzed memory configurations in a more compact way, reducing the analysis time by one order of magnitude. Besides, we have to stress that our new shape analysis is now ready to support an interprocedural data dependence test.

Now that we have automatically identified the parallel loops of these codes, the next step is to generate the parallel code. As a first approach we have just manually inserted an "omp parallel for" before the parallel loops, identifying the private variables. Sparse codes were tested with matrices of 10000×10000 and 20% density. In the em3d, we execute the parallel version of `compute_nodes()` with 1024 E nodes and 1024 H nodes and a degree of the bipartite graph of 50. The low speed-up of 2 on 8 processors is due to the lack of enough workload in each iteration of the `compute_nodes()` routine. Other authors report better speed-up but relying in iterating 500 or more times this routine [9].

5. Related work

Recently, there have been many interesting works in the shape analysis field. A important body of them use separation logic [6], [13], [1], [8] to describe the shape of data structures through recursively defined predicates. Some of these works rely on pre-defined recursive predicates [6], [13], whereas [1], [8] resort to inductive synthesis to infer recursive shape invariants. [1] is suitable for list-processing programs limiting the class of analyzable programs, whereas [8] can handle more general data structures, specially data types with tree-like backbone. Both of them are interprocedural. All these works are concerned in finding predicates that encode spatial relationships between heap locations, and typically are targeted towards program verification. Our work, on the contrary is concerned with finding the temporal relationships that may impose the data dependences that arise in the section of a program candidate to be parallelized (a loop or a function call, in our study). We believe that the kind of analysis that is suitable for verification is not easily adaptable towards dependence detection. Verification is mainly concerned with proving code correctness and obtaining structure invariants and therefore cannot make any assumptions about the program. In our approach, we assume code correctness, and we are not so much concerned with the structure invariants than actual access relationships, which determine dependencies.

Other approaches in shape analysis such as [18], [17], or [10] use the 3-valued logic analyzer (TVLA) parametric framework. TVLA can be used to instantiate different shape analyzers based on a collection of instrumentation predicates to accurately describe the structures. These predicates must be provided by the user, although in [12] machine learning is applied to automatically find recursive predicates. These works are concerned with codifying spatial shape invariant information, suitable for program verification client analyses, but as we have mentioned before, our study is on the contrary concerned with finding temporal data dependences. It must be stressed that although our work shares ideas with the TVLA framework (such as summarization, materialization, or abstract interpretation), we provide a new, unrelated shape analysis technique.

Regarding *expressiveness*, some of these works can provide more information about the data structures than our approach. For example, [17] recognizes that a sorting function returns a permutation of elements in the input, and [10] is able to find out that reversing a list twice yields the same list. Again, we think that kind of information is tailored for verification clients and is not so important for data dependence detection.

More related to our problem, i.e., finding data dependences in recursive/dynamic data structures using a shape analysis technique are the works [7], [9] and [14]. For instance, [7] proposed a test for identifying data dependences, test that relies on the identification of the shape of the overall data structure being traversed (among the pre-defined Tree, DAG or Cycle shapes), as well as on the computation of the access paths for the pointers in the statements being analyzed. The technique presented in [9] deduces the shape of the traversal patterns (among the pre-defined Tree, DAG or Cycle shapes) over the shape of overall data structure. Once they have extracted the traversal-pattern shape information, dependence analysis is applied to detect dependences. The main drawback in these works is that they are not useful in programs that perform destructive updates in the loops under test. The work in [14] goes a step further, and similarly to our approach, they make use of the notions of abstract interpretation and refinement, which allows the analyses to give support to destructive updates. However, in [14] the interprocedural support is very simplistic, and they do not support recursive functions, which is a major contribution in our work. Besides, they target only Java codes entirely based on manually-tuned collection libraries. Although they make use of the results of their analysis for parallelization, this is done entirely by the programmer, whereas we provide the means for automatic detection of loop-carried dependencies.

We differ from previous related works in that our technique let us annotate the memory locations reached by each heap-directed pointer, with read/write information. This feature let us capture quite accurately, the temporal relationship between the statements that visit the locations of the program heap. Our algorithm is flow sensitive, context sensitive and supports recursion, which lets us analyze quite accurately loops and functions that traverse and cre-

ate generic heap-based recursive/dynamic data structures in programs that perform destructive updates.

6. Conclusions and future work

We have presented in this work an interprocedural shape analysis technique and its application to the automatic parallelization of dynamic data structure based codes. The core intraprocedural analysis is based on the concept of coexistent links sets, which provide a compact way to represent possible connections of heap elements. This core technique is then extended with two key ideas: (i) a *new kind of link* (recursive flow links) that leaves a trace in our graph representation across recursive calls, so that we can recover pointer state when returning to enclosing calls; and (ii) a couple of *context change rules* (call-to-start and return-to-call) that describe how the heap representation is transformed when performing function calls or returning to a call site.

Also, we have conducted some preliminary experiments that show that the technique provides accurate results and outperforms significantly some previous approaches to the problem of interprocedural shape analysis. Besides, we have automatically detected the parallel loops of three codes, which once parallelized exhibit good speed-ups. Next, we plan to conduct more experiments over bigger benchmark programs and to put to work our shape analysis to perform an accurate interprocedural data dependence test in order to exploit not only loop-level parallelism but also function level parallelism.

References

[1] J. Berdine, C. Calcagno, B. Cook, D. Distefano, P. O'Hearn, T. Wies, and H. Yang. Shape analysis for composite data structures. *Lecture Notes in Computer Science*, 4590:178–192, 2007.

[2] Martin C. Carlisle and Anne Rogers. Software caching and computation migration in Olden. In *ACM Symposium on Principles and Practice of Parallel Programming (PPoPP)*, July 1995.

[3] R. Castillo, A. Tineo, F. Corbera, A. Navarro, R. Asenjo, and E.L. Zapata. Towards a versatile pointer analysis framework. In *European Conference on Parallel Computing (EURO-PAR) 2006*, 29th August - 1st September 2006.

[4] 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.

[5] F. Corbera, A. Navarro, R. Asenjo, A. Tineo, and E.L. Zapata. A formal presentation of shape analysis graphs and operations. In *Technical Report at http://www.ac.uma.es/∼asenjo/research/*, Dpt. Computer Architecture, Univ. of Malaga, Spain, July 2007.

[6] D. Distefano, P.W. O'Hearn, and H. Yang. A local shape analysis based on separation logic. *Lecture Notes in Computer Science*, 3920:287–302, 2006. Springer-Verlag.

[7] R. Ghiya, L. J. Hendren, and Y. Zhu. Detecting parallelism in C programs with recursive data structures. In *Proc. 1998 International Conference on Compiler Construction*, pages 159–173, March 1998.

[8] B. Guo, N. Vachharajani, and D. August. Shape analysis with inductive recursion synthesis. In *Programming Language Design and Implementation (PLDI'07)*, June 2007.

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

[10] B. Jeannet, A. Loginov, T. Reps, and M. Sagiv. A relational approach to interprocedural shape analysis. In *In Proceedings of the 11th International Static Analysis Symposium (SAS'04)*, Verona, Italy, August 2004.

[11] 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.

[12] A. Loginov, T. Reps, and M. Sagiv. Abstraction refinement via inductive learning. *Lecture Notes in Computer Science*, 3576, 2005. Springer-Verlag.

[13] S. Magill, A. Nannevski, E. Clarke, and P. Lee. Inferring invariants in separation logic for imperative list-procesing programs. In *Workshop on Semantics, Program Analysis and Computing Enviroments for Memory Management (SPACE)*, January 2006.

[14] Mark Marron, Darko Stefanovic, Manuel Hermenegildo, and Deepak Kapur. Heap analysis in the presence of collection libraries. In *7th ACM Workshop on Program Analysis for Software Tools and Engineering (PASTE'07)*, San Diego, June 2007.

[15] A. Navarro, F. Corbera, A. Tineo, R. Asenjo, and E.L. Zapata. Detecting loop-carried depedences in programs with dynamic data structures. *Journal of Parallel and Distributed Computing*, 67:47–62, 2007.

[16] 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.

[17] N. Rinetzky, M. Sagiv, and E. Yahav. Interprocedural shape analysis for cutpoint-free programs. In *12th International Static Analysis Symposium (SAS'05)*, London, England, September 2005.

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