Increasing the Accuracy of Shape and Safety Analysis of Pointer-based Codes

Pedro C. Diniz

University of Southern California / Information Sciences Institute
4676 Admiralty Way, Suite 1001
Marina del Rey, California, 90292
pedro@isi.edu

Abstract. Analyses and transformations of programs that manipulate pointer-based data structures rely on understanding the topological relationships between the nodes i.e., the overall *shape* of the data structures. Current static *shape* analyses either assume correctness of the code or trade-off accuracy for analysis performance, leading in most cases to shape information that is of little use for practical purposes. This paper introduces four novel analysis techniques, namely *structural fields*, *scan loops*, *assumed/verified shape properties* and *context tracing*. Analysis of *structural fields* allows compilers to uncover node configurations that play key roles in the data structure. Analysis of *scan* loops allows compilers to establish accurate relationship between pointer variables while traversing the data structures. *Assumed/verified* property analysis derives sufficient shape properties that guarantee termination of scan loops. These properties must then be verified during shape analysis for consistency. *Context tracing* allows the analyses to isolate data structure nodes by tracing relationships between pointer variables along control-flow paths in the program. We believe that future static shape and safety analysis algorithms will have to include some if not all of these techniques to attain a high level of accuracy. In this paper we illustrate the application of the proposed techniques to codes that build (correctly as well as incorrectly) data structures that are beyond the reach of current approaches.

1 Introduction

Codes that directly manipulate pointers can construct arbitrarily sophisticated data structures. To analyze and transform such codes compilers must understand the topological relationships between the nodes of these structures. For example, nodes might be organized as trees, directed acyclic graphs (DAGs) or and general cyclic graphs. Even when the overall structure has cycles, it might be important to understand that the induced data structure topology traversing only specific fields is acyclic or even a tree.

Statically uncovering the shape of pointer-based data structures is an extremely difficult problem. Current approaches interpret the statements in the program (ignoring safety issues) against an abstract representation of the data

structure. As pointer-based data structures have no predefined dimensions, compilers must summarize (or *abstract*) many nodes into a finite set of *summary* nodes in their internal representation of the data structure. The need to summarize nodes and symbolic relationships between pointers that manipulate the data structures leads to conservative, and often incorrect, determinations of the shape of data structures, *e.g.,* reporting that a data structure has a cycle when in reality it has not.

We believe the key to address many of the shortcomings of current shape analysis algorithms is to exploit the information that can be derived from both the predicates of conditional statements and from looping constructs. These construct, as the examples in this paper illustrate, can help compilers to derive accurate symbolic relationships between pointer variables. For example the `while` loop code below (left) *scans* a data structure along the `next` field. If the body of the loop executes, on exit we are guaranteed that {`t != NULL`} holds, but more importantly that {`p = t->next`}. This fact is critical to verifiy the *correct* insertion of an element in a linked-list. The code below (right) corresponds to an insertion in a doubly-linked list where the predicate clearly identifies the node denoted by `p` as the last node in the list by testing its `next` field. The node with the configuration {`next = NULL`}, therefore plays the important role of signaling the list's end.

The loop code also reveals that a sufficient[1] condition for its termination is that the data structure be acyclic along `next`. An analysis algorithm can operate this acyclicity assumption to ascertain termination and properties of other constructs and later verify the original acyclicity assumption.

```
t = NULL;
while(p != NULL){
  if(p->data < item)
    break;
  t = p;
  p = p->next;
}
```

```
if(p->next != NULL){
  p->next->prev = temp;
  temp->next = p->next;
  p->next = temp;
  temp->prev = p;
}
```

These examples illustrate that programmers fundamentally encode "state" in their programs via conditionals and loop constructs. Many loop constructs are used to scan the structures to position pointer variables at nodes that should be modified. Conditional statements define which operations should be performed. The fact that programmers used them to encode "state" and reason about the relative position of pointer variables and consequently nodes in the data structure is a clear indication that a shape and safety analysis algorithms should exploit the information conveyed in these statements.

[1] Although not a necessary condition as the programmer might insert sentinel values that prevent the code from ever reaching a section of the data structure with a cycle.

This paper presents a set of symbolic analyses techniques that we believe will extend the reach of current static shape and safety analysis algorithms for codes that manipulate pointer-based data structures, namely:

- **Structural Fields:** Uncovering value configurations or "states" of nodes that potentially play key roles in the data structure.
- **Scan Loops:** Symbolic execution of loops that only traverse (but do not modify) data structures. These loop will enable the extraction of all possible bindings of pointers to nodes in the abstract shape representation using the relationships imposed by the loop statements.
- **Assumed/verified Properties:** Derivation of sufficient shape properties that guarantee termination of *scan* loops. These properties must be verified during shape analysis.
- **Context Tracing:** The compiler can propagate contexts, *i.e.,* the set of bindings of variables to nodes in the data structure throughout the program and use conditionals to prune the sets of nodes pointer variables can point to at particular program points.

While the integration and effective exploitation of the knowledge gained by the techniques presented in this paper with actual shape and safety analysis algorithms are beyond the scope of this paper, this paper presents a set of techniques we believe will enhance the applicability and accuracy of these analysis algorithms. For example, identifying nodes with *selected* configurations can help *summarization* and *materialization* algorithms to retain *particular* nodes of the data structure. Retaining precise *symbolic* relationships between pointer variables can also allow materialization algorithm to preserve *structural invariants.*

This paper is organized as follows. The next section describes a specific example that illustrates potential of the proposed approach. Section 3 describe the set of basic symbolic analysis our algorithm relies on. We present experimental evidence of the success of this approach for both correct and incorrect codes in section 4. We survey related work in section 5 and then conclude.

2 Example

We now illustrate how a compiler can use the techniques presented in this paper to increase the accuracy of shape analysis and safety information for codes that manipulate sophisticated pointer-based data structures. This code, depicted in figure 1, builds a data structure by the successive invocation of the `insert` function. In this code the function `new_node(int)` allocates through the `malloc` function a node that is unaliased with any of the nodes in the data structure. In this example we assume an initial binding of the `node` argument to a single node with both `link` and `next` pointer fields equal to `NULL`.

The code starts by scanning (via what we call a *scan* loop) the data structure along the `link` field searching for the appropriate insertion point. Next it allocates the storage for a new node and inserts it "forward of" the node pointed to by `b` along the `next` field. It then conditionally links the node pointed to by

```
   typedef struct node {
     int data;
     int nchild;
     struct node *link, *next;
   }

01:void insert(node* n, int d){
02:  node *b, *t;
03:  b = n;
04:  while(b->link != NULL){
05:    if(b->link->data > d)
06:      break;
07:    b = b->link;
08:  }
09: t = new_node(d);
10: t->next = b->next;
11: b->next = t;
12: b->nchild++;
13: if(b->nchild == 2){
14:   b->next->next->link = b->link;
15:   b->link = b->next->next;
16:   b->nchild = 0;
17: }
18:}
```

Fig. 1. Pointer-Based Data Structure Insertion C Code.

b to the node denoted by b->next->next along the link field. This relinking step effectively splits a long sequence of nodes into two shorter sequence along the link field and resets the value of nchild field to 0. Figure 2 illustrates an instance with 8 nodes of a structure this code builds.

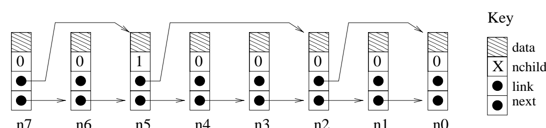

Fig. 2. Skip-List Pointer-Based Data Structure Example.

To understand the topology of the created data structure a compiler must be able to deduce the following:

1. The nodes are linked (linearly) along the next field.
2. At each insertion, the code increments the value of the nchild field starting with a 0 value.
3. Nodes with nchild = 1 and link == NULL have one child node along next.
4. Nodes with nchild = 1 and link $\neq$ NULL have 2 child nodes along next. The nodes satisfy the structural identity link = next.next.next.
5. When the field nchild transitions from 1 to 2 the node has 3 child nodes along next all of which have nchild set to 0.
6. When the condition in line 13 evaluates to true the code resets nchild to 0 while relinking the node along link. This link field jumps over 2 nodes along next and the node satisfies the structural identity link = next.next.

Using these facts, and more importantly retaining them in the internal abstract representation for the shape analysis, a compiler can recognize that the data structure is both acyclic along next and link but following both fields leads

to non-disjoint sets of nodes. Based on the structural properties the compiler can prove the `while` loop always terminates and that the statements in lines `14` and `15` are always safe.

The key to allow an analysis to deduce the facts enumerated above, lies in its ability to track the various "configurations" or combinations of values for `nchid`, `link` and `next`, in effect building a Finite-State-Machine (FSM) transitions for the various configurations and their properties. To accomplish this the analysis must be able to **trace all possible mappings context** of `b` to nodes of the data structure and record, *after the data structure has been manipulated*, the configuration and identities the newly modified nodes meet.

In this process the compiler uses *scan* loop analysis to recognize which sets of nodes in the internal shape analysis abstraction the various variables can point to. For this example the compiler determines that for the contexts reaching line `10` `b` must point to nodes in the abstraction following only the `link` fields and starting from the binding of the variables `n`. The compiler then uses the bindings to isolate and capture the modifications to the data structure. These updates to the internal shape representation generate other possible contexts the compiler needs to examine in subsequent iterations. For this example the compiler can *exhaustively prove* that for nodes pointed to by `b`, that satisfy the `b->link != NULL` only two *contexts* can reach `10` as shown below. In this illustration the field values denote node identifiers in the internal shape analysis abstract representation and structural identities are denoted by path expressions.

```
context1 = { id = s0,
  config = {nchild = 0, link = {s1}, next = {s2}
  prop: {link = next.next} acyclic(next), acyclic(link) }

context2 = { id = s3,
  config = {nchild = 1, link = {s4}, next = {s1}
    prop: {link = next.next.next} acyclic(next),acyclic(link)}
```

Using this data the compiler can propagate only the $context_2$ through the conditional section of the code in lines `14` through `16` leading to the creation of a new $context_1$. The compiler reaches a point where the bindings of contexts to the variables in the program is fixed. Since in any case the symbolic execution of these contexts preserves both *acyclicity* along `next` and `link` the compiler can therefore ensure termination of the `while` and preserve the abstract information regarding the data structure topology. For the example in figure 1 a possible abstract representation using a storage graph approach is depicted in figure 3. This representation is complemented with several boolean function per node indicating acyclicity and field interdependences. For example for node `s1` the predicate `link = next.next.next` holds.

This example suggest the inclusion of the knowledge derived from the basic techniques described in this paper into a shape and safety analysis algorithm. First, the analysis identifies which fields of the nodes of the data structure have a potential for affecting the topology. Next the analysis tracks the values of the various fields exhaustively across all possible execution paths using symbolic contexts. These symbolic contexts make references to the internal shape rep-

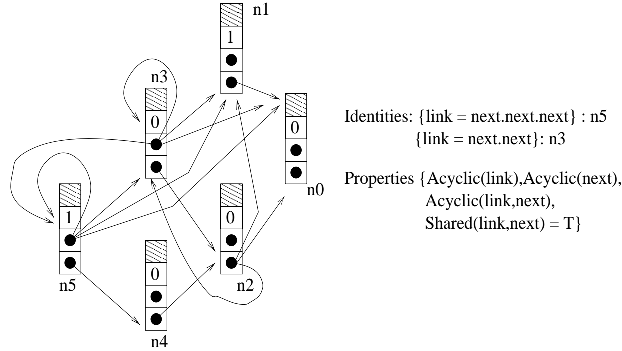

Fig. 3. Shape Graph Representation for Code in Figure 1.

resentation to capture acyclic and structural identities. Finally, the algorithm **assumes** properties about its data structure that guarantee termination of loop constructs. A compiler must later **verify** the properties uncovered during shape analysis to **ensure** they logically imply the properties assumed to derive them.

3 Basic Analyses

We now describe the set of analysis techniques and begin by outlining the set of assumptions they rely on. As we other approaches we assume, that the input program is type safe. For simplicity we also enforce that pointer handles hold addresses of user defined data structure types and not the address of fields within. We do not handle pointer arithmetic nor, for tractability, arrays of pointers. All the data structures are built with heap allocated nodes each of which has a discrete set of field names. For simplicity we also assume the data structures to be constructed by the repeated invocation of a set of identifiable functions.[2]

3.1 Structural Fields and Value Analysis

Certain fields of nodes of the data structure contain data that is intimately related to its structural identities. This case reveals itself when predicates that guard code that modifies the data structures test specific values for these fields. As such we define a field of a type declaration as *structural* if there exists at least one conditional statement in which the value of the field is tested, and the corresponding conditional statement guards the execution of a straight line code segment that (possibly) modifies the data structure. This conditions capture most of the relevant pointer fields of a data structure that are actively manipulated in the construction of the data structure. For non-pointer fields we require

[2] This constraint can be lifted in most cases by performing a call-graph analysis and identifying the functions that allocated and manipulate the data structures.

that a structural field must have all of its predicates in the form `ptr->field == <value>` where `<value>` is an integer value.

In many cases it is also useful, and possible, to determine statically the range of values a given structural field assumes. While for pointer fields we are only concerned with the two values {`nil, non-nill`}, for numeric integer fields we are interested in determining the set of values, and possibly the sequence(s) in which they are assumed.

In figure 4 we outline an algorithm that tracks the set of values for a given integer field and attempts to establish the "cycles" of sequences of values it assumes in essence trying to uncover the FSM of the values of each field. As with the symbolic analysis technique describe in a later section this algorithm assumes all modifications to fields in the data structures are performed via variables local to each procedure that manipulates the data structure.

The algorithm uses several auxiliary functions, namely:

- The function `PathRegions(`$\texttt{var}_i$`,` f_j`)` computes the control flow region of statements for which the local variables $\texttt{var}_i$ is used to update a numerical field f_j but not redefined. Such as region can encompass many control-flow paths that do not update $\texttt{var}_i$ or any of the numerical fields of interest. In these cases the control-flow can be contracted for algorithmic performance.
- The function `Transfer(`r_k`,`$\texttt{var}_i$`,` f_j`)` computes a symbolic transfer function along the control-flow paths in region r_k for variable $\texttt{var}_i$ and field f_j. This function symbolically evaluates the statements in the region folding relevant conditional statements into the symbolic transfer function. As a summarization result this function returns a tuple with the following elements: an `increment`; a `limit` and a `reset` value. When unable to derive constant values for the elements of this tuple, the algorithm simply returns an `unknown` symbolic value. When a given element is absent, however, the algorithm returns an `empty` indication. In its simplest implementation this function evaluates statements such as `var->field++` and predicates such as {`var->field == <value>`} and conservatively returns `unknown` when the updates and tests involve computations using temporary variables.
- The function `extractFSM(`f_j`, init(`f_j`),`T_*^*`)` determines the actual set of values the field f_j can assumes given the input set of symbolic transfer functions that update that particular field and the initial field values. The extracted FSM, other that the values and corresponding transitions, also labels the values of the field as *persistent* if across invocations of the code the fields can retain that specific value. Without accurate data the FSM consists of a simple node with the value `unknown`. Again, and in its simplest implementation this function can enforce that all transfer functions have the same increment value and a non-empty limit and reset values. A transfer function without a limit means that there is possibly a control flow path in the procedure that can increment the value of the field without bound. In this case the `extractFSM` function returns a FSM with unknown elements.

For the example in section 2 the algorithm outlined above would uncover a single path region defined by the statements in lines 12 through 17. For this

```
for all f_j ∈ structFields(type t_i) do
  1. compute S_i = PathRegions(var_i, f_j);
  2. if ∃ var_h ∈ S_i then
       mark f_j non-structural; continue;
  3. for each region r_k ∈ S_i do
       compute T^i_k = Transfer(r_k,var_i, f_j);
  4. if ∃ tuple t ∈ T^i_k with unknown then
       mark f_j non-structural; continue;
  5. extractFSM(f_j, init(f_j),T^*_*);
```

Fig. 4. Structural Field Values Analysis Algorithm.

region the algorithm could extract a single symbolic transfer function for the `nchild` field denoted by `nchild` $\mapsto$ `nchild + 1; if(nchild == 2) nchild` $\mapsto$ `0; else nchild` $\mapsto$ `nchild`; resulting in a tuple with `incr = 1; limit = 2; reset = 0`. Using this data and with an `init(nchild) =` $\{0\}$ the algorithm would uncover a FSM with 3 states corresponding the the values $\{0, 1, 2\}$ of which $\{0, 1\}$ are persistent and with transitions $0 \rightarrow 1; 1 \rightarrow 2; 2 \rightarrow 0$.

3.2 Node Configurations

It is often the case that nodes with different *configurations* in terms of nil and non-nil values for pointer fields or simple key numerical values occupy key places in the data structure. For this reason we define the *configuration* of a node as a specific combination of values of pointer and structural fields. We rely on the analysis in section 3.1 to define the set of persistent values for each field and hence define the maximum number of configurations. If the value analysis algorithm is incapable of uncovering a small number of values for a given non-pointer field the shape analysis algorithm ignores the corresponding field. This leads to a reduced number of configuration but possibly to inaccurate shape information.

3.3 Scan Loop Analysis

The body of a *scan* loop can only contain simple assignment statements of scalar variables (*i.e.,* non-pointer variables) or pointer assignment of the form `var = var->field`. A *scan* loop can have nested conditional statements and/or `break` or `continue` statements as illustrated below. One can trivially extend the definition of scan loops to include calls to functions such as `printf` or any functions whose call-chains include functions that do not modify the data structure.

Scan loops are pervasive in programs that construct sophisticated pointer-based data structures. Programmers typically use *scan* loops to position a small number of pointer handles into the data structures before performing updates. For example the `sparse` pointer-based code available from McCat Group [6] has 17 *scan* loops with at most 2 pointer variables. All these 17 loops can be statically analyzed for safety and termination as described in the next section.

```
while(p->next != NULL){
  if(p->data < 0)
    break;
  t = p;
  p = p->next;
}
```

The fundamental property of *scan* loops, is that they do not modify the data structure. This property allows the compiler to summarize their effects and "execute" them by simply matching the path expressions extracted from symbolic analysis against the abstract representation of the data structure. From the view point of abstract interpretation *scan* loops behave as *multi–valued* operation.

We use symbolic composition techniques to derive *path expressions* that reflect the symbolic bindings of pointer variables in *scan* loops. The algorithm derives symbolic path expressions for all possible entry and exit paths of the loop for the cases of zero-iterations and compute a symbolic closed-form expressions for one or more loop iterations. The number of these expressions is exponential in the nesting depth of the loop body and linear on the number of loop exit points. We use conservative symbolic path expression merge functions to limit the number of bindings for each pointer variables to 1.

For the example above our path expression extraction algorithm derives the binding $\texttt{p} \rightarrow \texttt{p}_{\texttt{in}}.(\texttt{next})^{+}$ where $\texttt{p}_{\texttt{in}}$ represents the value of `p` on entry of the loop and derives the binding $\texttt{p} \rightarrow \texttt{p}_{\texttt{in}}$ for the zero-iteration case. More importantly, however, is that the symbolic analysis can uncover the precise relationship between `p` and `t` as {`p = t->next`} on exit for the non-zero-iteration case.

3.4 Assumed Properties for Termination

Using the symbolic analysis of *scan* loops it is also possible to developed an algorithm that extracts conditions that guarantee the termination and safety of *scan* loops. We examine all the zero-iteration execution paths for initial safety conditions. Next we use inductive reasoning to validate the safety of a generic iteration based on safety assumptions for the previous iteration. To reason about the safety requirements of an iteration we extract the set of non-nil predicates each statement requires. In the case of conditionals we can also use the results of the test to generate new predicates that can ensure the safety of other statements.

For the code sample above the algorithm can derive that for the safe execution of the entire loop only the predicate $\{p_{in}!= \texttt{NULL}\}$ needs to hold at the loop entry. The algorithm can also determine that only the dereferencing of the predicate in the loop {`p->next != NULL`} header in the first iteration is potentially unsafe. Subsequent iterations use a new value of `p` assigned to the previous `p->next` expression, which by control flow has to be non-null. Finally the algorithm derives, whenever possible, shape properties that is guaranteed to imply the termination of the loop for all possible control flow paths. For the example above the property `Acyclic(next)` would guarantee termination of the loop.

As a by-product of the symbolic evaluation of each statement in the loop, a compiler can extract termination condition by deriving a symbolic transfer function of the scan loop. This transfer function which takes into account copies through temporary local variables and exposes which fields the loop uses for advancing through the data structures. On a typical null checking termination the algorithm derives conservative acyclicity properties along all of the traversed fields. For other termination conditions such as `p->next != p` (which indicates a self-loop terminated structure), the algorithm could hint the shape analysis algorithm that a node in the abstraction with a self-loop should be prevented from being summarized with other nodes.

3.5 Domain of Applicability - Segmented Codes

The techniques presented here are geared towards codes that are *segmented*, *i.e.,* they consists of a sequence of assignment or conditional statements and *scan* loops. *Segmented* codes allow our approach to handle the codes as if they were a sequence of conditional codes with straight-line sequences of statements, some of which multi-valued as is the case of *scan* loops. Fortunately codes that manipulate pointer-based data structures tend to be segmented. This is not surprising as programmers naturally structure their codes in phases that correspond to logic steps of the insertion/removal of nodes. Furthermore, both procedural abstractions and object-oriented paradigm promote this model.

4 Application Examples

We now describe the application of the base analysis techniques and suggest a shape representation using the information gathered by our techniques for the *jump-list* C code presented in section 2. We present results for both correct and "incorrect" constructions of the data structure. We assume the code repeatedly invokes the function `insert` and that the initial value for its argument `n` points to a node with both nil `link` and `next` fields.

For the correct construction code the various techniques presented here would uncover the information presented in figure 5. We manually performed a shape analysis algorithm (not presented here) that exploits this information for the materialization and abstraction steps and were able to capture the abstract shape graph presented in Section 2 (figure 3). Because of the need to trace all of the execution context the algorithmic complexity of this approach is expected to be high. In this experiment we have generated and traced 53 contexts and required 8 iterations to reach a fixed-point solution.

We now examine the algorithm's behavior for the trivial case where the programmer would simply remove the conditional statement in line `13`. In this case a shape analysis algorithm would immediately recognize that for the first analysis context $\#1 = \{b \rightarrow n1; n \rightarrow n1; t \rightarrow n0; id : \{b = n\}, \{t = b.next\}, \{t = n.next\}\}$ the dereferencing of the statement in line `14` would definitely generate an execution error.

```
Structural fields: {nchild, link, next }
Value Analysis:{nchild→{0,1},link→{nil,≠nil},next→{nil,≠nil}}

Configurations:
    c0 = {child = 0,link = nil,next = nil}
    c1 = {child = 0,link = nil,next ≠ nil}
    c2 = {child = 0,link ≠ nil,next = nil}
    c3 = {child = 0,link ≠ nil,next ≠ nil}
    c4 = {child = 1,link = nil,next = nil}
    c5 = {child = 1,link = nil,next ≠ nil}
    c6 = {child = 1,link ≠ nil,next = nil}
    c7 = {child = 1,link ≠ nil,next ≠ nil}

Scan Loops Safety and Termination:
    Body Transfer Function = { b_i → b_{i-1}.link }
    Closed Form Symbolic Expressions = { b → n_0.(link)* }
    Safety Requires (Zero-Trip): { {b_0 ≠ nil}, { n_0 ≠ nil } }
    Assumed Properties = { Acyclic(link, next)}

Shape Analysis Results:
    Number of Contexts: 53
    Number of Iterations: 8
    Stats.: Materializations: 86, Summarizations: 13
```

Fig. 5. Validation Analysis for Skip-List Example.

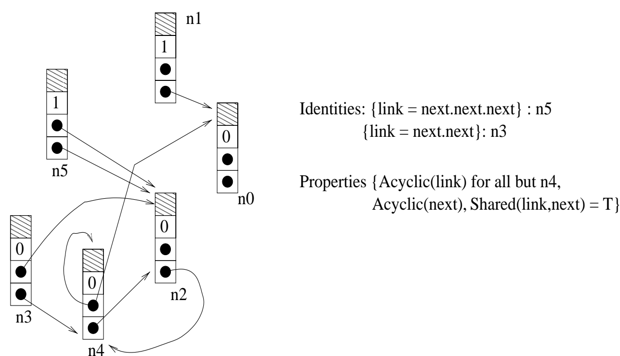

Fig. 6. Unintended Skip-List Construction Example.

We now examine the case where the programmer has swapped `link` with `next` in line `14` and instead used the sequence of instructions below.

```
13: if(b->nchild == 2){
14:   b->next->link = b->next;
15:   b->link = b->next->next;
16:   b->nchild = 0;
17: }
```

With this code a compiler can recognize that on the 2nd invocation the program creates a cycle in the data structure along the `link` field. More surprising is that this *true cycle* along `link` in $n4$ would not cause the program to go into an infinite loop!. Rather, the program would construct the ASG as outlined in figure 6 and with the properties shown. The analysis described in this would allow a shape analysis algorithm to realize this fact as while tracing the set of valid contexts it would verify that `b = n.(link)+` never visits nodes $n4$ but only the set node represented in the abstraction by $\{n1, n3, n5\}$. As such the compiler could verify that the scan loop assumption is still valid.

5 Related Work

We focus on related work for shape analysis of C programs both automated and with programmer assistance. We do not address work in pointer aliasing analysis (see *e.g.,* [3, 15]) or semi-automated program verification (see *e.g.,* [1, 11]).

5.1 Shape Analysis

The methods described by other researchers differ in terms of the precision in which nodes and their relationships are represented in *abstract-storage-graphs (ASGs)* [3]. Larus and Hilfinger [10] describe, possibly, the first automated shape analysis algorithm. This approach was refined by Zadeck *et al.* [2] by aggregating only nodes generated from the same heap allocation site. Plevyak *et al.* [12] addresses the issue of cycles by representing simple *invariants* between pointer fields. Sagiv *et al.,* [13, 14] describe a series of refinements to the naming scheme for nodes in the abstract storage as well as more sophisticated materialization and summarization operations. This refinement allows their approach to handle list-reversal type of operations and doubly-linked lists assuming those properties were known *a-priori*, that is before the reversal operations would be executed. Ghyia and Hendren [6] describe an approach that sacrifices precision for time and space complexity. They describe an interprocedural dataflow analysis in which

[3] For an example of a storeless alias approach see [5] where pointer relations are represented solely via path expressions.

for every pair of pointer variables, *pointer handles*, the analysis keeps track of *connectivity*, *direction* and *acyclicity*. Even in cases where the analysis yields incorrect shape, the information is still useful for application of transformations other than parallelism. Corbera *et al.* [4] have proposed an representation by allowing each statement in the program to be associated with more than a one ASG using invariant and property predicates at each program point to retain connectivity, direction and cycle information. Such expanded representation could use the wealth of symbolic information the analysis proposed in this paper has to offer to maintain the accuracy and reduce the possibly large space overhead multiple ASGs per program execution point imply.

5.2 Shape Specification and Verification

Hummel *et al.* [7, 8] describe an approach in which the programmer specifies, in the ADDS and ASAP languages, a set of data structures properties using *direction* and *orthogonality* attributes as well as structural invariants. The compiler is left with the task of checking if any of the properties is violated and every point of the execution of the program. The expressiveness power of both ADDS and ASAP languages is limited to the specification of *uniform* properties that must hold for all nodes of a given data structure. It is not therefore possible to isolate specific nodes of the data structure (assumed connected) and define properties that are different from properties of other nodes and is the case of a cycle-terminated linked list. This expressiveness constraint is due to decidability limitations of theorem proving in the presence of both universal and existential quantifiers.

Kuncak *et al.,* [9] describe a language that allows programmer to describe the referencing relationships of heap objects. The relationships determine the *role* of the various nodes of the data structures and allow an analysis tool to verify if the nodes comply with the *legal* alias relationships. Programmers explicitly augment the original code with role specifications at particular points in the code, in effect indicating to the analysis tool precisely where should the role specification be tested. This point-specific insertion is equivalent to choosing where to perform abstraction of the shape analysis, in itself, traditionally a hard problem.

6 Conclusion

In this paper we described four novel analysis techniques, namely, *structural fields*, *scan loops*, *assumed/verified shape properties* and *context tracing* to attain a high level of accuracy regarding the relationship between pointer handles that traverse pointer-based data structures. We have illustrated the application of the proposed analysis techniques to codes that build (correctly as well as incorrectly) sophisticated data structures that are beyond the reach of current approaches. This paper supports the thesis that compiler analysis algorithms must uncover and exploit information derived from conditional statements in the form of the techniques presented here if they are to substantially increase their accuracy.

References

1. T. Ball, R. Majumdar, T. Millstein and S. Rajamani. Automatic Predicate Abstraction of C Programs. In *Proc. of the ACM Conference on Programming Language Design and Implementation*, pages 203–213, ACM Press, New York, NY, June 2001.
2. D. Chase, M. Wegman and F. Zadek. Analysis of Pointers and Structures. In *Proc. of the ACM Conference on Program Language Design and Implementation*, pages 296–310, ACM Press, New York, NY, June 1990.
3. J. Choi, M. Burke, and P. Carini. Efficient Flow-Sensitive Interprocedural Computation of Pointer-induced Aliases and Side Effects. In *Proc. of the Twentieth Annual ACM Symp. on the Principles of Programming Languages*, ACM Press, pages 232–245, New York, NY, January 1993.
4. F. Corbera, R. Asenjo, and E. Zapata. Accurate Shape Analysis for Recursive Data Structures. In *Proc. of the Thirteenth Workshop on Languages and Compilers for Parallel Computing*, August 2000.
5. A. Deutsh. Interprocedural may-alias analysis for pointers: Beyond k-limiting. In *Proc. of the ACM Conference on Program Language Design and Implementation*, pages 230–241, ACM Press, New York, NY, June 1994.
6. R. Ghiya and L. Hendren. Is it a Tree, a DAG, or a Cyclic Graph? a Shape Analysis for Heap-directed Pointers in C. In *Proc. of the Twenty-third Annual ACM Symp. on the Principles of Programming Languages*, ACM Press, pages 1–15, New York, NY, January 1996.
7. L. Hendren, J. Hummel, and A. Nicolau. A General Data Dependence Test for Dynamic, Pointer-based Data Structures. In *Proc. of the ACM Conference on Program Language Design and Implementation*, ACM Press, pages 218–229, New York, NY, June 1994.
8. J.Hummel, L. Hendren, and A. Nicolau. A language for conveying the aliasing properties of pointer-based data structures. In *Proc. of the 8th International Parallel Processing Symposium*, pages 218–229, IEEE Computer Society Press, Los Alamitos, CA, April 1994.
9. V. Kuncak, P. Lam, and M. Rinard. Role Analysis. In *Proc. of the Twenty-nineth Annual ACM Symp. on the Principles of Programming Languages*, ACM Press, pages 17–32, New York, NY, 2002.
10. J. Larus and P. Hilfinger. Detecting Conflicts between Structure Accesses. In *Proc. of the ACM Conference on Program Language Design and Implementation*, ACM Press, pages 21–34, New York, NY, June 1988.
11. G. Necula and P. Lee. The Design and Implementation of a Certifying Compiler. In *Proc. of the ACM Conference on Programming Language Design and Implementation*, ACM Press, pages 333–344, New York, NY, 1998.
12. J. Plevyak, V. Karamcheti, and A. Chien. Analysis of Dynamic Structures for Efficient Parallel Execution. In *Proc. of the Sixth Workshop on Languages and Compilers for Parallel Computing*, Published as Lecture Notes in Computer Science (LNCS) Vol. 768, pages 37–57. Springer-Verlag, 1993.
13. M. Sagiv, T. Reps and R. Wilhelm. Parametric Shape Analysis via 3-valued Logic. In *Proc. of the Twenty-sixth Annual ACM Symp. on the Principles of Programming Languages*, ACM Press, New York, NY, January 1999.
14. 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.

15. R. Wilson and M. Lam. Efficient Context-Sensitive Pointer Analysis for C Programs. In *Proc. of the ACM Conference on Programming Language Design and Implementation*, ACM Press, pages 1–12, New York, NY, June 1995.