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 variables that traverse the data structure. *Assumed/verified* property analysis derives sufficient shape properties that guarantee termination of loops. These properties must be verified during shape analysis for consistency. *Context tracing* allows the compiler to isolate data structure nodes by evaluation of conditional statements and hence preserve shape accuracy.
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 accuracy. In this paper we illustrate the application of the proposed techniques to codes that build (correctly as well as incorrectly) sophisticated 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, acyclic graphs or cyclic graphs. Even when the overall structure has cycles, it might be important to understand that the induced topology traversing only specific fields is 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. In this process the algorithm tries to uncover important topological properties by looking at the relationships between the nodes the code manipulates. As pointer-based data structures have no predefined dimensions, compilers must resort to 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 the conservative, and often incorrect, determination of cyclic data structures.

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 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 on the 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 under the assumption that the data structure is acyclic along `next` to ascertain termination properties of other constructs and later verify the original acyclicity assumption.

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

These examples illustrate that programmers fundamentally encode "state" in their programs via conditionals and loop constructs. 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 analysis and safety algorithms should exploit the information conveyed in these statements.

[1] Although not a necessary condition as the programmer might have inserted a sentinel value that prevents the code from ever reaching a section of the data structure where there is a cycle.

This paper presents a set of symbolic analysis 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.
- **Scan Loops:** Symbolic execution of loops that only traverse (but do not modify) the structure and 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 propagates contexts, *i.e.,* the set of bindings of variables to nodes in the data structure throughout the program and uses conditionals to prune the sets of nodes pointer variables can point to at particular program points.

Part of the contribution of this paper is in presenting techniques that can enhance existing shape/safety analysis approaches. For example, identifying nodes with distinct configurations can help current *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.*

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. *A fundamental contribution of this paper is to show that in order to increase the accuracy of shape and safety analysis algorithms, compilers must exploit the knowledge encoded in conditional statements.*

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 figure1, builds a data structure by the successive invocation of the function `insert`. In this code the function call `new_node(int)` allocates indirectly through the `malloc` function a node that is unaliased with any of the nodes in the data structure. For the sake of 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`.

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

01void 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.

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

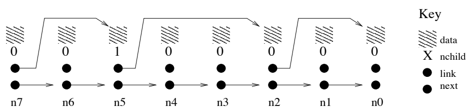


Fig. 2. Skip-List Pointer-Based Data Structure Example.

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. At each insertion, the code increments the value of the `nchild` field starting with a 0 value.
2. Nodes with `nchild = 1` and `link == NULL` have one child node along `next`.
3. Nodes with `nchild = 1` and `link` $\neq$ `NULL` have 2 child nodes along `next`. The nodes satisfy the structural identity `link = next.next.next`.
4. 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.
5. When the conditional statement 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, the 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 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 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 ids 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-

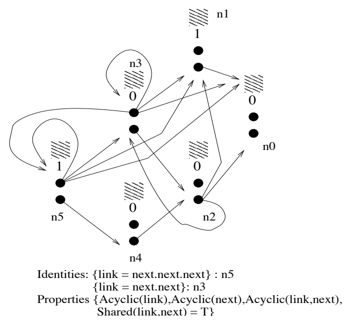

n1
1
n3
0
0
n0
1
0
0
n5
n2
n4
Identities: {link = next.next.next} : n5
{link = next.next}: n3
Properties {Acyclic(link),Acyclic(next),Acyclic(link,next),
Shared(link,next) = T}

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. The compiler must later **feedback the properties uncovered** during shape analysis to **ensure** they logically imply the properties assumed to derive them in the first place.

3 Basic Analyses

3.1 Structural Fields and Value Analysis

Certain fields of nodes of the data structure contains 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 used, and the corresponding conditional statement guards the execution of a straight line code segment that possibly modifies the data structure. For non-pointer fields we further require that a structural field must have all of the predicates of the form `ptr->field == <value>` where `<value>` is an integer value. For the example in figure 1 this algorithm finds the structural fields to be {`nchild, link, next`}.

Figure 4 depicts the algorithm that tracks the set of values a given structural field can assume. We focus on integer values only. The algorithm performs a simple fixed point computation over the control-flow graph determining for each statement the set of possible values for each field. Whenever possible the algorithm also evaluates conditional to rule out transitions between values of nodes, in essence attempting to find the underlying FSM for the values of each field. The algorithm also labels each value as persistence if across invocations of the code the field can retain that specific value.

For the example in section 2 the algorithm uncovers for the `nchild` field a FSM with 3 states corresponding the the values $\{0, 1, 2\}$ of which $\{0, 1\}$ are persistent and with transitions $0 \rightarrow 1; 1 \rightarrow 2; 2 \rightarrow 0$.

```
for all $f_i \in structFields(typet_i)$ do {
  init FSM($f_i$) with value from allocation sites;
  for every value $v_i \in V$ do
    for all CFG path $p_i \in Prog$ do
      for every value $v_i \in V$ do {
        for every statement $s_i$ along $p_i$ do
          if($s_i$ = var-> $f_i$ += <int const>)
            update FSM($f_i$) with $v_i \rightarrow v_i + const$;
          if($s_i$ = var-> $f_i$ += <unknownt>)
            update FSM($f_i$) with $v_i \rightarrow unknown$;
      }
    }
    mark values in $V$ as persistent;
}
```

Fig. 4. Structural Field Values Analysis Algorithm.

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. The number of configurations dictates the maximum number of nodes in the abstract representation of the data structure. 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 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.

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

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 McGill McCat Compiler Group has 64 *scan* loops with a maximum of 2 pointer variables. Of these 58 can be statically analyzed for *safety* as described in the next section.

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 this symbolic analysis against the abstract representation of the data structure. From the view point of abstract interpretation *scan* loops *behave as multi–value traversal operation.*

We use symbolic composition techniques to derive *path expressions* that reflect the symbolic bindings of pointer variables. The *scan* loop analysis algorithm derives symbolic path expressions for all possible entry and exit paths of the loop for the cases on zero-iterations and compute a symbolic closure for one or more 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.

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, the symbolic analysis can uncover the precise relationship `p = t->next` for the non-zero-iteration case. This ability to uncover precise relationships between pointer handles greatly increases the a compiler materialization algorithm to maintain accuracy and address abnormal program behavior.

The reader should not dismiss the lack of generality of *scan* loops. As our empirical experimental results show, *scan* loops are pervasive in programs that manipulated sophisticated pointer-based data structures. *Scan* loops and the predicates we can extract from them are instrumental in discriminating the insertion/deletion points in the data structures. The fact that programmers use them abundantly is a clear indication of their importance for compiler to be able to reason accurately about their behavior.

3.4 Assumed Properties for Termination

Using the symbolic analysis of *scan* loops we 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 an inductive reasoning to validate the safety of a generic iteration based on the 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 conditional we can also use the results of the test to generate new predicates that can ensure the safety of otehr statements. To trace symbolically the expression in a given loop iteration the algorithm builds the control flow of the loop body and examines the required predicates for safe execution of the dereferencing operations.

For he code sample above the algorithm derives that for the safe execution of the entire loop only the predicate p_{in}`!= NULL` needs to hold at the loop entry. The algorithm also determines 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. To extract this information the algorithm computes a symbolic transfer function (which takes into account copies through temporaries) and determines which fields does the loop use 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 compiler could possibly hint the shape analysis algorithm that a node in the abstraction with a self-loop should be prevented from being summarized with other non-similar nodes. It is up to the shape analysis algorithm to verify that indeed these properties hold at the point where they were assumed.

3.5 Segmented Codes

We gear our analysis to codes that are *segmented*. A *segmented* code consists of a sequence of assignment or conditional statements and *scan* loops. *Segmented* codes allow our implementation to handle the code as if it were a sequence of conditional codes with straight-line sequences of statements.

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. In addition both procedural abstractions and object-oriented paradigm promote and facilitate this model.

4 Application Examples

We now describe the application of the base techniques and shape analysis algorithm for the *jump-list* code presented in section 2 for **both correct and incorrect** constructions. We use the source C code presented in section 2. We have assumed the code repeatedly invokes the function `insert` and 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 "algorithm" is expected to be high. In this experiment we have generated and traced 53 contexts and required 8 as indicated below suggesting a number of iterations that is exponential in the depth of the abstract representation.

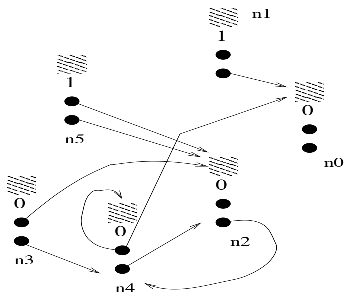
Structural fields: {nchild, link, next }
Value Analysis:{nchild→{0,1},link→{nil,≠nil},next→{nil,≠nil}}

Configurations:
    c_0 = {child = 0,link = nil,next = nil}
    c_1 = {child = 0,link = nil,next ≠ nil}
    c_2 = {child = 0,link ≠ nil,next = nil}
    c_3 = {child = 0,link ≠ nil,next ≠ nil}
    c_4 = {child = 1,link = nil,next = nil}
    c_5 = {child = 1,link = nil,next ≠ nil}
    c_6 = {child = 1,link ≠ nil,next = nil}
    c_7 = {child = 1,link ≠ nil,next ≠ nil}

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

Shape Analysis Results:
    Number of Contexts: 53
    Number of Iterations: 8
    Stats.: Materializations: 86
            Summarizations: 13
            Nodes,Edges: $\{6, 13\}$

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



Identities: {link = next.next.next} : n5
{link = next.next}: n3
Properties {Acyclic(link) for all but n4 ,Acyclic(next),
Shared(link,next) = T}

Fig. 6. Unintended Skip-List Construction Example.

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.

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 2^{nd} 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 $\{n1, n3, n5\}$. 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, 14]) or semi-automated program verification (see *e.g.,* [10, 1]).

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* Larus and Hilfinger [9] possibly describe the first automated shape analysis algorithm. This approach was refined Zadeck *et al.* [2] to aggregate only nodes generated from the same allocation site. Plevyak *et al.* [11] addresses the issue of cycles by representing simple *invariants* between fields of nodes. Sagiv *et al.,* [13, 12] describe a series of refinement to the naming scheme for nodes in the abstract storage and materialization and summarization of nodes which increases the precision. This refinement allows their approach to handle list-reversal type of operations and doubly-linked lists. Ghyia and Hendren [5] describe an approach that sacrifices precision for time and space complexity. They describe an interprocedural dataflow analysis in which for every pair of pointer variables *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 expanded the storage 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.

5.2 Shape Specification and Verification

Hummel *et al.* [7, 6] 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 structure invariants. The compiler is left with the task of checking if any of the statement in the program violate the axioms and reports if so. The expressiveness power of these languages does not allow for instance to select nodes with particular properties (*e.g.,* as is the case of a cycle-terminated linked list) we all of the properties need to apply to all nodes of the same data structure. The implicit assumption is that properties are applicable only to non-nil fields of the objects. This restriction is due to decidability limitations of theorem proving. Kuncak *et al.,* [8] 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 a analysis tool to verify if the nodes comply with the *legal* alias relationships. Programmers augment the original code with role specification at particular point in the code, in effect indicating to the analysis tool precisely where should the role specification be tested. This points-specific insertion is equivalent to choosing where to perform abstraction of the shape analysis and has traditionally been 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. 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 *SIGPLAN '01 Conference on Programming Language Design and Implementation,* SIGPLAN Notices 36(6), pages 203–213, New York, NY, June 2001. ACM Press.
2. D. Chase, M. Wegman, and F. Zadek. Analysis of pointers and structures. In *Proceedings of the ACM SIGPLAN '90 Conference on Program Language Design and Implementation*, pages 296–310, New York, NY, June 1990. ACM Press.
3. J. Choi, M. Burke, and P. Carini. Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side effects. In *Proceedings of the Twentieth Annual ACM Symposium on the Principles of Programming Languages*, pages 232–245, New York, NY, January 1993. ACM, ACM Press.

4. F. Corbera, R. Asenjo, and E.L. Zapata. Accurate shape analysis for recursive data structures. In *Proc. of the Thirteenth Workshop on Languages and Compilers for Parallel Computing*, August 2000.
5. 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 *Proceedings of the Twenty-third Annual ACM Symposium on the Principles of Programming Languages*, pages 1–15, New York, NY, January 1996. ACM Press.
6. L. Hendren, J. Hummel, and A. Nicolau. A general data dependence test for dynamic, pointer-based data structures. In *Proceedings of the ACM SIGPLAN '94 Conference on Program Language Design and Implementation*, pages 218–229, New York, NY, June 1994. ACM Press.
7. J.Hummel, L. Hendren, and A. Nicolau. A language for conveying the aliasing properties of pointer-based data structures. In *Proceedings of the 8th International Parallel Processing Symposium*, pages 218–229, Los Alamitos, CA, April 1994. IEEE Computer Society Press.
8. V. Kuncak, P. Lam, and M. Rinard. Role analysis. In *Proceedings of the Twenty-nineth Annual ACM Symposium on the Principles of Programming Languages*, pages 17–32, New York, NY, 2002. ACM Press.
9. J. Larus and P. Hilfinger. Detecting conflicts between structure accesses. In *Proceedings of the ACM SIGPLAN '88 Conference on Program Language Design and Implementation*, pages 21–34, New York, NY, June 1988. ACM Press.
10. G. Necula and P. Lee. The designa nd implementation of a certifying compiler. In *SIGPLAN '98 Conference on Programming Language Design and Implementation,* SIGPLAN Notices 33(6), pages 333–344, New York, NY, 1998. ACM Press.
11. 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*, volume 768, pages 37–57. Springer-Verlag, 1993.
12. M. Sagiv Mand T. Reps and R. Wilhelm. Parametric shape analysis via 3-valued logic. In *Proceedings of the Twenty-sixth Annual ACM Symposium on the Principles of Programming Languages*, New York, NY, January 1999. ACM Press.
13. 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.
14. R. Wilson and M. Lam. Efficient context-sensitive pointer analysis for C programs. In *SIGPLAN '95 Conference on Programming Language Design and Implementation,* SIGPLAN Notices 30(6), pages 1–12, New York, NY, June 1995. ACM Press.