To appear in Proceedings of the Twenty Third ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages, St. Petersburg Beach, Florida, January 21–24, 1996. ©1996 ACM (see notice below).

Is it a Tree, a DAG, or a Cyclic Graph?

A Shape Analysis for Heap-Directed Pointers in C *

Rakesh Ghiya and Laurie J. Hendren
School of Computer Science, McGill University
Montréal, Québec, CANADA H3A 2A7
{ghiya,hendren}@cs.mcgill.ca

Abstract

This paper reports on the design and implementation of a practical shape analysis for C. The purpose of the analysis is to aid in the disambiguation of heap-allocated data structures by estimating the shape (*Tree*, *DAG*, or *Cyclic Graph*) of the data structure accessible from each heap-directed pointer. This shape information can be used to improve dependence testing and in parallelization, and to guide the choice of more complex heap analyses.

The method has been implemented as a context-sensitive interprocedural analysis in the McCAT compiler. Experimental results and observations are given for 16 benchmark programs. These results show that the analysis gives accurate and useful results for an important group of applications.

1 Introduction and Related Work

Pointer analyses are of critical importance for optimizing/parallelizing compilers that support languages like C, C++ and FORTRAN90. The pointer analysis problem can be divided into two distinct subproblems: (i) analyzing pointers that point to statically-allocated objects (typically on the stack), and (ii) analyzing pointers that point to dynamically-allocated objects (typically in the heap). Pointers to stack objects are usually obtained using the address-of (`&a`) operator, while pointers to heap-objects are obtained using a memory allocating function like `malloc()`. Henceforth we will refer to these analyses respectively as *stack analysis* and *heap analysis*.

A considerable amount of work has been done in both of these areas. Initially, the focus was on heap analysis alone, for languages like Lisp and Scheme or for toy imperative languages that did not include all of the complexities of C [3, 4, 5, 12, 17, 18, 19, 20, 22, 23, 27, 30]. A recent trend has been to actually implement pointer analyses in real C and FORTRAN90 compilers, and to examine if practical and useful solutions can be obtained. The most recently proposed (and implemented) approaches [1, 6, 26, 29, 32, 33], mostly focus on the stack problem and only give conservative estimates for the heap problem. These approaches exploit the fact that pointer targets on the stack always possess a compile-time name. Using this property stack pointer relationships are accurately captured as points-to pairs [6] of the form `(p,x)` (denoting pointer variable `p` points to the data object `x`), or alternatively as alias pairs [26] of the form `(*p,x)` (denoting `*p` and `x` are aliased).

Unfortunately heap objects do not have any fixed name during their lifetime, as they are dynamically allocated and are inherently anonymous. Hence various schemes are used to name them: naming them according to the place (statement) in the program where they are allocated [3, 26, 29] or further qualifying these names with procedure strings to distinguish between objects allocated at the same statement but along different calling chains [1, 33]. These naming schemes can give the same name to completely unrelated heap objects, and hence tend to provide conservative results, and they cannot compute shape information.

Instead of adapting heap analysis to an abstraction actually designed for stack analysis (by naming heap objects), our approach is to decouple the two analyses and provide heap analyses that approximate relationships and attributes of heap-directed pointers. In our McCAT compiler we first perform a stack analysis called *points-to analysis* [6, 8] that resolves pointer relationships on the stack. It uses one abstract lo-

*This work supported by NSERC, FCAR, and the EPPP project (financed by Industry Canada, Alex Parallel Computers, Digital Equipment Canada, IBM Canada and the Centre de recherche informatique de Montréal).

cation called *heap* for all heap locations and reports all heap-directed pointers to be pointing to it. Depending on the characteristics of the program under analysis, we can then apply the appropriate heap analysis which gives more precise information about the relationships between heap-directed pointers. For programs with few uses of the heap, the *level-0* or *points-to* analysis is enough. For programs that use a number of dynamically-allocated arrays and/or non-recursive structures, the *level-1* or *connection* analysis is used which identifies if two heap-directed pointers point to the same structure [10, 11]. Scientific applications written in C typically exhibit this feature, as they use a number of disjoint dynamically-allocated arrays.

This paper focuses on the *level-2* heap analysis: *shape analysis*. The goal of shape analysis is to estimate the shape of the data structure accessible from a given heap-directed pointer: is it tree-like, DAG-like or a general graph containing cycles? More specifically, our focus is on identifying unaliased tree-like and acyclic DAG-like data structures built by the program, and provide conservative estimates otherwise. Shape analysis is designed for programs that primarily use recursive data structures, or a combination of arrays and recursive data structures. Shape information can be gainfully exploited to parallelize such programs [12, 17, 22, 24], or to apply optimizing transformations like loop unrolling [15] and software pipelining [16] on them.

Much of the previous work on heap analysis also primarily focused on some variation of the problem of shape estimation [3, 5, 17, 19, 21, 23, 28, 30]. In general, all of these approaches use a more complex abstraction than the one given in this paper, and as a result they *may* find a more precise answer. Rather than look for a complex abstraction, our approach is to start with simple abstractions that can be implemented in real compilers and examine the usefulness of these simple abstractions with respect to a set of representative benchmark programs. Thus, the main contribution of our work is the design and implementation of *practical* abstractions to perform shape analysis for an important class of C programs. We believe we are the first to implement such a method in an optimizing/parallelizing C compiler and collect empirical results for real C programs. Our results indicate that our shape analysis provides accurate results for programs that build tree-like and DAG-like data structures in a compositional manner.

The rest of the paper is organized as follows. In Section 2 we give a high-level overview of the analysis rules assuming a simple model where stack-directed pointers and heap-directed pointers are clearly separated. The method has been fully implemented in the McCAT compiler as a context-sensitive interprocedural analysis for C programs. In Section 3 we give a brief overview of our implementation of this method and discuss the most pertinent features. We present some empirical data in Section 4, to evaluate the cost and effectiveness of shape analysis. Conclusions and some future directions are given in Section 5.

2 Analysis Rules

The shape analysis is actually composed of three *store-less* [4] abstractions that work together and are computed together for each program point. For each heap-directed pointer, we approximate the attribute *shape*, and for each pair of heap-directed pointers we approximate the *direction* and *interference* relationships between them. These three abstractions are defined formally as follows.

Definition 2.1 *Given any heap-directed pointer* `p`*, the* shape *attribute* `p.shape` *is* Tree*, if in the data structure accessible from* `p` *there is a unique (possibly empty) (access) path between any two nodes (heap objects) belonging to it. It is considered to be* DAG *(directed acyclic graph), if there can be more than one path between any two nodes in this data structure, but there is no path from a node to itself (i.e. it is acyclic). If this data structure contains a node having a path to itself,* `p.shape` *is considered to be* Cycle*. Note that, as lists are a special case of tree data structures, their shape is also considered as* Tree*.*

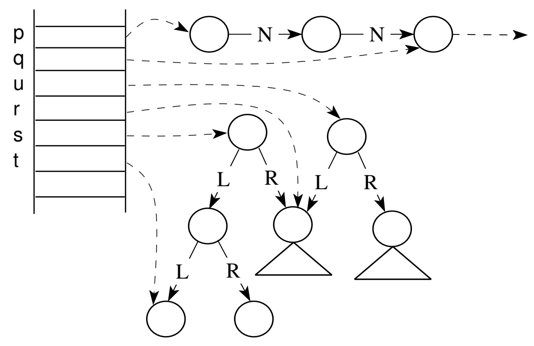
Definition 2.2 *Given any two heap-directed pointers* `p` *and* `q`*,* direction matrix `D` *captures the following relationships between them:*

- `D[p,q] = 1` *: An access path* possibly *exists in the heap,* from *the heap object pointed to by pointer* `p`*,* to *the heap object pointed to by pointer* `q`*. In this case, we simply state that pointer* `p` *has a path to pointer* `q`*.*
- `D[p,q] = 0` *: No access path exists* from *the heap object pointed to by* `p`*,* to *the heap object pointed to by* `q`*.*

Definition 2.3 *Given any two heap-directed pointers* `p` *and* `q`*,* interference matrix `I` *captures the following relationships between them:*

- `I[p,q] = 1` *: A common heap object can be* possibly *accessed, starting from pointers* `p` *and* `q`*. In this case, we state that* `p` *and* `q` *can* interfere*.*
- `I[p,q] = 0` *: No common heap object can be accessed, starting from pointers* `p` *and* `q`*. In this case, we state that* `p` *and* `q` *do not* interfere*.*

Direction relationships are used to actually estimate the shape attributes, while interference relationships are used for safely calculating direction relationships.



(a) Heap Structure

	p	q	r	s	t	u
p	1	1	0	0	0	0
q	0	1	0	0	0	0
r	0	0	1	0	0	0
s	0	0	1	1	1	0
t	0	0	0	0	1	0
u	0	0	1	0	0	1

(b) Direction Matrix

	p	q	r	s	t	u
p	1	1	0	0	0	0
q	1	1	0	0	0	0
r	0	0	1	1	0	1
s	0	0	1	1	1	1
t	0	0	0	1	1	0
u	0	0	1	1	0	1

(c) Interference Matrix

Figure 1: Example Direction and Interference Matrices

2.1 Illustrative Examples

We illustrate the direction and interference abstractions in Figure 1. Part (a) shows the heap structure at a program point, while parts (b) and (c) show the direction and interference matrices for it.

In Figure 1, an access path exists from pointer `p` to `q`, and also from pointer `s` to `t`, so the entries `D[p,q]` and `D[s,t]` are set to one. No access path exists from pointer `q` to `p`, or from pointer `r` to `t`, so the entries `D[q,p]` and `D[r,t]` are set to zero. Further, no path exists from pointer `s` to `u` and vice versa, so both the entries `D[s,u]` and `D[u,s]` are set to zero. However starting from both `u` and `s` the heap object pointed to by `r` can be accessed. To indicate this, the interference matrix entries `I[s,u]` and `I[u,s]` are set to one. This example also illustrates that: (i) direction relationships are not symmetric, (ii) interference relationships are symmetric, and (iii) interference relationships form a superset of direction relationships. The second property is used to reduce the storage requirement for the interference matrix by half, in the actual implementation. The third property follows from the fact that if an access path exists from pointer `p` to `q`, then they can also both access the object pointed to by `q`.

We now demonstrate how direction relationships help estimate the shape of heap data structures. In Figure 2, initially we have both `p.shape` and `q.shape` as *Tree*. Further `D[q,p]` is one, as there exists a path from `q` to `p` through the `next` link. The statement `p->prev = q`, sets up a path from `p` to `q` through the `prev` link. From direction matrix information we know that a path already existed from `q` to `p`, and now a path is being set also from `p` to `q`. Thus we can deduce the creation of a cycle between heap objects pointed to by `p` and `q`. Thus, after the statement, `D[p,q] = 1`, `D[q,p] = 1`, `p.shape` = *Cycle* and `q.shape` = *Cycle*.

It should be noted that for a heap-directed pointer `p`, `p.shape` only abstracts the shape of the data structure *accessible* from `p` and not the overall shape of the data structure pointed to by `p`. For example, in Figure 3, the overall shape of the data structure pointed to by `p` and `q` is *DAG*. However, if only the part of the data structure accessible from `p` or `q` is considered, its shape is *Tree*. So we have both `p.shape` and `q.shape` as *Tree*.

Knowledge about the shape of the data structure accessible from a heap-directed pointer, provides crucial information for disambiguating heap accesses originating from it. For a pointer `p`, if `p.shape` is *Tree*, then any two accesses of the form `p->f` and `p->g` will always lead to disjoint subpieces of the tree (assuming `f` and `g` are distinct fields). If `p.shape` is *DAG*, then two distinct field accesses `p->f->f` and `p->g` can lead to a com-

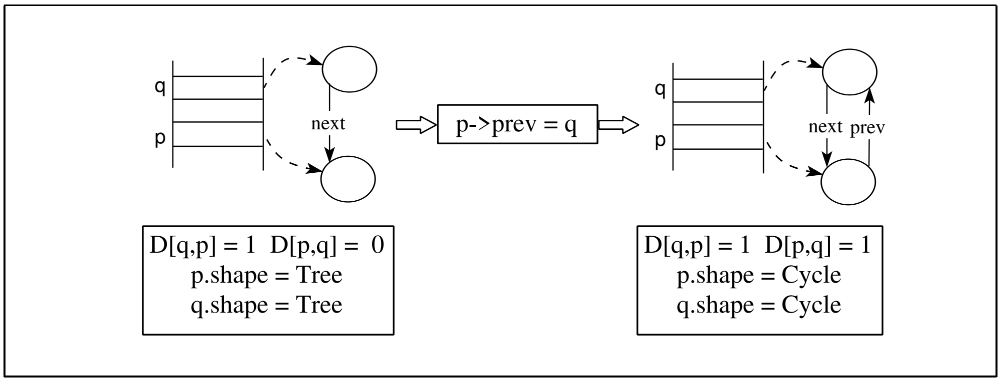


Figure 2: Example Demonstrating Shape Estimation

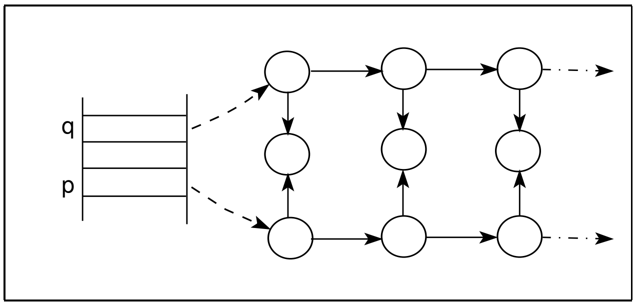


Figure 3: Estimating Shape with *accessibility* Criterion

mon heap object, as in Figure 4. However, if a dag-like structure is traversed using a sequence of links, every subsequence visits a distinct node. This information can be used to disambiguate heap accesses in different iterations of a loop, or different recursive calls, traversing such a data structure. Finally, if `p.shape` happens to be *Cycle*, we have effectively no information to disambiguate heap accesses originating from `p`.

Thus, the goal of shape analysis is to identify tree-like and dag-like data structures, and to retain this information as long as possible, during the analysis. We now present the rules to calculate direction and interference matrix abstractions, and to estimate shape information using them.

2.2 Analysis of Basic Statements

The McCAT compiler translates input C programs into a structured and compositional intermediate form called SIMPLE [31] . Using this form, there are eight basic statements that can access or modify heap data structures as listed in Figure 5(a). Variables `p` and `q` and the field `f` are of pointer type, variable `k` is of integer type, and `op` denotes the `+` and `-` operations. The overall structure of the analysis is shown in Figure 5(b). Given the direction and interference matrices `D` and `I` at program point `x`, before the given statement, we compute the matrices D_n and I_n at program point `y`. Additionally, we have the attribute matrix A, where for a pointer `p`, `A[p]` gives its shape attribute. The attribute matrix after the statement is represented as A_n.

For each statement, we compute the sets of direction and interference relationships it kills and generates. Using these sets, the new matrices D_n and I_n are computed as shown in Figure 5(c). Note that the elements in the gen and kill sets are denoted as D(p,q) for direction relationships, and I(p,q) for interference relationships. Thus a gen set of the form { D(x,y), D(y,z) }, indicates that the corresponding entries in the output direction matrix (D_n[x,y] and D_n[y,z]) should be set to one. We also compute the set of pointers H_s, whose shape attribute can be modified by the given statement. Another attribute matrix A_c is used to store the changed attribute of pointers belonging to the set H_s. The attribute matrix A_n is then computed using the matrices A and A_c as shown in Figure 5(c).

Let H be the set of pointers whose relationships/attributes are abstracted by matrices D, I and A. Assume that these pointers can only point to heap objects or to `NULL`. Further assume that updating an interference matrix entry I[p,q], implies identically updating the entry I[q,p]. This assumption is rendered valid due to the symmetric nature of interference relationships.

The actual analysis rules can be divided into three groups: (1) allocations, (2) pointer assignments, and (3) structure updates. In the following subsections we discuss the three rules.

2.2.1 Allocating new heap cells

`p = malloc()` : Pointer `p` points to a newly allocated heap object. All its existing relationships get killed. Pointer `p` now has an empty path to itself and and it also interferes with itself. This statement can change the attributes of only pointer `p`. Since the newly allocated object pointed to by `p` has no incoming or outgoing links, its shape attribute is Tree. This can be

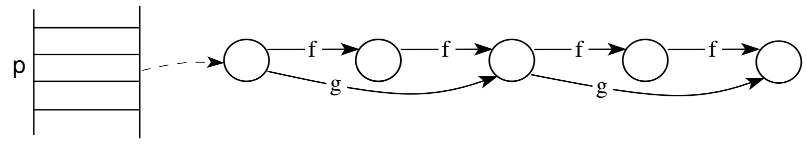


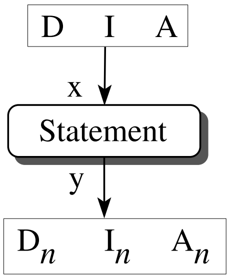
Figure 4: Acyclicity of Dag Data Structures

Allocations
1. `p = malloc();`

Pointer Assignments
2. `p = q;`
3. `p = q->f;`
4. `p = &(q->f);`
5. `p = q op k;`
6. `p = NULL;`

Structure Updates
7. `p->f = q;`
8. `p->f = NULL;`

(a) Basic statements



(b) Analysis Structure

Build the new matrices
$\forall$ r,s $\in$ H, D_n[r,s] = D[r,s], I_n[r,s] = I[r,s]
$\forall$ s $\in$ H, A_n[s] = A[s]

Delete killed relationships
$\forall$ entries D(r,s) $\in$ D_kill_set, D_n[r,s] = 0
$\forall$ entries I(r,s) $\in$ I_kill_set, I_n[r,s] = 0

Add generated relationships
$\forall$ entries D(r,s) $\in$ D_gen_set, D_n[r,s] = 1
$\forall$ entries I(r,s) $\in$ I_gen_set, I_n[r,s] = 1

Update shape attributes of affected pointers
Compute H_s and A_c
$\forall$ s $\in$ H_s, A_n[s] = A_c[s]

(c) General Form of Analysis Rules

Figure 5: The Overall Structure of the Analysis

summarized with the following rule.

$$\begin{aligned}
&\text{D_kill_set} = \{\ \text{D(p,s)} \mid \text{s} \in \text{H} \wedge \text{D[p,s]}\ \} \cup \\
&\qquad\qquad \{\ \text{D(s,p)} \mid \text{s} \in \text{H} \wedge \text{D[s,p]}\ \} \\
&\text{I_kill_set} = \{\ \text{I(p,s)} \mid \text{s} \in \text{H} \wedge \text{I[p,s]}\ \} \\
&\text{D_gen_set} = \{\ \text{D(p,p)}\ \} \quad \text{I_gen_set} = \{\ \text{I(p,p)}\ \} \\
&\text{H}_s = \{\ \text{p}\ \} \quad \text{A}_c\text{[p]} = \text{Tree}
\end{aligned}$$

Note that having D(p,p) in the gen set here simply implies that `p` presently points to a heap object. It does not imply that a cyclic data structure becomes accessible from `p` after this statement. In that case, we would also have A_c[p] = Cycle.

2.2.2 Pointer assignments

The next five basic heap statements (`p = q`, `p = q->f`, `p = &(q->f)`, `p = q op k` and `p = NULL`) update the stack-resident pointer `p`, and make it point to a new heap object. They kill all existing relationships of `p`, and can only change the shape attribute of pointer `p`. So the kill set and the set H_s for all these statements, are same as that for the statement `p = malloc()`. Below, we present the rules to calculate the gen set and the matrix A_c for these five statements.

`p = q` : Pointer `p` now points to the same heap object as `q`. It simply inherits the relationships and the shape attribute of pointer `q`. In case `q` presently points to `NULL`, `p` would also point to `NULL` after the statement. So we have D(p,p) and I(p,p) in the gen set, only if D[q,q] and I[q,q] are presently set to one (implying that `q` does not point to `NULL`). We have the following overall rule for the statement `p = q`.

$$\begin{aligned}
&\text{D_gen_set_from} = \{\ \text{D(s,p)} \mid \text{s} \in \text{H} \wedge \text{s} \neq \text{p} \wedge \text{D[s,q]}\ \} \\
&\text{D_gen_set_to} = \{\ \text{D(p,s)} \mid \text{s} \in \text{H} \wedge \text{s} \neq \text{p} \wedge \text{D[q,s]}\ \} \cup \\
&\qquad\qquad \{\ \text{D(p,p)} \mid \text{D[q,q]}\ \} \\
&\text{I_gen_set} = \{\ \text{I(p,s)} \mid \text{s} \in \text{H} \wedge \text{s} \neq \text{p} \wedge \text{I[q,s]}\ \} \cup \\
&\qquad\qquad \{\ \text{I(p,p)} \mid \text{I[q,q]}\ \} \\
&\text{D_gen_set} = \text{D_gen_set_from} \cup \text{D_gen_set_to} \\
&\text{H}_s = \{\ \text{p}\ \} \quad \text{A}_c\text{[p]} = \text{A[q]}
\end{aligned}$$

For purposes of our analysis we consider a pointer pointing to a specific field or at a specific offset of a heap object, to be pointing to the object itself. With this assumption, the statements `p = &(q->f)` and `p = q op k` are equivalent to the statement `p = q` for shape analysis and are analyzed using the same rule.

The statement `p = NULL` kills all relationships of `p` and does not generate any new relationships. Since `p` points to `NULL` after the statement, shape attribute is not relevant to it. As a default case, it is set as Tree.

`p = q->f` : This statement makes pointer `p` point to the heap object accessible from pointer `q` through the link `f`, as shown in Figure 6.[1] It generates following types of relationships: (i) new direction relationships because of pointer `p` having a path *from/to* other pointers, and (ii) new interference relationships with respect to pointer `p`.

From **Relationships** :
After the statement `p = q->f`, `p` will have a path *from* all pointers that presently have a path to `q` (`u`, `v` and `q` in Figure 6). Further, `p` can potentially also have a path from: (i) pointers to which `q` has a path to (pointer `l` in Figure 6), and (ii) pointers which interfere with `q` (pointer `t` in Figure 6). It is because of the second possibility that we abstract interference relationships. Note that due to these two possibilities, a number of spurious relationships can be generated. For example in Figure 6, we will generate the spurious relationships D(r,p) and D(s,p), as `q` also has a path to `r` and `p`, besides `l`. As all pointers `q` has paths from/to, also interfere with `q`, the set of *from* direction relationships can be stated as follows.

$$\text{D\_gen\_set\_from} = \{\ \text{D(s,p)} \mid \text{s} \in \text{H} \wedge \text{s} \neq \text{p} \wedge \text{I[s,q]}\ \}$$

To **Relationships** : Pointer `p` will have a path to all pointers `q` has a path to via the link `f`. ¿From the direction matrix, we can find *all* the pointers `q` has a path to, but cannot identify the pointers `q` has a path to via a specific link. So we conservatively assume `p` to be having a path to all the pointers `q` has a path to. In Figure 6, after the statement, `p` is reported to be having paths to pointers `l`, `r` and `s`, where the path from `p` to `s` is spurious.

Note that `q` has a path to itself, so according to the above assumption `p` should also be reported to have a path to `q` after the statement. However, if the data structure accessible from `q` is acyclic (i.e. A[q] = Tree or Dag), `p` cannot have a path back to `q`. In Figure 6, A[q] is Tree, hence `p` is not reported to be having a path to `q`.

Thus, the set of *to* direction relationships can be summarized as:

[1] In this Figure, for sake of clarity we have simply labeled each node with the stack-resident pointer that points to it, instead of explicitly representing the stack.

$$\begin{aligned}\text{D\_gen\_set\_to} = \{\ &\text{D(p,s)} \mid \text{s} \in \text{H} \wedge \text{s} \neq \text{q} \wedge \text{s} \neq \text{p} \wedge \\ &\text{D[q,s]}\ \} \cup \{\ \text{D(p,q)} \mid \text{A[q]} = \text{Cycle}\ \} \\ &\cup \{\ \text{D(p,p)} \mid \text{D[q,q]}\ \}\end{aligned}$$

The overall D_gen_set is obtained by unioning the *from* and *to* sets.

Interference **Relationships** : After the statement, pointer `p` can potentially interfere with all the pointers `q` presently interferes with. So we have the following set of newly generated interference relationships:

$$\begin{aligned}\text{I\_gen\_set} = \{\ &\text{I(p,s)} \mid \text{s} \in \text{H} \wedge \text{s} \neq \text{p} \wedge \text{I[q,s]}\ \} \cup \\ &\{\ \text{I(p,p)} \mid \text{I[q,q]}\ \}\end{aligned}$$

Despite potentially introducing several spurious relationships, this statement does not affect the shape of the data structure accessible from `q`. It only gives a new name to one of the heap objects belonging to this data structure. Since the data structure accessible from `p` is a subpiece of the data structure accessible from `q`, it is *safe* to assign `p` the shape attribute of `q`, giving the following attribute matrix A_c:

$$A_c[p] = A[q]$$

It is possible that A[q] is Cycle, while the shape of the structure accessible from `q` via the `f` link is Tree. However, we cannot detect this from the information available, and must conservatively say that $A_n[p]$ is a Cycle. But if A[q] is Tree, we do not lose the tree attribute, and if it is Dag we still preserve the acyclic property of the data structure accessible from `p`. Note that if we simply deduce the shape attribute of `p` from its direction relationships after the statement, we may lose its Tree or Dag attribute. Thus, separately abstracting the shape attribute proves to be critical in identifying tree-like and dag-like data structures.

2.2.3 Structure Updates

Structure updates are of the form `p->f = NULL` and `p->f = q`. These are the "nasty" statements for shape analysis in imperative programs because such statements can drastically change the shape and connectivity of heap structures. The goal is to get accurate kill and gen information without using an overly complex abstraction. The choice we have made in this practical technique is to use our simple abstractions and live with overly conservative gen and kill sets for these types of statements. However, using a combination of our three abstractions, we are still able to perform accurate shape estimation in many important cases. We discuss the analysis of the two statements in detail below.

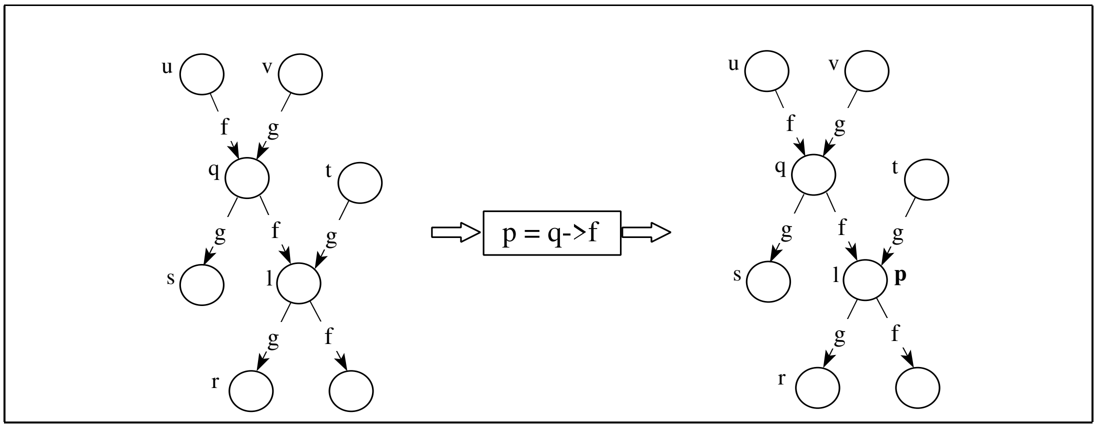


Figure 6: Analyzing Basic Heap Statement `p = q->f`

`p->f = NULL` : This statement breaks the link `f` emanating from the heap object pointed to by `p`. After the statement, `p` should no longer have paths to pointers, it presently has paths to exclusively via the link `f`. As already discussed, this information cannot be obtained from direction/interference matrices. So no relationships can be killed. Further, the statement does not generate any new relationships.

The shape attribute of pointer `p` may change, if this statement disconnects the subpiece of the data structure, due to which A[p] is Dag or Cycle. But the direction/interference information does not suffice to detect such cases, and we err conservatively leaving the attributes unchanged. Note that due to the lack of precise kill information for this statement, if a tree-like structure temporarily becomes dag-like or cyclic, and becomes a tree again (e.g. when swapping the children of a tree), our analysis would continue to report its shape as Dag or Cycle.

`p->f = q` : This statement first breaks the link `f`, and then resets it thereby linking the heap object pointed to by `p`, to the heap object pointed to by `q`, as shown in Figure 7. As already discussed, the relationships killed on breaking the link `f`, cannot be obtained with the information available. However, resetting the link `f` results in generating some new relationships and modifying the attributes of several pointers, as discussed below.

All pointers having a path to `p` (including `p` itself), will now have a path to `q` via the link `f`. Further, these pointers will have paths to all pointers `q` has paths to. In Figure 7, pointers `u`, `v` and `p` will have paths to pointers `q`, `r` and `s` after the statement. Thus, the set of direction relationships generated can be summarized as follows:

$$\text{D\_gen\_set} = \{ \text{D(r,s)} \mid \text{r,s} \in \text{H} \wedge \text{D[r,p]} \wedge \text{D[q,s]} \}$$

In Figure 7, pointer `q` interferes with pointer `t`, before the statement. After the statement, pointers `u`, `v` and `p` will also interfere with `t`. This demonstrates that all pointers having a path to `p`, can potentially interfere with all pointers `q` interferes with. Thus we get the following set of new interference relationships:

$$\text{I\_gen\_set} = \{ \text{I(r,s)} \mid \text{r,s} \in \text{H} \wedge \text{D[r,p]} \wedge \text{I[q,s]} \}$$

This statement can considerably affect the shape attribute of pointers, which have direction relationships with pointers `p` and `q`. We can have the following situations, depending on the current attributes and direction relationships of pointers `p` and `q`:

Pointer `q` already has a path to `p` (D[q,p] = 1) : After the statement `p->f = q`, `p` will also have a path back to `q`. Thus, a cycle will be generated between `p` and `q`. We have already illustrated this case in Figure 2. Further, this cycle will also be accessible from all pointers that presently have a path to `p` or `q` (including `p` and `q` themselves), and the shape attribute of all these pointers will become Cycle. We summarize this case as follows:

$$\text{H}_s = \{ \text{s} \mid \text{s} \in \text{H} \wedge (\text{D[s,q]} \vee \text{D[s,p]}) \}$$
$$\forall\ \text{s} \in \text{H}_s,\ \text{D[q,p]} \Rightarrow \text{A}_c\text{[s]} = \text{Cycle}$$

If the above situation does not arise, we have the following possibilities:

A[q] = Tree : In this case another tree-like structure becomes accessible from all the pointers that presently have a path to `p`. If the data structures pointed to by `p` and `q` are initially completely disjoint, then the statement simply connects a tree substructure to the data structure pointed to by `p` and does not affect the shape attribute of any pointer. Figure 7 illustrates this case. Otherwise the shape attribute of pointers that initially have a path to `p` and also interfere with `q`, becomes Dag (if it is presently Tree). Pointers `u` and `v` in Figure 8 fall in this category. Finally, if the shape attribute of such a pointer is already Dag or Cycle, it remains unchanged. In other words, the shape attribute of these

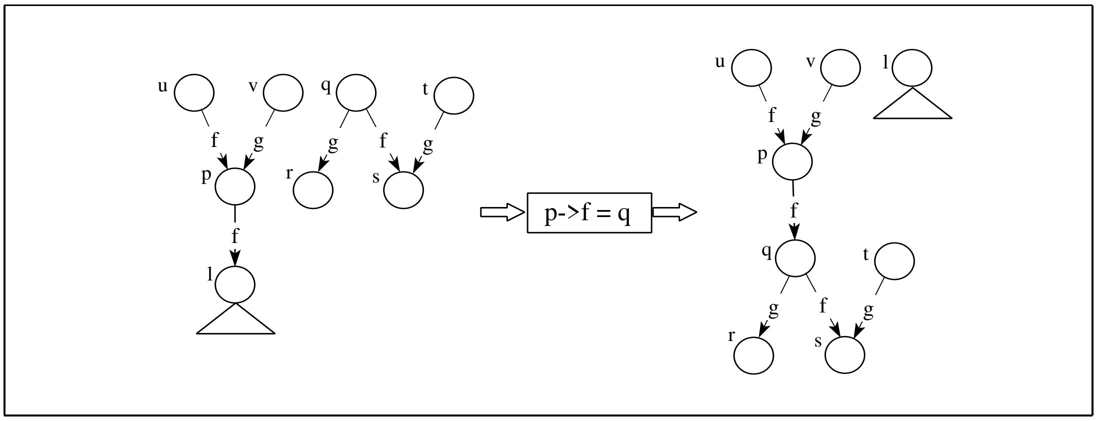


Figure 7: Analyzing Basic Heap Statement `p->f = q`

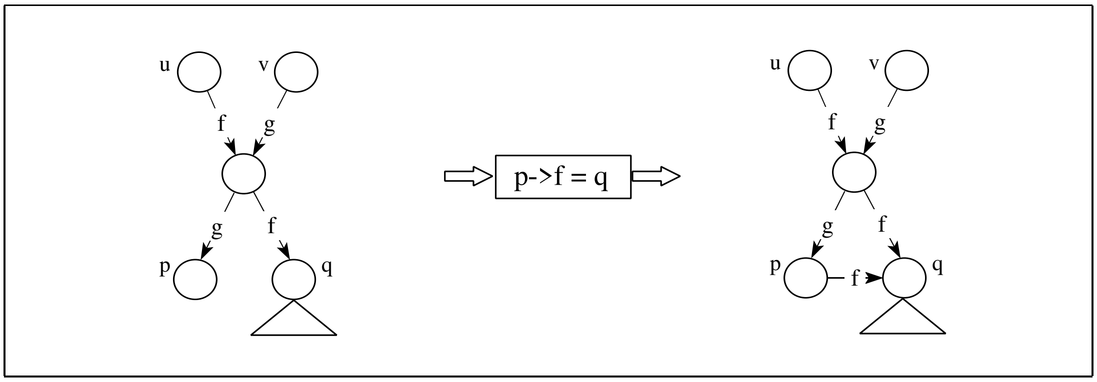


Figure 8: Direction Relationships Impacting Shape Attribute

pointers, becomes the merge of their current attribute and the Dag attribute, where the merge operator $\bowtie$ for the shape attribute is defined as follows:

$\bowtie$	*Tree*	*Dag*	*Cycle*
Tree	*Tree*	*Dag*	*Cycle*
Dag	*Dag*	*Dag*	*Cycle*
Cycle	*Cycle*	*Cycle*	*Cycle*

This case can be formally summarized as follows:

$$\begin{array}{l} \mathrm{H}_s = \{\ \mathrm{s} \mid \mathrm{s} \in \mathrm{H} \wedge (\mathrm{I[s,q]} \wedge \mathrm{D[s,p]})\ \} \\ \forall\ \mathrm{s} \in \mathrm{H}_s,\ ((\neg \mathrm{D[q,p]}) \wedge (\mathrm{A[q]} = \mathrm{Tree})) \Rightarrow \\ \qquad \mathrm{A}_c[\mathrm{s}] = \mathrm{A[s]} \bowtie \mathrm{Dag} \end{array}$$

<u>A[q] $\neq$ Tree</u> : In this case, the shape attribute of all pointers that have path to `p` is merged with the shape attribute of `q`. This is required because the data structure accessible from `q`, will also become accessible from all these pointers after the statement. We summarize the case as follows:

$$\begin{array}{l} \mathrm{H}_s = \{\ \mathrm{s} \mid \mathrm{s} \in \mathrm{H} \wedge \mathrm{D[s,p]}\ \} \\ \forall\ \mathrm{s} \in \mathrm{H}_s,\ ((\neg \mathrm{D[q,p]}) \wedge (\mathrm{A[q]} \neq \mathrm{Tree})) \Rightarrow \\ \qquad \mathrm{A}_c[\mathrm{s}] = \mathrm{A[s]} \bowtie \mathrm{A[q]} \end{array}$$

From the rules presented above, it can be noticed that a considerable number of spurious direction and interference relationships can be introduced during the analysis. However, empirical results presented in Section 4, indicate that our analysis provides effective shape information for a broad range of programs in an efficient manner.

3 Implementing Shape Analysis in the McCAT C Compiler

Shape analysis has been implemented as a context-sensitive interprocedural analysis in the McCAT optimizing/parallelizing C compiler. It is a flow-sensitive analysis and collects program-point-specific information. The analysis is performed on the SIMPLE intermediate representation which is a simplified, compositional subset of C [7, 13, 31]. Shape analysis is performed after points-to analysis [6, 8] and focuses only on the subset of pointers reported to be pointing to heap by points-to analysis. This reduces the storage requirements for the abstractions, and makes the implementation easier as well as more efficient. The overall analysis framework is similar to that used for points-to analysis. It should be noted, however, that other dataflow frameworks could also use the basic abstraction and rules presented in the previous section. Our particular implementation is structured as a simple analysis for each basic statement of the form presented in Section 2, a compositional rule for each control construct, and a context-sensitive approach for handling procedure calls.

While presenting the basic analysis rules in Section 2.2, we did not consider the presence of stack-directed pointers. However, we take them into account in the actual implementation, and this requires some additional checks. The subtle point is that references of the form `p->f` may refer to the stack, the heap, or to both the stack and heap. For example, in one calling context, `p` may point to a stack-allocated object that has a name, while in another calling context `p` may point to a heap-allocated object. Consider a statement of the form `p->f = q`. If `p` points to a stack-allocated object with the name `x`, then the appropriate analysis rule is `x.f = q`, whereas if `p` points to a heap-allocated object, the appropriate rule is `p->f = q`. Thus, our implementation first uses points-to information to resolve all references of the form `p->f` into a set of possible stack and heap locations, and then applies the appropriate simple shape analysis rules, merging the results of all the outputs.

Our strategy for handling control constructs is illustrated in Figure 9, which gives the analysis rule for the `while` statement. It also defines the merge operators for our three abstractions. The merge operator for the direction and interference relationships is simply the logical OR ($\vee$) operation, as they are both *possible* (or *may*) relationships. The merge operator $\bowtie$ for the shape attribute has already been defined in Section 2.2.3. Also note that we consider the loop condition as a simple assignment, when feasible. For example, when it involves a pointer equality test like (`p == NULL`) or (`p == q`).

To accurately handle procedure calls, we use the interprocedural analysis framework built by points-to analysis [6, 8, 14]. It provides us the complete invocation graph of the program which is constructed by a simple depth-first traversal of the invocation structure of the program. Since the invocation structure is not known statically for recursive and indirect calls, they are handled in a special manner. Recursive calls are represented by special pairs of recursive and approximate nodes, where the approximate node represents all possible unrollings of recursion. Indirect calls through function pointers are represented by nodes indicating the possible set of functions invocable from the given call-site (resolved during points-to analysis).

Based on the above framework, we use the context-sensitive interprocedural strategy depicted in Figure 10. Complete rules for our interprocedural analysis scheme are described in [11]. Here we briefly discuss only the most pertinent issues. The general idea is that, first, the three matrices (for direction, interference and attribute abstractions) at the call-site are *mapped* to prepare the input matrices for the called procedure. Next, the body of the procedure is analyzed with these input matrices, and the output matrices obtained are *unmapped* and returned to the call-site. With this approach, every time a procedure call is analyzed for some call-chain, there exists a unique invocation graph node corresponding to it. Recursive calls are handled via an interprocedural fixed-point computation, using the special recursive and approximate nodes in the invocation graph. Indirect calls are handled by separately analyzing each invocable procedure from the given call-site, and then merging all outputs.

```
/ D,I,A : Input matrices, H : Set of pointers abstracted,
  ign : Current invocation graph node /
fun process_while(cond,body,D,I,A,H,ign) =
  do
    prevD = D; prevI = I; prevA = A;
    [D1,I1,A1] = process_basic_stmt(cond,D,I,A,H);
    [D2,I2,A2] = process_stmt(body,D1,I1,A1,H,ign);
    D = Merge(D,D2); I = Merge(I,I2);
    A = Merge(A,A2);
  while ((D != prevD) and (I != prevI) and
        (A != prevA))
  return([D,I,A]);

  Merge(D,D1) ⇒ ∀ r,s ∈ H, D[r,s] = D[r,s] ∨ D1[r,s]
  Merge(I,I1) ⇒ ∀ r,s ∈ H, I[r,s] = I[r,s] ∨ I1[r,s]
  Merge(A,A1) ⇒ ∀ s ∈ H, A[s] = A[s] ⋈ A1[s]
```

Figure 9: Analyzing a `while` Statement

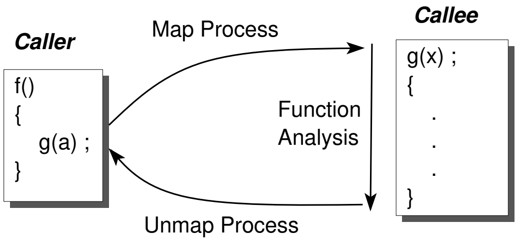


Figure 10: The Interprocedural Strategy

The main issue related with *map* and *unmap* processes, is identifying the set of pointers whose attributes/relationships can be modified by the procedure call. This set includes pointers which are: (i) global in scope, (ii) not in the scope of the callee but are accessible via an indirect reference (*invisible variables*), and (iii) not at all accessible in the callee, but have a direction/interference relationship with some pointer accessible in the callee (*inaccessible variables*). Special symbolic names must be generated to represent all invisible and inaccessible variables and capture their attributes/relationships in the calling context.

Points-to analysis already generates symbolic names for *invisible* variables, and we simply reuse them. For *inaccessible* variables we generate two symbolic names for each parameter or global that is related to some inaccessible variable(s). If the name of the parameter or global variable is `x`, then we use the name `0-x` to represent all inaccessible pointers that have paths *to* `x`, and the name `0+x` to represent all inaccessible pointers that have paths *from* `x` or interfere with `x`. For each procedure call a mapping is stored between names in the calling context and names in the called context (globals, parameters and symbolic names), and is used while *unmapping*. Complete description of the rules for *map* and *unmap* processes can be found in [11].

Finally, in order to get the full context-sensitivity at a reduced price, we have implemented a simple memoization scheme. Every time we finish analyzing a procedure call, we store the currently computed pairs of input and output matrices, in the invocation graph node corresponding to it. When this call is re-analyzed along this call-chain, we simply use the stored output from its invocation node, if the current input is identical to the stored input. We are also currently exploring other techniques to optimize our interprocedural algorithm, which include: (i) excluding the functions from the invocation graph, which neither update nor access heap-directed pointer variables, (ii) building the invocation graph in a lazy manner, as the demand for different invocation contexts arises during the analysis [9], and (iii) performing more extensive memoization by trying to memoize all calls to a procedure (except the first one) irrespective of the call-chain they appear on.

4 Experimental Results

4.1 Benchmarks

We have collected a number of small and medium sized benchmarks from a variety of sources. Table 1 summarizes the characteristics of the benchmark programs. The first section gives the source lines including comments, counted using the *wc* utility, and the number of statements in the SIMPLE intermediate representation (this number gives a good estimate of program size from the analysis point of view). The second section gives the minimum, maximum and average number of variables abstracted by the direction/interference matrices over all functions in the program (this includes symbolic variables introduced by our analysis). These numbers indicate the size of the abstractions and the memory requirements of the analysis for a given program. Note that the average varies from 7 (*stanford*) to 83 (*sim*), which is quite reasonable with respect to space requirements (we use bit matrices). The third section gives the total number of indirect references in the program, and the number of indirect references where the dereferenced pointer can point to a stack location, to a heap location and to both a stack and a heap location (this typically happens when a formal parameter receives a stack-directed pointer in one invocation of the function and a heap-directed pointer in another). All the benchmarks in Table 1 have substantial number of indirect references, with majority of the indirect references refering to heap locations (except for the two benchmarks: *assembler* and *loader*). Thus the given benchmark set is well suited for evaluating a heap analysis for C.

4.2 Results

We estimate the effectiveness of shape analysis for this set, using the following measurements (Table 2(a)):

Refs: The number of heap-related indirect references in the benchmark.

T, D, C : These three columns respectively give the number of heap-related indirect references in the program, where the dereferenced pointer, say `p`, points to a tree-like, dag-like or cyclic data structure: i.e. A[p] = *Tree*, *DAG* or *Cycle*, where A is the attribute matrix at the given program point.

The multi-columns labeled `*a/(*a).b` and `a[i]` (where `a` itself is a heap-directed pointer) in Table 2(a), separately give the above measurements for indirect references of the respective form.[2] The multi-column labeled Overall gives the overall statistics for the given program. Further, in Table 2(b) we compare the actual shape of data structure(s) a program builds with that reported by the analysis and we observe if the shape information would be useful in improving dependence information and/or parallelization. Both tables have the programs divided into 2 groups. The top group corresponds to programs that build tree-like data structures, and are thus good candidates for our shape analysis. In 9 of these 11 programs, we can determine that the structures are in fact *Trees* and this information is useful. In the remaining 2 programs, `reverse` and `sim`, we conservatively find that the structures are *DAGS*, so our shape information is only slightly useful. The bottom group of programs build structures that are inherently dag-like or graph-like, and so even

[2] Note that an access of the form `x = a[i]` is simply considered a pointer reference (and not an indirect reference) when `a[i]` is a pointer, but `a` itself is not a pointer. Indirect references of the form `x = *(a[i])` are counted as indirect references of the form `*a` because of our simplification which expresses this as { `temp = a[i]; x = *temp;` }. For this reason, benchmarks using arrays of pointers may not always have indirect references of type `a[i]`.

Program	Source Lines	SIMPLE stmts	Min vars	Max vars	Avg vars	Ind Refs	To Stack	To Heap	Stack/ Heap
bintree	351	342	4	23	10	50	10	40	0
xref	153	139	20	40	24	31	0	31	0
misr	277	235	2	10	8	47	39	35	27
chomp	430	476	20	27	22	127	45	82	0
stanford	885	880	4	14	7	28	0	28	0
hash	257	110	4	6	11	14	7	7	0
power	681	641	16	23	18	180	29	151	0
reverse	123	49	9	18	12	16	0	16	0
assembler	3361	3071	22	36	24	718	666	52	0
loader	1539	1055	13	28	17	170	106	64	0
sim	1422	1760	76	111	83	374	34	340	0
paraffins	381	180	6	31	21	37	2	35	0
blocks2	876	1070	56	82	61	373	98	275	0
nbody	2204	703	24	36	27	134	24	116	6
sparse	2859	1495	24	60	32	468	3	465	0
pug	2400	2089	32	153	48	822	147	688	13

Table 1: Benchmark Characteristics

when our shape analysis gives correct shape (4 of the 5 programs), the shape information is not really detailed enough for improving dependence testing and/or parallelization. The shape information is useful, however, as a way of automatically determining that a higher-level heap analysis should be applied.

4.3 Observations

Based on the data presented in Tables 2(a) and 2(b), and our examination of the benchmark programs, we make the following observations.

If a program builds a tree-like data structure in such a manner, that a new node is always appended at the beginning/end of the existing structure, then shape analysis always successfully infers the shape of this data structure as *Tree*. This happens because in this case the data structure does not even temporarily lose its tree attribute. In our benchmark set, *bintree*, *xref*, *stanford* and *chomp* build binary trees by appending the new node to a leaf node, while *hash*, *misr*, *loader* and *assembler* build linked lists by appending a new item at the beginning/end of the list. The shape attribute for these programs is accurately estimated.

If a tree or dag-like data structure is built (or modified) by inserting new nodes between existing nodes, shape analysis provides conservative estimates and reports the shape to be dag-like or cyclic. The following code fragment illustrates this case:

```
S1: old_node->f = q;
S2: new_node = newNode();
S3: new_node->f = q;
S4: old_node->f = new_node;
```

Just before statement `S4` both `old_node` and `new_node` have a path to `q`. Statement `S4` *kills* the path from `old_node` to `q` (which is via the link `f`), and sets a path from `old_node` to `new_node`. Our analysis cannot detect the kill information, and finds `old_node` to have an additional path to `q` via `new_node` (which is now actually the only path). So it reports its shape attribute as *DAG*. If further insertions are done into this apparently dag-like structure, analysis becomes overly conservative and reports its final shape attribute as *Cycle*. This case applies to benchmarks *sim*, *blocks2* and *nbody*.

If a data structure temporarily becomes dag-like or cyclic and then becomes tree-like again, shape analysis cannot detect this, and continues to report its shape as dag-like or cyclic. The benchmark *reverse* that recursively swaps a binary tree represents this case.

Shape analysis abstracts all pointers from an array of pointers, say `a[10]`, as one pointer `a`. So the relationships and attribute of this pointer, represent the merge of the relationships and attributes of all the pointers it denotes (all pointers `a[i]`). If the shape attribute of such a pointer is reported to be *Tree*, one is guaranteed that all array indices point to tree-like structures, completely disjoint from each other (if the structures pointed to by two array indices, say `a[i]` and `a[j]` share a node, the shape attribute for `a` is reported as *DAG* or *Cycle*).

In our benchmark set, *hash* uses an array of pointers to linked lists, and *power* implements a power network tree [2, 25] with its root having an array of pointers to disjoint subtrees. We get the shape attribute of these arrays as *Tree*. For example, the *simplified* version of the loop that builds the power tree is as follows:

Program	`*a / (*a).b`				`a[i]`				Overall			
	Refs	T	D	C	Refs	T	D	C	Refs	T	D	C
bintree	36	36	0	0	4	4	0	0	40	40	0	0
xref	29	29	0	0	2	2	0	0	31	31	0	0
misr	35	35	0	0	0	0	0	0	35	35	0	0
chomp	56	56	0	0	26	26	0	0	82	82	0	0
stanford	28	28	0	0	0	0	0	0	28	28	0	0
hash	7	7	0	0	0	0	0	0	7	7	0	0
power	147	147	0	0	4	4	0	0	151	151	0	0
reverse	16	11	5	0	0	0	0	0	16	11	5	0
assembler	45	45	0	0	7	7	0	0	52	52	0	0
loader	55	55	0	0	9	9	0	0	64	64	0	0
sim	96	29	67	0	244	221	23	0	340	250	90	0
paraffins	26	8	18	0	9	3	6	0	35	11	24	0
blocks2	119	16	37	66	156	64	43	49	275	80	80	115
nbody	74	22	0	52	42	14	0	28	116	36	0	80
sparse	384	14	0	370	0	0	0	0	384	14	0	370
pug	514	16	0	498	174	1	0	173	688	17	0	671

(a) Empirical Measurements for Shape Analysis

Program	Actual Data Structure(s) Built	Shape Reported	Shape Correct	Shape Info. Useful
bintree	A binary tree	*Tree*	yes	yes
xref	A binary tree with a linked list hanging from each node	*Tree*	yes	yes
misr	A linked list	*Tree*	yes	yes
chomp	A game tree and a linked list	*Tree*	yes	yes
stanford	A binary tree for tree-sort	*Tree*	yes	yes
hash	A hash table using an array of linked lists	*Tree*	yes	yes
power	A tree implementing a power network	*Tree*	yes	yes
reverse	A binary tree which is recursively swapped	*DAG*	no	slightly
assembler	A linked list	*Tree*	yes	yes
loader	A linked list	*Tree*	yes	yes
sim	An array of linked lists	*DAG*	no	slightly
paraffins	3 arrays of interconnected linked lists (DAG)	*DAG*	yes	slightly
blocks2	A constraint graph data structure (DAG)	*DAG/Cycle*	partially	slightly
nbody	A leaf-linked octree (DAG)	*Cycle*	no	no
sparse	Sparse matrix using linked lists (cyclic)	*Cycle*	yes	no
pug	A complex cyclic structure	*Cycle*	yes	no

(b) Accuracy and Usefulness of Shape Information

Table 2: Experimental Results

```
t = (struct root*) malloc();
for (i = 0 ; i <= N; i = i + 1)
 { temp_1 = (i * 10);
   l = build_lateral(temp_1,20);
   temp_2 = (* t).feeders;
   temp_2[i] = l;
 }
```

In the above loop, the function call `build_lateral` returns a tree in each iteration, which is then connected to the ith index of the pointer array `temp_2`. Since in each iteration a disjoint tree is connected, the overall shape of the pointer array `temp_2` is deduced as *Tree*. Further, most of the computation in this benchmark, is performed in a loop which iterates over this array. The important segment of this loop in the *simplified* format is as follows:

```
for (i = 0; i <= N; i = i + 1)
 { temp_0 = (* r).feeders;
   l = temp_0[i];
   theta_R = (* r).theta_R;
   theta_I = (* r).theta_I;
   a = Compute_Lateral(l,theta_R,theta_I,
                       theta_R,theta_I);
    ...
  }
```

For this loop, we know from shape information that in each iteration pointer `l` points to a disjoint tree, which is then operated upon by the function `Compute_Lateral`. Thus this loop can be effectively parallelized provided there are no dependencies due to other variables (there are none in this case). This demonstrates how shape analysis can provide critical information for dependence analysis and parallelization.

The benchmark *paraffins* also uses arrays of pointers to linked lists. However these lists share some nodes, and consequently the shape gets reported as *DAG*. The benchmarks *sparse* and *pug* use inherently cyclic data structures with back pointers. So majority of indirect references for them fall in the *Cycle* category. The ones in the *Tree* category represent newly allocated nodes, before they are connected to the main data structure of the program.

Besides shape information, direction and interference relationships can also be useful on their own. For example, to identify if data structures accessible from two pointers say `p` and `q` share a node, one needs to simply check if the interference matrix entry `I[p,q]` is set to one. Similarly, direction matrix information can aid the programmer in *safely* deallocating memory. At a call-site like `free(p)`, if any (live) pointer can have a path to `p`, then it may not be a safe deallocation. We are currently exploring the effectiveness of direction and interference information, for these and other applications.

4.4 Interprocedural Measurements

Shape analysis is a context-sensitive interprocedural analysis. In Table 3, we present the invocation graph characteristics of the benchmarks. The first three columns in this table, give the total number of functions actually called in the program, the total number of call-sites in the program, and the total number of nodes in its invocation graph. The last three columns give the number of recursive and approximate nodes, and the number of nodes per call-site.

In Table 4 we present some *dynamic* interprocedural measurements for shape analysis. The first three columns give the total number of procedure calls analyzed, the number of procedure calls that get memoized, and the actual number of procedure calls that get analyzed. The last three columns give the average number of procedure calls actually analyzed (given in the column labeled Act) per function, per call-site and per invocation graph node. These averages are calculated by dividing the number in the Act column, with the appropriate number from the first three columns in Table 3.

Program	fns	call sites	ig	Rec	App	nodes/ call-site
bintree	17	31	32	2	4	1.03
xref	8	14	15	2	4	1.07
misr	5	7	7	0	0	1.00
chomp	20	47	98	7	7	2.09
stanford	8	12	13	2	4	1.08
hash	5	8	8	0	0	1.00
power	18	31	53	6	6	1.71
reverse	5	10	11	2	4	1.10
assembler	52	263	642	0	0	2.44
loader	30	82	125	2	2	1.52
sim	14	26	44	2	8	1.70
paraffins	7	6	7	0	0	1.16
blocks2	20	28	28	1	2	1.00
nbody	34	67	118	2	2	1.76
sparse	28	76	121	0	0	1.59
pug	41	69	101	0	0	1.46

Table 3: Static Interprocedural Measurements

There are several interesting observations to be made from the results in Tables 3 and 4. The first is that for these benchmarks, the size of the invocation graph does not explode, and we can do a complete context-sensitive analysis with reasonable cost. There are, however, other benchmarks that do have very large invocation graphs, so we are exploring more aggressive memoization techniques for handling these programs, as discussed in Section 3. It is also interesting to note that a large number of procedure calls get memoized, even with our simple scheme, that only reuses output values when the *same* invocation node is visited with a previously computed input. Finally, it can be observed from Table 3 that majority of the benchmarks, have recur-

Program	Calls Analyzed			Avf	Avc	Avi
	Tot	Mem	Act			
bintree	59	12	47	2.76	1.51	1.47
xref	60	13	47	5.88	3.36	3.13
misr	7	0	0	1.40	1.00	1.00
chomp	390	196	194	9.70	4.13	1.98
stanford	36	6	30	3.75	2.50	2.31
hash	11	1	10	2.00	1.25	1.25
power	112	49	63	3.50	2.03	1.19
reverse	52	16	36	7.20	3.60	3.27
paraffins	9	0	9	1.29	1.50	1.29
nbody	252	42	210	6.17	3.13	1.78
assembler	1057	221	836	16.08	3.18	1.30
loader	328	126	202	6.73	2.46	1.62
sim	161	8	153	10.93	6.19	3.66
blocks2	690	432	258	12.90	9.21	8.90
sparse	195	64	131	4.68	1.72	1.08
pug	213	53	160	3.90	2.39	1.36

Table 4: Dynamic Interprocedural Measurements

sive and approximate invocation graph nodes. Since most of these programs use recursive data structures, they also employ recursion as the control structure to traverse and modify them. This implies that to be useful, any shape analysis must handle interprocedural analysis, and it must handle recursive programs in a safe and accurate manner.

5 Conclusions and Future Work

In this paper we have presented an analysis that approximates the shape of dynamic data structures in C programs. Shape analysis is part of a hierarchy of pointer analyses implemented in the McCAT C compiler, and it is directed at programs that use simple recursive data structures that are built compositionally. The analysis has been completely implemented and tested on 16 benchmark programs. The experimental results show that it does provide accurate results for the those programs that build simple data structures. Thus, for programs building lists, trees, and arrays of lists or trees, we can often provide useful information for optimization and parallelization.

For programs that make major structural changes to the data structure, our shape abstraction is not powerful enough to give accurate results, although the results will be safe. Other analyses can handle some of these cases [3, 5, 17, 30], but they are substantially more complicated and more difficult to implement in real compilers. Our approach is to use the cheapest and simplest analysis possible for each program under consideration. Thus, if the program fits into the target class for our shape analysis, we will not apply a more expensive or complex analysis.

We plan to extend our shape analysis to create the *level-3* analysis in our hierarchy by enriching the direction abstraction to keep information about the first link on the path (a partial implementation of path matrices [17]) and by using a more complex attribute matrix that abstracts the shape of the data structure with respect to certain links. It is hoped that this analysis will be able to handle structures like leaf-linked trees, trees with parent pointers, and to also improve upon the accuracy of the information in the face of structural updates by giving better kill information.

Based on the positive results from the experiments presented in this paper, we also plan to apply all of our heap analyses to larger programs and to continue our development of more efficient interprocedural strategies.

References

[1] J. Choi, M. Burke, and P. Carini. Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side-effects. In *Conference Record of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 232–245, January 1993.

[2] A. Rogers, M. C. Carlisle, J. H. Reppy, and L. J. Hendren. Supporting dynamic data structures on distributed-memory machines. *ACM Transactions on Programming Languages and Systems*, 17(2):231–263, March 1995.

[3] D. R. Chase, M. Wegman, and F. K. Zadeck. Analysis of pointers and structures. In *Proceedings of the SIGPLAN '90 Conference on Programming Language Design and Implementation*, pages 296–310, June 1990.

[4] A. Deutsch. A storeless model of aliasing and its abstractions using finite representations of right-regular equivalence relations. In *Proceedings of the 1992 International Conference on Computer Languages*, pages 2–13, April 1992. IEEE Computer Society Press.

[5] A. Deutsch. Interprocedural may-alias analysis for pointers: Beyond *k*-limiting. In *Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation*, pages 230–241, June 1994.

[6] M. Emami, R. Ghiya, and L. J. Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In *Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation*, pages 242–256, June 1994.

[7] A. M. Erosa and L. J. Hendren. Taming control flow: A structured approach to eliminating goto statements. In *Proceedings of the 1994 International Conference on Computer Languages*, pages 229–240, May 1994.

[8] M. Emami. A practical interprocedural alias analysis for an optimizing/parallelizing C compiler. Master's thesis, McGill University, July 1993.

[9] E. Gagnon. A fast-forward and lazy points-to analysis. ACAPS Project Report 1995.622B.03, McGill University, May 1995.

[10] R. Ghiya and L. J. Hendren. Connection analysis: A practical interprocedural heap analysis for C. In *Proceedings of the Eight Workshop on Languages and Compilers for Parallel Computing*, August 1995.

[11] R. Ghiya. Practical techniques for interprocedural heap analysis. Master's thesis, School of Computer Science, McGill University, May 1995.

[12] W. L. Harrison III. The interprocedural analysis and automatic parallelization of Scheme programs. *Lisp and Symbolic Computation*, 2(3/4):179–396, 1989.

[13] L. Hendren, C. Donawa, M. Emami, G. Gao, Justiani, and B. Sridharan. Designing the McCAT compiler based on a family of structured intermediate representations. In *Proceedings of the 5th International Workshop on Languages and Compilers for Parallel Computing*, number 757 in Lecture Notes in Computer Science, pages 406–420, August 1992. Springer-Verlag. Published in 1993.

[14] L. J. Hendren, M. Emami, R. Ghiya, and C. Verbrugge. A practical context-sensitive interprocedural analysis framework for C compilers. ACAPS Technical Memo 72, School of Computer Science, McGill University, July 1993.

[15] L. J. Hendren and G. R. Gao. Designing programming languages for analyzability: A fresh look at pointer data structures. In *Proceedings of the 1992 International Conference on Computer Languages*, pages 242–251, April 1992. IEEE Computer Society Press.

[16] L. J. Hendren, J. E. Hummel, and A. Nicolau. Abstractions for recursive pointer data structures: Improving the analysis and transformation of imperative programs. In *Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation*, pages 249–260, June 1992.

[17] L. J. Hendren and A. Nicolau. Parallelizing programs with recursive data structures. *IEEE Transactions on Parallel and Distributed Systems*, 1(1):35–47, January 1990.

[18] S. Horwitz, P. Pfeiffer, and T. Reps. Dependence analysis for pointer variables. In *Proceedings of the SIGPLAN '89 Conference on Programming Language Design and Implementation*, pages 28–40, June 1989.

[19] N. D. Jones and S. S. Muchnick. *Program Flow Analysis, Theory and Applications*, chapter 4, Flow Analysis and Optimization of LISP-like Structures, pages 102–131. Prentice-Hall, 1981.

[20] N. D. Jones and S. S. Muchnick. A flexible approach to interprocedural data flow analysis and programs with recursive data structures. In *Conference Record of the Ninth Annual ACM Symposium on Principles of Programming Languages*, pages 66–74, January 1982. ACM SIGACT and SIGPLAN.

[21] N. Klarlund and M. Schwartzbach. Graph types. In *Conference Record of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 196–205, January 1993.

[22] J. R. Larus. Compiling Lisp programs for parallel execution. *Lisp and Symbolic Computation*, 4:29–99, 1991.

[23] J. R. Larus and P. N. Hilfinger. Detecting conflicts between structure accesses. In *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation*, pages 21–34, June 1988.

[24] J. R. Larus and P. N. Hilfinger. Restructuring Lisp programs for concurrent execution. In *Proceedings of the ACM/SIGPLAN PPEALS 1988 — Parallel Programming: Experience with Applications, Languages and Systems*, pages 100–110, July 1988.

[25] S. Lummetta, L. Murphy, X. Li, D. Culler, and I. Khalil. Decentralized optimal power pricing. In *Proceedings of Supercomputing 93*, pages 243–249, November 1993.

[26] W. A. Landi and B. G. Ryder. A safe approximate algorithm for interprocedural pointer aliasing. In *Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation*, pages 235–248, June 1992.

[27] J. Plevyak, A. Chien, and V. Karamcheti. Analysis of dynamic structures for efficient parallel execution. In *Proceedings of the 6th International Workshop on Languages and Compilers for Parallel Computing*, number 768 in Lecture Notes in Computer Science, pages 37–56, August 1993. Springer-Verlag. Published in 1994.

[28] T. Reps. Shape analysis as a generalized path problem. In *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation (PEPM)*, pages 1–11, June 1995.

[29] E. Ruf. Context-insensitive alias analysis reconsidered. In *Proceedings of the ACM SIGPLAN '95 Conference on Programming Language Design and Implementation*, pages 13–22, June 1995.

[30] M. Sagiv, T. Reps, and R. Wilhelm. Solving shape-analysis problems in languages with destructive updating. In *Conference Record of the Twenty Third Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, January 1996.

[31] B. Sridharan. An analysis framework for the McCAT compiler. Master's thesis, McGill University, September 1992.

[32] B. Steensgaard. Points-to analysis in almost linear time. In *Conference Record of the Twenty Third Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, January 1996.

[33] R. P. Wilson and M. S. Lam. Efficient context-sensitive pointer analysis for C programs. In *Proceedings of the ACM SIGPLAN '95 Conference on Programming Language Design and Implementation*, pages 1–12, June 1995.