15-745 Project Final Report: Shape Analysis

Kunming Jiang (kunmingj), Andrew Haberlandt (ahaberla)

Apr 25, 2023

1 Introduction

Pointer analysis is crucial for optimizing languages like C and C++, and is a foundational analysis in the compilers for these langauges. A well-established pointer analysis is foundational for reasoning on dependencies within memory, identifying indirect control-flow targets, and bug detection, as well as enabling a variety of optimizations [1]. Despite its numerous benefits, applying pointer analysis is also resource and time intensive, and often difficult to achieve precision [1]. Conventional pointer analysis involves constructions of either an inclusion-based relationship [2], or a point-to graph to demonstrate the relationship between pointers within a program [3]. These methods are costly and often ignore higher-level semantics of these pointers, i.e. the underlying data structure these pointers represent. A direct acyclic graph (DAG), for instance, can rule out the possibility of a cyclic point-to relationship, and disinclude edges in the relation graph that are otherwise difficult to eliminate.

Shape Analysis [4] proposes a framework to infer potential data structures allocated in the heap. The basic construct involves a dataflow analysis that generates a direction and inference matrix between all pointers, and separates the shape of the data structure each pointer points to into three categories: tree, DAG, or cycle. Using this information, the compiler can efficiently deduce the updates to the point-to information of each pointer operation.

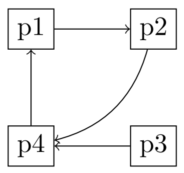
While the original framework for shape analysis was proposed in 1996, the idea is seldomly implemented in modern compiler machines. In this project, we implement the algorithm described in [4] in LLVM14 and demonstrate how it can be used to augment traditional pointer analysis methods.

2 Identifying Shapes of Pointers

The main goal of shape analysis is to identify the data structure expressed by a pointer and its successors. In particular, shape analysis is concerned with what pointers are *reachable* from a pointer p. We can illustrate the relationship between pointers using a directed *points-to graph*. Each node in the graph represents a pointer, and an edge $N_1 \to N_2$ indicates that N_2 is a member of the heap object pointed to by N_1. In particular, if written in C code, an edge $N_1 \to N_2$ exists if and only if N_1 points to a `struct` or array, and N_2 is a member of the `struct` or array. For example, Figure 1 shows a snippet of C code and the corresponding points-to graph.

In this project, for a given node n, we identify the subgraph consisted of all the nodes reachable from n. We refer to the shape of this subgraph as *shape of* n, which can be separated into three categories: `TREE`, `DAG`, and `CYCLE`. For a node n to be categorized as `TREE`, we require that for any decendent d of n, there exists exactly one path from n to d. `DAG` relaxes the condition by allowing multiple paths from n to its decendents, but forbids any cycle from appearing in the graph. Thus, for

```
struct Node {
  struct Node *succ;
};
struct Node *p1 = malloc(sizeof(struct Node));
struct Node *p2 = malloc(sizeof(struct Node));
struct Node *p3 = malloc(sizeof(struct Node));
struct Node *p4 = malloc(sizeof(struct Node));
p1->succ = p2;
p2->succ = p4;
p3->succ = p4;
p4->succ = p1;
```



$$\begin{tikzcd}
\boxed{p1} \arrow[r] & \boxed{p2} \arrow[dl, bend left] \\
\boxed{p4} \arrow[u] & \boxed{p3} \arrow[l]
\end{tikzcd}$$

Figure 1: C code for a node in a DAG, and the points-to relationships

n to be `DAG`, no path in the subgraph can reach a node twice. Finally, `CYCLE` describes all subgraphs that contains cyclic paths, however, note that categorizing n as `CYCLE` only implies the existence of a cycle downstream, and n might not be part of any cycle.

Shape of nodes provide valuable information that is otherwise hard to obtain. For instance, if a node n has shape `TREE`, then two traversals from different children of n will never reach the same node, and can thus be safely parallelized. Similarly, a `DAG` node can be safely processed in parallel with any of its decendents, since the two nodes are guaranteed to be different. We show how shape analysis can be used to augment traditional pointer analysis in section 6.

3 Using Data Structures to Represent Shapes

Our implementation of shape analysis largely follows the framework proposed by Ghiya et al. [4]. Concretely, we designed a dataflow analysis, where each state is consisted of three elements: a *direction matrix* (`D`), an *interference matrix* (`I`), and a mapping `S` from pointers to shapes (`TREE`, `DAG`, or `CYCLE`).

The direction matrix `D` maps every pair of nodes to a bit value and reasons the relationship between them. For two nodes `p` and `q`,

- `D[p,q] = 1` indicates that a path *might* exist from `p` to `q`.
- `D[p,q] = 0` indicates `q` is definitely not reachable from `p`.

Since the graph is directed, direction matrix might not be symmetric.

The interference matrix `I` also maps every pair of nodes to a bit value. It reasons the relationship between their decendents. For two nodes `p` and `q`,

- `I[p,q] = 1` indicates that there *might* be a node reachable by both `p` and `q`.
- `I[p,q] = 0` indicates `p` and `q` do not have common descendent.

Unlike `D`, `I` is symmetric, so we only need to fill in half of its entries.

Finally, `S` maps each node to a shape: `TREE`, `DAG`, or `CYCLE`. To facilitate the dataflow analysis described in section 4, we expand the definition in section 2 and integrate the three shapes into a semi-lattice of form:

$$\top = \texttt{TREE} \leq \texttt{DAG} \leq \texttt{CYCLE} = \bot$$

Thus, every TREE is also a DAG, and everything is a CYCLE. This lattice allows us to be over-conservative in certain occasions, since categorizing every node to CYCLE only reduces the effectiveness of the analysis, not the soundness.

4 Using a Dataflow Analysis in LLVM to Identify Shapes

To implement the paper's framework in LLVM, we designed a dataflow analysis that uses the dataflow framework we developed during the assignments of this class. The analysis is a forward analysis that runs on each function in the program.

The lattice element consists of a tuple of the following:

1. A direction matrix D that maps every pair of pointers to a bit value.
2. An interference matrix I that maps every pair of pointers to a bit value.
3. A mapping S from pointers to shapes.

The direction and interference matrices are essentially bitvector sublattices, with a Top representing that no pair of pointers has a direction/interference relation, and a Bottom representing that all pointers are reachable or interfere with each other. At the entry block, the direction and interference matrices are initialized to Top. The meet operator for this relation is union ($\cup$), since the direction/interference relations represent that a pair of pointers *might* have a relation.

The meet operator for the shape mapping is defined just as in the original paper [4]:

	TREE	DAG	CYCLE
TREE	TREE	DAG	CYCLE
DAG	DAG	DAG	CYCLE
CYCLE	CYCLE	CYCLE	CYCLE

For the entry state of the analysis, we initialize the shape mapping to TREE for all pointers, and initialize the direction and interference matrices to Top (no pair of pointers have a relation). At the end of the analysis, we meet over all exits to obtain the final shape mapping for each pointer.

4.1 Handling LLVM Instructions

Although the set of rules for updating the interference, direction, and shape values for each pointer are defined in [4], it was not exactly trivial to map these to LLVM instructions. We describe the semantics of each instruction below:

Calls to malloc: Just as in the original paper, calls to malloc initialize a "new" pointer, which has no direction or interference relations (other than self-relations), and whose shape is initialized to TREE.

Loads: An LLVM LoadInst is equivalent to the p = q->f; semantics described in the original paper. We summarize the updates as:

1. All pointers with a path to q now have a path to p
2. All pointers with a path *from* q now have a path *to* p.
3. All pointers that interfere with q now have a *path* to p.

4. All pointers that have a path *from* `q` now have a path *from* `p`.

Stores: An LLVM `StoreInst` is equivalent to the `p->f = q;` semantics described in the original paper. We summarize the updates as:

1. Clearly, `p` now leads to `q`.

2. All pointers that lead to `p` now potentially lead to `q`.

3. All pointers that lead to `p` now potentially lead to all pointers `q` leads to.

4. All pointers that lead to `p` now potentially interfere with all pointers `q` interferes with.

GetElementPointer: This is a instruction unique to LLVM, which is responsible for computing the address of a particular member of a structure or array. It is usually used to compute the address of a particular field of a structure (or even a particular field in a structure in an array of structures).

This instruction is not exactly covered by the original paper. We treat this instruction as pointer assignment; this is assuming that the 'source' pointer is actually a pointer to the same heap object that is computed by the GEP instruction. This assumption holds true for most struct field accesses.

Bitcast / other instructions: Few other instructions are used with pointers, and they mostly preserve any pointer relations. The most common example is Bitcast, which simply casts a pointer to another type, without changing which *heap object* it points to.

PHI nodes: We treat PHI nodes as explicit merge points in the dataflow analysis. For each pointer, we take the union of the direction and interference relations of all incoming pointers. We also merge the shapes, using the merge operator described in the previous section.

5 Interprocedural Shape Identification via Function Summaries

So far, we have described how we implemented the shape analysis from the original paper in our dataflow framework. However, we found that the analysis was not very useful in practice, because it was not interprocedural. In particular, functions that modify heap objects typically are broken into many individual functions that each perform one simple operation (e.g. insertion and removal from a linked list are often distinct functions). These functions typically modify the shape of arguments, or return a result whose chape is a function of the shape of the input argument.

Without an interprocedural analysis, function calls must be assumed to cause all arguments and return values to all have direction and interference relationships, and forces all of these pointers to be categorized as `CYCLE`. This is clearly not very useful, since most nontrivial functions contain function calls.

Since actual use of heap structures such as linked lists, trees, or DAGs requires reasoning about the heap structures across function boundaries, we added a simple interprocedural analysis that uses *function summaries* to propagate shape information across Call instructions in the dataflow analysis transfer function.

5.1 Per-callsite Function Summaries

When a callsite is encountered, a recursive call is made to perform the shape analysis on the callee, *with the initial state of the arguments being copied from the caller.* This means that all direction and interference relations *at the callsite* will be available during the shape analysis of the callee. After the callee has been analyzed, the direction and interference relations are copied back to the caller.

We found that it is not sufficient to only copy the argument and return value relations, since the function may also modify the shape of other pointers reachable from the arguments. The per-instruction transfer function already accounts for this possibility, but it only considers relations to pointers that are already present in the interference/direction matrices. To account for this, we copy the entire direction/interference state of the caller into the callee when computing function summaries. Values in the caller are mapped to a distinct name in the callee, allowing the callee to modify the shape of the caller's pointers and then allowing that state to be copied back into the caller after the *CallInst*.

This is fairly expensive, since it requires a separate (potentially recursive) dataflow analysis for each callsite. However, it is also very precise, since it allows us to propagate shape information across function boundaries.

This interprocedural analysis does not currently support recursive function calls. It also fails to reason about side-effects involving globally accessible heap pointers.

6 Using Shapes to Improve Alias Analysis

While the benefit of shape analysis mainly manifests in loop parallelization, implementing such an application is time-consuming and unachievable in the timespan of this project. Instead, we focus on more direct results of shape analysis: i.e. its effect on alias analysis.

6.1 Alias Analysis

Alias analysis is a form of pointer analysis that answers the question of whether two pointers potentially alias the same memory object. In LLVM in particular, there are four ways to express the alias relationship between two pointers:

- `NoAlias`: Two pointers are guaranteed to be pointing to different memory locations.
- `MustAlias`: Two pointers always point to the same memory location.
- `PartialAlias`: There are overlaps between objects pointed by the two pointers (e.g. one points to address 0 with size 16, the other points to address 8 with size 8).
- `MayAlias`: The alias relationship between two pointers are undecidable.

Alias relations are symmetric.

6.2 From Shape to Alias

Since `NoAlias`, `MustAlias`, and `PartialAlias` are all precise answers already, we focus on converting `MayAlias` relationships into `NoAlias`. To do so, we list two optimization steps from shape analysis to alias analysis: for two pointers A and B that do not have relation `MayAlias`,

1. If A is a parent of B or if A is alias to a parent of B, and A has shape `TREE` or `DAG`, then A and B have relationship `NoAlias`. Similar property holds if B is a parent of A.

2. If A and B are in different subtrees of a common ancestor (either through equality test or aliasing), and the ancestor has shape `TREE`, then A and B have relationship `NoAlias`.

Note that we are allowed to be over-conservative by undercounting the number of parents and ancestors. Indeed, supplying an incomplete list of parents or ancestors only results in missing identifiable `NoAlias` relations, and would not produce incorrect deduction.

6.3 Finding Subtrees

We record parents of each node using a mapping `P` from a node to a set of nodes that are definitely its parents. While one might naively assume that one can obtain the ancestors of each node from the direction matrix, this is not a valid approach because the direction matrix can only identify *potential* ancestors, not *definite* ones. As a result, we define a mapping `A` from a node to a set of its definite ancestors. For each pointer instruction we encounter during our dataflow analysis, `P` and `A` are generated as following:

Instruction	P	A
`p = q`	$\mathtt{P[p]} \leftarrow \mathtt{P[q]}$	$\mathtt{A[p]} \leftarrow \mathtt{A[q]}$
`p = q->f`	$\mathtt{P[p]} \leftarrow \{\mathtt{q}\}$	$\mathtt{A[p]} \leftarrow \mathtt{A[q]} \cup \{\mathtt{q}\}$
`p->f = q`	$\mathtt{P[q]} \leftarrow \mathtt{P[q]} \cup \{\mathtt{p}\}$	$\mathtt{A[q]} \leftarrow \mathtt{A[q]} \cup \mathtt{A[p]} \cup \{\mathtt{p}\}$
`n = phi(p, q)`	$\mathtt{P[n]} \leftarrow \mathtt{P[p]} \cap \mathtt{P[q]}$	$\mathtt{A[n]} \leftarrow \mathtt{A[p]} \cap \mathtt{A[q]}$

Next, we use a mapping `subtree` from pairs of nodes to a semi-lattice, defined as:

$$\mathtt{subtree[p,q]} = \mathtt{x} \Longrightarrow \mathtt{q} \text{ is an ancestor of } \mathtt{p} \text{ and } \mathtt{p} \text{ is reachable from } \mathtt{q} \to \mathtt{x}$$

If `p` can be accessed from multiple fields of `q`, set `subtree[p, q]` to $\perp$, and if `q` is not an ancestor of `p`, `subtree[p, q]` is $\top$. `subtree` is generated by:

Instruction	subtree
`p = q`	$\forall \mathtt{a} \in \mathtt{A[q]}, \mathtt{subtree[p,a]} \leftarrow \mathtt{subtree[q,a]}$
`p = q->f`	$\forall \mathtt{a} \in \mathtt{A[q]}, \mathtt{subtree[p,a]} \leftarrow \mathtt{subtree[q,a]};$ $\mathtt{subtree[p,q]} = \mathtt{subtree[p,q]} \wedge f$
`p->f = q`	$\forall \mathtt{a} \in \mathtt{A[p]}, \mathtt{subtree[q,a]} \leftarrow \mathtt{subtree[q,a]} \wedge \mathtt{subtree[p,a]};$ $\mathtt{subtree[q,p]} = \mathtt{subtree[q,p]} \wedge f$
`n = phi(p, q)`	$\forall \mathtt{a} \in \mathtt{A[n]}, \mathtt{subtree[n,a]} \leftarrow \mathtt{subtree[p,a]} \wedge \mathtt{subtree[q,a]}$

where $\wedge$ is the meet operator for the semi-lattice of `subtree`. We add the above elements (`P`, `A`, and `subtree`) to our dataflow analysis described in section 4, and modify the meet operator accordingly.

Since by "different subtree" in step 2, we are referring to them being disjoint, "$\neq$" of values of `subtree` is only defined when the values are neither $\top$ nor $\perp$. With this definition of $\neq$, we can now implement the two steps in section 6.2:

1. Run LLVM's native alias analysis.
2. For any two pointer pairs `p` and `q` that have `MayAlias` relation:
    - Iterate through `P[p]`. If any entry matches or is alias to q, update the relation between `p` and `q` to `NoAlias`.
    - Iterate through `P[q]`. If any entry matches or is alias to p, update the relation between `p` and `q` to `NoAlias`.
    - Iterate through every pair $(\mathtt{a_p} \in \mathtt{A[p]}, \mathtt{a_q} \in \mathtt{A[q]})$. If $\mathtt{a_p}$ is equal or alias to $\mathtt{a_q}$ and $\mathtt{subtree[p,a_p]} \neq \mathtt{subtree[q,a_q]}$, update the relation between `p` and `q` to `NoAlias`.

7 Evaluation

For our evaluation, we constructed four benchmark exmaples with various combinations of trees, DAGs, and cycles. We run shape analysis of these programs using our shape analysis (implemented for LLVM 14), and evaluate the effectiveness using two metrics:

1. What percentage of pointers can we identify the shape correctly?
2. What percentage of `MayAlias` can be converted into a `NoAlias`?

Benchmark	**# Pointers**	**✓Shape**	**# Alias**	`MayAlias`	`MayAlias` →`NoAlias`
`Linked Lists`	11	11	3	0	0
`DAG`	20	20	16	0	0
`Cycle`	21	21	21	6	3
`Interprocedual`	13	8	6	3	2

Figure 2: Test results of various benchmarks. Entries of a row from left to right are: benchmark name, number of pointers, number of pointers with correctly identified shape, number of alias relations, number of `MayAlias` relations, and number of `MayAlias` relations successfully converted to `NoAlias` relations.

We present our result of the four benchmarks in figure 2. We only considered the relations between pointers which were the subject of a load or store, which is why the number of alias relations is much smaller than the number of distinct pairs of pointers. For most benchmarks, we have successfully identified the shape of all pointers. For `Interprocedual`, the five failed shape identifications are due to function calls of `printf` and `fgets`. Since `printf` could potentially alter the heap, all known matrix and shape values need to be discarded (`D` and `I` to 1, `S` to `CYCLE`).

For improvement on alias analysis, we first note that LLVM can already analyze simple examples like linked lists and DAG. Benchmark `Cycle` uses a combination of `CYCLE`s and `TREE`s to create uncertainty for LLVM, but our shape analysis successfully identifies the structure and the consequent alias relationship. Benchmark `Interprocedual` implements operations of a stack structure, with separate functions for pushing and popping using a linked list. Our shape analysis correctly assesses that the stack structure is always a `TREE`regardless of pushes and pops, and thus one would never encounter the same node twice when walking through the stack.

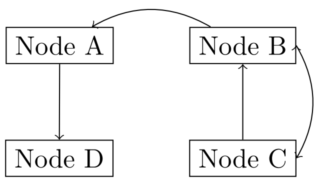


Figure 3: The points-to relationships between heap objects in `test_cyc`

A more detailed depiction of the benchmark `Cycle` is shown in figure 3. In the graph, Node B is in a cycle with Node C, while also having a path leading to Node A and Node D. Our shape analysis successfully identifies B and C as `CYCLE`, and A and D as `TREE`. As a result, it reasons that A and D must be different, and any fields of A must be different from any fields of D. LLVM, on the other hand, fail to deduce the `NoAlias` relation between A and D.

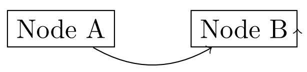


Figure 4: The points-to relationships between heap objects in `test_interp`. This example is exceedingly simple, as it was designed with two separate functions (to test our interprocedural support), one for pushing and one for popping, treating the linked list as a stack.

Benchmark `Interprocedual` is depicted in figure 4. The program begins with an empty stack. It then pushes in node A and node B and immediately pops them out. LLVM fails to recognize that the two popped nodes are different, while our shape analysis lists every pointer to the structure as `TREE`, and thus sets the relation between all popped nodes as `NoAlias`.

8 Conclusion and Future Work

We were able to successfully implement a shape analysis for LLVM, and improved upon it by adding interprocedural support and reasoning about subtree relationships. Our shape analysis is able to identify the shape of nearly all pointers in our benchmark examples, and improve the LLVM alias analysis by converting `MayAlias` to `NoAlias` in some cases.

However, our shape analysis has a few notable limitations. We assume that a GEP instruction is computing an address *within* the heap object that is given by the base address, so we would reason about arrays of structures as if they were a single object. Furthermore, since we followed the algorithm in [4] we do not reason precisely about which struct member is written to, so overwrites to an existing struct member do not clear any of the relations caused by the member's previous value. Also, we did not have time to evaluate against pointer analyses other than the "basic" alias analysis provided by LLVM.

Future work should consider how this technique can be modified to support reasoning about individual nested structures and individual structures within arrays, as well as how an accurate shape analysis can be applied in practice to enable optimizations such as loop parallelization.

9 Work Distribution

The work was distributed 50/50.

References

[1] V. Kanvar and U. P. Khedker, "Heap abstractions for static analysis," *ACM Comput. Surv.*, vol. 49, jun 2016.

[2] P. Liu, Y. Li, B. Swain, and J. Huang, "Pus: A fast and highly efficient solver for inclusion-based pointer analysis," in *Proceedings of the 44th International Conference on Software Engineering*, ICSE '22, (New York, NY, USA), p. 1781–1792, Association for Computing Machinery, 2022.

[3] P. M. Gharat, U. P. Khedker, and A. Mycroft, "Generalized points-to graphs: A precise and scalable abstraction for points-to analysis," *ACM Trans. Program. Lang. Syst.*, vol. 42, may 2020.

[4] R. Ghiya and L. J. Hendren, “Is it a tree, a dag, or a cyclic graph? a shape analysis for heap-directed pointers in c,” in *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL ’96, (New York, NY, USA), p. 1–15, Association for Computing Machinery, 1996.