A Survey of Shape Analysis Techniques

Vedant Kumar, vsk@berkeley.edu

May 11, 2014

1 Introduction

Shape analysis techniques statically determine whether the contents of a program's memory can satisfy a set of structural invariants. The basic problems in shape analysis are (1) to decompose a program into a set of locations, (2) to conservatively determine what these locations point to, and (3) to use this 'points-to' information to uncover the shape of in-memory structures. In short, the goal of shape analysis is to answer questions about a program's memory-usage patterns without actually running it.

The ability to answer these questions is powerful and broadly applicable. For example, in the field of compile-time optimization the task of automatically transforming a program into parallel fragments requires shape analysis to flag conflicting memory operations [LH88]. The graph structures used in several shape analysis methods (e.g [HPR89], [LH88], and [SRW96]) are amenable to solving this problem and related dependence analysis problems. In the field of program verification, shape analysis can be used to check if variables satisfy sophisticated invariants such as 'is-a-list?', 'is-a-tree?', and 'contains-cycles?' [SRW96]. Static enforcement of such shape invariants is useful.

This paper is organized in a top-down fashion. So far in Section 1 we have characterized shape analysis, stated the basic problems in the field, and motivated further study. Section 2 delves into the fundamental approaches to solving shape analysis problems. Section 3 discusses the main achievements in the field. Section 4 presents the challenges remaining in the field, and Section 5 concludes.

2 Solving shape analysis problems

There are different approaches to solving the shape analysis problem. Instead of describing each approach individually, we discuss their commonalities and make generalizations whenever appropriate. In this section, we focus on the following recurring motifs: specialized program representation, abstract interpretation, and the construction of shape graphs via lattice operations.

2.1 Program representation

The choice of program representation affects the scope and complexity of any analysis. In the literature there are two common classes of representations. The first is the Lisp-y `cons`-cell model espoused by [JM82], [LH88], and [SRW96]. The other is the C-like model employed by [Steen95], [AG01], and [KS13]. The Lisp model handles loads, stores, reference passing, and recursive data structures. The C model is similar, but considers pointer dereferencing and a slew of unsafe memory operations as well. While the C model enables analysis of a larger class of programs, some authors eschew it in order to focus on fundamentals. Apart from the choice of program representation, preprocessing is a common way to simplify analyses. For example, 'kill' instructions are injected before all assignments in [SRW96] to make the dataflow relations more compact. Such 'frontend-level' differences can be significant and drive diversity in the literature.

2.2 Abstract interpretation

Abstract interpretation is akin to normal program interpretation (i.e execution), but with a specialized operational semantics. The key ideas are (1) to replace concrete program values with abstract values, and (2) to simulate the resulting program under the new operational semantics. Shape analysis algorithms utilize abstract interpretation to compute dataflow relations between the various locations in a program, resulting in a model of in-memory structures.

The first step in abstract interpretation is to safely replace concrete program values with abstract ones. In [JM82] [1], the authors specify a concrete-value lattice A, an accompanying abstract approximation lattice A', and a pair of functions which translate between the two lattices: $abs : A \to A'$ and $conc : A' \to A$. They then state an important *safe approximation criterion*: given an n-ary concrete operation $\varphi : A^n \to A$, an approximation operation $\varphi' : (A')^n \to A'$ is safe if for all $a_1, ..., a_n$: $\{abs(\varphi(a_1, ..., a_n)) \mid \forall i.a_i \in conc(a'_i)\} \sqsubseteq \varphi'(a'_1, ..., a'_n)$. The intuition behind this requirement is that the abstract n-ary operation must contain all values that result from abstracting any feasible concrete version of the abstract operation. All operations in the abstract operational semantics must satisfy this criterion.

The second step in abstract interpretation is to define concrete and abstract versions of the program state. In [JM82], the concrete state of a program is an element $\sigma \in Q \times A \times L$, where Q is the set of control-flow edges and L is the set of locations. The abstract state replaces the location set with a token set T. In addition, each state is equipped with a partial *retrieval function* ($\tau : T \to A' \times 2^{T \times T}$), which either maps tokens to atomic abstract values, or to two more tokens. These τ-functions allow the abstract representation to model exactly two program data types: atomic values and binary lists.

[1] To focus our discussion, we treat definitions from [JM82] as representative throughout this subsection.

The third step in abstract interpretation is to define the semantic action of each program construct. This is a difficult but mechanical step, so we curtail its discussion here. It suffices to say that coupled with an equivalence relation on abstract states, these semantics allow computation of the subset $S' \subseteq \Delta$ of reachable states in a program. The set Δ is ordered by subset inclusion: by the Tarski-Knaster theorem, a least fixed-point to a function $f : \Delta \rightarrow \Delta$ exists provided that there are no infinitely ascending chains in Δ and that f is continuous. Let f be our abstract simulation function: we iterate it until the least fixed-point is reached, thereby completing the abstract interpretation. This simulation can capture accurate descriptions of recursive data structures, perform interprocedural analysis, and trade speed for precision by adjusting its token sets. It is a flexible and foundational concept.

2.3 Shape graphs

Shape graphs are an alternate form of abstract state representation introduced in [SRW96]. We focus on shape graphs because they have been used to drive key advances in the field. The shape graph is conceptually similar to the alias graph in [LH88] and (to some extent) the retrieval functions of [JM82], so our discussion will not be too idiosyncratic. We define shape graphs [2], examine how they are constructed, discuss some interesting properties, and compare them to other abstract state representations.

The shape graph is a finite digraph consisting of *shape nodes*, *variable edges*, and *selector edges*. It is formally defined as a tuple $\langle E_s, E_v \rangle$ in [SRW96]. Variable edges of the form $[x, n]$ reside in E_v, where x is a pointer variable and n is a shape node (i.e a graph vertex). Selector edges of the form $[s, sel, t]$ reside in E_s, where sel is a selector and s and t are shape nodes. The sets E_v and E_s fully describe the shape graph. The class of deterministic shape graphs is $\mathcal{DSG}$: it contains graphs which may only represent the ephemeral effects of one execution sequence. Formally, graphs in $\mathcal{DSG}$ satisfy $\forall x.|E_v(x)| \leq 1$ and $\forall x.\forall sel.|E_s(x, sel)| \leq 1$. We require a separate class of static shape graphs to conservatively represent subsets of $\mathcal{DSG}$ for our abstract semantics. This new class is the lattice $\mathcal{SSG}$, ordered by component-wise subset inclusion.

Our goal of computing useful static program representations is within reach. First, we define a concrete semantics as a $\mathcal{DSG}$-transformer: $[\![st]\!]_{\mathcal{DSG}} : \mathcal{DSG} \rightarrow \mathcal{DSG}$. Second, we represent the control flow nodes of the program as a set V. Third, we define a *collecting semantics*, $c : V \rightarrow 2^{\mathcal{DSG}}$, which generates all feasible deterministic shape graphs for a given program point. Explicitly, $c(v) = \{[\![st(v_k)]\!]_{\mathcal{DSG}}(..., [\![st(v_1)]\!]_{\mathcal{DSG}}(\langle \phi, \phi \rangle)) \mid v_1, ..., v_k \in pathsTo(v)\}$, where $pathsTo : V \rightarrow 2^V$ yields all paths through the CFG which may transition to v. Next, we define an abstraction function, $\alpha : 2^{\mathcal{DSG}} \rightarrow \mathcal{SSG}$. Finally, we define the abstract semantics as a $\mathcal{SSG}$-transformer: $[\![st]\!]_{\mathcal{SSG}} : \mathcal{SSG} \rightarrow \mathcal{SSG}$. Putting all the pieces together, the shape analysis algorithm can be thought of as the function composition $\alpha \circ c : V \rightarrow 2^{\mathcal{DSG}} \rightarrow \mathcal{SSG}$ followed by a fixed-point iteration.

[2]All results in this subsection are drawn directly from [SRW96], unless explicitly mentioned otherwise.

A discussion of the exact concrete and abstract semantics is omitted because it would require pages of dense equations and exposition. For reference, figures 2 and 6 in [SRW96] fully specify $[\![st]\!]_{\mathcal{DSG},\mathcal{SSG}}$.

Shape graphs have interesting properties, such as a fluid naming scheme and strong nullification. The naming scheme is simple: the shape nodes at a given CFG node v are named by the set of v-local variables which all point to the same run-time location. Variables can be added, removed, and moved between shape nodes to model $\mathcal{DSG}$-transformations as precisely as possible. Shape nodes are not forced to irreversibly partion memory with a fixed variable labelling: they may be *materialized* (split into more granular nodes) as well as *un-materialized* (coalesced into summary nodes) as necessary. The variable-set naming scheme enables the second property, *strong nullification*, the condition that all variable edges emanating from a shape node are removed on a *nil*-assignment. This can be expressed as $[\![x := \mathbf{nil}]\!]_{\mathcal{SSG}}(\langle E_v, E_s\rangle) \approx \langle E_v - [x, *], E_s\rangle$ where all shape nodes n_X s.t $x \in X$ are renamed to $n_{X-\{x\}}$ [3]. Strong nullification is cited as a crucial factor in model-checking the 'is-list?' invariant on list reversal functions: without it, it would be difficult to precisely distinguish the list head pointers of the old and new lists.

In comparison to the method described in [JM82], the shape graph formalism permits more flexibility in terms of supported structural invariants. We are not limited to atoms and binary lists, and have a straightforward way to model-check new structures using graph traversal methods. In comparison to the alias graphs presented in [LH88], shape graphs have a simplfied naming scheme which dispenses with access path concatenation and aggregate labels (i.e node names containing regular expressions). While conceptually similar, these two structures are motivated in very different contexts. The method presented in [LH88] solves the alias analysis problem in order to detect structure access conflicts, while alias analysis is proposed as a basic extension of the method from [SRW96]. Clearly, alias analysis, pointer analysis, and shape analysis are highly interrelated topics.

3 Main achivements

The results in [SRW02] stand out as major achievements in shape analysis. First, the authors characterize the abstract state representations used in previous works as variants of the shape graph. They then build upon shape graphs to create a logical framework for expressing a wide range of structural invariants. These invariants are highly expressive: they may encode type information, structure connectivity properties, as well as ordering properties. In addition, the authors implemented a domain-specific language for 'programming' these invariants. An analyzer-generator was implemented to transform these invariants into a program that enforces them. These results are significant for their general nature and emphasis on practicality.

[3]We omit some details here for simplicity. The actual abstract semantics are not defined in this way.

Over the years there have been several other major achievements in the field. Shape analysis techniques may be applied interprocedurally, due to consistent consideration throughout the literature (e.g in [JM82], [LH88], [SRW*], etc). Work has also been done to reduce the space complexity of shape analysis: merged shape nodes are discussed in [SRW96], while summary nodes and compact hammock graphs are discussed in [LH88]. In [AG98], implmentation techniques are established to make dataflow analysis more efficient (notably demand-driven analysis). The popular work from [Steen95], Steensgaard's almost linear-time points-to algorithm, has been widely implemented (e.g in the LLVM compiler framework).

4 Limitations and open problems

According to results cited in [LH88], precisely solving the structural alias problem is NP-complete, placing fundamental limits on the efficiency of shape analysis algorithms. Moreover, we are faced with an unavoidable loss of precision when collapsing sets of deterministic shape graphs into a single static shape graph. Language features such as non-determinism, destructive updating, and recursion make perfectly precise static shape analysis impossible. It is not clear how to quantify the loss of precision, or if there are techniques that minimize this loss for certain classes of programs. Space limits are also a pressing issue: some algorithms use hardcoded size limits on shape nodes (e.g *sl* limits in [LH88]) or perform merging in order to keep graphs compact, at the expense of precision. Supporting interprocedural analysis and context sensitivity exacerbates the time and space complexity concerns mentioned earlier. Parallelizing the shape analysis algorithm to take advantage of SMP and distributed systems is an underexplored problem. A final non-trivial problem is the difficulty of implementing shape analysis techniques for complex, unsafe languages such as C++.

5 Conclusion

Shape analyses determine the structure of dynamically-updated storage statically, answering questions about how memory is organized. In the area of program verification, shape analysis can be used to statically perform model-checking of functions and structural invariants. In the area of optimization, the closely-related alias analysis and pointer analysis algorithms are crucial for inferring data dependences. Important optimizations such as lifting instructions out of loops, code-elimination, and instruction reordering heavily depend upon this information.

In summary, we have defined shape analysis and discussed its applicability. Common features of shape analysis algorithms are explained in some detail, and some comparative notes are included. Finally, the achievements, limits and open problems of the field are presented.

6 Bibliography

1. D. Atkinson and W. Griswold. Effective Whole-Program Analysis in the Presence of Pointers. In *SIGSOFT '98/FSE-6 Proceedings of the 6th ACM SIGSOFT'*, pages 46-55, 1998.

2. D. Atkinson and W. Griswold. Implementation Techniques for Efficient Data-Flow Analysis of Large Programs. In *Proceedings of the International Conference on Software Maintenance*, pages 52-61, 2001.

3. LLVM, `http://llvm.org/`.

4. S. Horwitz, P. Pfeiffer, and T. Reps. Dependence Analysis for Pointer Variables. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 28-40, 1989.

5. N. Jones and S. Muchnick. A Flexible Approach to Interprocedural Data Flow Analysis and Programs with Recursive Data Structures. In *ACM Symposium on Principles of Programming Languages*, pages 66-74, 1982.

6. G. Kastrinis and Y. Smaragdakis. Hybrid Context-Sensitivity for Points-To Analysis. In *Proceedings of the 34th ACM SIGPLAN conference on Programming Language Design and Implementation*, pages 423-434, 2013.

7. J. Larus and P. Hilfinger. Detecting Conflicts Between Structure Accesses. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 21-34, 1988.

8. M. Sagiv, T. Reps, and R. Wilhelm. Solving Shape-Analysis Problems in Languages with Destructive Updating. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages.* 1996.

9. M. Sagiv, T. Reps, and R. Wilhelm. Parametric Shape Analysis via 3-Valued Logic. In *ACM Symposium on Principles of Programming Languages.* 2002.

10. B. Steensgaard. Points-to Analysis in Almost Linear Time. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages*, pages 32-41, 1996.

11. T. Tok, S. Guyer, and C. Lin. Efficient Flow-Sensitive Interprocedural Data-Flow Analysis in the Presence of Pointers. In *LNCS, Springer-Verlag*, pages 17-31, 2006.

12. X. Zhang, M. Naik, and H. Yang. Finding Optimum Abstractions in Parametric Dataflow Analysis. In *Proceedings of the 34th ACM SIGPLAN conference on Programming Language Design and Implementation*, pages 365-375, 2013.