Revamping TVLA: Making Parametric Shape Analysis Competitive

Technical Report TR-2007-01-01

Igor Bogudlov[1], Tal Lev-Ami[1], Thomas Reps[2], and Mooly Sagiv[1]

[1] School of Comp. Sci., Tel Aviv Univ., {igorbogu,tla,msagiv}@post.tau.ac.il
[2] Comp. Sci. Department, Univ. of Wisconsin, Madison, reps@cs.wisc.edu

Abstract. TVLA is a parametric framework for shape analysis that can be easily instantiated to create different kinds of analyzers for checking properties of programs that use linked data structures. We report on dramatic improvements in TVLA's performance, which make the cost of parametric shape analysis comparable to that of the most efficient specialized shape-analysis tools (which restrict the class of data structures and programs analyzed) without sacrificing TVLA's parametricity. The improvements were obtained by employing well-known techniques from the database community to reduce the cost of extracting information from shape descriptors and performing abstract interpretation of program statements and conditions. Compared to the prior version of TVLA, we obtained as much as 50-fold speedup.

1 Introduction

The abstract-interpretation technique of [1] for static analysis allows one to summarize the behavior of a statement on an infinite set of possible program states. This is sometimes called an *abstract semantics* for the statement. With this methodology it is necessary to show that the abstract semantics is *conservative*, i.e., it summarizes the (*concrete*) *operational semantics* of the statement for every possible program state. Intuitively speaking, the operational semantics of a statement is a formal definition of an interpreter for this statement. This operational semantics is usually quite natural. However, designing and implementing sound and reasonably precise abstract semantics is quite cumbersome (the best induced abstract semantics defined in [1] is usually not computable). This is particularly true in problems like shape analysis and pointer analysis (e.g., see [2, 3]), where the operational semantics involves destructive memory updates.

1.1 An Overview of the TVLA System.

In this paper, we review recent improvements to TVLA (**T**hree-**V**alued-**L**ogic **A**nalyzer), a system for automatically generating a static-analysis implementation from the operational semantics of a given program [4, 3]. In TVLA, a language's small-step structural operational semantics is written in a meta-language based on First-Order Logic with Transitive Closure (FO(TC)). The main idea is that program states are represented as logical structures, and the program's transition system is defined using first-order logical formulas. The abstraction is controlled using a set of *Instrumentation Predicates*,

which are defined using FO(TC) formulas and dictate what extra information is tracked for each program state. *Integrity constraints* can be provided in the form of FO(TC) formulas; these express invariant properties of the operational semantics (e.g., each program pointer can point to at most one concrete location).

TVLA is a parametric framework based on the theory of [3]. Given the concrete operational semantics, instrumentation predicates, and integrity constraints, TVLA automatically generates the abstract semantics, and, for each program point, produces a conservative abstract representation of the program states at that point. The idea of automatically generating abstract semantics from concrete semantics was proposed in [5]. TVLA is intended as a testbed in which it is easy to try out new ideas for shape abstractions.

One of the attractive features of the TVLA system is the fact that it is very flexible yet it guarantees soundness. For example, it can be utilized to handle prove properties of concurrent systems even with unbounded number of threads [6, 7]. For a short formal description of the theory behind TVLA we refer the reader to [8].

A unique aspect of TVLA is that it automatically generates the abstract transformers from the concrete semantics; these transformers are (i) guaranteed to be sound, and (ii) rather precise—the number of false alarms reported in our applications is very small. The abstract transformers in TVLA are computed in 4 stages:

(i) *Focus*—a partial concretization operation in which each heap cell that will be updated is materialized as a singleton (non-summary) individual, so that it is possible to perform a strong update;

(ii) *Update* — in which the update formulas are evaluated using Kleene semantics on the abstract structure to achieve a sound abstract transformer;

(iii) *Coerce* — a semantic reduction in which an internal Datalog-style constraint solver uses the instrumentation predicates and integrity constraints to improve the precision of the analysis; and

(iv) *Blur*—in which the abstraction function is re-applied (which ensures that the analyis terminates).

1.2 Contributions.

Compared to specialized shape-analysis approaches, the above operations incur interpretative overhead. In this paper, we show that using techniques taken from the realm of databases, such as semi-naive evaluation and query optimization [9], TVLA reduces this overhead, and thereby achieves performance comparable to that of state-of-the-art specialized shape analysis without changing TVLA's functionality.

Extensions to TVLA have been developed to handle interprocedural analysis [10] and concurrency [6, 7]. Our improvements support both these extensions.

2 Key Improvements

2.1 Preliminaries

The bottleneck in previous versions of TVLA has been Coerce, which needs to consider interactions among all the predicates. The complexity of Coerce stems from the fact that the number of constraints is linear in the size of the program, and the number of tuples

that need to be considered during the evaluation of a constraint is exponential in the number of variables in the constraint.

Coerce translates the definitions of instrumentation predicates and the integrity constraints to Datalog-like constraint rules of the form $R_1 \wedge ... \wedge R_k \Rightarrow H$[3], where the left hand side is called the base of the rule and is a conjunction of general formulas, and H is called the head of the rule and is a literal. The head and the conjuncts of the base are called atoms. Each atom induces a relation of the tuples that satisfy it. Coerce then applies a constraint by searching for assignments to the free variables of the rule such that the base is known to hold (i.e., evaluates to 1), and the head either does not hold (i.e., evaluates to 0) or may not hold (i.e., evaluates to $\frac{1}{2}$). In the first case it safely discards the structure as inconsistent, and in the second case it attempts to coerce the value to 1 by updating tuples of the head relation in the structure (more details can be found in [3]).

This search for violating assignments is done by performing k Datalog-join[4] operations on $\overline{H}, R_1, ..., R_k$ and projecting the result on the free variables of H in a way similar to evaluation of a Datalog rule

$$H \text{ :- } R_1, ..., R_k.$$

Worst-case complexity of the constraint evaluation is proportional to the product of the relation sizes $|\overline{H}||R_1|...|R_k|$, but in practice depends very much on the order of the Datalog-join operations and their dependencies.

Constraints in the Coerce procedure are organized into strongly connected components according to their inter-dependencies. Each component is examined in turn, and its constraints are repeatedly evaluated and applied. This process continues until a fixed point is reached in a way similar to evaluation of Datalog rules in databases [9].

Unlike Datalog, however, in TVLA's three-valued logic setting the atoms R_i may be general formulas with transitive closure, including negation. This is possible because in the realm of TVLA 3-valued logic, the Coerce procedure is guaranteed to converge [3]. We view negated predicates as dual relations in which all 1-tuples (i.e. tuples evaluating to 1 in the structure) in the original relation evaluate to 0, and 0-tuples in the original relation evaluate to 1. The additional $\frac{1}{2}$ value can be viewed as an absence of information and the goal of Coerce is to minimize the amount of $\frac{1}{2}$ tuples with respect to the given constraints, thus maximizing the available information and improving analysis precision.

In practice, we do not maintain a separate relation for the negated predicates but rather compute them on demand. We only store the potentially satisfying 1-tuples and $\frac{1}{2}$-tuples of the predicate in the predicate relation for memory efficiency reasons.

2.2 Improvements Overview

In order to improve Coerce performance, we address several factors that affect its running time. We adapt the database semi-naive evaluation technique to TVLA, to reduce

[3] Here and in the rest of the paper, $\wedge$, $\vee$ and the overline denote the logical and, or and not operations, respectively, in 3-valued Kleene logic.

[4] To distinguish between the database join operation $\bowtie$ and the lattice join operation $\sqcup$, the former will be called Datalog-join.

the number of tuples examined during constraint evaluation and avoid unnecessary computations. We use formula rewriting and query optimization to further reduce average constraint evaluation time. Finally, we introduce the concept of multi-constraints to reduce the effective number of constraints evaluated during the Coerce procedure. These improvements are discussed below.

Many other improvements and techniques were introduced into TVLA, including the following: precomputing information that only depends on constraint structure, such as atom types and dependencies; tracking modified predicates in each structure for quick constraint filtering; caching and on-demand recomputation of transitive closure; caching of recently-used predicate values and tuple lists for predicate negation.

In addition, we did extensive re-engineering and optimization of the TVLA core geared toward improved performance.

2.3 View Maintenance with Semi-Naive Evaluation

Semi-naive evaluation is a well-known technique in database view maintenance to speed up the bottom-up evaluation of Datalog rules [9]. On each iteration of the algorithm, it evaluates the rules in *incremental* fashion; i.e., it only considers variable assignments containing relation tuples that were changed during the previous iteration. All other assignments must have been examined during the previous iteration and thus cannot contribute any new information. Because the number of changed tuples is usually small compared to the size of the relation, this avoids a considerable amount of computation.

The following outlines the standard semi-naive evaluation algorithm:

for $i := 1$ **to** m **do begin**
    $\Delta P_i := Eval(p_i, P_1, ..., P_m)$;
    $P_i = \Delta P_i$;
end;
repeat
    for $i := 1$ **to** m **do**
        $\Delta Q_i := \Delta P_i$
    for $i := 1$ **to** m **do begin**
        $\Delta P_i := EvalIncr(p_i, P_1, ..., P_m, \Delta Q_1, ..., \Delta Q_m)$;
        $\Delta P_i := \Delta P_i - P_i$;
    end;
    for $i := 1$ **to** m **do**
        $P_i := P_i \cup \Delta P_i$;
until $\Delta P_i = \emptyset$ **for all** $i, 1 \leq i \leq m$;

The evaluation procedures in the above algorithm are defined as follows in the context of Coerce: Let $EvalRule(r, T_1, ..., T_n)$ be the algebraic expression which computes the violating assignments for the constraint r when relation T_i is used for the atom i. Then the relation for the predicate p, denoted by $Eval(p, P_1, ..., P_m)$, is the union of the tuples produced by $EvalRule$ for each constraint r whose head is the predicate p.

$\Delta P_1, ..., \Delta P_n$ are the *incremental relations*: the set of tuples added to $P_1, ..., P_n$ in the last round of computation. The incremental evaluation procedure *EvalIncr* uses both the full relations P_i and the incremental tuples ΔP_i to update the relation. The new tuples produced by the rule can be found if we substitute the full relation for all but one of the atoms, and substitute only the incremental tuples for the remaining atom. This is done for every atom in the rule that has new tuples.

We thus define the incremental relation for a rule r to be:

$$EvalIncr(r, T_1, ..., T_n, \Delta T_1, ..., \Delta T_n) := \bigcup_{1 \leq i \leq n} EvalRule(r, T_1, ..., T_{i-1}, \Delta T_i, T_{i+1}, ..., T_n)$$

And the incremental relation for a predicate p, denoted by

$$EvalIncr(p, P_1, ..., P_m, \Delta P_1, ..., \Delta P_m);$$

is the union of what $EvalIncr$ produces for each rule r for predicate p.

Note that in the standard semi-naive evaluation algorithm, all the tuples in the structure are examined during the first iteration, while subsequent iterations proceed incrementally.

We take the idea of semi-naive evaluation one step further by using properties of the TVLA abstract transformer step to avoid fully evaluating the constraints even once: all of the constraints hold before the *Focus* and *Update* steps, and thus the set of violating assignments for the structure is initially empty. Any assignment that violates the constraints must therefore be due to a tuple changed by *Focus* or *Update*. Thus we may compute the set of these assignments by incremental evaluation of the rule where ΔP_i are the tuples whose values have been changed by the potentially violating step. We save the structure before and after the potentially violating step, and calculate the difference. This *delta structure* is used during the first iteration of the semi-naive algorithm instead of the full evaluation:

for $i := 1$ **to** m **do begin**
$\Delta P_i := (P_i^{new} \setminus P_i^{old}) \cup \{(tuple, 0) | (tuple, value) \in (P_i^{old} \setminus P_i^{new})\}$
$P_i = \Delta P_i;$
end;

The incremental predicates ΔP_i in the delta-structure contain all the tuples that differ between the new and the old structure, along with their values *in the new structure* (including 0, 1 and $\frac{1}{2}$). Removed tuples, which exist (i.e. evaluate to 1 or $\frac{1}{2}$) in the old structure, but not the new one, are included in the delta structure with the tuple value set to 0 (as stated in formal terms above).

To evaluate a rule r incrementally we should be able to find an increment ΔT_i efficiently for each of the rule atoms according to the definition of $EvalRule$. This is easy to do for literal atoms - a predicate, an equality or their negations. For a predicate atom T, its increment ΔT contains all the 1-tuples (and $\frac{1}{2}-$tuples, if T is the head atom) in the corresponding entry of the delta structure, ΔP. Whereas an increment of a negated predicate $T = \overline{P}$ contains all the 0-tuples (and $\frac{1}{2}-$tuples for the head atom)

in ΔP. For the special case of equality literal, the corresponding predicate is the built-in unary *summary predicate* that determines whether a given node is a *summary* node. (Equality evaluates to $\frac{1}{2}$ for equal summary nodes. See [3]).

However, some constraints' atoms are complex formula, not simple literals. Usually such atoms contain universal quantifiers and transitive closure formulas, that cannot be decomposed into several simpler constraints by TVLA. For such "bad" constraints we check if any of the predicates the complex atom depends on, has been modified. If not, then the incremental evaluation can still be invoked, since the delta is empty. Otherwise the rule is evaluated in a usual non-incremental way using all of the tuples. Indeed, the "bad" constraints now constitute a major bottleneck of the Coerce procedure, since they are not computed incrementally. Solving this problem is one of the future research directions we intend to pursue.

2.4 Query Optimization.

We employ standard database query optimization techniques in order to minimize expected cost of evaluation. These steps are performed on the constraints and update formulas before running the analysis.

In order to evaluate a constraint $R_1, ..., R_k \Rightarrow \overline{H}$, we have to enumerate all the combinations of 1-tuples (i.e. tuples evaluating to 1) from R_1, 1-tuples from R_2, etc. and both 1-tuples and $\frac{1}{2}$-tuples from H. Each such combination induces an assignment on free variables of R_1, ..., H for which the constraint does not hold. The evaluation proceeds recursively: we first generate all satisfying tuples of R_1. Then for each tuple we assign the free variables of R_1 to the appropriate values of the tuple, and enumerate on the rest of the constraint with these variables bound.

Note, that this procedure performance depends greatly on the evaluation order of the atoms $R_1, ..., R_k, H$. We would like to examine the smallest relations first, as well as relations that bind the most variables in subsequent atoms. In the new version of TVLA we examine the types of atoms in the constraint, their arity and mutual dependence in order to find an evaluation order that will minimize the expected cost of Datalog-join operations. Note, that the head relation is also included in the reordering along with the subgoal atoms.

Standard algorithms exist in the realm of database query optimization that suggest such reordering, such as the Wong-Youssefi algorithm [11]. Since we mostly deal with relations of small arity and size, we use a simplified set of heuristics, that prove to work very well in practice. For example, quantified formulas and negated predicates are evaluated last. In the incremental evaluation of the semi-naive algorithm, the delta-predicates are always evaluated first, because they usually contain few tuples.

Another issue to consider is speeding up tuples generation and evaluation. For the predicate relations we maintain both a fast hash map data structure with caching of recently-used predicate values, and a linked list of the tuples, to optimize both single tuple evaluation and satisfying tuples list traversal.

In sparse structures evaluation of negations and quantified formulas can be especially costly. Consequently we transform formulas so that negations only appear before atomic formulas, the $\exists$ and $\forall$ symbols enclose smallest possible formulas and appear at the end of conjunctions or disjunctions. These measures minimize the chance that

we will have to evaluate these formulas, as well as the cost of their evaluation. We also use caching of tuple lists for predicate negation to avoid repeated re-computation in different constraints. Transitive closure values are also cached and recomputed on demand.

2.5 Multi-Constraint Evaluation.

TVLA safety properties are often symmetric, i.e., give rise to a set of constraints, all consisting of the same atoms, such that for each such constraint a different atom serves as the (negated) head of the rule and the rest remain in the base.

For example, the "Unique" property of a predicate is defined:

$$Unique[P] := \forall v_1, v_2 : \overline{P(v_1)} \vee \overline{P(v_2)} \vee (v_1 = v_2)$$

It gives rise to two constraints with the same set of atoms $\{P(v_1), P(v_2), v_1 \neq v_2\}$:

$$P(v_1) \wedge P(v_2) \Rightarrow (v_1 = v_2)$$
$$P(v_1) \wedge (v_1 \neq v_2) \Rightarrow \overline{P(v_2)}$$

We introduced the notion of a *multi-constraint* to represent a set of rules that have the same set of atoms. An atom corresponding to a head of some rule is called a *head atom*. Instead of evaluating these rules one-by-one, we can evaluate them all at once at a cost comparable to that of a single rule evaluation.

A constraint is violated when all of the atoms in the base evaluate to 1 while the negated head evaluates to either 1 or $\frac{1}{2}$. Similarly, a multi-constraint is violated when all of its atoms evaluate to 1, except at most one atom that evaluates to $\frac{1}{2}$ and is the head of some rule. If all of the atoms evaluate to 1, the violation cannot be fixed, and we must discard the structure. Otherwise we fix the predicate tuple evaluating to $\frac{1}{2}$ according to the usual *Coerce* rules [3], effectively choosing the head of the rule to be fixed according to the location of the $\frac{1}{2}$ tuple in the assignment.

We evaluate the multi-constraint efficiently by keeping count of the number of $\frac{1}{2}$ values for an assignment while enumerating the relations' tuples. For non-head atoms we enumerate 1-tuples only. If no $\frac{1}{2}$-tuple has been encountered yet in the current assignment, we will examine both the 1 and $\frac{1}{2}$-tuples for head atoms. As soon as a $\frac{1}{2}$ tuple is encountered, we will examine only the 1-tuples in subsequent head atoms. The generalization of this procedure to semi-naive evaluation is also straightforward.

This technique usually cuts the effective number of constraints in half, and affects the running time of *Coerce* accordingly.

3 Experimental Results

General Benchmarks We incorporated the above-mentioned techniques into TVLA. The empirical results from running the new tool on various benchmark examples are presented in Table 1. Table 1 compares the running time of each analysis with both the previously available TVLA version, as well as some specialized shape-analysis tools [12, 13].

The benchmark suite consisted of the following examples: singly-linked lists operations, including Merge, Delete and Reverse; sorting of linked lists, including insertion

sort, bubble sort, and a recursive version of Quicksort (using the extension of [10]); sorted-tree insertion and deletion; analysis of set data structures from [14]; analysis of the Lindstrom scanning algorithm [15, 16]; insertion into an AVL tree [17].

For each program, Table 1 uses the following shorthand to indicate the set of properties that the analysis established: CL—cleanness, i.e., absence of memory leaks and null-pointer dereferences; DI—data-structure invariants were maintained (e.g., treeness in the case of a tree-manipulation program); IS—the output result is isomorphic to the input; TE—termination; SO—the output result is sorted. The column labeled "Structs" indicates the total number of (abstracted) logical structures attached to the program's control-flow graph at the end of the analysis. "N/A" denotes absence of available empirical data for the tool. "B/S" denotes that the analysis is beyond the scope of the tool. The tests were done on a 2.6GHz Pentium PC with 1GB of RAM running XP.[5]

Table 1. Running time comparison results

Program	Properties	Structs	New TVLA	Old TVLA	[12]	[13]
LindstromScan	CL, DI	1285	8.21	63.00	10.85	B/S
LindstromScan	CL, DI, IS, TE	183564	2185.50	18154.00	B/S	B/S
SetRemove	CL, DI, SO	13180	106.20	5152.80	B/S	B/S
SetInsert	CL, DI, SO	299	1.75	22.30	B/S	B/S
DeleteSortedTree	CL, DI	2429	6.14	47.92	4.22	B/S
DeleteSortedTree	CL, DI, SO	30754	104.50	1267.70	B/S	B/S
InsertSortedTree	CL, DI	177	0.85	1.94	0.89	B/S
InsertSortedTree	CL, DI, SO	1103	2.53	12.63	B/S	B/S
InsertAVLTree	CL, DI, SO	1855	27.40	375.60	B/S	B/S
Merge	CL, DI	231	0.95	4.34	0.45	0.15
Delete	CL, DI	110	0.59	1.50	0.06	N/A
Reverse	CL, DI	57	0.29	0.45	0.07	0.06
InsertSort	CL, DI	712	3.02	23.53	0.09	0.06
BubbleSort	CL, DI	518	1.70	8.45	0.07	N/A
RecQuickSort	CL, DI	5097	3.92	16.04	B/S	0.30
RecQuickSort	CL, DI, SO	5585	9.22	75.01	B/S	B/S

Coerce Performance Table 2 gives further insight to the performance of the Coerce step and the semi-naive technique. We compare the new TVLA version with and without the semi-naive algorithm enabled. The old TVLA version timing is also presented for comparison.

We break down Coerce timing to *eval_inc* - incremental evaluation of constraints; *eval_full* - full non-incremental evaluation that is usually invoked for "bad" constraints that contain complex formula as their atoms, dependent on some modified predicates. "Bad" constraints cannot be evaluated with the incremental evaluation procedure, and full evaluation is used instead; *calc_delta* - calculation of the delta structure in our adapted semi-naive algorithm.

[5] Establishing total correctness of Lindstrom Scan was performed on a 2.4GHz Core 2 Duo with 4GB of RAM. The tests of [13] were done on a 2GHz Pentium Linux PC with 512MB of RAM.

Table 2. Coerce performance comparison

Measurement	New TVLA		Old TVLA
	w/o SNE	w/ SNE	
InsertAVLTree			
Total time	30.470	27.450	375.670
Coerce time	23.540	19.650	355.140
eval_inc	6.287	2.142	N/A
eval_full	16.595	16.595	N/A
calc_delta	0.000	0.545	N/A
DeleteSortedTree			
Total time	135.015	104.516	1267.703
Coerce time	84.070	53.570	1083.520
eval_inc	42.677	8.912	N/A
eval_full	38.893	38.893	N/A
calc_delta	0.000	3.501	N/A
SetRemove			
Total time	152.406	106.297	5152.766
Coerce time	109.690	62.160	4788.580
eval_inc	58.233	10.042	N/A
eval_full	49.069	49.069	N/A
calc_delta	0.000	1.366	N/A

Conclusion Table 1 shows that our techniques indeed resulted in considerable speedup vis-a-vis the old version of TVLA: most of the examples run an order of magnitude faster, and in one case a factor of 50 is achieved. Moreover, the new tool's performance is comparable with that of specialized analysis tools, especially on larger examples.

4 Related and Future Work

In this paper we have shown that database optimizations are powerful enough to make generic shape analysis comparable with specialized shape analysis. The use of Datalog techniques and algorithms in program analysis has also been explored in [18, 19], though not in the context of parametric shape analysis. Database optimizations were also used to develop a tool for context sensitive program analysis [19]. Indeed, many applications of database optimizations in program verification are yet to be explored.

In the future, we plan to combine our optimizations with other techniques in order to handle more realistic programs such as the Apache web server. For example, coarser abstractions such as the ones suggested in [20, 21] are needed to reduce the state space.

References

1. Cousot, P., Cousot, R.: Systematic design of program analysis frameworks. In: POPL. (1979)
2. Sagiv, M., Reps, T., Wilhelm, R.: Solving shape-analysis problems in languages with destructive updating. TOPLAS **20**(1) (1998)

3. Sagiv, M., Reps, T., Wilhelm, R.: Parametric shape analysis via 3-valued logic. TOPLAS (2002)
4. Lev-Ami, T., Sagiv, M.: TVLA: A system for implementing static analyses. In: SAS. (2000)
5. Cousot, P.: Abstract interpretation based static analysis parameterized by semantics. In: SAS. (1997)
6. Yahav, E.: Verifying safety properties of concurrent Java programs using 3-valued logic. In: POPL. (2001)
7. Yahav, E., Sagiv, M.: Automatically verifying concurrent queue algorithms. In: Electronic Notes in Theoretical Computer Science. Volume 89., Elsevier (2003)
8. Reps, T., Sagiv, M., Wilhelm, R.: Static program analysis via 3-valued logic. In: CAV. (2004)
9. Ullman, J.: Database and Knowledge Base Systems, vol. I. W.H. Freeman and Co. (1988)
10. Rinetzky, N., Sagiv, M., Yahav, E.: Interprocedural shape analysis for cutpoint-free programs. In: SAS. (2005)
11. Wong, E., Youssefi, K.: Decompositiona strategy for query processing. ACM Transactions on Database Systems (TODS) **1**(3) (1976) 223–242
12. Lev-Ami, T., Immerman, N., Sagiv, M.: Abstraction for shape analysis with fast and precise transformers. In: CAV. (2006)
13. Gotsman, A., Berdine, J., Cook, B.: Interprocedural shape analysis with separated heap abstractions. In: SAS. (2006)
14. Reineke, J.: Shape analysis of sets. Master's thesis, Saarland University (2005)
15. Lindstrom, G.: Scanning list structures without stacks or tag bits. IPL **2**(2) (1973)
16. Loginov, A., Reps, T., Sagiv, M.: Automatic verification of the Deutsch-Schorr-Waite tree-traversal algorithm. In: SAS. (2006)
17. Parduhn, S.: Algorithm animation using shape analysis with special regard to binary trees. Master's thesis, Saarland University (2005)
18. Whaley, J., Avots, D., Carbin, M., Lam, M.S.: Using datalog and binary decision diagrams for program analysis. In Yi, K., ed.: Proceedings of the 3rd Asian Symposium on Programming Languages and Systems. Volume 3780 of Lec. Notes in Comp. Sci., Springer-Verlag (2005)
19. Lam, M., Whaley, J., Livshits, V., Martin, M., Avots, D., Carbin, M., Unkel, C.: Context-sensitive program analysis as database queries. In: PODS. (2005)
20. Arnold, G.: Specialized 3-valued logic shape analysis using structure-based refinement and loose embedding. In: SAS. (2006)
21. Manevich, R., Berdine, J., Cook, B., Ramalingam, G., Sagiv, M.: Shape analysis by graph decomposition. In: TACAS. (2007)