On Abstraction Refinement for Program Analyses in Datalog

Xin Zhang[†] Ravi Mangal[†] Radu Grigore[*] Mayur Naik[†] Hongseok Yang[*]

[†] Georgia Institute of Technology [*] Oxford University

Abstract

A central task for a program analysis concerns how to efficiently find a program abstraction that keeps only information relevant for proving properties of interest. We present a new approach for finding such abstractions for program analyses written in Datalog. Our approach is based on counterexample-guided abstraction refinement: when a Datalog analysis run fails using an abstraction, it seeks to generalize the cause of the failure to other abstractions, and pick a new abstraction that avoids a similar failure. Our solution uses a boolean satisfiability formulation that is general, complete, and optimal: it is independent of the Datalog solver, it generalizes the failure of an abstraction to as many other abstractions as possible, and it identifies the cheapest refined abstraction to try next. We show the performance of our approach on a pointer analysis and a typestate analysis, on eight real-world Java benchmark programs.

1. Introduction

Building a successful program analysis requires solving high-level conceptual issues, such as finding an abstraction of programs that keeps just enough information for a given verification problem, as well as handling low-level implementation issues, such as coming up with efficient data structures and algorithms for the analysis.

One popular approach for addressing this problem is to use Datalog [5, 15, 23, 24]. In this approach, a program analysis only specifies how to generate Datalog constraints from program text. The task of solving the generated constraints is then delegated to an off-the-shelf Datalog constraint solver, such as that underlying BDDBDDB [25], Doop [22], Jedd [16], and Z3's fixpoint engine [12], which in turn relies on efficient symbolic algorithms and data structures, such as Binary Decision Diagrams (BDDs).

The benefits of using Datalog for program analysis, however, are currently limited to the automation of low-level implementation issues. In particular, finding an effective program abstraction is done entirely manually by analysis designers, which results in undesirable consequences such as ineffective analyses hindered by inflexible abstractions or undue tuning burden for analysis designers.

In this paper, we present a new approach for lifting this limitation by automatically finding effective abstractions for program analyses written in Datalog. Our approach is based on counterexample-guided abstraction refinement (CEGAR), which was developed in the model-checking community and has been applied effectively for software verification with predicate abstraction [1, 6, 8, 11, 20]. A counterexample in Datalog is a derivation of an output tuple from a set of input tuples via Horn-clause inference rules: the rules specify the program analysis, the set of input tuples represents the current program abstraction, and the output tuple represents a (typically undesirable) program property derived by the analysis under that abstraction. The counterexample is spurious if there exists some abstraction under which the property *cannot* be derived by the analysis. The CEGAR problem in our approach is to find such an abstraction from a given family of abstractions.

We propose solving this problem by formulating it as a boolean satisfiability (SAT) problem. We give an efficient construction of SAT constraints from a Datalog solver's solution in each CEGAR iteration. Our main theoretical result is that, regardless of the Datalog solver used, its solution contains information to reason about *all* counterexamples. This result seems unintuitive because a Datalog solver performs a least fixed-point computation that can stop as soon as each output tuple that is derivable has been derived (i.e., the solver need not reason about *all* possible ways to derive a tuple).

The above result ensures that solving our SAT constraints generalizes the cause of verification failure in the current CEGAR iteration to the maximum extent possible, eliminating not only the current abstraction but all other abstractions destined to suffer a similar failure. There is still the problem of deciding which abstraction to try next. We show that an optimization extension of the SAT problem, called the maximum satisfiability (MAXSAT) problem, enables to identify the *cheapest* refined abstraction. Our approach avoids unnecessary refinement by using this abstraction in the next CEGAR iteration.

We have implemented our approach and applied it to two realistic static analyses written in Datalog, a pointer analysis and a typestate analysis, for Java programs. These two analyses differ significantly in aspects such as flow sensitivity (insensitive vs. sensitive), context sensitivity (cloning-based vs. summary-based), and heap abstraction (weak vs. strong updates), which demonstrates the generality of our approach. On a suite of eight real-world Java benchmark programs, our approach searches a large space of abstractions, ranging from 2^{1k} to 2^{5k} for the pointer analysis and 2^{13k} to 2^{54k} for the typestate analysis, for hundreds of analysis queries considered simultaneously in each program, thereby showing the practicality of our approach.

We summarize the main contributions of our work:

1. We propose a CEGAR-based approach to automatically find effective abstractions for analyses in Datalog. The approach enables Datalog analysis designers to specify high-level knowledge about abstractions while continuing to leverage low-level implementation advances in off-the-shelf Datalog solvers.
2. We solve the CEGAR problem using a boolean satisfiability formulation that has desirable properties of generality, completeness, and optimality: it is independent of the Datalog solver, it fully generalizes the failure of an abstraction, and it computes the cheapest refined abstraction.
3. We show the effectiveness of our approach on two realistic analyses written in Datalog. On a suite of real-world Java benchmark programs, the approach explores a large space of abstractions for a large number of analysis queries simultaneously.

2. Overview

We illustrate our approach using a graph reachability problem that captures the core concept underlying a precise pointer analysis.

The example program in Figure 1 allocates an object in each of methods `f` and `g`, and passes it to methods `id1` and `id2`. The pointer analysis is asked to prove two queries: query `q1` stating that `v6` is not aliased with `v1` at the end of `g`, and query `q2` stating that `v3` is not aliased with `v1` at the end of `f`. Proving `q1` requires a *context sensitive* analysis that distinguishes between different calling contexts of methods `id1` and `id2`. Query `q2`, on the other hand, cannot be proven since `v3` is in fact aliased with `v1`.

```
f() { v1 = new ...;          g() { v4 = new ...;
      v2 = id1(v1);                v5 = id1(v4);
      v3 = id2(v2);                v6 = id2(v5);
  q2: assert(v3 != v1);        q1: assert(v6 != v1);
}                              }
id1(v) { return v; }         id2(v) { return v; }
```

Figure 1: Example program.

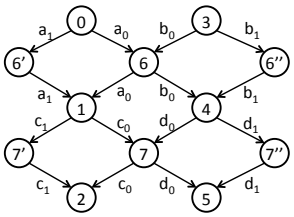
Input relations:

$\texttt{edge}(i, j, n)$ (edge from node i to node j labeled n)
$\texttt{abs}(n)$ (edge labeled n is allowed)

Derived relations:

$\texttt{path}(i, j)$ (node j is reachable from node i)

Rules: (1): $\texttt{path}(i, i)$.
(2): $\texttt{path}(i, j)$:- $\texttt{path}(i, k)$, $\texttt{edge}(k, j, n)$, $\texttt{abs}(n)$.



Input tuples:	Derived tuples:
$\texttt{edge}(0, 6, a_0)$	$\texttt{path}(0, 0)$
$\texttt{edge}(6, 1, a_0)$	$\texttt{path}(0, 6)$
$\texttt{edge}(1, 7, c_0)$	$\texttt{path}(0, 1)$
...	...
$\texttt{abs}(a_0)$	**Query tuples:**
$\texttt{abs}(c_0)$	$\texttt{path}(0, 5)$
...	$\texttt{path}(0, 2)$

Figure 2: Graph reachability example in Datalog.

A common approach to distinguish between different calling contexts is to clone (i.e., inline) the body of the called method at a call site. However, cloning each called method at each call site is infeasible even in the absence of recursion, as it grows program size exponentially and hampers the scalability of the analysis. We seek to address this problem by cloning selectively.

For exposition, we recast this problem as a reachability problem on the graph in Figure 2. In that graph, nodes 0, 1, and 2 represent basic blocks of `f`, while nodes 3, 4, and 5 represent basic blocks of `g`. Nodes 6 and 7 represent the bodies of `id1` and `id2` respectively, while nodes $6'$, $6''$, $7'$ and $7''$ are their clones at different call sites. Edges denoting matching calls and returns have the same label. A choice of labels constitutes a valid abstraction of the original program if, for each of a, b, c, and d, either the zero (non-cloned) or the one (cloned) version is chosen. Then, proving query `q1` corresponds to showing that node 5 is unreachable from node 0 under some valid choice of labeled edges, which is the case if edges labeled $\{a_1, b_0, c_1, d_0\}$ are chosen; proving query `q2` corresponds to finding a valid combination of edge labels that makes node 2 unreachable from node 0, but this is impossible.

Our graph reachability problem can be expressed in Datalog as shown in Figure 2. A Datalog program consists of a set of input relations, a set of derived relations, and a set of rules that express how to compute the derived relations from the input relations. There are two input relations in our example: `edge`, representing the possible labeled edges in the given graph; and `abs`, containing labels of edges that may be used in computing graph reachability. Relation `abs` specifies a program abstraction in our original setting; for instance, $\texttt{abs} = \{a_1, b_0, c_1, d_0\}$ specifies the abstraction in which only the calls to methods `id1` and `id2` from `f` are inlined.

The derived relation `path` contains each tuple (i, j) such that node j is reachable from node i along a path with only edges whose labels appear in relation `abs`. This computation is expressed by rules (1) and (2) both of which are Horn clauses with implicit universal quantification. Rule (1) states that each node is reachable from itself. Rule (2) states that if node k is reachable from node i and edge (k, j) is allowed, then node j is reachable from node i. Queries in the original program correspond to tuples in relation `path`. Proving a query amounts to finding a valid instance of relation `abs` such that the tuple corresponding to the query is *not* derived.

run	used abstraction	eliminated abstractions: within run, q_1	eliminated abstractions: within run, q_2	eliminated abstractions: across runs, q_2
1	$a_0\, b_0\, c_0\, d_0$	$a_0\, b_0\, *\, d_0$ $a_0\, *\, c_0\, d_0$	$a_0\, *\, c_0\, *$	
2	$a_1\, b_0\, c_0\, d_0$	$a_1\, *\, c_0\, d_0$	$a_1\, *\, c_0\, *$	
3	$a_1\, b_0\, c_1\, d_0$		$a_1\, *\, c_1\, *$	$a_0\, *\, c_1\, *$

Table 1: Each iteration (run) eliminates a number of abstractions. Some are eliminated by analyzing the current Datalog run (within run); some are eliminated because of the derivations from the current run interact with derivations from previous runs (across runs).

Hard constraints:	**Soft constraints:**
$path(0, 0)\ \wedge$	$(abs(a_0)\ \texttt{weight}\ 1)\ \wedge$
$(path(0, 6) \vee \neg path(0, 0) \vee \neg abs(a_0))\ \wedge$	$(abs(b_0)\ \texttt{weight}\ 1)\ \wedge$
$(path(0, 1) \vee \neg path(0, 6) \vee \neg abs(a_0))\ \wedge$	$(abs(c_0)\ \texttt{weight}\ 1)\ \wedge$
$(path(0, 7) \vee \neg path(0, 1) \vee \neg abs(c_0))\ \wedge$	$(abs(d_0)\ \texttt{weight}\ 1)\ \wedge$
$(path(0, 4) \vee \neg path(0, 6) \vee \neg abs(b_0))\ \wedge$	$(\neg path(0, 5)\ \texttt{weight}\ 5)\ \wedge$
$\cdots$	$(\neg path(0, 2)\ \texttt{weight}\ 5)$

Figure 4: Formula from the Datalog run's result in the first iteration.

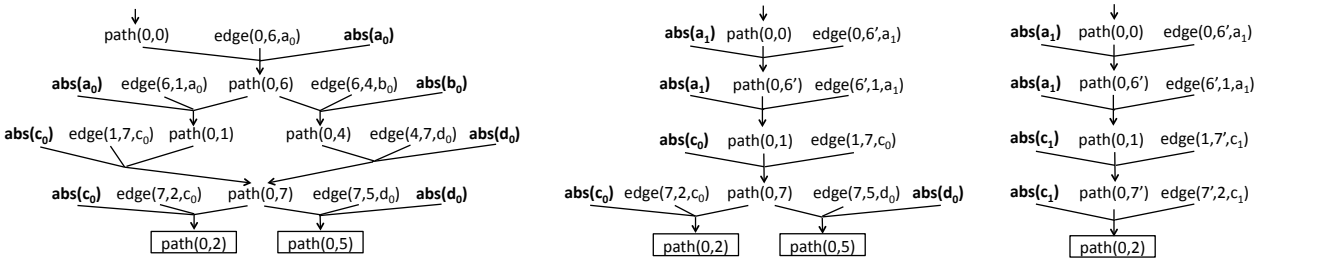
Our two queries `q1` and `q2` correspond to tuples $\texttt{path}(0, 5)$ and $\texttt{path}(0, 2)$ respectively. There are in all 16 abstractions, each involving a different choice of the zero/one versions of labels a through d in relation `abs`. Since we wish to minimize the amount of cloning in the original setting, abstractions with more zero versions of edge labels are cheaper. Our approach, outlined next, efficiently finds the cheapest abstraction $\texttt{abs} = \{a_1, b_0, c_1, d_0\}$ that proves `q1`, and shows `q2` cannot be proven by any of the 16 abstractions.

Our approach is based on iterative counterexample-guided abstraction refinement. Table 1 illustrates its iterations on the graph reachability example. In the first iteration, the cheapest abstraction $\texttt{abs} = \{a_0, b_0, c_0, d_0\}$ is tried. It corresponds to the case where neither of nodes 6 and 7 is cloned (i.e., a fully context-insensitive analysis). This abstraction fails to prove both of our queries. Figure 3(a) shows all the possible derivations of the two queries using this abstraction. Each set of edges in this graph, incoming into a node, represents an application of a Datalog rule, with the source nodes denoting the tuples in the body of the rule and the target node denoting the tuple at its head.

The first question we ask is: how do we generalize the failure of the current abstraction to avoid picking another that will suffer a similar failure? Our solution is to exploit a monotonicity property of Datalog: more input tuples can only derive more output tuples. It follows from this property that the maximum generalization of the failure can be achieved if we find *all minimal subsets* of the set of tuples in the current abstraction that suffice to derive queries. From the derivation in Figure 3(a), we see that these minimal subsets are $\{a_0, b_0, d_0\}$ and $\{a_0, c_0, d_0\}$ for query $\texttt{path}(0, 5)$, and $\{a_0, c_0\}$ for query $\texttt{path}(0, 2)$. We thus generalize the current failure to the maximum possible extent, eliminating any `abs` that is a superset of $\{a_0, b_0, d_0\}$ or $\{a_0, c_0, d_0\}$ for query $\texttt{path}(0, 5)$ and any `abs` that is a superset of $\{a_0, c_0\}$ for query $\texttt{path}(0, 2)$.

The next question we ask is: how do we pick the abstraction to try next? Our solution is to use the cheapest abstraction of the ones not eliminated so far for both the queries. From the fact that label a_0 is in both minimal subsets identified above, and that zero labels are cheaper than the one labels, and we conclude that this abstraction is $\texttt{abs} = \{a_1, b_0, c_0, d_0\}$, which corresponds to only cloning method `id1` at the call in `f`.

In the second iteration, our approach uses this abstraction, and again fails to prove both queries. But this time the derivation, shown in Figure 3(b), is different from that in the first iteration. This time, we eliminate any `abs` that is a superset of $\{a_1, c_0, d_0\}$ for query $\texttt{path}(0, 5)$, and any `abs` that is a superset of $\{a_1, c_0\}$ for query $\texttt{path}(0, 2)$. The cheapest of the remaining abstractions, not



(a) Using $\texttt{abs} = \{a_0, b_0, c_0, d_0\}$. (b) Using $\texttt{abs} = \{a_1, b_0, c_0, d_0\}$. (c) Using $\texttt{abs} = \{a_1, b_0, c_1, d_0\}$.

Figure 3: Derivations after different iterations of our approach on our graph reachability example.

eliminated for both the queries, is $\texttt{abs} = \{a_1, b_0, c_1, d_0\}$, which corresponds to cloning methods `id1` and `id2` in `f` (but not in `g`).

Using this abstraction in the third iteration, our approach succeeds in proving query $\texttt{path}(0, 5)$, but still fails to prove query $\texttt{path}(0, 2)$. As seen in the derivation in Figure 3(c), this time $\{a_1, c_1\}$ is the minimal failure subset for query $\texttt{path}(0, 2)$ and we eliminate any `abs` that is its superset.

At this point, four abstractions remain in trying to prove query $\texttt{path}(0, 2)$. However, another novel feature of our approach allows us to eliminate these remaining abstractions without any more iterations. After each iteration, we accumulate the derivations generated by the current run of the Datalog program with the derivations from all the previous iterations. Then, in our current example, at the end of the third iteration, we have the following derivations available:

$$\begin{array}{l}
\texttt{path}(0,6)\ \texttt{:-}\ \texttt{path}(0,0),\ \texttt{edge}(0,6,a_0),\ \texttt{abs}(a_0)\\
\texttt{path}(0,1)\ \texttt{:-}\ \texttt{path}(0,6),\ \texttt{edge}(6,1,a_0),\ \texttt{abs}(a_0)\\
\texttt{path}(0,7')\texttt{:-}\ \texttt{path}(0,1),\ \texttt{edge}(1,7',c_1),\texttt{abs}(c_1)\\
\texttt{path}(0,2)\ \texttt{:-}\ \texttt{path}(0,7'),\texttt{edge}(7',2,c_1),\texttt{abs}(c_1)
\end{array}$$

The derivation of $\texttt{path}(0, 1)$ using $\texttt{abs}(a_0)$ comes from the first iteration of our approach. Similarly, the derivation of path(0,2) using path(0,1) and $\texttt{abs}(c_1)$ is seen during the third iteration. However, accumulating the derivations from different iterations allows our approach to explore the above derivation of $\texttt{path}(0, 2)$ using $\texttt{abs}(a_0)$ and $\texttt{abs}(c_1)$ and detect $\{a_0, c_1\}$ to be an additional minimal failure subset for query $\texttt{path}(0, 2)$. Consequently, we remove any `abs` that is a superset of $\{a_0, c_1\}$. This eliminates all the remaining abstractions for query $\texttt{path}(0, 2)$ and our approach concludes that the query cannot be proven by any of the 16 abstractions.

We summarize the three main strengths of our approach over previous CEGAR approaches. (1) Given a single failing run of a Datalog program on a single query, it reasons about all counterexamples that cause the proof of the query to fail and generalizes the failure to the maximum extent possible. (2) It reasons about failures and discovers new counterexamples across iterations by mixing the already available derivations from different iterations. (3) It generalizes the causes of failure for multiple queries simultaneously. Together, these three features enable faster convergence of our approach.

Encoding as MAXSAT**.** In the graph reachability example, our approach searched a space of 16 different abstractions for each of two queries. In practice, we seek to apply our approach to real-world programs and analyses, where the space of abstractions is 2^x for x of the order of tens of thousands, for each of hundreds of queries. To handle such large spaces efficiently, our approach formulates a boolean satisfiability problem from the output of the Datalog computation in each iteration, and solves it using an off-the-shelf solver to obtain the abstraction to try in the next iteration. For our example, Figure 4 shows the boolean formula it constructs at the end of the first iteration. This formula contains a boolean variable for each tuple in the Datalog computation, denoted by the tuple itself. It has two kinds of clauses: hard clauses, which *must* be satisfied, and soft clauses, each of which is associated with a weight and may be left unsatisfied. We seek to find a solution that maximizes the sum of the weights of satisfied soft clauses. This problem is an optimization extension of the boolean satisfiability (SAT) problem, called the (weighted) maximum satisfiability (MAXSAT) problem.

Briefly, the formula has three parts. The first part is hard constraints that encode the derivation using one Horn clause per derivation constraint. Input tuples besides those in the abstraction relation are replaced by true since they are fixed. For instance, the following derivation in Figure 3(a):

$$\texttt{path}(0,6)\ \texttt{:-}\ \texttt{path}(0,0),\ \texttt{edge}(0,6,a),\ \texttt{abs}(a_0)$$

yields the clause $(path(0,6) \vee \neg path(0,0) \vee \neg abs(a_0))$.

The second part is soft constraints that guide the MAXSAT solver to pick the cheapest abstraction among those that satisfy the hard constraints. In our example, a weight of 1 is accrued when the solver picks an abstraction that contains zero label and, implicitly, a weight of 0 when it contains a one label.

The third part of the formula is soft constraints that negate each unproven query so far. The reason is that, to prove a query, we must find an abstraction that avoids deriving the query. One may wonder why we make these soft instead of hard constraints. The reason is that certain queries (e.g., $\texttt{path}(0, 2)$) cannot be proven by any abstraction; these would make the entire formula unsatisfiable and prevent other queries (e.g., $\texttt{path}(0, 5)$) from being proven. But we must be careful in the weights we attach: these weights can affect the convergence characteristics of our approach. Returning to our example, making such a soft clause unsatisfiable incurs a weight of 5, which implies that no abstraction – not just the cheapest – can prove that query, the reason being that even the most expensive abstraction incurs a weight of 4.

The formula constructed in each subsequent iteration conjoins those from all previous iterations with the formula encoding the derivation of the current iteration.

3. Parametric Dataflow Analyses

Datalog is a logic programming language capable of naturally expressing many static analyses [22, 25]. In this section, we review this use of Datalog, especially for developing ***parametric*** static analyses. Such an analysis takes (an encoding of) the program to analyze, a set of queries, and a setting of parameters that dictate the degree of program abstraction. The analysis then outputs queries that it could successfully verify using the chosen abstraction. Our goal is to develop an efficient algorithm for automatically adjusting the setting of these abstraction parameters for a given program and set of queries. We formally state this problem in this section and present our CEGAR-based solution in the subsequent section.

3.1 Datalog Syntax and Semantics

We start with a review of Datalog's syntax and semantics. Figure 5 shows the syntax of Datalog. A Datalog program is a list of constraints. (We use overbar, such as $\bar{c}$, to denote zero, one, or

$$\begin{array}{rrcl@{\qquad}rrcl} \textit{(program)} & C & ::= & \bar{c} & \textit{(constraint)} & c & ::= & l \text{ :- } \bar{l} \\ \textit{(literal)} & l & ::= & r(\bar{a}) & \textit{(argument)} & a & ::= & v \mid d \end{array}$$

Figure 5: Syntax of Datalog used for analyses.

$$\begin{array}{rl@{\qquad}rl} \textit{(relations)} & r \in \mathbf{R} = \{\mathtt{a}, \mathtt{b}, ..\} & \textit{(variables)} & v \in \mathbf{V} = \{x, y, ..\} \\ \textit{(constants)} & d \in \mathbf{D} = \{0, 1, ..\} & \textit{(tuples)} & t \in \mathbf{T} = \mathbf{R} \times \mathbf{D}^* \\ \textit{(queries)} & q \in \mathbf{Q} \subseteq \mathbf{T} & \textit{(abstractions)} & A \in \mathbf{A} \subseteq \mathcal{P}(\mathbf{T}) \\ \textit{(substitutions)} & \sigma \in \mathbf{\Sigma} = \mathbf{V} \to \mathbf{D} & & \end{array}$$

Figure 6: Auxiliary definitions and notations.

$$\begin{array}{l} [\![C]\!] \in \mathcal{P}(\mathbf{T}) \qquad F_C, f_c \in \mathcal{P}(\mathbf{T}) \to \mathcal{P}(\mathbf{T}) \\ [\![C]\!] = \mathsf{lfp}\, F_C \qquad F_C(T) = T \cup \bigcup\{\, f_c(T) \mid c \in C \,\} \\ \qquad\qquad f_{l_0 \text{ :- } l_1,\ldots,l_n}(T) = \{\, \sigma(l_0) \mid \sigma(l_k) \in T \text{ for } 1 \le k \le n \,\} \end{array}$$

Figure 7: Semantics of Datalog.

more occurrences separated by a comma.) Constraints have a head consisting of one literal, and a body consisting of a list of literals. Each literal is a relation name together with several arguments, each of which is either a variable or a constant. We assume that Datalog programs are finite. We define ***tuples*** as literals not containing variables; they are also called ***ground literals***. Finally, we consider only well-formed programs, for which all the variables occurring in the head of a constraint also appear in the body of that constraint.

Each Datalog program C denotes a set of tuples derived using its constraints. The details are given in Figure 7. In this semantics of Datalog, each constraint $l_0 \text{ :- } l_1, \ldots, l_n$ is interpreted as a rule for deriving a tuple from known tuples — it says that if there exists a substitution σ such that $\sigma(l_1), \sigma(l_2), \ldots, \sigma(l_n)$ are all known tuples, $\sigma(l_0)$ is derived. Note that if the constraint body is empty, $n = 0$, then l_0 is a tuple that the analysis derives. The program denotes the least fixed-point (lfp) of repeated applications of the constraints. The following standard result will be used tacitly in later arguments.

Proposition 1 (Monotonicity). If $C_1 \subseteq C_2$, then $[\![C_1]\!] \subseteq [\![C_2]\!]$.

3.2 Abstractions and Queries

So far we discussed Datalog in general. We now turn to Datalog programs that implement parametric static analyses. Such Datalog programs contain three types of constraints: (1) those that encode the abstract semantics of the programming language, (2) those that encode the program being analyzed, and (3) those that determine the degree of program abstraction used by the abstract semantics.

Example 2. Our graph reachability example (Figure 2) is modeled after a parametric pointer analysis, and is specified using these three types of constraints. The two rules in the figure describe inference steps for deriving tuples of the `path` relation. These rules apply for all graphs, and they are constraints of type (1). Input tuples of the `edge` relation encode information about a specific graph, and are of type (2). The remaining input tuples of the `abs` relation specify the amount of cloning and control the degree of abstraction used in the analysis; they are thus constraints of type (3). ∎

Hereafter, we refer to the set A of constraints of type (3) as the ***abstraction***, and the set C of constraints of types (1) and (2) as the ***analysis***. This further grouping reflects the different treatment of constraints by our refinement approach — the constraints in the abstraction change during iterative refinement, whereas the constraints in the analysis do not. Given an analysis, only abstractions from a certain family $\mathbf{A}$ make sense. We say that A is a ***valid abstraction*** when $A \in \mathbf{A}$. The result for evaluating the analysis C with such a valid abstraction A is the set $[\![C \cup A]\!]$ of tuples.

A ***query*** $q \in \mathbf{Q}$ is just a particular kind of tuple that describes a bug or an undesirable program property. We assume that a set $Q \subseteq \mathbf{Q}$ of queries is given in the specification of a verification problem. The goal of a parametric static analysis is to show, as much as possible, that the bugs or properties described by Q do not arise during the execution of a given program. We say of a valid abstraction A that it ***rules out*** a query q if and only if $q \notin [\![C \cup A]\!]$. Note that an abstraction either derives a query, or rules it out. Different abstractions rule out different queries, and we will often refer to the set of queries ruled out by several abstractions taken together. We denote by $\mathcal{R}(\mathcal{A}, Q)$ the set of queries out of Q that are ruled out by some abstraction $A \in \mathcal{A}$:

$$\mathcal{R}(\mathcal{A}, Q) \;=\; Q \setminus \bigcap\{\, [\![C \cup A]\!] \mid A \in \mathcal{A} \,\}$$

Conversely, we say that an abstraction A is ***unviable*** with respect to a set Q of queries if and only if A does not rule out any query in Q; that is, $Q \subseteq [\![C \cup A]\!]$.

Example 3. In our graph reachability example, the family $\mathbf{A}$ of valid abstractions consists of sets $\{\mathtt{abs}(a_i), \mathtt{abs}(b_j), \mathtt{abs}(c_k), \mathtt{abs}(d_l)\}$ for all i, j, k, l. They describe 16 options of cloning nodes in the graph. The set of queries is $Q = \{\mathtt{path}(0, 5), \mathtt{path}(0, 2)\}$. ∎

We assume that the family $\mathbf{A}$ of valid abstractions is equipped with the precision preorder $\sqsubseteq$ and the efficiency preorder $\preceq$. Intuitively, $A_1 \sqsubseteq A_2$ holds when A_1 is at most as precise as A_2, and so it rules out less queries. Formally, we require that the precision preorder obeys the following condition:

$$A_1 \sqsubseteq A_2 \;\Rightarrow\; [\![C \cup A_1]\!] \cap \mathbf{Q} \supseteq [\![C \cup A_2]\!] \cap \mathbf{Q}$$

Some analyses have a most precise abstraction $A_\top$, which can rule out most bugs. This abstraction, however, is often impractical, in the sense that computing $[\![C \cup A_\top]\!]$ requires too much time or space. The efficiency preorder captures the notion of abstraction efficiency: $A_1 \preceq A_2$ denotes that A_1 is at most as efficient as A_2. Often, the two preorders point in opposite directions.

Example 4. Abstractions of our running example are ordered as follows. Let $A = \{\mathtt{abs}(a_i), \mathtt{abs}(b_j), \mathtt{abs}(c_k), \mathtt{abs}(d_l)\}$ and $B = \{\mathtt{abs}(a_{i'}), \mathtt{abs}(b_{j'}), \mathtt{abs}(c_{k'}), \mathtt{abs}(d_{l'})\}$. Then, $A \sqsubseteq B$ if and only if $i \le i' \wedge j \le j' \wedge k \le k' \wedge l \le l'$. Also, $A \preceq B$ if and only if $(i + j + k + l) \ge (i' + j' + k' + l')$. These relationships formally express that cloning more nodes can improve the precision of the analysis but at the same time it can slow down the analysis. ∎

3.3 Problem Statement

Our aim is to solve the following problem:

Definition 5 (Datalog Analysis Problem). Suppose we are given an analysis C, a set $Q \subseteq \mathbf{Q}$ of queries, and an abstraction family $(\mathbf{A}, \sqsubseteq, \preceq)$. Compute the set $\mathcal{R}(\mathbf{A}, Q)$ of queries that can be ruled out by some valid abstraction.

Since $\mathbf{A}$ is typically finite, a brute force solution is possible: simply apply the definition of $\mathcal{R}(\mathbf{A}, Q)$. However, $|\mathbf{A}|$ is often exponential in the size of the analyzed program. Thus, it is highly desirable to exploit the structure of the problem to obtain a better solution. In particular, the information embodied by the efficiency preorder $\preceq$ and by the precision preorder $\sqsubseteq$ should be exploited.

Our general approach, in the vein of CEGAR, is to run Datalog, in turn, on a finite sequence $A_1, \ldots, A_n$ of abstractions. In the ideal scenario, every query $q \in Q$ is ruled out by some abstraction in the sequence, and the combined cost of running the analysis for all the abstractions in the sequence is as small as possible. The efficiency preorder $\preceq$ provides a way to estimate the cost of running an analysis without actually doing so; the precision preorder $\sqsubseteq$ could be used to restrict the search for abstractions. We describe the approach in detail in the next section.

Example 6. What we have described for our running example provides the instance $(C, Q, (\mathbf{A}, \sqsubseteq, \preceq))$ of the Datalog Analysis problem. Recall that $Q = \{\mathtt{path}(0, 2), \mathtt{path}(0, 5)\}$. As we explained in Section 2, among these two queries, only $\mathtt{path}(0, 5)$ can be ruled out

by some abstraction. A cheapest such abstraction according to the efficiency order $\preceq$ is $A = \{\texttt{abs}(a_1), \texttt{abs}(b_0), \texttt{abs}(c_1), \texttt{abs}(d_0)\}$, which clones two nodes, while the most expensive one is $B = \{\texttt{abs}(a_1), \texttt{abs}(b_1), \texttt{abs}(c_1), \texttt{abs}(d_1)\}$ with four clones. Hence, the answer $\mathcal{R}(\mathbf{A}, Q)$ for this problem is $\{\texttt{path}(0, 5)\}$, and our goal is to arrive at this answer in a small number of refinement iterations, while mostly trying a cheap abstraction in each iteration, such as the abstraction A rather than B. ∎

4. Algorithm

In this section, we present our CEGAR algorithm for parametric analyses expressed in Datalog. Our algorithm frees the designer of the analysis from the task of describing how to do refinement. All they must do is to describe which abstractions are valid. We achieve such a high degree of automation while remaining efficient due to two main ideas. The first is to record the result of a Datalog run using a boolean formula that compactly represents large sets of unviable abstractions. The second is to reduce the problem of finding a good abstraction to a MAXSAT problem.

We begin by describing the first idea (Section 4.1): how Datalog runs are encoded in boolean formulas, and what properties this encoding has. In particular, we observe that conjoining the encoding of multiple Datalog runs gives an under-approximation for the set of unviable abstractions (Theorem 9). This observation motivates the overall structure of our CEGAR-based solution to the Datalog analysis problem, which we describe next (Section 4.2). The algorithm relies on a subroutine for choosing the next abstraction. While arguing for the correctness of the algorithm, we formalize the requirements for this subroutine: it should choose a cheap abstraction not yet known to be unviable. We finish by describing the second idea (Section 4.3), how choosing a good abstraction is essentially a MAXSAT problem, thus completing the description of our solution.

4.1 From Datalog Derivations to SAT Formulae

In the CEGAR algorithm, we iteratively call a Datalog solver. It is desirable to do so as few times as possible, so we wish to eliminate as many unviable abstractions as possible without calling the solver. To do so, we need to rely on more information than the binary answer of a Datalog run, on whether an abstraction derives or rules out a query. Intuitively, there is more information, waiting to be exploited, in *how* a query is derived. Theorem 8 shows that by recording the Datalog run for an abstraction A as a boolean formula it is possible to partly predict what Datalog will do for other abstractions that share tuples with A. Perhaps less intuitively, Theorem 9 shows that it is sound to mix (parts of) derivations seen for different runs of Datalog. Thus, in some situations we can predict that an abstraction A_1 will derive a certain query by combining tuple dependencies observed in runs for two other abstractions A_2 and A_3.

For each tuple $t \in \mathbf{T}$, we introduce a unique boolean variable which we also denote t, and consider boolean formulas ϕ that use such t's as atomic formulas. We assume these formulas have the syntax from Figure 8, and we define their semantics using the satisfaction relation $T \models \phi$ between a set T of tuples and a formula ϕ. It states that ϕ is true under the assignment where each boolean variable in T is true and the remaining ones in $\mathbf{T}$ are false:

Definition 7 (Satisfaction of boolean formulae). The relation $T \models \phi$ of boolean satisfaction is defined inductively, as follows:

$$\begin{array}{ll} T \models t \text{ iff } t \in T & T \models \mathsf{True} \text{ iff always} \\ T \models \neg\phi \text{ iff } T \not\models \phi & T \models \phi_1 \wedge \phi_2 \text{ iff } T \models \phi_1 \text{ and } T \models \phi_2 \end{array}$$

If $T \models \phi$, we say that T is a ***model*** of ϕ. We say that ϕ is ***valid*** and write $\models \phi$ when $T \models \phi$ for all $T \subseteq \mathbf{T}$. The other boolean connectives can be defined in terms of $\neg$ and $\wedge$.

$$\begin{array}{l} \text{(boolean formula) } \phi \in \mathbf{\Phi} \qquad \phi ::= \mathsf{True} \mid t \mid \neg\phi_1 \mid \phi_1 \wedge \phi_2 \\ [C] \in \mathcal{P}(\mathbf{T}) \to \mathbf{\Phi} \\ [C]_T = \bigwedge \{ \sigma(l_1) \wedge \ldots \wedge \sigma(l_n) \to \sigma(l_0) \mid \\ \qquad (l_0 \texttt{ :- } l_1, \ldots, l_n) \in C \text{ and } \sigma(l_0), \ldots, \sigma(l_n) \in T \} \end{array}$$

Figure 8: From Datalog analysis results to boolean formulas.

Figure 8 shows our routine for producing a boolean formula from the result T of analysis C. We denote this formula with $[C]_T$. Intuitively, the formula encodes information about the dependency among tuples in derivations, and its models are those sets of tuples that are consistent with the observed dependencies. The following theorem states that $[C]_T$ allows us to predict what Datalog would do for those abstractions $A \subseteq T$.

Theorem 8. *Let T be a fixed-point of function F_C used to define $[\![C]\!]$ in Figure 7, and let T' be a subset of T; thus $F_C(T) = T$ and $T' \subseteq T$. Then $[\![C \cup T']\!]$ is fully determined by $[C]_T$, as follows:*

$$t \in [\![C \cup T']\!] \quad \Leftrightarrow \quad \models \big((\bigwedge T' \wedge [C]_T) \to t\big)$$

One interesting instantation of the theorem is the case that T is the result of running Datalog with some valid abstraction A, i.e., $T = [\![C \cup A]\!]$. To see the importance of this case, consider the first iteration of the running example (Section 2), where A is $\{\texttt{abs}(a_0), \texttt{abs}(b_0), \texttt{abs}(c_0), \texttt{abs}(d_0)\}$. The theorem says that it is possible to predict exactly what Datalog would do for subsets of A. If such a subset derives a query then, by monotonicity (Proposition 1), so do all its supersets. In other words, we can give lower bounds for which queries do other abstractions derive. (All other abstractions are supersets of subsets of A.)

When the analysis C is run multiple times with different abstractions, the results $T_1, \ldots, T_n$ of these runs lead to boolean formulas $[C]_{T_1}, \ldots, [C]_{T_k}$. The next theorem points out the benefit of considering these formulas together, as illustrated in Section 2. It implies that by conjoining these formulas, we can mix derivations from different runs, and identify more unviable abstractions.

Theorem 9. *Let $T_1, \ldots, T_n$ be fixed-points of F_C. For all T',*

$$\models \big((\bigwedge T' \wedge [C]_{T_1} \wedge \ldots \wedge [C]_{T_n}) \to t\big) \quad \Rightarrow \quad t \in [\![C \cup T']\!]$$

4.2 The Algorithm

Our main algorithm (Algorithm 1) classifies the queries Q into those that are ruled out by some abstraction and those that are impossible to rule out using any abstraction. The algorithm maintains its state in two variables, $R \in \mathcal{P}(\mathbf{Q})$ and $\phi \in \mathbf{\Phi}$, where R is the set of queries that have been ruled out so far and ϕ is a boolean formula that encodes the Datalog runs observed so far.

The call $\mathsf{choose}(\phi, Q')$ evaluates to $\mathsf{impossible}$ only if all queries in Q' are impossible to rule out, according to the information encoded in ϕ. Thus, the algorithm terminates only if all queries that can be ruled out have been ruled out. Conversely, choose never returns an abstraction whose analysis was previously recorded in ϕ. Intuitively, ϕ represents the set of abstractions known to be unviable for the remaining set of queries $Q' = (Q \setminus R)$. Formally, this notion is captured by the concretization function γ, whose definition is justified by Theorem 9.

$$\begin{array}{ll} \gamma & \in \mathbf{\Phi} \times \mathcal{P}(\mathbf{Q}) \to \mathcal{P}(\mathbf{A}) \\ \gamma(\phi, Q') & \triangleq \{\, A \in \mathbf{A} \mid \models \big(\bigwedge A \wedge \phi\big) \to \bigwedge Q' \,\} \end{array}$$

The condition that $\big(\bigwedge A \wedge \phi\big) \to \bigwedge Q'$ is valid appears complicated, but it is just a formal way to say that all queries in Q' are derivable from A using derivations encoded in ϕ. Hence, $\gamma(\phi, Q')$ contains the abstractions known to be unviable with respect to Q'; thus $\mathbf{A} \setminus \gamma(\phi, Q')$ is the set of valid abstractions that, according to

Algorithm 1 CEGAR-based Datalog analysis.

1: **INPUT:** Queries Q
2: **OUTPUT:** A partition (R, I) of Q, where R contains queries that have been ruled out and I queries impossible to rule out.
3: **var** $R := \emptyset$, $\phi :=$ True
4: **loop**
5: $\quad A := \mathsf{choose}(\phi, Q \setminus R)$
6: $\quad$ **if** $(A = \mathsf{impossible})$ **return** $(R, Q \setminus R)$
7: $\quad T := [\![C \cup A]\!]$
8: $\quad R := R \cup (Q \setminus T)$
9: $\quad \phi := \phi \wedge [C]_T$
10: **end loop**

ϕ, might be able to rule out some queries in Q'. The function $\mathsf{choose}(\phi, Q')$ chooses an abstraction from $\mathbf{A} \setminus \gamma(\phi, Q')$, if this set is not empty.

Each iteration of Algorithm 1 begins with a call to choose (Line 5). If all remaining queries $Q \setminus R$ are impossible to rule out, then the algorithm terminates. Otherwise, a Datalog solver is run with the new abstraction A (Line 7). The set R of ruled out queries is updated (Line 8) and the relevant Datalog rule groundings are recorded in ϕ (Line 9).

Theorem 10. *If* $\mathsf{choose}(\phi, Q')$ *evaluates to an element of the set* $\mathbf{A} \setminus \gamma(\phi, Q')$ *whenever such an element exists, and to* impossible *otherwise, then Algorithm 1 is partially correct: it returns* (R, I) *such that* $R = \mathcal{R}(\mathbf{A}, Q)$ *and* $I = Q \setminus R$*. In addition, if* $\mathbf{A}$ *is finite, then Algorithm 1 terminates.*

The next section gives one definition of the function choose that satisfies the requirements of Theorem 10, thus completing the description of a correct algorithm. The definition of choose from Section 4.3 makes use of the efficiency preorder $\preceq$, such that the resulting algorithm is not only correct, but also efficient. Our implementation also makes use of the precision preorder $\sqsubseteq$ to further improve efficiency. The main idea is to constrain the sequence of used abstractions to be ascending with respect to $\sqsubseteq$. For this to be correct, however, the abstraction family must satisfy an additional condition: for any two abstractions A_1 and A_2 there must exist another abstraction A such that $A \sqsupseteq A_1$ and $A \sqsupseteq A_2$. The proofs are available in the long version of the paper.

4.3 Choosing Good Abstractions via MaxSat

The requirement that function $\mathsf{choose}(\phi, Q')$ should satisfy was laid down in the previous section: it should return an element of $\mathbf{A} \setminus \gamma(\phi, Q')$, or say impossible if no such element exists. In this subsection we describe how to choose the cheapest element, according to the preorder $\preceq$. The type of function choose is

$$\mathsf{choose} \;\in\; \mathbf{\Phi} \times \mathcal{P}(\mathbf{Q}) \;\to\; \mathbf{A} \uplus \{\mathsf{impossible}\}$$

The function choose is essentially a reduction to the standard weighted MaxSat problem [18]. This section begins with a brief review of the MaxSat problem and then continues with an explanation for one definition of choose.

The input to a (partial and weighted) MaxSat problem is the hard constraint ψ_0 and pairs (w_k, ψ_k) of weights w_k and soft constraints ψ_k for $1 \le k \le n$. The goal is to find a model of the hard constraint such that the weight sum of the satisfied soft constraints is maximized. Formally, $\textsc{MaxSat}\big(\psi_0, \{(w_1, \psi_1), \ldots, (w_n, \psi_n)\}\big)$

$$\begin{array}{ll} \text{finds} & T \subseteq \mathbf{T} \text{ that} \\ \text{maximizes} & \sum\{\, w_i \mid T \models \psi_i \text{ and } 1 \le i \le n \,\} \\ \text{subject to} & T \models \psi_0 \end{array}$$

If there is no feasible solution then MaxSat returns impossible. In textbook definitions, the constraints are required to be clauses, i.e., disjunctions of atomic formulas. To simplify the presentation, we do not list individual clauses, but use what may appear to be arbitrary boolean formulas. However, these formulas are always in conjunctive normal form, or trivially convertible to that form.

To define choose in terms of MaxSat, we simply have to say which are the hard and soft constraints. Before describing these in general, let us examine an example.

Example 11. Consider an analysis with parameters $p_1, p_2, \ldots, p_n$, each taking a value from $\{1, 2, \ldots, k\}$. The set of valid abstractions is $\mathbf{A} = \{1, 2, \ldots, n\} \to \{1, 2, \ldots, k\}$. The fact that p_i has value j is encoded by a Datalog tuple which in turn is encoded as a boolean variable. Let us write $[p_i = j]$ for the boolean variable that encodes the fact $p_i = j$. Let the queries be $\{q_1, q_2, q_3\}$, and suppose that the accumulated constraints are

$$\phi = ([p_1{=}1] \wedge [p_2{=}1] \to q_1) \wedge ([p_1{=}1] \to q_2) \wedge ([p_3{=}1] \to q_3)$$

We construct the hard constraint such that (1) only valid abstractions are considered, and (2) abstractions known to be unviable with respect to $\{q_1, q_2, q_3\}$ are avoided.

$$\psi_0 \;=\; \delta_{\mathbf{A}} \wedge \phi \wedge (\neg q_1 \vee \neg q_2 \vee \neg q_3)$$

The formula $\delta_{\mathbf{A}}$, which restricts abstractions to be valid, could be defined as $\bigwedge\{\, \delta^i_{\mathbf{A}} \mid 1 \le i \le n \,\}$ where

$$\begin{aligned} \delta^i_{\mathbf{A}} \triangleq & \big(\bigvee\{\, [p_i = j] \mid 1 \le j \le k \,\}\big) \wedge \neg[p_i < 1] \\ & \wedge \big(\bigwedge\{\, \neg([p_i < j] \wedge [p_i = j]) \mid 1 \le j \le k \,\}\big) \\ & \wedge \big(\bigwedge\{\, ([p_i < j] \vee [p_i = j]) \leftrightarrow [p_i < j+1] \mid 1 \le j < k \,\}\big) \end{aligned}$$

The formula $\delta^i_{\mathbf{A}}$ says that p_i takes exactly one value. The definition uses auxiliary boolean variables $[p_i < j]$, so that $\delta^i_{\mathbf{A}}$ grows linearly with k, rather than quadratically as the naive encoding does. One interesting feature of the hard constraint ψ_0 defined above is that a large part of ψ_0 can be rewritten in a Horn conjunctive normal form.

It remains to construct the soft constraints. To a large extent, this is done based on knowledge about the efficiency characteristics of a particular analysis, and thus is left to the designer of the analysis. For example, if the designer knows from experience that the analysis tends to take longer as $\sum p_i$ increases, then they could include $[p_i = j]$ as a soft constraint with weight $k - j$, for each i and j.

The remaining soft constraints that we include are independent of the particular analysis, and they express that abstractions should be preferred when they could potentially help with more queries. In this example, the extra soft constraints are $\neg q_1$, $\neg q_2$, $\neg q_3$, all with some large weight w. Suppose an abstraction A_1 is known to imply q_1, and an abstraction A_2 is known to imply both q_1 and q_2. Then A_1 is preferred over A_2 because of the last three soft constraints. Note that an abstraction is not considered at all only if it is known to imply all three queries, because of the hard constraint. ∎

In general, choose is defined as follows:

$$\mathsf{choose}(\phi, Q) \triangleq \textsc{MaxSat}\big(\delta_{\mathbf{A}} \wedge \phi \wedge \alpha(Q),\ \eta_{\mathbf{A}} \cup \beta_{\mathbf{A}}(Q)\big)$$

The boolean formula $\delta_{\mathbf{A}}$ encodes the set $\mathbf{A}$ of valid abstractions, and the soft constraints $\eta_{\mathbf{A}}$ encode the efficiency preorder $\preceq$. Formally, letting $T_{\mathbf{A}} = \bigcup\{A \mid A \in \mathbf{A}\}$, the set of tuples used in valid abstractions, we require $\delta_{\mathbf{A}}$ and $\eta_{\mathbf{A}}$ to satisfy the conditions

$$\begin{array}{rcl} T \models \delta_{\mathbf{A}} & \Leftrightarrow & T \cap T_{\mathbf{A}} \in \mathbf{A} \\ A \preceq B & \Leftrightarrow & \sum\{\, w_i \mid A \models \eta^{(i)}_{\mathbf{A}} \,\} \le \sum\{\, w_i \mid B \models \eta^{(i)}_{\mathbf{A}} \,\} \end{array}$$

where $\eta_{\mathbf{A}}$ has the form $\big\{(w_1, \eta^{(1)}_{\mathbf{A}}), \ldots, (w_n, \eta^{(n)}_{\mathbf{A}})\big\}$. Finally, the hard constraint $\alpha(Q)$ and the soft constraints $\beta_{\mathbf{A}}(Q)$ are

$$\alpha(Q) \triangleq \bigvee\{\, \neg q \mid q \in Q \,\} \qquad \beta_{\mathbf{A}}(Q) \triangleq \{\, (w_{\mathbf{A}}{+}1, \neg q) \mid q \in Q \,\}$$

where $w_{\mathbf{A}}$ is an upper bound on the weight given by $\eta_{\mathbf{A}}$ to a valid abstraction; for example, the sum $\sum_{i=1}^{n} w_i$ of all weights would do.

Discussion. The function $\mathsf{choose}(\phi, Q)$ reasons about a possibly very large set $\gamma(\phi, Q)$ of abstractions known to be unviable, and it does so by using the compact representation ϕ of previous Datalog runs, together with several helper formulas, such as $\delta_{\mathbf{A}}$ and $\eta_{\mathbf{A}}$. Moreover, the reduction to a MAXSAT problem is natural and involves almost no transformation of the formulas involved. (In practice, any acrobatics related to keeping formulas in conjunctive normal form while reducing a problem to MAXSAT have the tendency to degrade overall performance.)

Finally, it is tempting to remark that the MAXSAT instances we produce are easy because they tend to consist of mostly Horn formulas. However, MAXSAT is known to be hard even in the restricted case of Horn formulas [14]. It is possible to design Datalog analyses for which $\delta_{\mathbf{A}}$ and $\eta_{\mathbf{A}}$ are extremely simple, the MAXSAT instances would be Horn formulas, and nevertheless finding the best next abstraction amounts to solving an NP-hard problem.

5. Empirical Evaluation

In this section, we empirically evaluate our approach on real-world analyses and programs. The experimental setup is described in Section 5.1 and the evaluation results are discussed in Section 5.2.

5.1 Experimental Setup

We evaluate our approach on two static analyses written in Datalog, a pointer analysis and a typestate analysis, for Java programs. We study the results of applying our approach to each of these analyses on eight Java benchmark programs described in Table 2. The programs are from Ashes Suite and DaCapo Suite.

We implemented our approach using MAXSAT and open-source Datalog solvers without any modification. Both our analyses are expressed and solved using `bddbddb` [24], a BDD-based Datalog solver. Throughout our experiments, we use the algorithm of Section 4.2 with the optimizations at the end of that section. We use the MiFuMaX solver [13] to solve MAXSAT formulas generated by this algorithm. All our experiments were done using Oracle HotSpot JVM 1.6 on a Linux machine with 128GB memory and 3.0GHz processors. We next describe our two analyses in more detail.

Pointer Analysis. Our pointer analysis is a flow-insensitive but context- and object-sensitive analysis based on k-object-sensitivity [19]. It computes a call graph simultaneously with points-to results, because the precision of the call graph and points-to results is inter-dependent due to dynamic dispatching in OO languages like Java.

The precision and efficiency of this analysis depends on how many distinctions it makes between different calling contexts of methods in the program. A common approach to distinguish contexts for OO languages is to use the receiver object (i.e., the object on which the method is called) at a method call site. The context of a call is defined to be the allocation site of the receiver object obj, the allocation site of the allocator object obj' of obj, (i.e., the receiver object of the method that made the allocation of obj) and so on.

This analysis associates a string of such allocation sites of length upto k, which we call *allocation string*, with each abstract object and uses these *allocation strings* to distinguish contexts and separate the facts inferred for a method in different contexts. A different k value can be associated with each allocation site and this limits the length of the *allocation string* for all the abstract objects allocated at that site. The *allocation string* associated with an abstract object also aids the analysis precision by allowing the analysis to distinguish between objects allocated at the same site in different calling contexts.

Since the precision and efficiency of the analysis depends heavily on how many distinctions of calling contexts it makes, the abstraction A we use to parametrize this analysis specifies the k value independently for each object allocation site in the program. Any such abstraction yields a sound analysis but different abstractions affect its precision and efficiency. Using notation $A(h)$ to denote the k value of site h, abstractions are ordered as follows with respect to these aspects:

$$\begin{array}{lrcl} \text{(precision)} & A_1 \sqsubseteq A_2 & \Leftrightarrow & \forall h:\ A_1(h) \leq A_2(h) \\ \text{(efficiency)} & A_1 \preceq A_2 & \Leftrightarrow & \Sigma_h A_1(h) \geq \Sigma_h A_2(h) \end{array}$$

which reflects the intuition that making more calling context distinctions makes the analysis more precise but also less efficient.

Typestate analysis. Our typestate analysis is based on that by Fink et al. [9]. It differs in three major ways from the pointer analysis described above. First, it is fully flow-sensitive, whereas the pointer analysis is fully flow-insensitive. Second, it is fully context-sensitive, using procedure summaries instead of cloning, and therefore capable of precise reasoning for programs with arbitrary call chain depth, including recursive ones. It is based on the tabulation algorithm [21] that we expressed in Datalog. Third, it performs both may- and must-alias reasoning; in particular, it can do strong updates, whereas our pointer analysis only does weak updates. These differences between our two analyses highlight the versatility of our approach.

More specifically, the typestate analysis computes at each program point, a set of abstract states of the form (h, t, a) that collectively over-approximate the typestates of all objects at that program point. The meaning of these components of an abstract state is as follows: h is an allocation site in the program, t is the typestate in which a certain object allocated at that site might be in, and a is a finite set of heap access paths with which that object is definitely aliased (called *must set*). The precision and efficiency of this analysis depends heavily on how many access paths it tracks in *must sets*. Hence, the abstraction A we use to parametrize this analysis is a set of access paths that the analysis is allowed to track: any *must set* in any abstract state computed by the analysis must be a subset of the current abstraction A. The specification of this parametrized analysis differs from the original analysis in that the parametrized analysis simply checks before adding an access path p to a *must set* m whether $p \in A$: if not, it does not add p to m; otherwise, it proceeds as before. Note that it is always safe to drop any access path from any *must set* in any abstract state, which ensures that it is sound to run the analysis using any set of access paths as the abstraction. Different abstractions, however, do affect the precision and efficiency of the analysis, and are ordered as follows with respect to these aspects:

$$\begin{array}{lrcl} \text{(precision)} & A_1 \sqsubseteq A_2 & \Leftrightarrow & A_1 \subseteq A_2 \\ \text{(efficiency)} & A_1 \preceq A_2 & \Leftrightarrow & |A_1| \geq |A_2| \end{array}$$

which reflects the intuition that tracking more access paths makes the analysis more precise but also less efficient.

Using the typestate analysis client, we compare our refinement approach to a scalable CEGAR-based approach for finding optimal abstractions proposed by Zhang et al. [26]. A similar comparison is not possible for the pointer analysis client since the work by Zhang et al. cannot handle non-disjunctive analyses. Instead, we compare the precision and scalability of our approach on the pointer analysis client with an optimized Datalog-based implementation of k-object-sensitive pointer analysis that uses $k = 4$ for all allocation sites in the program. Using a higher k value caused this baseline analysis to timeout on our larger benchmarks.

5.2 Evaluation Results

Table 3 summarizes the results of our experiments. It shows the numbers of queries, abstractions, and iterations of our approach (**CURRENT**) and the baseline approaches (**BASELINE**) for each analysis and benchmark.

The "total" column under queries shows the number of queries posed by the analysis on each benchmark. For the pointer analysis, each query corresponds to proving that a certain dynamically

	description	# classes		# methods		bytecode (KB)		KLOC	
		app	total	app	total	app	total	app	total
toba-s	java bytecode to C compiler	25	158	149	745	32	56	6	69
javasrc-p	java source code to HTML translator	49	135	461	789	43	60	13	66
weblech	website download/mirror tool	11	576	78	3,326	6	208	12	194
hedc	web crawler from ETH	44	353	230	2,134	16	140	6	153
antlr	A parser/translator generator	111	350	1,150	2,370	128	186	29	131
luindex	document indexing and search tool	206	619	1,390	3,732	102	235	39	190
lusearch	text indexing and search tool	219	640	1,399	3,923	94	250	40	198
schroeder-m	sampled audio editing tool	109	936	617	6,435	37	352	12	334

Table 2: Benchmark characteristics. All numbers are computed using a 0-CFA call-graph analysis.

	pointer analysis						typestate analysis					
	queries			abstraction size		iterations	queries		abstraction size		iterations	
	total	resolved		final	max.		total	resolved	final	max.		
		CURRENT	BASELINE								CURRENT	BASELINE
toba-s	7	7	0	17	1,782	10	543	543	62	14,781	15	159
javasrc-p	46	46	0	47	1,845	13	159	159	89	13,653	14	92
weblech	5	5	2	14	3,095	10	13	13	33	25,781	14	16
hedc	47	47	6	73	2,948	18	24	24	14	23,622	7	10
antlr	143	143	5	97	2,917	15	77	77	66	24,815	12	45
luindex	138	138	67	116	4,055	26	248	248	79	33,835	16	72
lusearch	322	322	29	146	3,936	17	45	45	74	33,526	13	52
schroeder-m	51	51	25	45	5,826	15	194	194	71	54,741	9	49

Table 3: Results showing statistics of queries, abstractions, and iterations of our approach (**CURRENT**) and the baseline approaches (**BASELINE**).

dispatching call site in the benchmark is monomorphic, i.e., has a single target method. We excluded queries that could be proven by a context-insensitive pointer analysis. For the typestate analysis, each query corresponds to a typestate assertion. We tracked typestate properties for the objects from the same set of classes as used by Fink et al. [9] in their evaluation.

The "resolved" column shows the number of queries proven or shown to be impossible to prove using any abstraction in the search space. For the pointer analysis, impossibility means that a call site cannot be proven monomorphic no matter how high the k values are. For the typestate analysis, impossibility implies that the typestate assertion cannot be proven even by tracking all program variables. In our experiments, we found that our approach successfully resolved all the queries for the pointer analysis, by using a maximum k value of 10 at any allocation site. However, the baseline 4-object-sensitive analysis without refinement could only resolve up to 50% of the queries. Selectively increasing the k value allowed our approach to scale better and try higher k values, leading to greater precision. For the typestate analysis client, both of our approach and the baseline approach resolved all queries.

The "max." column under abstractions shows the *abstraction size*—the quantity x such that the space of abstractions considered by our approach is of size 2^x (i.e., $x = \log_2 |\mathbf{A}|$). For our benchmarks, the abstraction size ranges from 1k to 5k for the pointer analysis, and from 13k to 54k for the typestate analysis. For the pointer analysis, the cheapest abstraction that our approach considers is equivalent to a context-insensitive analysis with all $k = 0$ and the most expensive is equivalent to a 10-object-sensitive analysis. For the typestate analysis, the cheapest abstraction is the empty set of access paths, and the most expensive is the set of all program variables of reference type. Both our analyses ran out of memory on the more expensive abstractions in these spaces, even for our smallest benchmark, emphasizing the need for our CEGAR approach.

The "final" column shows the abstraction size that was used in the last iteration of our approach. For all the benchmarks and both clients, these sizes are below 5% of the maximum abstraction size, shown in the "max." column. Lastly, the "iterations" column shows the number of iterations that were taken by our approach. They show that our approach is capable of exploring a huge space of abstractions for a large number of queries simultaneously, in under a few iterations. In comparison, the baseline approach (**BASELINE**) of Zhang et al. invokes the typestate client analysis far more frequently as it refines each query individually. For example, the baseline approach took 159 iterations to finish the typestate analysis on `toba-s`, while our approach only needed 15 iterations. Since the baseline for the pointer analysis client is not a refinement-based approach, it invokes the client analysis just once and is not comparable with our approach.

In the rest of this section, we evaluate the performance of the Datalog solver and the MAXSAT solver in more detail.

Performance of Datalog solver. Table 4 shows statistics of the running time of the Datalog solver in different iterations of our approach. These statistics include the minimum, maximum, and average running time over all iterations for a given analysis and benchmark. The numbers in Table 4 indicate that the abstractions chosen by our approach are small enough to allow the analyses to scale. For `schroeder-m`, one of our largest benchmarks, the change in running time from the slowest to the fastest run is only 2X for both client analyses. This further indicates that our approach is able to resolve all posed queries simultaneously before the sizes of the chosen abstractions start affecting the scalability of the client Datalog analyses. In contrast, the baseline k-object-sensitive analysis could only scale upto $k = 4$ on our larger benchmarks. Even with $k = 4$, the Datalog solver ran for over six hours on our largest benchmark when using the baseline approach. With our approach, on the other hand, the longest single run of the Datalog solver for the pointer analysis client was only seven minutes.

Figures 9 and 10 show the change in abstraction size and the analysis running time across iterations for the pointer and typestate analysis, respectively, applied on `schroeder-m`. There is a clear correlation between the growth in abstraction size and the increase in the running times. For both analyses, since our approach only chooses the cheapest viable abstraction in each iteration, the abstraction size grows almost linearly, as expected. Further, for typestate analysis, an increase in abstraction size typically results in an almost linear growth in the number of abstract states tracked. Consequently, the linear growth in the running time for the typestate analysis is also expected behavior. However, for the pointer analysis, typically, the number of distinct calling contexts grows exponentially with the increase in abstraction size. The linear curve for the running

	running time of the Datalog solver (in seconds)							running time of the MAXSAT solver (in seconds)					
	pointer analysis				typestate analysis			pointer analysis			typestate analysis		
	BASELINE	min.	max.	avg.	min.	max.	avg.	min.	max.	avg.	min.	max.	avg.
toba-s	11	5	7	6	49	82	68.1	2	7	3.1	1	6	3.1
javasrc-p	29	7	11	9	76	152	120.8	<1	4	1.6	2	19	6.4
weblech	2,574	44	54	47.5	121	172	146.6	5	11	6.7	3	8	5.3
hedc	5,058	21	37	27.9	52	58	54.3	1	23	3.7	1	2	1.7
antlr	3,723	30	55	39.3	193	325	264.8	11	44	24.1	5	27	13.25
luindex	913	59	84	76.4	311	512	426.7	8	48	16.3	6	26	14.7
lusearch	7,040	59	85	72.7	238	437	343.9	7	62	23.9	6	29	15.9
schroeder-m	23,038	192	428	289.6	1,778	2,681	2,304.6	34	257	114	37	308	138.6

Table 4: Running time of the Datalog and MAXSAT solvers in each iteration.

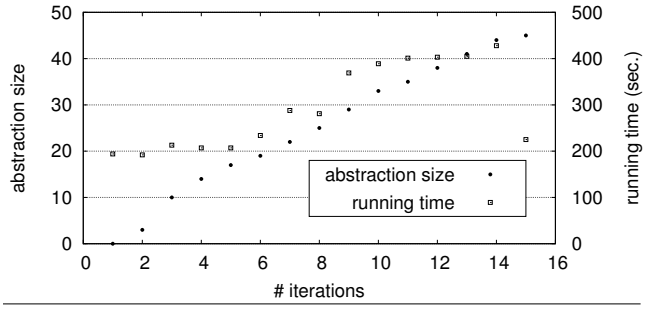


Figure 9: Running time of the Datalog solver and abstraction size for *pointer analysis* on `schroeder-m` in each iteration.

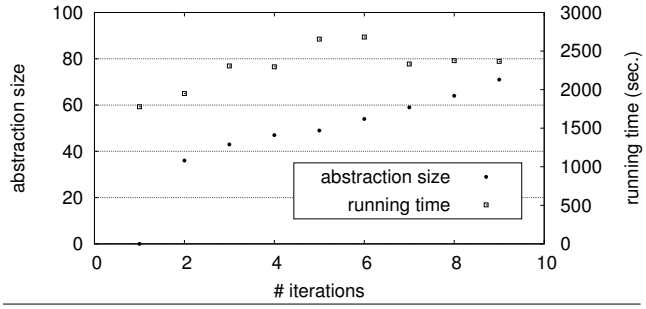


Figure 10: Running time of the Datalog solver and abstraction size for *typestate analysis* on `schroeder-m` in each iteration.

time in Figure 9 indicates that the abstractions chosen by our approach are small enough to limit this exponential growth.

Performance of MAXSAT solver. Table 4 shows statistics of the running time of the MAXSAT solver in different iterations of our approach. The metrics reported are the same as those for the Datalog solver. Although the performance of MAXSAT solvers is not completely deterministic, it is largely affected by two factors, (1) the size of boolean constraints posed to the solver, and (2) the structure of these constraints. For both analyses, as seen previously, the abstraction size increases with the number of iterations while the number of unresolved queries decreases. Growth in abstraction size increases the complexity of the client Datalog analyses, causing an increase in the number of boolean constraints generated. On the other hand, fewer queries tends to simplify the structure of the constraints to be solved.

Figure 11 shows the running time of the MAXSAT solver across all iterations for the pointer analysis applied to our largest benchmark `schroeder-m`. Initially, the solver running time shows an increasing trend but this reverses towards the end. We believe that the two conflicting factors of size and structure of the constraints are at play here. While the complexity of the constraints increases initially due to their growing size, after a certain iteration, the number of unresolved queries becomes small enough to suitably simplify the structure of the constraints and overwhelm the effect of

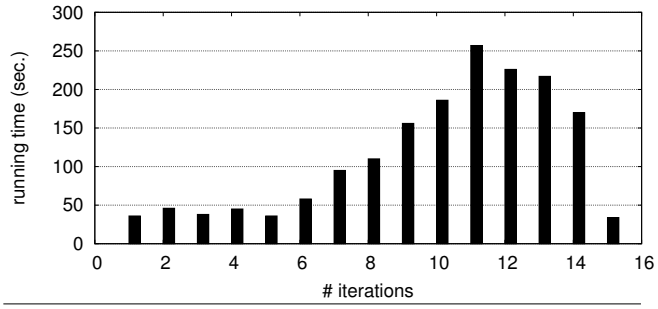


Figure 11: Running time of the MAXSAT solver for *pointer analysis* on `schroeder-m` in each iteration.

	pointer analysis		typestate analysis	
	# variables	# clauses	# variables	# clauses
toba-s	784k	1,485k	741k	938k
javasrc-p	470k	877k	1,022k	1,333k
weblech	1,620k	3,307k	1,374k	1,807k
hedc	1,245k	2,664k	606k	751k
antlr	3,621k	6,875k	2,318k	3,009k
luindex	2,406k	5,643k	2,829k	3,784k
lusearch	2,103k	5,011k	2,626k	3,524k
schroeder-m	6,706k	23,680k	16,293k	22,257k

Table 5: Statistics of MAXSAT formula in the final iteration.

growing constraint size. For the remaining benchmarks and analyses, we observed a similar trend, which we omit for the sake of brevity.

Finally, Table 5 shows the number of variables and clauses in the largest constraint that the MAXSAT solver had to solve for a given analysis and benchmark. Though the structure of the constraints is not apparent from these numbers, the large size of the constraints indicates the difficulty of the problems that the solver is tackling.

6. Related Work

Our approach is broadly related to work on constraint-based analysis, including analysis based on boolean constraints, set constraints, and SMT constraints. Constraint-based analysis has well-known benefits that our approach also avails, such as the ability to reason *about* the analysis and leveraging sophisticated solvers to implement the analysis. A key difference is that constraint-based analyses typically solve constraints generated from program text, whereas our approach solves constraints generated from an analysis run, which is itself obtained by solving constraints generated from program text.

Our approach is also related to work on CEGAR-based model checking and program analyses using Datalog, as we discuss next.

CEGAR-based Model Checking. CEGAR was originally proposed to enable model checkers to scale to even larger state-spaces than those possible using symbolic approaches such as BDDs [8]. Our motivation for using CEGAR, in contrast, is to enable designers of analyses in Datalog to express flexible abstractions. Moreover, our notions of counterexamples and refined abstractions differ radically

from those in model checkers. Despite these differences, however, there are similarities. Notably, SAT-based approaches have also been applied in model checkers [3, 6, 7], albeit for different purposes: they use SAT either to perform bounded model checking itself [7] or to decide whether a counterexample is real or spurious [3, 6], whereas we use SAT to generalize a counterexample and find a refined abstraction.

Our work is most closely related to recent work on synthesizing software verifiers from proof rules for safety and liveness properties in the form of Horn-like clauses [2, 4, 10]. Their approach is also CEGAR-based but differs in two key ways: (1) they can identify internal nodes of derivations where information gets lost due to the current abstraction, which they subsequently refine, whereas we focus on leaf nodes of derivations; and (2) they use CEGAR to solve difficult Horn constraints formulated even on infinite domains, whereas we use CEGAR for finding a better abstraction, which is then used to generate new Horn constraints. As such, their approach is more expressive and flexible, but ours appears to scale better.

Zhang et al. [26] propose a CEGAR-based approach for efficiently finding an optimal abstraction in a parametric program analysis. Our approach improves on Zhang et al. in three aspects. First, their counterexample generation requires a parametric static analysis to be disjunctive (which implies path-sensitivity), whereas any analysis written in Datalog, including non-disjunctive ones, can be handled by our approach. As a result, their approach is not applicable to the pointer analysis in Section 5. Second, their approach relies on a nontrivial backward analysis for analyzing a counterexample and selecting a next abstraction to try, but this backward analysis is not generic and should be designed for each parametric analysis. Our approach, on the other hand, uses a generic MAXSAT-based algorithm for the counterexample analysis and the abstraction selection, which only requires users to define the cost model of abstractions. Conversely, [26] converges faster for certain problems. Finally, the approach in [26] cannot mix counterexamples across iterations to generate new counterexamples for free, a feature that is present in our approach as illustrated in Section 2.

Program Analysis Using Datalog. Recent years have witnessed a surge of interest in using Datalog for program analysis (see Section 1). Datalog solvers have simultaneously advanced, using either symbolic representations of relations such as BDDs (e.g., BDDBDDB [25] and Jedd [16]) or even explicit representations (e.g., Doop [22]). More recently the popular Z3 SMT solver has been extended to compute least fixpoints of constraints expressed in Datalog [12]. Our CEGAR approach is independent of the underlying Datalog solver and leverages these advances.

Liang et al. [17] propose a cause-effect dependence analysis technique for analyses in Datalog. The technique identifies input tuples that definitely do not affect output tuples. More specifically, it computes the transitive closure of all derivations of an output tuple to identify an over-approximation of the set of input tuples needed in any derivation (e.g., $\{t_1, t_2, t_3\}$). In contrast, our approach identifies the exact set of input tuples needed in each of even exponentially many derivations (e.g., $\{\{t_1\}, \{t_2, t_3\}\}$). Thus, in our example, their approach prunes abstractions that contain $\{t_1, t_2, t_3\}$, whereas ours also prunes those that only contain $\{t_1\}$ or $\{t_2, t_3\}$.

7. Conclusion

We presented a novel CEGAR-based approach to automatically find effective abstractions for program analyses written in Datalog. We formulated the abstraction refinement problem for each iteration as a MAXSAT problem that not only successfully eliminates all abstractions which fail in a similar way but also finds the next cheapest viable abstraction. We showed the generality of our approach by applying it to two different and realistic static analyses. Finally, we demonstrated its practicality by evaluating it on a suite of eight real-world Java benchmarks.

References

[1] T. Ball and S. Rajamani. The SLAM project: Debugging system software via static analysis. In *POPL*, 2002.

[2] T. Beyene, C. Popeea, and A. Rybalchenko. Solving existentially quantified Horn clauses. In *CAV*, 2013.

[3] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu. Symbolic model checking without BDDs. In *TACAS*, 1999.

[4] N. Bjørner, K. McMillan, and A. Rybalchenko. On solving universally quantified Horn clauses. In *SAS*, 2013.

[5] M. Bravenboer and Y. Smaragdakis. Strictly declarative specification of sophisticated points-to analyses. In *OOPSLA*, 2009.

[6] S. Chaki, E. Clarke, A. Groce, S. Jha, and H. Veith. Modular verification of software components in C. In *ICSE*, 2003.

[7] E. Clarke, A. Gupta, J. Kukula, and O. Shrichman. SAT based abstraction-refinement using ILP and machine learning techniques. In *CAV*, 2002.

[8] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement for symbolic model checking. *JACM*, 50 (5), 2003.

[9] S. Fink, E. Yahav, N. Dor, G. Ramalingam, and E. Geay. Effective typestate verification in the presence of aliasing. *ACM TOSEM*, 17(2), 2008.

[10] S. Grebenshchikov, A. Gupta, N. Lopes, C. Popeea, and A. Rybalchenko. HSF(C): A software verifier based on Horn clauses. In *TACAS*, 2012.

[11] T. Henzinger, R. Jhala, R. Majumdar, and K. McMillan. Abstractions from proofs. In *POPL*, 2004.

[12] K. Hoder, N. Bjørner, and L. de Moura. μZ - an efficient engine for fixed points with constraints. In *CAV*, 2011.

[13] M. Janota. MiFuMax — a literate MaxSat solver, 2013.

[14] B. Jaumard and B. Simeone. On the complexity of the maximum satisfiability problem for Horn formulas. *IPL*, 26(1):1–4, 1987.

[15] G. Kastrinis and Y. Smaragdakis. Hybrid context sensitivity for points-to analysis. In *PLDI*, 2013.

[16] O. Lhoták and L. Hendren. Jedd: a BDD-based relational extension of Java. In *PLDI*, 2004.

[17] P. Liang and M. Naik. Scaling abstraction refinement via pruning. In *PLDI*, 2011.

[18] V. Manquinho, J. M. Silva, and J. Planes. Algorithms for weighted boolean optimization. In *SAT*, 2009.

[19] A. Milanova, A. Rountev, and B. Ryder. Parameterized object sensitivity for points-to and side-effect analyses for Java. In *ISSTA*, 2002.

[20] J. Quielle and J. Sifakis. Specification and verification of concurrent systems in CESAR. In *Proceedings of the 5th Intl. Symp. on Programming*, pages 337–350, 1982.

[21] T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *POPL*, 1995.

[22] Y. Smaragdakis and M. Bravenboer. Using Datalog for fast and easy program analysis. In *Datalog 2.0 Workshop*, 2010.

[23] Y. Smaragdakis, M. Bravenboer, and O. Lhoták. Pick your contexts well: Understanding object-sensitivity. In *POPL*, 2013.

[24] J. Whaley and M. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *PLDI*, 2004.

[25] J. Whaley, D. Avots, M. Carbin, and M. Lam. Using Datalog with binary decision diagrams for program analysis. In *APLAS*, 2005.

[26] X. Zhang, M. Naik, and H. Yang. Finding optimum abstractions in parametric dataflow analysis. In *PLDI*, 2013.

A. Proofs for Results of Section 3

Lemma 12. *Let L be a complete lattice and $f, g \in L \to L$ two monotone functions. Then* lfp *preserves order, in the sense that*

$$(\forall x : f(x) \leq g(x)) \Rightarrow \mathsf{lfp}\, f \leq \mathsf{lfp}\, g$$

Proof. By the well-known Tarski's theorem, $f(x) \leq x$ implies $\mathsf{lfp}\, f \leq x$. We have $f(\mathsf{lfp}\, g) \leq g(\mathsf{lfp}\, g) = \mathsf{lfp}\, g$. Thus, $\mathsf{lfp}\, f \leq \mathsf{lfp}\, g$. □

Proposition 1 (Monotonicity). If $C_1 \subseteq C_2$, then $[\![C_1]\!] \subseteq [\![C_2]\!]$.

Proof. The result holds because $\mathcal{P}(\mathbf{T})$ is a complete lattice, the functions $F_{C_1}, F_{C_2} \in \mathcal{P}(\mathbf{T}) \to \mathcal{P}(\mathbf{T})$ are pointwise ordered with respect to each-other, and each of them is monotone:

$$\begin{array}{l} F_{C_1}(T) \subseteq F_{C_2}(T) \\ T_1 \subseteq T_2 \;\Rightarrow\; F_{C_k}(T_1) \subseteq F_{C_k}(T_2) \qquad \text{for } k \in \{1, 2\} \end{array}$$

These properties can be readily verified from the definitions given in Figure 7, and from the assumption $C_1 \subseteq C_2$. By Lemma 12, $\mathsf{lfp}\, F_{C_1} \subseteq \mathsf{lfp}\, F_{C_2}$, which is the desired result. □

B. Proofs for Results of Section 4

B.1 Proofs of Theorems 8 and 9

Recall that for each Datalog constraint c, we have the function $f_c : \mathcal{P}(\mathbf{T}) \to \mathcal{P}(\mathbf{T})$ in Figure 7, which determines the meaning of the constraint. We first prove a lemma that connects this function with a SAT formula generated by $[c]$.

Lemma 13. *For every constraint c and all $T, T' \subseteq \mathbf{T}$ such that $f_c(T) \subseteq T$ and $T' \subseteq T$,*

$$T' \models [c]_T \quad \Leftrightarrow \quad f_c(T') \subseteq T'$$

Proof. Consider c, T', T such that $f_c(T) \subseteq T$ and $T' \subseteq T$.

First, we show the only-if direction of the equivalence. Assume

$$T' \models [c]_T.$$

Pick $t \in f_c(T')$. By the definitions of f_c and $[c]$, there exist $t_1, \ldots, t_n$ such that

1. $t_1, \ldots, t_n \in T' \wedge t \in f_c(\{t_1, \ldots, t_n\})$; and
2. when we overload $t, t_1, \ldots, t_n$ and make them refer to variables corresponding to these tuples.

   $$\phi = (t_1 \wedge \ldots \wedge t_n \to t)$$

   is a conjunct in $[c]_{T'}$.

Since $T' \subseteq T$ and $[c]$ is monotone with respect to the subset order, the formula ϕ is also a conjunct of $[c]_T$. By assumption, $T' \models [c]_T$, so $T' \models \phi$. But all of $t_1, \ldots, t_n$ are in T'. Thus, this satisfaction relationship implies that $t \in T'$.

Next, we prove the if direction. Suppose that $f_c(T') \subseteq T'$. Pick one conjunct ϕ of $[c]_T$. We have to prove that $T' \models \phi$. By the definition of $[c]$, there are tuples $t_1, \ldots, t_n \in T$ and a tuple t such that when we use $t, t_1, \ldots, t_n$ to refer to variables corresponding to these tuples,

$$\phi = (t_1 \wedge \ldots t_n \to t)$$

and $t \in f_c(\{t_1, \ldots, t_n\})$. If some of $t_1, \ldots, t_n$ is missing in T', we have $T' \models \phi$, as desired. Otherwise,

$$t \in f_c(\{t_1, \ldots, t_n\}) \subseteq f_c(T') \subseteq T',$$

where we use the monotonicity of f_c with respect to $\subseteq$. From $t \in T'$ that we have just proved follows the desired $T' \models \phi$. □

Next, we prove a similar result for Datalog programs C. Recall that as in the case of Datalog constraints, every Datalog program C has the corresponding function $F_C : \mathcal{P}(\mathbf{T}) \to \mathcal{P}(\mathbf{T})$ in Figure 7 that determines the meaning of the program. We prove that this function is closely connected with a SAT formula generated by $[C]$.

Lemma 14. *For all T, T' such that $F_C(T) = T$,*

$$T' \models [C]_T \quad \Leftrightarrow \quad [\![C \cup (T' \cap T)]\!] \subseteq T'$$

Proof. Consider T', T such that $F_C(T) = T$. Let $T'' = T' \cap T$.

First, we show the only-if direction of the equivalence in the lemma. Suppose that

$$T' \models [C]_T.$$

Let

$$F(T_0) = T_0 \cup \bigcup_{c \in C \cup T''} f_c(T_0).$$

Then, $[\![C \cup T'']\!] = \bigcup_{i \geq 0} F^i(\emptyset)$. Furthermore, $T'' \subseteq T'$. Hence, to prove $[\![C \cup T'']\!] \subseteq T'$, it is sufficient to show that

$$\forall n \geq 0.\, F^n(\emptyset) \subseteq T''.$$

We prove this sufficient condition by induction on n.

1. **Base case** $n = 0$**.** Since $F^0(\emptyset) = \emptyset$, this case is immediate.
2. **Inductive case** $n > 0$**.** In this case,

   $$F^n(\emptyset) = F^{n-1}(\emptyset) \cup \bigcup_{c \in C \cup T''} f_c(F^{n-1}(\emptyset)).$$

   Pick $t \in F^n(\emptyset)$. If t belongs to the LHS $F^{n-1}(\emptyset)$ of the above union, we have the desired $t \in T''$ by the induction hypothesis. Otherwise, there is $c \in C \cup T''$ such that $t \in f_c(F^{n-1}(\emptyset))$. If $c \in T''$, c must be a tuple and be the same as t. Hence, $t \in T''$, as desired. Otherwise, there exist $c \in C$ and tuples

   $$t_1, \ldots, t_k \in F^{n-1}(\emptyset)$$

   such that $t \in f_c(\{t_1, \ldots, t_k\})$. By the induction hypothesis, $t_1, \ldots, t_k \in T''$. Hence, our proof obligation $t \in T''$ can be discharged if we show that

   $$f_c(T'') \subseteq T''.$$

   We show this subset relationship using Lemma 13. Specifically, we show the following three properties:

   $$T'' \subseteq T, \quad f_c(T) \subseteq T, \quad \text{and} \quad T'' \models [c]_T.$$

   The first property holds because $T'' = T' \cap T$, and the second is true because T is a fixed-point of F_C and c is a constraint in C. We now show the third property. By assumption, $T' \models [C]_T$. Hence,

   $$T' \models [c]_T.$$

   Since $f_c(T) \subseteq T$, for every variable in the formula $[c]_T$, the corresponding tuple appears in T. Hence, the above satisfaction relationship implies

   $$(T' \cap T) \models [c]_T.$$

   That is, $T'' \models [c]_T$, the very property that we want to prove.

Next, we prove the if direction of the equivalence. Suppose that

$$[\![C \cup T'']\!] \subseteq T'.$$

We should show that $T' \models [C]_T$. Suppose not. Then, there exist $c \in C$ and tuples $t_1, \ldots, t_n \in T$ and t such that

1. $t_1, \ldots, t_n \in T'$ but $t \notin T'$; and

2. when $t_1, \ldots, t_n, t$ are used as variables corresponding to these tuples,
$$\phi = (t_1 \wedge \ldots \wedge t_n \to t)$$
is one conjunct of $[c]_T$.

Then, $t \in f_c(\{t_1, \ldots, t_n\})$. Since $T'' = T' \cap T$,
$$t_1, \ldots, t_n \in T''.$$
Hence,
$$t \in [\![\{c\} \cup \{t_1, \ldots, t_n\}]\!] \subseteq [\![C \cup T'']\!].$$
But $[\![C \cup T'']\!] \subseteq T'$. Thus, $t \in T'$. But this contract the fact that $t \notin T'$. □

Theorem 8. *Let T be a fixed-point of function F_C used to define $[\![C]\!]$ in Figure 7, and let T' be a subset of T; thus $F_C(T) = T$ and $T' \subseteq T$. Then $[\![C \cup T']\!]$ is fully determined by $[C]_T$, as follows:*
$$t \in [\![C \cup T']\!] \quad \Leftrightarrow \quad \models \big((\bigwedge T' \wedge [C]_T) \to t\big)$$

Proof. Consider T, T' such that $T' \subseteq T$ and $F_C(T) = T$.

First, we prove the only-if direction. Pick $t \in [\![C \cup T']\!]$. Consider T'' such that $T'' \models \bigwedge T' \wedge [C]_T$. This means that
$$T' \subseteq T'' \quad \text{and} \quad T'' \models [C]_T. \tag{1}$$
Using these facts, we will prove that
$$t \in T''. \tag{2}$$
The key step of our proof is to show that
$$[\![C \cup (T'' \cap T)]\!] \subseteq T''. \tag{3}$$
This gives the set membership in (2), as shown in the following reasoning:
$$t \in [\![C \cup T']\!] \subseteq [\![C \cup (T'' \cap T)]\!] \subseteq T''.$$
The first subset relationship here holds because T' is a subset of both T (by the choice of T') and T'' (by the first conjunct in (1)), and $[\![-]\!]$ is monotone. Now it remains to discharge the subset relationship in (3). This is easy, because it is an immediate consequence of Lemma 14 and the fact that $T'' \models [C]_T$, the second conjunct in (1).

Second, we show the if direction. Consider $t \in \mathbf{T}$ such that
$$\forall T'' \subseteq \mathbf{T} : T'' \models (\bigwedge T' \wedge [C]_T) \to t. \tag{4}$$
Let $T'' = [\![C \cup T']\!]$. Then,
$$T'' \models \bigwedge T' \quad \text{and} \quad [\![C \cup (T'' \cap T)]\!] \subseteq T''. \tag{5}$$
The first conjunct holds because $T' \subseteq T''$. The second conjunct holds because
$$[\![C \cup (T'' \cap T)]\!] \subseteq [\![C \cup T'']\!] = T''.$$
Here the subset relationship follows from the monotonicity of $[\![-]\!]$ with respect to $\subseteq$, and the equality holds because T'' includes T', it is a fixed-point of $F_{C \cup T'}$, and every fixed-point T_0 of $F_{C \cup T'}$ equals $[\![C \cup T' \cup T_0]\!]$. By Lemma 14, the second conjunct in (5) implies
$$T'' \models [C]_T.$$
What we have proved so far and our choice of t in (4) imply that $T'' \models t$. That is, $t \in T''$. If we unroll the definition of T'', we get the desired set membership:
$$t \in [\![C \cup T']\!].$$
□

Theorem 9. *Let $T_1, \ldots, T_n$ be fixed-points of F_C. For all T',*
$$\models \big((\bigwedge T' \wedge [C]_{T_1} \wedge \ldots \wedge [C]_{T_n}) \to t\big) \quad \Rightarrow \quad t \in [\![C \cup T']\!]$$

Proof. Let $T_1, \ldots, T_n$ be fixed-points of F_C. Consider t and T' such that
$$\models \big((\bigwedge T' \wedge \phi) \to t\big) \tag{6}$$
where $\phi = \bigwedge\{\, [C]_{T_k} \mid 1 \le k \le n \,\}$. We should show that
$$t \in [\![C \cup T']\!].$$
Let $T'' = [\![C \cup T']\!]$. Because of our assumption in (6), it suffices to prove that
$$T'' \models \bigwedge T' \quad \text{and} \quad T'' \models \phi.$$
The first conjunct holds because $T' \subseteq [\![C \cup T']\!] = T''$. Showing the second conjunct is slightly more involved, and it will form the contents of the rest of this proof. Suppose that the second conjunct does not hold. Then, there exist a constraint $c \in C$, an index $1 \le k \le n$, and tuples $t_0, t_1, \ldots, t_m$ such that

1. $t_1, \ldots, t_m \in T''$ but $t_0 \notin T''$;
2. $t_1, \ldots, t_m \in T_k$ and $t_0 \in f_c(\{t_1, \ldots, t_m\})$; and
3. when we overload $t_0, t_1, \ldots, t_m$ and treat them as variables,
$$\psi = (t_1 \wedge \ldots \wedge t_m \to t_0)$$
is a conjunct in $[c]_{T_k}$, and hence in ϕ.

Since $t_0 \in f_c(\{t_1, \ldots, t_m\})$,
$$t_0 \in [\![\{c\} \cup \{t_1, \ldots, t_m\}]\!] \subseteq [\![C \cup T'']\!].$$
But $[\![C \cup T'']\!] = T''$ because T'' includes T', it is a fixed-point of $F_{C \cup T'}$ and every fixed-point T_0 of $F_{C \cup T'}$ equals $[\![C \cup T' \cup T_0]\!]$. Hence, $t_0 \in T''$, which contradicts the fact that $t_0 \notin T''$. □

B.2 Proof of Theorem 10

Theorem 10. *If $\mathsf{choose}(\phi, Q')$ evaluates to an element of the set $\mathbf{A} \setminus \gamma(\phi, Q')$ whenever such an element exists, and to $\mathsf{impossible}$ otherwise, then Algorithm 1 is partially correct: it returns (R, I) such that $R = \mathcal{R}(\mathbf{A}, Q)$ and $I = Q \setminus R$. In addition, if $\mathbf{A}$ is finite, then Algorithm 1 terminates.*

Proof. The key part of our proof is to show that Algorithm 1 has the following loop invariant: letting $I = Q \setminus R$,

1. $R \subseteq \mathcal{R}(\mathbf{A}, Q)$;
2. ϕ has the form $\bigwedge\{[C]_{T_k} \mid 1 \le k \le n\}$ for some fixed-points $T_1, \ldots, T_n$ of F_C; and
3. $\mathcal{R}(\mathbf{A} \setminus \gamma(\phi, I), I) = \mathcal{R}(\mathbf{A}, I)$.

Let us start by proving that the invariant holds initially. When the loop of Algorithm 10 is entered for the first time,
$$R = \emptyset \wedge \phi = \mathsf{True} \wedge I = Q.$$
Hence, the first and second conditions of our invariant hold in this case. For the third condition, we notice that
$$\begin{aligned}\gamma(\phi, I) = \gamma(\mathsf{True}, Q) &= \{A \in \mathbf{A} \mid \models (\bigwedge A \to \bigwedge Q)\} \\ &= \{A \in \mathbf{A} \mid Q \subseteq A\},\end{aligned}$$
and also that
$$\mathcal{R}(\mathbf{A}, Q) = \mathcal{R}(\{A \in \mathbf{A} \mid Q \not\subseteq A\}, Q).$$
The third condition follows from these two facts and the equality $I = Q$.

Next, we prove that our invariant is preserved by the loop of Algorithm 1. Assume that the invariant holds for R, ϕ, and I. Also, assume that the result A of $\mathsf{choose}(\phi, I)$ is not $\mathsf{impossible}$. Let
$$R' = R \cup (Q \setminus [\![C \cup A]\!]), \quad \phi' = \phi \wedge [C]_{[\![C \cup A]\!]}, \quad I' = Q \setminus R'.$$

We should show that the three conditions of our invariant hold for R', ϕ', and I'. The first condition is

$$R' \subseteq \mathcal{R}(\mathbf{A}, Q),$$

which holds because $R \subseteq \mathcal{R}(\mathbf{A}, Q)$ and $(Q \setminus [\![C \cup A]\!]) \subseteq \mathcal{R}(\mathbf{A}, Q)$. The second condition also holds because ϕ has the required form, the newly added conjunct in the definition of ϕ' is $[C]_{[\![C\cup A]\!]}$, and $F_C([\![C \cup T_0]\!]) = [\![C \cup T_0]\!]$ for every T_0. It remains to prove the third condition:

$$\mathcal{R}(\mathbf{A} \setminus \gamma(\phi', I'), I') = \mathcal{R}(\mathbf{A}, I').$$

For this, we will show the following sufficient condition:

$$\forall A' \in \gamma(\phi', I').\ I' \setminus [\![C \cup A']\!] = \emptyset.$$

Pick $A' \in \gamma(\phi', I')$. Then,

$$\models \bigwedge A' \wedge \phi' \to I'.$$

Equivalently,

$$\forall t \in I'.\ \models \bigwedge A' \wedge \phi' \to t.$$

Since ϕ' is the conjunction of $[C]_{T_j}$ for some fixed-point T_j of F_C, by Theorem 9, the above formula implies that

$$\forall t \in I'.\ t \in [\![C \cup A']\!].$$

Hence, $I' \setminus [\![C \cup A']\!] = \emptyset$, as desired.

We now use our invariant to show the partial correctness of Algorithm 1. Assume that the invariant holds for ϕ, R, and I. Suppose that

$$\mathsf{choose}(\phi, Q \setminus R) = \mathsf{impossible}.$$

Then, $\mathbf{A} \setminus \gamma(\phi, I) = \emptyset$ by our assumption on choose. Because ϕ, R, and I satisfy our loop invariant,

$$\mathcal{R}(\mathbf{A}, Q \setminus R) = \mathcal{R}(\mathbf{A}, I) = \mathcal{R}(\mathbf{A} \setminus \gamma(\phi, I), I) = \mathcal{R}(\emptyset, I) = \emptyset,$$

where The last equality uses the definition of $\mathcal{R}$. But $R \subseteq \mathcal{R}(\mathbf{A}, Q)$ by the loop invariant. Hence, by the definition of $\mathcal{R}$,

$$\mathcal{R}(\mathbf{A}, Q) = R.$$

This means that when Algorithm 1 returns, its result (R, I) satisfies $R = \mathcal{R}(\mathbf{A}, Q)$ and $I = Q \setminus R$, as claimed in the theorem.

Finally, we show that if $\mathbf{A}$ is finite, Algorithm 1 terminates. Our proof is based on the fact that the set

$$\gamma(\phi, Q \setminus R) \subseteq \mathbf{A}$$

is strictly increasing. Consider ϕ, R, I satisfying the loop invariant. Assume that the result A of $\mathsf{choose}(\phi, I)$ is not $\mathsf{impossible}$. Let

$$R' = R \cup (Q \setminus [\![C \cup A]\!]), \quad \phi' = \phi \wedge [C]_{[\![C\cup A]\!]}, \quad I' = Q \setminus R'.$$

Since I' is a subset of I and ϕ' contains ϕ as its conjunct,

$$(\models \phi' \to \phi) \quad \text{and} \quad (\models \bigwedge I \to \bigwedge I')$$

This implies that for every $A \in \mathbf{A}$,

$$\models (\bigwedge A \wedge \phi \to \bigwedge I) \to (\bigwedge A \wedge \phi' \to \bigwedge I').$$

Hence,

$$\gamma(\phi, I) \subseteq \gamma(\phi', I').$$

It remains to show that this subset relationship is strict. This is the case because $A \notin \gamma(\phi, I)$ by the assumption on choose, but it is in $\gamma(\phi', I')$. To see why $A \in \gamma(\phi', I')$, notice $I' \subseteq [\![C \cup A]\!]$, $A \subseteq [\![C \cup A]\!]$, and $F_C([\![C \cup A]\!]) = [\![C \cup A]\!]$. Hence, by Theorem 8,

$$\models (\bigwedge A \wedge [C]_{[\![C\cup A]\!]} \to \bigwedge I').$$

This implies

$$\models (\bigwedge A \wedge \phi \wedge [C]_{[\![C\cup A]\!]} \to \bigwedge I'),$$

which is equivalent to $A \in \gamma(\phi', I')$. □