Rethinking Static Analysis by Combining Discrete and Continuous Reasoning

Mayur Naik

Department of Computer and Information Science
University of Pennsylvania
mhnaik@cis.upenn.edu

Abstract. Static analyses predominantly use discrete modes of logical reasoning to derive facts about programs. Despite significant strides, this form of reasoning faces new challenges in modern software applications and practices. These challenges concern not only traditional analysis objectives such as scalability, accuracy, and soundness, but also emerging ones such as tailoring analysis conclusions based on relevance or severity of particular code changes, and needs of individual programmers.
We advocate seamlessly extending static analyses to leverage continuous modes of logical reasoning in order to address these challenges. Central to our approach is expressing the specification of the static analysis in a constraint language that is amenable to computing provenance information. We use the logic programming language Datalog as proof-of-concept for this purpose. We illustrate the benefits of exploiting provenance even in the discrete setting. Moreover, by associating weights with constraints, we show how to amplify these benefits in the continuous setting.
We also present open problems in aspects of analysis usability, language expressiveness, and solver techniques. The overall process constitutes a fundamental rethinking of how to design, implement, deploy, and adapt static analyses.

Keywords: Static analysis, constraint solving, provenance, probabilistic logics, alarm ranking, inductive logic programming

1 Introduction

Static analysis has made remarkable strides in theory and practice over the decades since the seminal work of Cousot and Cousot on abstract interpretation [10]. The practical impact of static analysis tools includes triumphs such as Astrée [11] for verifying memory safety properties of C programs used in Airbus controller software, SLAM [6] for verifying temporal safety properties that device drivers on the Windows operating system must obey, Coverity [7] for checking a wide variety of programming errors based on semantic inconsistencies in large enterprise C/C++ applications, and Infer [9] for modularly checking various safety properties of C, C++, Objective C, and Java code in Android and iOS mobile applications.

At the same time, new programming languages with rich dynamic features, such as Javascript and Python, and new software engineering practices such as continuous integration and continuous deployment (CI/CD) are becoming increasingly popular. These settings favor programmer productivity but pose new challenges to static analysis, such as tailoring analysis conclusions based on relevance or severity of code changes by individual developers in large teams [19, 30]. The resulting trend in the growth and diversity of software applications is challenging even traditional objectives of static analysis, such as scalability, accuracy, and soundness [21].

Static analyses predominantly use discrete modes of logical reasoning to derive facts about programs: the facts and the process of deriving them are discrete in nature. For instance, such analyses typically work by applying deductive rules of the form $A \Rightarrow B$ on program text. The undecidability of the static analysis problem lends such rules to be necessarily incomplete, deriving consequent B which may be false even if antecedent A is true.

In this paper, we argue that leveraging continuous modes of logical reasoning opens promising avenues to address the above challenges. For instance, we can extend the syntax of the above deductive rule with a real-valued weight $w \in [0, 1]$, and extend its semantics to the continuous domain, which allows to selectively violate instances of the rule as well as associate a *confidence score* with each derived fact. This enables to leverage inference procedures for conventional probabilistic graphical models such as Bayesian networks [31] (e.g. [32]) or Markov Logic Networks (MLN) [35] (e.g. [24]). We can even learn the weights and structure of the rules from (possibly partial or noisy) input-output data (e.g. labeled alarms on program text) rather than being hand-engineered by human experts. By replacing the traditional operations ($\wedge$, $\vee$) and values $\{true, false\}$ of the Boolean semiring with the corresponding operations ($\times$, max) and values [0, 1] of the Viterbi semiring [14], we can leverage ideas from numerical relaxation in optimization problems, such as Newton's root-finding method, MCMC-based random sampling, and stochastic gradient descent [36]. This opens the door to the invention of new program approximations and to the customization of static analyses by end-users.

Crucially, we advocate to seamlessly *extend* rather than *replace* existing methods, by synergistically combining discrete and continuous forms of logical reasoning in static analysis. In particular, we presume that the analysis is expressed in a constraint language that is amenable to computing provenance information in the form of proof trees that explain how the analysis derives output facts (e.g., alarms) from input facts (e.g., program text). Such information allows to answer questions such as whether a particular alarm is relevant to a particular code change in a continuously evolving codebase. Such information is useful even in the discrete setting but its benefits are amplified in the continuous setting—for instance, allowing to answer questions such as the *extent* to which an alarm is relevant to a code change. Throughout, we use the logic programming language Datalog [1] as proof-of-concept for the constraint language, since it suffices to express a wide range of analyses in the literature, and efficient procedures exist

for evaluating Datalog programs, computing provenance information, and extending the Datalog language and solvers with capabilities such as statistical relational models and mathematical optimization procedures [3].

The rest of the paper is organized as follows. Section 2 illustrates the key ingredients of our approach on the problem of improving the effective accuracy of a static analysis by incorporating user feedback. Section 3 outlines the landscape of challenges in static analysis where similar ideas are applicable and discusses open problems. Finally, Section 4 concludes.

2 Illustrative Overview

We illustrate our approach using an example from [32] which applies a static analysis to a multi-threaded Java program called Apache FTP server. Figure 1 shows a code fragment from the program. The `RequestHandler` class is used to handle client connections. An object of this class is created for every incoming connection to the server. The `close()` method is used to clean up and close an open client connection, and the `getRequest()` method is used to access the `request` field. Both these methods can be invoked from other parts of the program by multiple threads in parallel on the same `RequestHandler` object.

Dataraces are a common and insidious kind of error that plague multi-threaded programs. Since `getRequest()` and `close()` may be called on the same `RequestHandler` object by different threads in parallel, there exists a datarace between lines 10 and 20: the first thread may read the `request` field while the second thread concurrently sets the `request` field to `null`.

On the other hand, even though the `close()` method may also be simultaneously invoked by multiple threads on the same `RequestHandler` object, the atomic test-and-set operation on lines 13–16 ensures that for each object instance, lines 17–24 are executed at most once. There is therefore no datarace between the pair of accesses to `controlSocket` on lines 17 and 18, and similarly no datarace between the accesses to `request` on lines 19 and 20, and so forth.

We may use a static analysis to find dataraces in this program. However, due to the undecidable nature of the problem, the analysis may also report alarms on lines 17–24. In the rest of this section, we illustrate how our approach generalizes from user feedback to guide the analysis away from the false positives and towards the actual datarace.

A Static Datarace Analysis. Figure 1 shows a simplified version of the analysis in Chord, a static datarace detector for Java programs [29]. The analysis is expressed in Datalog as a set of logical rules over relations.

The analysis takes relations $\mathcal{N}(p_1, p_2)$, $\mathcal{U}(p_1, p_2)$, and $\mathcal{A}(p_1, p_2)$ as input, and produces relations $\mathcal{P}(p_1, p_2)$ and $\mathcal{R}(p_1, p_2)$ as output. In all relations, variables p_1 and p_2 range over the domain of program points. Each relation may be visualized as the set of tuples indicating some known facts about the program. For our example program, $\mathcal{N}(p_1, p_2)$ may contain the tuples $\mathcal{N}(l_1, l_2)$, $\mathcal{N}(l_2, l_3)$, etc. While some input relations, such as $\mathcal{N}(p_1, p_2)$, may be directly obtained from the text of the program being analyzed, other input relations, such as $\mathcal{U}(p_1, p_2)$

```
1  package org.apache.ftpserver;
2
3  public class RequestHandler {
4    FtpRequestImpl request;
5    FtpWriter writer;
6    BufferedReader reader;
7    Socket controlSocket;
8    boolean isClosed;
9    public FtpRequest getRequest() {
10     return request;          // l0
11   }
12   public void close() {
13     synchronized(this) {     // l1
14       if (isClosed) return;  // l2
15       isClosed = true;       // l3
16     }
17     controlSocket.close();   // l4
18     controlSocket = null;    // l5
19     request.clear();         // l6
20     request = null;          // l7
21     writer.close();
22     writer = null;
23     reader.close();
24     reader = null;
25   }
26 }
```

Input relations:

$\mathcal{N}(p_1, p_2)$ (program point p_1 is an immediate successor of program point p_2)

$\mathcal{U}(p_1, p_2)$ (no common lock guards program points p_1 and p_2)

$\mathcal{A}(p_1, p_2)$ (instructions at program points p_1 and p_2 may access the same memory location, and constitute a possible datarace)

Output relations:

$\mathcal{P}(p_1, p_2)$ (different threads may reach program points p_1 and p_2 in parallel)

$\mathcal{R}(p_1, p_2)$ (datarace may occur between different threads while executing the instructions at program points p_1 and p_2)

Analysis rules:

$r_1: \mathcal{P}(p_1, p_3) \text{ :- } \mathcal{P}(p_1, p_2), \mathcal{N}(p_2, p_3), \mathcal{U}(p_1, p_3)$

$r_2: \mathcal{P}(p_2, p_1) \text{ :- } \mathcal{P}(p_1, p_2)$

$r_3: \mathcal{R}(p_1, p_2) \text{ :- } \mathcal{P}(p_1, p_2), \mathcal{A}(p_1, p_2)$

Fig. 1: **Java program and simplified static datarace analysis in Datalog.**

or $\mathcal{A}(p_1, p_2)$, may themselves be the result of earlier analyses (in this case, a lockset analysis and a pointer analysis, respectively).

The rules are intended to be read from right-to-left, with all variables universally quantified, and the :- operator interpreted as implication. For example, the rule r_1 may be read as saying, "For all program points p_1, p_2, p_3, if p_1 and p_2 may execute in parallel ($\mathcal{P}(p_1, p_2)$), and p_3 may be executed immediately after p_2 ($\mathcal{N}(p_2, p_3)$), and p_1 and p_3 are not guarded by a common lock ($\mathcal{U}(p_1, p_3)$), then p_1 and p_3 may themselves execute in parallel."

Observe that the analysis is *flow-sensitive*, i.e. it takes into account the order of program statements, represented by the relation $\mathcal{N}(p_1, p_2)$, but *path-insensitive*, i.e. it disregards the satisfiability of path conditions and predicates along branches. This is an example of an approximation to enable the analysis to scale to large programs.

Applying the Analysis to a Program. To apply the above analysis to our example program, one starts with the set of input tuples, and repeatedly applies the inference rules r_1, r_2, and r_3, until no new facts can be derived. Starting with the tuple $\mathcal{P}(l_4, l_2)$, we show a portion of the derivation graph thus obtained in Figure 2. Each box represents a tuple and is shaded gray if it is an input tuple. Nodes identified with rule names represent grounded clauses: for example, the node $r_1(l_4, l_2, l_3)$ indicates the "*grounded instance*" of the rule r_1 with $p_1 = l_4$, $p_2 = l_2$, and $p_3 = l_3$. This clause takes as hypotheses the tuples $\mathcal{P}(l_4, l_2)$, $\mathcal{N}(l_2, l_3)$, and $\mathcal{U}(l_4, l_3)$, and derives the conclusion $\mathcal{P}(l_4, l_3)$, and the arrows represent these dependencies.

Observe that clause nodes are conjunctive: a rule fires iff all of its antecedents are derivable. On the other hand, tuple nodes are disjunctive: a tuple is derivable iff there exists at least one derivable clause of which it is the conclusion. For instance, the tuple $\mathcal{P}(l_6, l_7)$ can be derived in one of two ways: either by instantiating r_1 with $p_1 = l_6$, $p_2 = l_6$, and $p_3 = l_7$ (as shown in Figure 2), or by instantiating r_2 with $p_1 = l_7$ and $p_2 = l_6$ (not shown).

Observe that lines l_4 and l_2 can indeed execute in parallel, and the original conclusion $\mathcal{P}(l_4, l_2)$, in Figure 2, is true. However, the subsequent conclusion $\mathcal{P}(l_4, l_3)$ is spurious, and is caused by the analysis being incomplete: the second thread to enter the `synchronized` block will necessarily leave the method at line l_2. Among others, four subsequent false alarms—$\mathcal{R}(l_4, l_5)$, $\mathcal{R}(l_5, l_5)$, $\mathcal{R}(l_6, l_7)$, and $\mathcal{R}(l_7, l_7)$—result from the analysis incorrectly concluding $\mathcal{P}(l_4, l_3)$.

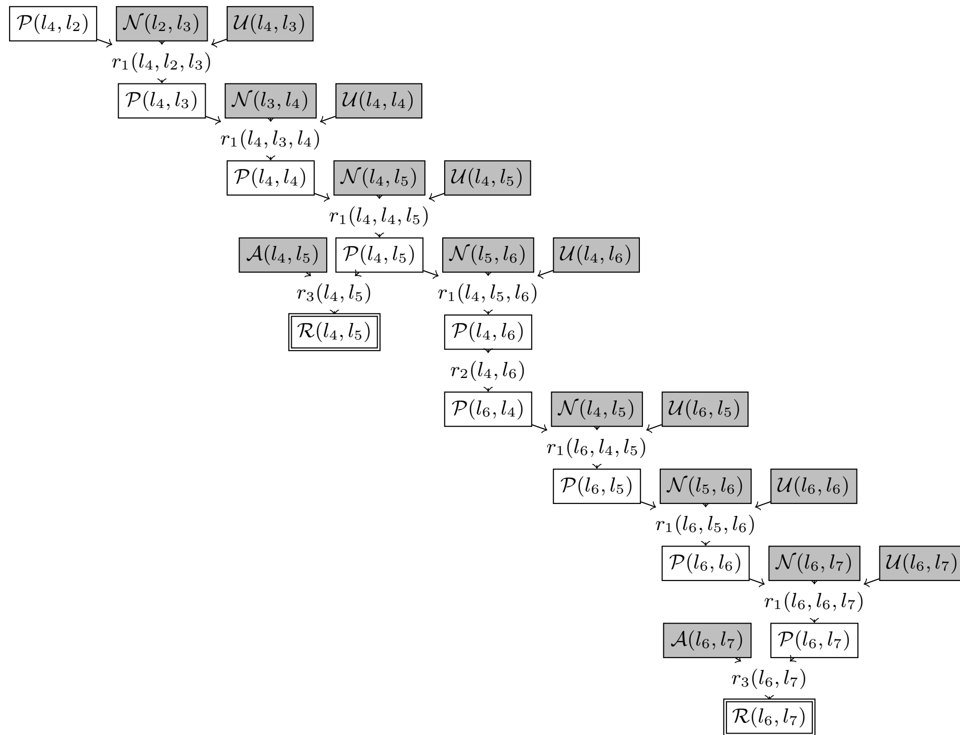

Fig. 2: A portion of the derivation graph obtained by applying the static datarace analysis to the program in Figure 1. The central focus of this section is following: if the user identifies $\mathcal{R}(l_4, l_5)$ as a false alarm, then how should this affect our confidence in the remaining alarms?

Quantifying Incompleteness using Probabilities. Incomplete analysis rules are the principal cause of false alarms: although $\mathcal{P}(l_4, l_2)$, $\mathcal{N}(l_2, l_3)$, and $\mathcal{U}(l_4, l_3)$ are all true, it is not the case that $\mathcal{P}(l_4, l_3)$. To address this problem, we relax the interpretation of clause nodes, and treat them probabilistically:

$$\Pr(r_1(l_4, l_2, l_3) \mid h_1) = 0.95, \text{ and} \tag{1}$$
$$\Pr(\neg r_1(l_4, l_2, l_3) \mid h_1) = 1 - 0.95 = 0.05, \tag{2}$$

where $h_1 = \mathcal{P}(l_4, l_2) \wedge \mathcal{N}(l_2, l_3) \wedge \mathcal{U}(l_4, l_3)$ is the event indicating that all the hypotheses of $r_1(l_4, l_2, l_3)$ are true, and $p_1 = 0.95$ is the probability of the clause "correctly firing". By setting p_1 to a value strictly less than 1, we make it possible for the conclusion of $r_1(l_4, l_2, l_3)$, $\mathcal{P}(l_4, l_3)$ to still be false, even though all the hypotheses in h_1 hold.

In this new setup, as before, if any of the antecedents of $r_1(l_4, l_2, l_3)$ is false, then it is itself definitely false:

$$\Pr(r_1(l_4, l_2, l_3) \mid \neg h_1) = 0, \text{ and} \tag{3}$$
$$\Pr(\neg r_1(l_4, l_2, l_3) \mid \neg h_1) = 1. \tag{4}$$

We also continue to treat tuple nodes as regular disjunctions:

$$\Pr(\mathcal{P}(l_6, l_7) \mid r_1(l_6, l_6, l_7) \vee r_2(l_7, l_6)) = 1, \tag{5}$$
$$\Pr(\mathcal{P}(l_6, l_7) \mid \neg(r_1(l_6, l_6, l_7) \vee r_2(l_7, l_6))) = 0, \tag{6}$$

and treat all input tuples t as being known with certainty: $\Pr(t) = 1$.

These rule probabilities can be learnt using an expectation maximization (EM) algorithm from training data. For now, we associate the rule r_3 with firing probability $p_3 = 0.95$, and r_2 with probability $p_2 = 1$. Finally, to simplify the discussion, we treat $\mathcal{P}(l_0, l_1)$ and $\mathcal{P}(l_1, l_1)$ as input facts, with $\Pr(\mathcal{P}(l_0, l_1)) = 0.40$ and $\Pr(\mathcal{P}(l_1, l_1)) = 0.60$.

From Derivation Graphs to Bayesian Networks. By attaching conditional probability distributions (CPDs) such as equations 1–6 to each node of Figure 2, we view the derivation graph as a Bayesian network. Specifically, we perform marginal inference on the network to associate each alarm with the probability, or *belief*, that it is a true datarace. This procedure generates a list of alarms ranked by probability, shown in Table 1a. For example, it computes the probability of $\mathcal{R}(l_4, l_5)$ as follows:

$$\begin{aligned}
\Pr(\mathcal{R}(l_4, l_5)) &= \Pr(\mathcal{R}(l_4, l_5) \wedge r_3(l_4, l_5)) + \Pr(\mathcal{R}(l_4, l_5) \wedge \neg r_3(l_4, l_5)) \\
&= \Pr(\mathcal{R}(l_4, l_5) \wedge r_3(l_4, l_5)) \\
&= \Pr(\mathcal{R}(l_4, l_5) \mid r_3(l_4, l_5)) \cdot \Pr(r_3(l_4, l_5)) \\
&= \Pr(r_3(l_4, l_5) \mid \mathcal{P}(l_4, l_5) \wedge \mathcal{A}(l_4, l_5)) \cdot \Pr(\mathcal{P}(l_4, l_5)) \cdot \Pr(\mathcal{A}(l_4, l_5)) \\
&= 0.95 \cdot \Pr(\mathcal{P}(l_4, l_5)) = 0.95^4 \cdot \Pr(\mathcal{P}(l_4, l_2)) \\
&= 0.95^8 \cdot \Pr(\mathcal{P}(l_1, l_1)) = 0.398.
\end{aligned}$$

The user now inspects the top-ranked report, $\mathcal{R}(l_4, l_5)$, and classifies it as a false alarm. The key idea underlying our approach is that *generalizing from feedback is conditioning on evidence*. By replacing the prior belief $\Pr(a)$, for each alarm a, with the posterior belief, $\Pr(a \mid \neg\mathcal{R}(l_4, l_5))$, our approach effectively propagates the user feedback to the remaining conclusions of the analysis. This results in the updated list of alarms shown in Table 1b. Observe that the belief in the closely related alarm $\mathcal{R}(l_6, l_7)$ drops from 0.324 to 0.030, while the belief in the unrelated alarm $\mathcal{R}(l_0, l_7)$ remains unchanged at 0.279. As a result, the entire family of false alarms drops in the ranking, so that the only true datarace is now at the top.

The computation of the updated confidence values occurs by a similar procedure as before. For example:

$$\begin{aligned}
&\Pr(\mathcal{R}(l_6, l_7) \mid \neg\mathcal{R}(l_4, l_5)) \\
&\quad = \Pr(\mathcal{R}(l_6, l_7) \wedge \mathcal{P}(l_4, l_5) \mid \neg\mathcal{R}(l_4, l_5)) + \Pr(\mathcal{R}(l_6, l_7) \wedge \neg\mathcal{P}(l_4, l_5) \mid \neg\mathcal{R}(l_4, l_5)) \\
&\quad = \Pr(\mathcal{R}(l_6, l_7) \wedge \mathcal{P}(l_4, l_5) \mid \neg\mathcal{R}(l_4, l_5)).
\end{aligned}$$

Next, $\mathcal{R}(l_4, l_5)$ and $\mathcal{R}(l_6, l_7)$ are conditionally independent given $\mathcal{P}(l_4, l_5)$ as it occurs on the unique path between them. So,

$$\begin{aligned}
&\Pr(\mathcal{R}(l_6, l_7) \wedge \mathcal{P}(l_4, l_5) \mid \neg\mathcal{R}(l_4, l_5)) \\
&\quad = \Pr(\mathcal{R}(l_6, l_7) \mid \mathcal{P}(l_4, l_5)) \cdot \Pr(\mathcal{P}(l_4, l_5) \mid \neg\mathcal{R}(l_4, l_5)) \\
&\quad = 0.95^5 \cdot \Pr(\mathcal{P}(l_4, l_5) \mid \neg\mathcal{R}(l_4, l_5)).
\end{aligned}$$

Finally, by Bayes' rule, we have:

$$\begin{aligned}
\Pr(\mathcal{P}(l_4, l_5) \mid \neg\mathcal{R}(l_4, l_5)) &= \frac{\Pr(\neg\mathcal{R}(l_4, l_5) \mid \mathcal{P}(l_4, l_5)) \cdot \Pr(\mathcal{P}(l_4, l_5))}{\Pr(\neg\mathcal{R}(l_4, l_5))} \\
&= \frac{0.05 \cdot 0.95^7 \cdot 0.60}{0.60} = 0.03.
\end{aligned}$$

Our prior belief in $\mathcal{P}(l_4, l_5)$ was $\Pr(\mathcal{P}(l_4, l_5)) = 0.42$, so that $\Pr(\mathcal{P}(l_4, l_5) \mid \neg\mathcal{R}(l_4, l_5)) \ll \Pr(\mathcal{P}(l_4, l_5))$, but is still strictly greater than 0. This is because one eventuality by which $\neg\mathcal{R}(l_4, l_5)$ may occur is for $\mathcal{P}(l_4, l_5)$ to be true, but for the clause $r_3(l_4, l_5)$ to misfire. We may now conclude that $\Pr(\mathcal{R}(l_6, l_7) \mid \neg\mathcal{R}(l_4, l_5)) = 0.95^5 \cdot 0.03 = 0.030$.

The Interaction Model. In summary, given an analysis and a program to be analyzed, our approach takes as input the set of tuples and grounded clauses produced by the Datalog solver at fixpoint, and constructs the belief network. Next, it performs Bayesian inference to compute the probability of each alarm, and presents the alarm with highest probability for inspection by the user. The user then indicates its ground truth, and our approach incorporates this feedback as evidence for subsequent iterations.

There are several possible stopping criteria by which the user could cease interaction, e.g., only inspect alarms with confidence higher than some threshold p_0, and stop once the confidence of the highest ranked alarm drops below p_0;

or only inspect n alarms, and stop after n iterations. In all these situations, we lose any soundness guarantees provided by the underlying analysis, but given the large number of alarms typically emitted by analysis tools, this approach strikes a useful tradeoff between accuracy and soundness.

Table 1: List of alarms produced (a) before and (b) after the feedback $\neg\mathcal{R}(l_4, l_5)$. Observe how the real datarace $\mathcal{R}(l_0, l_7)$ rises in ranking as a result of feedback.

(a) $\Pr(a)$.

Rank	Belief	Program points
1	0.398	l_4, l_5
2	0.378	l_5, l_5
3	0.324	l_6, l_7
4	0.308	l_7, l_7
5	0.279	l_0, l_7

(b) $\Pr(a \mid \neg\mathcal{R}(l_4, l_5))$.

Rank	Belief	Program points
1	0.279	l_0, l_7
2	0.035	l_5, l_5
3	0.030	l_6, l_7
4	0.028	l_7, l_7
5	0	l_4, l_5

3 A Taxonomy of Research Directions

In this section, we outline the landscape of challenges in static analysis, argue how techniques similar to those in the preceding section can be used to address them, and discuss open problems. We classify the landscape into three broad categories: i) balancing analysis tradeoffs (Section 3.1), ii) tailoring analysis results (Section 3.2), and iii) specifying and implementing analyses (Section 3.3). Note that these challenges are agnostic of specific analyses and apply broadly to a variety of different analyses.

3.1 Balancing Analysis Tradeoffs

The undecidability of the static analysis problem necessitates tradeoffs between accuracy, cost, and soundness. We focus on two of the most common tradeoffs: accuracy vs. cost, and accuracy vs. soundness.

Analysis Accuracy vs. Cost. This tradeoff concerns balancing the cost of the program abstraction against the accuracy of the analysis result. A popular paradigm to suitably strike this tradeoff is counterexample-guided abstraction refinement (CEGAR).

In [39], we show how to enable CEGAR for arbitrary analyses specified in Datalog. It uses a formulation of maximum satisfiability (MaxSAT), an optimization extension of the Boolean satisfiability problem, wherein the hard constraints encode provenance information relating analysis results (e.g. alarms) to abstractions, while the soft constraints encode the relative costs of different abstractions.

The objective is to either find the cheapest abstraction that suffices to prove a program property of interest or show that no such abstraction exists.

Open problems in this space include supporting richer analyses, such as those that employ widening, and considering not only the costs of different abstractions but also their likelihood of success (e.g. [15]).

Analysis Accuracy vs. Soundness. This tradeoff concerns balancing the accuracy of the analysis result against the soundness of the analysis. Arguably the most common tradeoff struck by static analyses in practice, it incurs false positives as well as false negatives, unlike the accuracy vs. cost tradeoff which only incurs false positives.

In [24] and [32], we show how to enable this tradeoff for arbitrary analyses specified in Datalog. The approach in [32] was illustrated in Section 2 wherein we perform marginal inference on a Bayesian network induced by provenance constraints to rank alarms produced by the analysis. In [24], we employ a different approach by performing MAP inference on a Markov Logic Network (MLN). The main difference between the two approaches, besides the different probabilistic models, is that we obtain a confidence score for each derived fact in [32], whereas we obtain the "most likely world" in [24]. The former is more amenable to user interaction both because it allows to rank analysis alarms and because incorporating user feedback translates into conditioning on evidence (in contrast, [24] requires additional constraints to propagate the user feedback).

Open problems in this space include how to transfer user feedback across programs, providing a rigorous semantics of rule weights, and richer probabilistic models which allow rule weights to depend on finer-grained program contexts.

3.2 Tailoring Analysis Results

A relatively recent area of exploration in static analysis concerns how to improve usability by tailoring analysis results. We consider unguided vs. interactive approaches, batch vs. continuous approaches, classification vs. ranking approaches, and different metrics for ranking.

Unguided vs. Interactive. Conventional static analyses are unguided in that they cannot tailor results to individual users. As we illustrated in Section 2, continuous modes of logical reasoning allow analyses to incorporate and generalize user feedback, but no longer guarantee soundness. Our earlier work [38] enables interaction while preserving soundness, but does not generalize user feedback; instead, the objective is to minimize the user burden by prioritizing questions that maximize the alarms to be resolved. Even in this discrete setting, provenance information is used to relate alarms to questions.

Open problems in this space include coping with noise inherent in interactive approaches, how to generalize user feedback effectively within a program, and how to transfer the knowledge learnt from interaction across programs.

Batch vs. Continuous Reasoning. Conventional static analyses operate in batch mode in that different parts of the program are presumed to be equally relevant to the user. However, in prevalent settings of continuous integration and continuous deployment, the user is interested only in alarms relevant to their code change [19, 30].

In [16], we show how to compute differential provenance information between two program versions in order to prioritize alarms relevant to the code change, even in discrete modes of reasoning. Moreover, by incorporating continuous modes of reasoning, we show how to amplify the benefits by ranking the alarms, and by incorporating and generalizing user feedback.

Open problems in this arena include how to guarantee the soundness of differential analysis even in the discrete setting, and the related problem of what code granularity to use for identifying the changes between two program versions (e.g. AST-based vs. line-based).

Alarm Clustering vs. Ranking. Another dimension to tailor analysis results is to cluster related alarms in order to reduce inspection effort. Provenance information can be used to identify dependencies between alarms and cluster correlated alarms together (e.g. [20]). However, clustering treats all alarms uniformly, which is seldom useful in practice.

Ranking alarms on the other hand opens the door to different metrics to prioritize alarms. We showed in Section 2 how continuous modes of reasoning can be combined with provenance information to rank alarms based on *ground truth*. However, alternative metrics of ranking are possible, such as *relevance* to code changes, and alarm *severity*. An important open problem in this setting is how to quantify severity.

3.3 Analysis Specification and Implementation

Another set of challenges concerns how to specify and implement constraint-based analyses to effectively support the use-cases discussed above. We discuss how to synthesize analyses automatically from data, expressiveness issues of the language for specifying analyses, and capabilities required of analysis solvers.

Synthesizing Analyses from Data. The problem of synthesizing analyses from input-output data (e.g., programs with labeled alarms) is motivated by two reasons: first, allowing end-users to customize analyses to diverse settings, and secondly, overcoming limitations of hand-engineered analyses that hinder use-cases discussed above. For example, the effectiveness of generalizing user feedback across alarms relies heavily on the quality of analysis rules, as rule weights can only go so far to compensate for it.

In recent work [2, 33, 36], we have developed increasingly scalable approaches to synthesize Datalog programs from input-output data. Provenance information is crucial to scaling the approach in [33] which follows the counterexample-guided inductive synthesis (CEGIS) paradigm for program synthesis. The key idea is to use a Datalog solver to not only produce counterexamples for the candidate

Datalog program with respect to given input-output data, but also explain them using "why" and "why-not" provenance. This in turn allows the iterative CEGIS process to converge faster and scale better.

The Datalog synthesis problem can also be seen as an instance of the classic Inductive Logic Programming (ILP) problem [26, 28]. A key difference is that ILP techniques focus on learning relations, often probabilistic ones, from vast amounts of mined data, e.g., biological data [27]. On the other hand, in our setting, and in a large class of synthesis techniques, the goal is to interactively infer a program from a small, representative set of input-output examples.

Open problems in this arena include active learning to reduce the burden on providing labeled data upfront, coping with noisy data, avoiding the need for syntactic rule templates, and synthesizing analyses with expressive features such as invented predicates, recursion, negation, and aggregation. Note that coping with noisy data is fundamentally necessary because, even if the data perfectly captures the concrete semantics of program behavior, the synthesized analysis must follow an abstract semantics that approximates the data.

Expressiveness of Analysis Language. The use of Datalog for static analysis dates back to Reps's work on demand-driven analysis [34]. The desire to express a wide variety of analyses in Datalog has led to extending the language with features such as value construction, negation, aggregation, and higher-order predicates. These extensions include LogicBlox's LogiQL [3] which forms the basis of the Doop static analysis framework [8], Semmle's QL [5] which allows Datalog programs to be written over the target program's syntax, the higher-order functional Datalog language Datafun [4], and Flix for specifying static analyses [22]. Finally, many works extend the semantics of logic programming to the continuous domain, such as Markov Logic Networks [35], ProbLog [12], and its extensions such as DeepProbLog [25] and aProbLog [18].

Capabilities of Analysis Solvers. A benefit of constraint-based analysis lies in the ability to leverage off-the-shelf solvers. The need for more expressive features in the constraint language is counterbalanced by the need to efficiently execute analyses specified in the language. Moreover, the use-cases discussed above require features besides just efficient execution, notably efficient computation of provenance information. Efficient algorithms for "why" and "why not" provenance for Datalog remain areas of active research [37, 40], and notions of provenance for more expressive logics are further beyond [13]. Finally, another interesting direction of exploration is the integration of Datalog solvers with solvers for other theories, such as SMT solvers [17] and solvers for mathematical optimization (e.g. MaxSAT and Integer Linear Programming) [3, 23].

4 Conclusions

We proposed a new approach to static analysis that builds upon the long-standing constraint-based approach while providing fundamentally new capabilities. The approach aims to seamlessly combine discrete and continuous modes

of logical reasoning. To this end, it relies on static analyses being specified in a constraint language that is amenable to computing provenance information. We showed how provenance plays a crucial role in a rich variety of applications of our approach. Finally, we outlined a taxonomy of research directions and described open problems in the field.

Acknowledgments: I thank the following for making vital contributions to the body of research summarized in this paper: PhD students Sulekha Kulkarni, Xujie Si, and Xin Zhang; postdocs Kihong Heo, Woosuk Lee, and Mukund Raghothaman; and collaborators Radu Grigore, Aditya Nori, and Hongseok Yang. I am grateful to Peter O'Hearn for providing useful feedback at various stages of this work. This research was supported by DARPA award #FA8750-15-2-0009, NSF awards #1253867 and #1526270, ONR award #N00014-18-1-2021, and gifts from Facebook, Google, and Microsoft.

References

1. Abiteboul, S., Hull, R., Vianu, V.: Foundations of Databases. Addison-Wesley (1995)
2. Albarghouthi, A., Koutris, P., Naik, M., Smith, C.: Constraint-based synthesis of Datalog programs. In: Proceedings of the 23rd International Conference on Principles and Practice of Constraint Programming (CP) (2017)
3. Aref, M., ten Cate, B., Green, T.J., Kimelfeld, B., Olteanu, D., Pasalic, E., Veldhuizen, T.L., Washburn, G.: Design and implementation of the LogicBlox system. In: Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD) (2015)
4. Arntzenius, M., Krishnaswami, N.: Datafun: A functional Datalog. In: Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP) (2016)
5. Avgustinov, P., de Moor, O., Jones, M.P., Schäfer, M.: QL: Object-oriented queries on relational data. In: Proceedings of the 30th European Conference on Object-Oriented Programming (ECOOP) (2016)
6. Ball, T., Rajamani, S.: The SLAM project: Debugging system software via static analysis. In: Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL) (2002)
7. Bessey, A., Block, K., Chelf, B., Chou, A., Fulton, B., Hallem, S., Henri-Gros, C., Kamsky, A., McPeak, S., Engler, D.: A few billion lines of code later: Using static analysis to find bugs in the real world. Communications of the ACM **53**(2) (Feb 2010)
8. Bravenboer, M., Smaragdakis, Y.: Strictly declarative specification of sophisticated points-to analyses. In: Proceedings of the ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA) (2009)
9. Calcagno, C., Distefano, D., Dubreil, J., Gabi, D., Hooimeijer, P., Luca, M., O'Hearn, P., Papakonstantinou, I., Purbrick, J., Rodriguez, D.: Moving fast with software verification. In: NASA Formal Method Symposium. Springer (2015)
10. Cousot, P., Cousot, R.: Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Proceedings of the 4th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL) (1977)

11. Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., Rival, X.: The ASTRÉE analyzer. In: Proceedings of the 14th European Symposium on Programming (ESOP) (2005)
12. De Raedt, L., Kimmig, A., Toivonen, H.: Problog: A probabilistic Prolog and its application in link discovery. In: Proceedings of the 20th International Joint Conference on Artificial Intelligence (2007)
13. Grädel, E., Tannen, V.: Semiring provenance for first-order model checking (2017), http://arxiv.org/abs/1712.01980
14. Green, T., Karvounarakis, G., Tannen, V.: Provenance semirings. In: Proceedings of the 26th Symposium on Principles of Database Systems (PODS) (2007)
15. Grigore, R., Yang, H.: Abstraction refinement guided by a learnt probabilistic model. In: Proceedings of the 43rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL) (2016)
16. Heo, K., Raghothaman, M., Si, X., Naik, M.: Continuously reasoning about programs using differential bayesian inference. In: Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI) (2019)
17. Hoder, K., Bjørner, N., De Moura, L.: μz: An efficient engine for fixed points with constraints. In: Proceedings of the 23rd International Conference on Computer-Aided Verification (CAV) (2011)
18. Kimmig, A., den Broeck, G.V., De Raedt, L.: An algebraic Prolog for reasoning about possible worlds. In: Proceedings of the 25th AAAI Conference on Artificial Intelligence (2011)
19. Lahiri, S., Vaswani, K., Hoare, C.A.R.: Differential static analysis: Opportunities, applications, and challenges. In: Proceedings of the FSE/SDP Workshop on Future of Software Engineering Research (FoSER) (2010)
20. Lee, W., Lee, W., Yi, K.: Sound non-statistical clustering of static analysis alarms. In: Proceedings of the 13th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI) (2012)
21. Livshits, B., Sridharan, M., Smaragdakis, Y., Lhoták, O., Amaral, J.N., Chang, B.Y.E., Guyer, S.Z., Khedker, U.P., Møller, A., Vardoulakis, D.: In defense of soundiness: A manifesto. Communications of the ACM **58**(2) (Jan 2015)
22. Madsen, M., Yee, M.H., Lhoták, O.: From Datalog to Flix: A declarative language for fixed points on lattices. In: Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI) (2016)
23. Mangal, R., Zhang, X., Naik, M., Nori, A.V.: Volt: A lazy grounding framework for solving very large MaxSAT instances. In: Proceedings of the International Conference on Theory and Applications of Satisfiability Testing (SAT) (2015)
24. Mangal, R., Zhang, X., Nori, A., Naik, M.: A user-guided approach to program analysis. In: Proceedings of the Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE) (2015)
25. Manhaeve, R., Dumancic, S., Kimmig, A., Demeester, T., De Raedt, L.: Deepproblog: Neural probabilistic logic programming. In: Advances in Neural Information Processing Systems (2018)
26. Muggleton, S.: Inductive logic programming. New generation computing **8**(4) (1991)
27. Muggleton, S.: Scientific knowledge discovery using inductive logic programming. Communications of the ACM **42**(11) (Nov 1999)

28. Muggleton, S., de Raedt, L., Poole, D., Bratko, I., Flach, P., Inoue, K., Srinivasan, A.: ILP turns 20 - biography and future challenges. Machine Learning **86**(1) (Jan 2012)
29. Naik, M., Aiken, A., Whaley, J.: Effective static race detection for Java. In: Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI) (2006)
30. O'Hearn, P.W.: Continuous reasoning: Scaling the impact of formal methods. In: Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science (LICS) (2018)
31. Pearl, J.: Probabilistic reasoning in intelligent systems: Networks of plausible inference. Morgan Kaufmann (1988)
32. Raghothaman, M., Kulkarni, S., Heo, K., Naik, M.: User-guided program reasoning using Bayesian inference. In: Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI) (2018)
33. Raghothaman, M., Mendelson, J., Zhao, D., Scholz, B., Naik, M.: Provenance-guided synthesis of Datalog programs. Tech. rep. (2019)
34. Reps, T.: Demand interprocedural program analysis using logic databases. In: Ramakrishnan, R. (ed.) Applications of Logic Databases, pp. 163–196. Springer (1995)
35. Richardson, M., Domingos, P.: Markov Logic Networks. Machine Learning **62**(1-2) (2006)
36. Si, X., Raghothaman, M., Heo, K., Naik, M.: Synthesizing Datalog programs using numerical relaxation. In: Proceedings of the 28th International Joint Conference on Artificial Intelligence (IJCAI) (2019)
37. Xu, J., Zhang, W., Alawini, A., Tannen, V.: Provenance analysis for missing answers and integrity repairs. IEEE Data Eng. Bull. **41**(1) (2018)
38. Zhang, X., Grigore, R., Si, X., Naik, M.: Effective interactive resolution of static analysis alarms. In: Proceedings of the ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications (OOPSLA) (2017)
39. Zhang, X., Mangal, R., Grigore, R., Naik, M., Yang, H.: On abstraction refinement for program analyses in Datalog. In: Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI) (2014)
40. Zhao, D., Subotic, P., Scholz, B.: Provenance for large-scale Datalog (2019), http://arxiv.org/abs/1907.05045