

Combining the Logical and the Probabilistic in Program Analysis

Xin Zhang

Georgia Institute of Technology, USA
xin.zhang@gatech.edu

Xujie Si

University of Pennsylvania, USA
xsi@cis.upenn.edu

Mayur Naik

University of Pennsylvania, USA
mhnaik@cis.upenn.edu

Abstract

Conventional program analyses have made great strides by leveraging logical reasoning. However, they cannot handle uncertain knowledge, and they lack the ability to learn and adapt. This in turn hinders the accuracy, scalability, and usability of program analysis tools in practice. We seek to address these limitations by proposing a methodology and framework for incorporating probabilistic reasoning directly into existing program analyses that are based on logical reasoning. We demonstrate that the combined approach can benefit a number of important applications of program analysis and thereby facilitate more widespread adoption of this technology.

CCS Concepts • **Theory of computation** → *Program analysis; Constraint and logic programming; Program verification*;
• **Software and its engineering** → *Formal software verification*;
• **Computing methodologies** → *Probabilistic reasoning*

Keywords Program Analysis, Logic, Probability, Markov Logic Network, Maximum Satisfiability

1. Introduction

Program analyses are algorithms that discover a wide range of useful artifacts about programs, including bugs, proofs, and specifications. Existing program analyses are expressed in the form of logical axiom/inference rules that are handcrafted by experts. This logic-based approach provides important benefits. First, logical rules are human-comprehensible, making it convenient for analysis writers to express their domain knowledge. Secondly, the results produced by solving logical rules often come with explanations (e.g., provenance information), making analysis tools easy to use. Last but not least, logical rules enable program analyses to provide rigorous formal guarantees such as soundness.

While logic-based program analyses have achieved remarkable success, however, they have significant limitations: they cannot handle uncertain knowledge and they lack the ability to learn and adapt. Although the semantics of most programs are deterministic, uncertainties arise in many scenarios due to reasons such as imprecise specifications, missing program parts, imperfect environment models, and many others. Current program analyses rely on experts to manually choose their representations, and such representations cannot be changed once they are shipped to end-users. However,

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.

Copyright is held by the owner/author(s). Publication rights licensed to ACM.

MAPL'17, June 18, 2017, Barcelona, Spain
ACM. 978-1-4503-5071-6/17/06...\$15.00
<http://dx.doi.org/10.1145/3088525.3088563>

the diversity of usage scenarios prevents such fixed representations from addressing the needs of individual end-users. Moreover, the analysis does not improve as it reasons about more programs, and therefore repeats past mistakes.

To address the drawbacks of the existing logical approach, we propose to combine logical and probabilistic reasoning in program analysis. While the logical part preserves the benefits of the current approach, the probabilistic part enables handling uncertainties and provides the additional ability to learn and adapt. Moreover, such a combined approach enables to incorporate probability directly into existing program analyses, leveraging a rich literature.

Rest of the paper. Section 2 illustrates how combining logic and probability can benefit important applications of program analysis. These applications are presented in more detail in [17, 46, 47]. Section 3 describes a declarative language for expressing such applications and presents a general recipe for incorporating probabilistic reasoning into existing program analyses that are based on logical reasoning. It also presents a scalable learning and inference engine for the language using techniques elaborated in [18, 19, 48]. Section 4 surveys related work, Section 5 discusses future directions, and Section 6 concludes.

Key Insights:

- Combining logical and probabilistic reasoning in program analysis provides the best of both worlds, such as soundness guarantees on one hand and the ability to adapt on the other.
- Program analyses are usually specified using axiom/inference rules that admit only logical reasoning. We enable incorporating probabilistic reasoning by attaching weights to such rules.
- We adopt the semantics of Markov Logic Networks (MLNs) from the AI community for weighted rules. We propose learning and inference algorithms that achieve scalability and accuracy by leveraging domain insights from program analysis.

2. Motivating Applications

We present three prominent applications of program analysis to motivate our approach: automated verification, interactive verification, and static bug detection. For ease of exposition, we presume the given analysis operates on an abstract program representation in the form of a directed graph. We illustrate our three applications using a static information-flow analysis applied to the two graphs depicted in Figure 1.

2.1 Automated Verification

A central problem in automated verification concerns picking an abstraction of the program that balances accuracy and scalability. An ideal abstraction should keep only as much information relevant to proving a program property of interest. Efficiently finding

such an abstraction is challenging because the space of possible abstractions is typically exponential in program size or even infinite.

Consider the graph in Figure 1(a). Suppose the possible abstractions are indicated by dotted ovals, each of which independently enables the analysis to lose distinctions between the contained nodes, and thereby trade accuracy for scalability. We thus have a total of $2^3 = 8$ abstractions. We denote each abstraction using a bitstring $b_2b_4b_6$ where bit b_i is 0 iff the distinction between nodes $i, i + 1$ is lost by that abstraction. The least precise but cheapest abstraction 000 loses all distinctions, whereas the most precise but costliest abstraction 111 keeps all distinctions. Suppose we wish to prove that this graph does not have a path from node 1 to node 8. The absence of such a path may, for instance, imply the absence of malicious information flow in the original program. The ideal abstraction for this purpose is 010, that is, it loses distinctions between nodes 2, 3 and nodes 6, 7 but differentiates between nodes 4, 5.

Limits of purely logical reasoning. A purely logical approach, such as one based on the popular CEGAR (counter-example guided abstraction refinement) technique, starts with the cheapest abstraction and iteratively refines parts of it by generalizing the cause of failure of the abstraction used in the current iteration. For instance, in our example, it starts with abstraction 000, which fails to prove the absence of a path from node 1 to node 8. However, it faces a non-deterministic choice of whether to refine b_2 , b_4 , or b_6 next. A poor choice hinders scalability or even termination of the analysis (in the case of an infinite space of abstractions).

Incorporating probabilistic reasoning. A probabilistic approach can help guide a logical approach to better abstraction selection. For instance, it can leverage the success probability of each abstraction, which in turn can be obtained from a probability model built from training data. In our example, such a model may predict that refining b_4 has a higher success probability than refining b_2 or b_6 .

The case for a combined approach. The above two approaches in fact address complementary aspects of abstraction selection. A combined approach stands to gain their benefits without suffering from their limitations. For instance, a logical approach can infer with certainty that refining b_2 is futile due to the presence of the edge from node 1 to node 4. However, it is unable to decide whether refining b_4 or b_6 next is more likely to prove our property of interest. Here, a probabilistic approach can provide the needed bias towards refining b_4 over b_6 , enabling the combined approach to select abstraction 010 next.

In summary, the combined approach attempts only two cheap abstractions, 000 and 010, before successfully proving the given property. Besides allowing logical and probabilistic elements to interact in a fine-grained manner and amplify the benefits of these individual approaches, the combined approach also allows to encode other *objective functions* uniformly with probabilities. These may include, for instance, the relative costs of different abstractions and rewards for proving different properties. The combined approach thus allows to strike a more effective balance between accuracy and scalability than the individual approaches.

Our results using a combined approach. We have devised such an approach in recent work [7, 45, 46] and demonstrated its effectiveness on widely-used analyses, including pointer analysis, type-state analysis, and concurrency analysis. For instance, the plot in Figure 2 shows the distribution of the cheapest abstractions identified by our approach for proving different memory accesses (called “queries”) thread-local in a multi-threaded Java benchmark program using a thread-escape analysis, which is part of a static race detection tool for Java. The program comprises 340 KLOC and each abstraction has $\sim 10K$ binary components—one per object allocation site in the program—for a search space of 2^{10K} . The plot

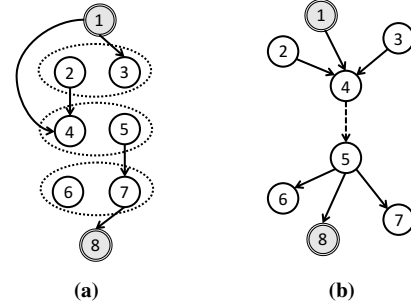


Figure 1: Graphs depicting how different applications of our approach enable a program analysis to avoid reporting false information flow from node 1 to node 8 in two programs.

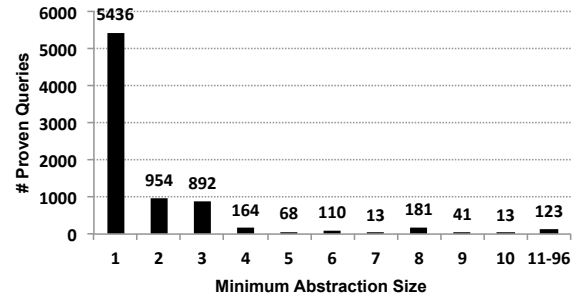


Figure 2: Minimum abstraction sizes needed for proving different queries by a thread-escape analysis in a Java program.

shows that only a tiny fraction of these components suffices to be refined for proving the vast majority of queries, but different components must be refined for queries occurring in different parts of the program. The sparsity of this data also highlights the promise of training a probabilistic approach to predict which components to refine for which queries, not only within the program but even across programs.

2.2 Interactive Verification

Automated verification is inherently incomplete due to undecidability reasons. Interactive verification seeks to address this limitation by introducing a human in the loop. A central challenge for an interactive verifier concerns reducing user effort by deciding which questions to the user are expected to yield the highest payoff.

Consider the graph in Figure 1(b). Suppose we once again wish to prove that this graph does not have a path from node 1 to node 8. Suppose the dotted edge from node 4 to node 5 is spurious, that is, it is present due to the incompleteness of the verifier. This spurious edge results in the verifier reporting a false alarm. Suppose the questions that the user is capable of answering are of the form: “Is edge (x, y) spurious?”. Then, the ideal set of questions to ask in this example is the single question: “Is edge $(4, 5)$ spurious?”.

Limits of purely logical reasoning. A purely logical approach can help prune the space of possible questions to ask. In particular, for our example, it can determine that it is fruitless to ask the user whether any of edges $(2, 4)$, $(3, 4)$, $(5, 6)$, and $(5, 7)$ is spurious. But it faces a non-deterministic choice of whether to ask the user about the spuriousness of edge $(1, 4)$, $(4, 5)$, or $(5, 8)$. In the worst case, this approach ends up asking all three questions, instead of just $(4, 5)$.

Incorporating probabilistic reasoning. A probabilistic approach can help guide a logical approach to better question selection in interactive verification. In particular, it can leverage the likelihood of different answers to each question, which in turn can be obtained from a probability model built from dynamic or static heuristics. In our example, for instance, test runs of the original program may reveal that edges (1, 4) and (5, 8) are definitely not spurious, but edge (4, 5) *may* be spurious. Similarly, a static heuristic might state that an edge (x, y) with a high in-degree for x and a high out-degree for y is likely spurious—a criterion that only edge (4, 5) meets in our example.

The case for a combined approach. The above two approaches can be combined to compute the expected payoff of each question. For instance, the inference by the probabilistic approach that edge (4, 5) is likely spurious can be combined with the inference by the logical approach that no path exists from node 1 to node 8 if edge (4, 5) is absent, thereby proving our property of interest. This approach can thus infer that the question of whether edge (4, 5) is spurious is the one with the highest payoff.

The combined approach allows to encode other objective functions that may be desirable in interactive verification. Consider a scenario in which multiple false alarms arise from a common root cause. In our example, such a scenario arises when we wish to verify that there is no path from any node in $\{1, 2, 3\}$ to any node in $\{6, 7, 8\}$. Maximizing the payoff in this scenario involves asking the least number of questions that are likely to rule out the most number of these paths. In our example, even assuming equal likelihood of each answer to any question, we can conclude that the payoff in this scenario is maximized by asking whether edge (4, 5) is spurious: it has a payoff of 9/1 compared to, for instance, a payoff of 5/2 for the set of questions $\{(1, 4), (5, 8)\}$ (since 5 of the 9 paths are ruled out if both edges in this set are deemed spurious by the user).

Our results using a combined approach. We have devised such an approach in recent work [47] and showed its effectiveness on a concurrency analysis. The approach achieves two key goals: *generalization* and *prioritization*. Generalization ensures that the number of questions asked is much smaller than the number of false alarms eliminated while prioritization aims to prioritize asking questions that eliminate the most false alarms.

Figure 3 shows the efficacy of our approach at resolving false alarms of a static datarace analysis on a multi-threaded Java benchmark program comprising 160 KLOC. Only 18 questions regarding the thread-locality of memory accesses suffice to resolve 87% of the 594 false alarms, highlighting the impact of generalization; and the first 3 questions eliminate over 300 false alarms, highlighting the impact of prioritization. The red points plot the results of an idealized oracle whereas the blue points plot the results obtained using a decision tree trained to predict users’ answers from simple static and dynamic heuristics. The coincidence of the two plots highlights the accuracy of the combined approach at nearly perfectly predicting the payoff.

2.3 Static Bug Detection

Another widespread application of program analysis is bug detection. Its key challenge lies in the need to avoid false positives (or false bugs) and false negatives (or missed bugs). They arise because of various approximations and assumptions that an analysis writer must necessarily make. However, they are absolutely undesirable to analysis users.

Consider the graph in Figure 1(b). Suppose this time all edges in the graph are real but certain paths are spurious, resulting in a mix of true positives and false positives among the paths from nodes in $\{1, 2, 3\}$ to nodes in $\{6, 7, 8\}$.

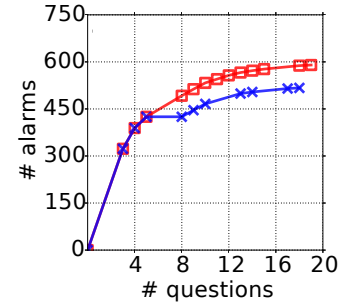


Figure 3: Number of questions asked and false alarms resolved by interactive static race detection on a Java program.

Limits of purely logical reasoning. A purely logical approach allows the analysis writer to express idioms for bug detection. An idiom in our graph example is:

“If there is an edge (x, y) then there is a path (x, y) .”

Another idiom captures the transitivity rule:

“If there is a path (x, y) and an edge (y, z) , then there is a path (x, z) .”

These idioms enable to suppress certain false positives, e.g., they prevent reporting a path from node 8 to node 1. However, they cannot incorporate feedback from an analysis user about retained false positives and generalize it to suppress similar false positives. For instance, suppose subpath (1, 5) is spurious. Even if the analysis user labels paths (1, 6) and (1, 7) as spurious, a purely logical approach cannot deduce that the subpath (1, 5) is the likely source of imprecision, and generalize it to suppress reporting path (1, 8). As a result, an analysis user must manually inspect each of the 9 paths to sift the true positives from the false positives.

Incorporating probabilistic reasoning. A probabilistic approach can provide the ability to associate a probability with each analysis fact and compute it based on a model trained using past data. In our example, it can compute a probability for each path in the graph. For instance, a model might predict that paths of longer length are less likely. Ranking-based bug detection tools exemplify this approach.

The case for a combined approach. The above two approaches can be combined to incorporate positive (resp. negative) feedback from an analysis user about true (resp. false) positives, and learn from it to retain (resp. suppress) similar true (resp. false) positives. For this purpose, we associate a probability with each idiom written by the analysis writer using the logical approach, and obtain the probability by training on previously labeled data. In our example, then, suppose the analysis user labels paths (1, 6) and (1, 7) as spurious. These labels are themselves treated probabilistically. The objective function seeks to balance the confidence of the analysis writer in the idioms with the confidence of the analysis user in the labels. In our example, the optimal solution involves suppressing the subpath (1, 5), which in turn prevents deducing path (1, 8).

Our results using a combined approach. We have devised such an approach [17] and demonstrated its effectiveness on a datarace analysis, a call-graph analysis, and an information-flow analysis for Java programs. Figure 4 shows the extent of generalization of user feedback on (5%, 10%, 15%, 20%) of randomly selected bug reports by a datarace analysis on a multi-threaded Java program comprising 200 KLOC. The underlying analysis produces 338 false reports and 700 true reports for a total of 1,038 reports. With only 5% feedback on these reports (i.e., by labeling roughly 50 of these reports as true vs. false), our approach is able to eliminate almost 70% of the false reports while retaining almost 100% of the true

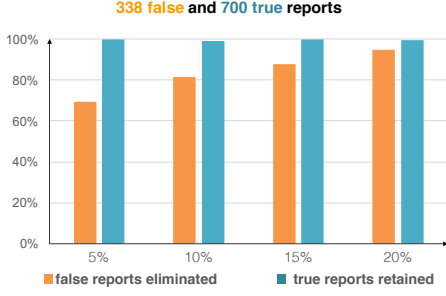


Figure 4: Generalization of feedback provided on bugs reported by a static race detection analysis on a Java program.

reports. This highlights the promise of a combined logical and probabilistic approach in improving the accuracy of bug detection through limited feedback.

3. Our Approach

In this section, we first introduce a declarative language for combining logic and probability. Then we describe a general framework built upon it for integrating probabilistic reasoning into an existing logical analysis. Finally, we present an effective learning and inference engine for this language to support our framework.

3.1 Markov Logic Network

Combining logical and probabilistic reasoning has been an active research area in artificial intelligence [36]. Towards this end, researchers have proposed various languages [10, 12, 21, 23, 34]. We choose *Markov Logic Network* (MLN) [34] as the language in our framework as it extends first-order logic, which and whose variants are often used to express program analyses.

A Markov Logic Network is a set of pairs (F_i, w_i) where F_i is a first-order formula and w_i is a real number. Given a set of constants C , the MLN defines a distribution of outputs, each of which is a set of tuples. We call $\llbracket F_i \rrbracket_C$ the grounding of F_i with respect to C , which is a set of formulae produced by instantiating all quantified variables in F_i by constants in C . Given a set of tuples x , its probability is defined by:

$$P(X = x) := \frac{1}{Z} \exp\left(\sum_i w_i n_i(x)\right),$$

where Z is a normalization factor, \exp is the exponential function, w_i is the weight for rule i , and $n_i(x)$ is the number of formulae satisfied by x in the grounding of F_i .

We solve an MLN by performing a probabilistic inference computation that finds the most likely output set of tuples.

3.2 A Framework to Combine Logic and Probability

We illustrate our framework for integrating probabilistic reasoning into existing analyses by means of the application of incorporating user feedback in static bug detection. Specifically, we illustrate our approach using the example of applying the static race detection tool Chord [24] to a real-world multi-threaded Java program, Apache FTP server [1].

Figure 5 shows a code fragment from the program. The `RequestHandler` class is used to handle client connections and 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, while the `getRequest()` method is used to access the `m_request` field. Both these methods can be invoked from various components of the program (not shown), and thus can be simultaneously executed by multiple threads in parallel on the

```

1 package org.apache.ftpserver;
2 public class RequestHandler {
3     Socket m_controlSocket;
4     FtpRequestImpl m_request;
5     FtpWriter m_writer;
6     BufferedReader m_reader;
7     boolean m_isConnectionClosed;
8     public FtpRequest getRequest() {
9         return m_request;
10    }
11    public void close() {
12        synchronized(this) {
13            if (m_isConnectionClosed)
14                return;
15            m_isConnectionClosed = true;
16        }
17        m_request.clear(); // x1
18        m_request = null; // x2
19        m_writer.close(); // y1
20        m_writer = null; // y2
21        m_reader.close();
22        m_reader = null;
23        m_controlSocket.close();
24        m_controlSocket = null;
25    }
26 }

```

Figure 5: Java code snippet of Apache FTP server.

Analysis Input Relations:

$\text{next}(p_1, p_2)$	(program point p_1 is immediate successor of program point p_2)
$\text{guarded}(p_1, p_2)$	(at least one common lock guards program points p_1 and p_2)
$\text{mayAlias}(p_1, p_2)$	(instructions at program points p_1 and p_2 may access the same memory location, and constitute a possible datarace)

Analysis Output Relations:

$\text{parallel}(p_1, p_2)$	(different threads may reach program points p_1 and p_2 in parallel)
$\text{race}(p_1, p_2)$	(datarace may occur between different threads while executing the instructions at program points p_1 and p_2)

Analysis Rules:

$$\text{parallel}(p_1, p_2) \wedge \text{next}(p_3, p_1) \implies \text{parallel}(p_3, p_2) \quad (1)$$











$$\text{parallel}(p_1, p_2) \implies \text{parallel}(p_2, p_1) \quad (2)$$

$$\left(\begin{array}{c} \text{parallel}(p_1, p_2) \quad \wedge \\ \text{mayAlias}(p_1, p_2) \quad \wedge \\ \neg \text{guarded}(p_1, p_2) \end{array} \right) \implies \text{race}(p_1, p_2) \quad (3)$$



Figure 6: Simplified race detection analysis.

same `RequestHandler` object. To ensure that this parallel execution does not result in any dataraces, the `close()` method uses a boolean flag `m_isConnectionClosed`. If this flag is set, all calls to `close()` return without any further updates. If the flag is not set, then it is first updated to true, followed by execution of the clean-up code (lines 17–24). To avoid dataraces on the flag itself, it is read and updated while holding a lock on the `RequestHandler` object (lines 12–16). All the subsequent code in `close()` is free from dataraces since only the first call to `close()` executes this section. However, note that an actual datarace still exists between the two accesses to field `m_request` on line 9 and line 18.

Logical Datarace Analysis. To scale to large real-world programs, Chord employs a context-sensitive but path-insensitive datarace analysis. This is a common design choice for balancing precision and scalability of static analyses. The analysis in Chord is expressed using Datalog [5], a declarative logic pro-

Detected Races	
R1: Race on field <code>org.apache.ftpserver.RequestHandler.m_request</code>  	
<code>org.apache.ftpserver.RequestHandler: 9</code>	<code>org.apache.ftpserver.RequestHandler: 18</code>
R2: Race on field <code>org.apache.ftpserver.RequestHandler.m_request</code>  	
<code>org.apache.ftpserver.RequestHandler: 17</code>	<code>org.apache.ftpserver.RequestHandler: 18</code>
R3: Race on field <code>org.apache.ftpserver.RequestHandler.m_writer</code>  	
<code>org.apache.ftpserver.RequestHandler: 19</code>	<code>org.apache.ftpserver.RequestHandler: 20</code>
R4: Race on field <code>org.apache.ftpserver.RequestHandler.m_reader</code>  	
<code>org.apache.ftpserver.RequestHandler: 21</code>	<code>org.apache.ftpserver.RequestHandler: 22</code>
R5: Race on field <code>org.apache.ftpserver.RequestHandler.m_controlSocket</code>  	
<code>org.apache.ftpserver.RequestHandler: 23</code>	<code>org.apache.ftpserver.RequestHandler: 24</code>
Eliminated Races	
E1: Race on field <code>org.apache.ftpserver.RequestHandler.m_isConnectionClosed</code>	
<code>org.apache.ftpserver.RequestHandler: 13</code>	<code>org.apache.ftpserver.RequestHandler: 15</code>

(a) Before feedback.

Detected Races	
R1: Race on field <code>org.apache.ftpserver.RequestHandler.m_request</code>  	
<code>org.apache.ftpserver.RequestHandler: 9</code>	<code>org.apache.ftpserver.RequestHandler: 18</code>
Eliminated Races	
E1: Race on field <code>org.apache.ftpserver.RequestHandler.m_isConnectionClosed</code>	
<code>org.apache.ftpserver.RequestHandler: 13</code>	<code>org.apache.ftpserver.RequestHandler: 15</code>
E2: Race on field <code>org.apache.ftpserver.RequestHandler.m_request</code>	
<code>org.apache.ftpserver.RequestHandler: 17</code>	<code>org.apache.ftpserver.RequestHandler: 18</code>
E3: Race on field <code>org.apache.ftpserver.RequestHandler.m_writer</code>	
<code>org.apache.ftpserver.RequestHandler: 19</code>	<code>org.apache.ftpserver.RequestHandler: 20</code>
E4: Race on field <code>org.apache.ftpserver.RequestHandler.m_reader</code>	
<code>org.apache.ftpserver.RequestHandler: 21</code>	<code>org.apache.ftpserver.RequestHandler: 22</code>
E5: Race on field <code>org.apache.ftpserver.RequestHandler.m_controlSocket</code>	
<code>org.apache.ftpserver.RequestHandler: 23</code>	<code>org.apache.ftpserver.RequestHandler: 24</code>

(b) After feedback.

Figure 7: Race reports produced for Apache FTP server. Each report specifies the field involved in the race, and line numbers of the program points with the racing accesses. The user feedback is to “dislike” report R2.

programming language widely used for specifying program analyses [4, 16, 41, 44, 46]. Figure 6 shows a simplified subset of the logical inference rules used by Chord. These rules are used to produce output relations from input relations, where the input relations express known program facts and output relations express the analysis outcome. These rules express the idioms that the analysis writer deems to be the most important for capturing dataraces in Java programs. For example, Rule (1) in Figure 6 conveys that if a pair of program points (p_1, p_2) can execute in parallel, and if program point p_3 is an immediate successor of p_1 , then (p_3, p_2) are also likely to happen in parallel. Rule (2) conveys that the parallel relation is commutative. Rule (3) expresses the idiom that only program points not guarded by a common lock can be potentially racing. In particular, if program points (p_1, p_2) can happen in parallel, can access the same memory location, and are not guarded by any common lock, then there is a potential datarace between p_1 and p_2 .

Figure 7 (a) shows the bug reports produced by applying the above analysis on the example program. Chord successfully captures the real datarace between line 9 and line 18 on field `m_request` (denoted by R1). However, it also reports another four false alarms (denoted by R2–R5). To avoid producing these false alarms, Chord needs to precisely track the control dependencies between threads. However, to scale to real-world programs including the Apache FTP sever, Chord is designed to be path-insensitive and therefore overapproximates the control-flow dependencies between threads. As a result, Chord concludes that the clean-up code (line 17–24) can be executed by different threads simultaneously, which in turn leads to the four aforementioned false alarms.

Conventionally, the interaction between the analysis and the analysis user stops after the reports are produced and the user has to inspect each report manually. In the current example, the user can quickly get frustrated due to the high false positive rate (80%).

Combining Logic with Probability. To address the above conundrum, our approach enables the analysis to consider the user’s feedback as the user inspects each report and update the reports accordingly. Currently, our approach considers feedback in the form of binary likes and dislikes. Suppose the user inspects R2 which is de-

noted by the output tuple $\text{race}(x_1, x_2)$, and concludes that it is a false alarm. By considering this negative feedback on R2, our approach will eliminate the other three false alarms (R3–R5) that are derived for similar reasons. Figure 7 (b) shows the reports after incorporating the feedback. We next illustrate how we can achieve this effect by combining logical and probabilistic reasoning.

Since all the rules in the logical analysis are rigid, directly adding the negative feedback on R2 as a rule $\neg \text{race}(x_1, x_2)$ will make the logical system unsatisfiable. Ideally, we wish to violate certain rule instances that introduce imprecision, and thus guide the analysis to produce results that are more desirable to the user.

Towards this end, we attach a weight to each imprecise rule and convert it from a conventional hard logical rule into a soft probabilistic rule. This in turn converts the analysis specified in Datalog into an analysis specified in MLN, which combines logical and probabilistic reasoning. Intuitively, the weight for each rule represents the confidence of the analysis writer in each rule: the higher the weight is, the less likely the rule will introduce false alarms. Such weights can be either manually specified by the analysis writer or automatically trained using programs whose reports are fully labeled. Finally, the rules considered to be precise remain as hard logical rules.

The MLN analysis defines a distribution of possible sets of bug reports rather than a single set of bug reports, and the most likely set constitutes the final bug reports. Moreover, it allows us to incorporate user feedback as a new rule to the system, which can change the output distribution and the most likely set. Since the user can make mistakes, however, we also add the user feedback as a soft rule to the system: its weight represents the user’s confidence in it and can be also trained from labeled data. Intuitively, the bug reports produced after feedback are the ones that the analysis writer and the analysis user most likely agree upon.

We next discuss how our approach can generalize the effect of user feedback on a single bug report to multiple bug reports. Our key insight is that most false alarms are symptoms of a few common root causes. For example, all false alarms in the example are derived because Chord falsely concludes that the clean-up code (line 17–24) can be executed by multiple threads simultaneously. If we can rectify these few sources of imprecision, we can significantly

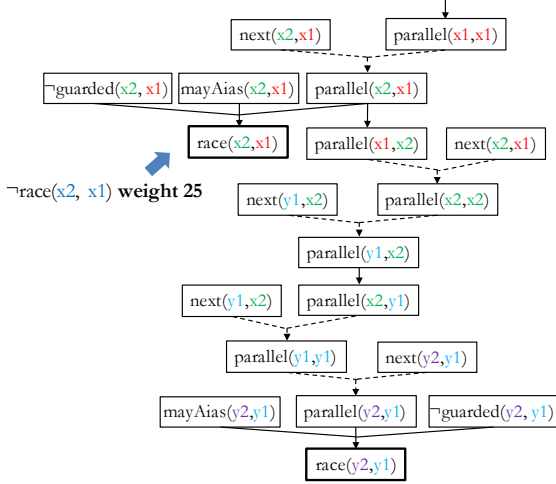


Figure 8: Derivation graph of race analysis and a user feedback.

improve the accuracy of the analysis. We use the example analysis to explain how our approach achieves this effect below.

In the example datarace analysis, we convert Rule (1) into a soft rule as it considers neither conditional statements nor synchronization operations. On the other hand, the other two rules remain as hard rules as they rarely introduce imprecision. This leads to the analysis specification below:

$$\text{parallel}(p_1, p_2) \wedge \text{next}(p_3, p_1) \implies \text{parallel}(p_3, p_2) \quad \text{weight 5 (1)}$$

$$\text{parallel}(p_1, p_2) \implies \text{parallel}(p_2, p_1) \quad (2)$$

$$\left(\begin{array}{c} \text{parallel}(p_1, p_2) \\ \text{mayAlias}(p_1, p_2) \\ \neg \text{guarded}(p_1, p_2) \end{array} \wedge \right) \implies \text{race}(p_1, p_2) \quad (3)$$

where the weight of Rule (1) is learnt from training programs whose reports are fully labeled.

By running the MLN analysis on the example program, we obtain the same set of bug reports as running the original Datalog analysis. The execution of the analysis can be visualized via a derivation graph, part of which is shown in Figure 8. In the graph, each node represents either an input tuple or a derived output tuple, while each hyperedge represents a rule instance: the dotted edges represent soft rule instances, and the solid edges represent hard rule instances.

Suppose the user inspects R2 (denoted by $\text{race}(x_2, x_1)$) and gives a negative feedback on it. This feedback is translated into a soft rule $\neg \text{race}(x_2, x_1)$ **weight 25** where the weight is learnt offline from fully labeled data. We add the feedback rule to the MLN analysis and it forms a conflict with the following two rule instances:

$$\text{parallel}(x_1, x_1) \wedge \text{next}(x_2, x_1) \implies \text{parallel}(x_2, x_1) \quad \text{weight 5 (a)}$$

$$\left(\begin{array}{c} \text{parallel}(x_2, x_1) \\ \text{mayAlias}(x_2, x_1) \\ \neg \text{guarded}(x_2, x_1) \end{array} \wedge \right) \implies \text{race}(x_2, x_1) \quad (b)$$

Since rule instance (b) is hard and rule instance (a) has a lower weight compared to the feedback rule, the most likely output is a set of tuples that violate rule instance (a) but satisfy rule instance (b) and the feedback rule in order to maximize the corresponding probability. This in turn prevents deriving $\text{parallel}(x_2, x_1)$ and $\text{race}(x_2, x_1)$. At the same time, we also implicitly enforce the least fixpoint semantics of the original Datalog analysis¹. As a result,

¹This can be achieved by either adding additional soft rules or by ensuring that the underlying inference engine always returns a minimal model.

all the other tuples that were derived using $\text{parallel}(x_2, x_1)$ are eliminated, which includes the other three false alarms (R3–R5).

3.3 Learning and Inference

To support the aforementioned framework, we developed an engine to solve learning and inference problems for MLNs.

Learning. We use a standard algorithm [40] to automatically learn weights from training data which consists of a set of input tuples and a set of expected output tuples. The initial weight of every rule is computed as the log of the ratio of the number of its instances satisfied by the expected output tuples over the number of its instances violated by the expected output tuples. Then the algorithm iteratively updates weights as follows: first, it invokes the inference algorithm to produce the most likely output under the current weights; then, it increases (decreases) the weight of any rule that has more (less) violations in the current output than it has in the expected output, so that the output produced in next iteration is more likely to match the expected output. The algorithm terminates when the difference between the produced output and the expected output is within a predefined bound or it reaches a given number of iterations. Since the learning algorithm is built upon the inference algorithm, its scalability and accuracy is largely dictated by the inference engine, which we discuss next.

Inference. The AI community has developed several solvers for the inference problem [11, 25, 26, 35]. However, none of them suffices for our applications. First, the solutions produced by most of these solvers do not necessarily satisfy hard rules in the problem formulation, as typically there are no hard rules in AI problems. On the other hand, hard rules are essential for program analysis applications to enforce formal guarantees such as soundness. Secondly, none of these solvers scale to the instances produced by such applications on real-world programs as they fail to exploit domain knowledge in program analysis. To address this challenge, we have developed a scalable and accurate inference engine that exploits various insights in program analysis.

Our inference algorithm is divided into two phases: *grounding* and *solving*. In the grounding phase, the MLN inference problem is reduced to a Weighted Partial Maximum Satisfiability (WPMS) problem by instantiating all the quantified variables with all constants. In the solving phase, the WPMS problem is solved by an off-the-shelf WPMS solver. Both phases are challenging to scale: for grounding, naively instantiating all quantified variables with all constants can lead to an intractable WPMS formula (comprising upto 10^{30} clauses); for solving, the WPMS problem itself is also a combinatorial optimization problem, which is known for its intractability. We next describe how we address this challenge by exploiting various domain insights.

Our first insight is *sparsity*, the observation that the solutions to most inference problems only contain a small subset of the universe of tuples. Based on sparsity, we have developed an iterative lazy grounding algorithm [19]. It starts by constructing an empty WPMS formula. Then in each iteration, it solves the current WPMS formula and expands the formula by adding clauses that are violated by the current solution. The algorithm terminates when the objective of the WPMS formula no longer improves. By exploiting sparsity, we effectively reduce the size of the grounded formula.

The running time of the above algorithm can be impaired due to an excessive number of iterations. To reduce the number of iterations, we start with a non-empty WPMS formula by exploiting the structure of logical formulae in program analyses [19]. In particular, we leverage the observation that most logical rules in program analyses are Horn clauses.

To further speed up the solving phase, we observe that the above grounding algorithm is solving a sequence of similar WPMS prob-

lems, where each problem is a subset of its immediate successor. By exploiting this insight, we have developed incremental solving which speeds up WMPS solving by reusing past results [39].

Finally, we observe that often only a small fraction of output tuples are of interest, which we refer to as query tuples. Such tuples typically concern the program analysis output (e.g., tuples representing bug reports). Intuitively, if we only care about these few tuples, it is unlikely that we need to reason about the whole problem. This *locality* hypothesis is akin to that exploited by query-driven program analyses. Based on this insight, we have developed an effective algorithm that lazily resolves a given set of query tuples [48].

4. Related Work

Our approach is related to data-driven approaches to program analysis, statistical bug ranking, and statistic-learning-based approaches to program synthesis.

Data-Driven Program Analysis. Data-driven approaches have been adopted to learn various knobs of existing program analyses from different sources of data. These approaches can be broadly classified into two categories: learning program invariants from program executions [6, 27, 30, 37, 38] and learning tuning parameters from past analysis runs [7, 8, 28, 29]. Such tuning parameters balance various tradeoffs in the analysis such as the tradeoff between precision and scalability [7, 29] and the tradeoff between soundness and completeness [8]. Similar to our approach, they also combine both probabilistic and logical reasoning, but in a different manner. While in these approaches, probabilistic methods are used as a pre-processor for the logical analysis, in our approach, probabilistic and logical reasoning are combined seamlessly using a unified language.

Statistical Bug Ranking. Statistical error ranking techniques [9, 13, 14] employ statistical methods and heuristics to rank errors reported by an underlying static analysis. Similar to data-driven program analysis, these techniques combine logical and probabilistic reasoning in a superficial manner. But probabilistic methods are used as a post-processor for the logical analysis rather than as a pre-processor.

Program Synthesis via Statistical Learning. Program synthesis is a class of techniques that aims to automate programming. Conventional synthesis techniques have mainly employed logical approaches [20, 31, 42]. Recently, probabilistic approaches have been proposed for various synthesis tasks, including code completion [32], code deobfuscation [3], variable naming [33], and program repair [15]. Combining logical and probabilistic reasoning in program synthesis is an interesting future direction. For instance, probabilistic reasoning can help logical approaches to handle noisy data while logical reasoning can help prune the search space of probabilistic approaches.

5. Future Challenges

We identify a set of key challenges to building applications that leverage combined logical and probabilistic reasoning.

Languages. Markov Logic Networks enable the aforementioned applications but possess certain limitations. First, a more fine-grained way to integrate probabilities into logical formulae stands to further improve the performance of these applications and enable new applications. Specifically, in an MLN, given a pair (F_i, w_i) , all instances in the grounding of the logical formula F_i are assigned the same weight w_i . However, in practice, one can envision scenarios for assigning separate weights to individual instances. For instance, whether a given analysis rule holds is closely related to

the context of the code fragment that it is applied to. Secondly, MLNs lack built-in support for the least fixpoint operator, which is used prevalently in abstract-interpretation-based program analyses. In our current applications, we mitigate this issue by adding additional soft constraints to enforce the least fixpoint semantics. However, a more fundamental and elegant solution may be conceivable by refining the language design.

Guarantees. We need statistical guarantees to reason about the correctness of the new combined approach since conventional logical guarantees such as soundness are not expressive enough or applicable to certain applications. For instance, our static bug detection application can introduce false negatives after incorporating user feedback, and no longer guarantees soundness. By applying statistical guarantees such as *precision* and *recall*, we can effectively quantify the false positive and false negative rates. One possible direction to enforce such guarantees is to leverage the literature of probably approximately correct learning (PAC learning) [43].

Learning. Until now, we have assumed that the logical formulae come from existing logical analyses and the weights are the only learnt parameters. However, there are cases that could benefit from learning the logical formulae as well. For instance, we may lack specifications for certain program properties (e.g., security vulnerabilities). In this case, both the logical part and the probabilistic part of the analysis specification could be obtained by learning from labeled data. There are also cases where the specification may exist but is too imprecise such that simply making the rules probabilistic does not suffice to improve the performance unless new rules are introduced. One possible direction to enable such learning is to leverage the literature of inductive logic programming [22] and program synthesis [2].

Inference. The inference engine is the key component that affects the scalability and accuracy of our approach as even the learning problem is solved by solving a series of inference problems. However, the inference problem is a combinatorial optimization problem, which is known for its intractability. While the general problem is computationally challenging in theory, it is feasible to build an inference engine that is scalable and accurate in practice by exploiting various domain insights. Currently, by leveraging the insights described in Section 3.2, we are able to scale the learning and inference algorithms to analyses of programs comprising upto half a million lines of code without sacrificing accuracy. In the future, we plan to leverage other domain insights such as modularity in programs to further improve scalability.

6. Conclusion

We presented a novel approach to program analysis that combines logical and probabilistic reasoning. While the logical part preserves the benefits of conventional analyses, the probabilistic part enables handling uncertainty and provides the ability to adapt and learn. Then we described an end-to-end system for supporting the approach, which includes a declarative language for combining logic and probability, a general framework for incorporating probability in existing program analyses using this language, and an effective learning and inference engine for the language. We believe such a combined approach will help address long-standing open problems in program analysis and enable new applications.

Acknowledgments

We thank the referees for useful feedback. This work was supported by DARPA under agreement #FA8750-15-2-0009, NSF awards #1253867 and #1526270, and a Facebook Fellowship. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright thereon.

References

- [1] Apache FTP Server. <http://mina.apache.org/ftpserver-project/>.
- [2] R. Alur, R. Bodík, G. Juniwal, M. Martin, M. Raghothaman, S. Seshia, R. Singh, A. Solar-Lezama, E. Torlak, and A. Udupa. Syntax-guided synthesis. In *FMCAD*, 2013.
- [3] B. Bichsel, V. Raychev, P. Tsankov, and M. T. Vechev. Statistical deobfuscation of android applications. In *CCS*, 2016.
- [4] M. Bravenboer and Y. Smaragdakis. Strictly declarative specification of sophisticated points-to analyses. In *OOPSLA*, 2009.
- [5] S. Ceri, G. Gottlob, and L. Tanca. What you always wanted to know about datalog (and never dared to ask). *IEEE Trans. Knowl. Data Eng.*, 1989.
- [6] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Trans. Software Eng.*, 2001.
- [7] R. Grigore and H. Yang. Abstraction refinement guided by a learnt probabilistic model. In *POPL*, 2016.
- [8] K. Heo, H. Oh, and K. Yi. Machine-learning-guided selectively unsound static analysis. In *ICSE*, 2017.
- [9] Y. Jung, J. Kim, J. Shin, and K. Yi. Taming false alarms from a domain-unaware C analyzer by a bayesian statistical post analysis. In *SAS*, 2005.
- [10] A. Kimmig, S. Bach, M. Broecheler, B. Huang, and L. Getoor. A short introduction to probabilistic soft logic. In *Proceedings of the NIPS Workshop on Probabilistic Programming: Foundations and Applications*, 2012.
- [11] S. Kok, M. Sumner, M. Richardson, P. Singla, H. Poon, D. Lowd, and P. Domingos. The Alchemy system for statistical relational AI. Technical report, University of Washington, Seattle, WA, 2007.
- [12] D. Koller. Probabilistic relational models. In *ILP*, 1999.
- [13] T. Kremenek and D. Engler. Z-ranking: Using statistical analysis to counter the impact of static analysis approximations. In *SAS*, 2003.
- [14] T. Kremenek, K. Ashcraft, J. Yang, and D. Engler. Correlation exploitation in error ranking. In *FSE*, 2004.
- [15] F. Long and M. Rinard. Automatic patch generation by learning correct code. In *POPL*, 2016.
- [16] M. Madsen, M. Yee, and O. Lhoták. From Datalog to Flix: a declarative language for fixed points on lattices. In *PLDI*, 2016.
- [17] R. Mangal, X. Zhang, A. V. Nori, and M. Naik. A user-guided approach to program analysis. In *FSE*, 2015.
- [18] R. Mangal, X. Zhang, A. V. Nori, and M. Naik. Volt: A lazy grounding framework for solving very large maxsat instances. In *SAT*, 2015.
- [19] R. Mangal, X. Zhang, A. Kamath, A. V. Nori, and M. Naik. Scaling relational inference using proofs and refutations. In *AAAI*, 2016.
- [20] Z. Manna and R. J. Waldinger. A deductive approach to program synthesis. *ACM Trans. Program. Lang. Syst.*, 1980.
- [21] B. Milch, B. Marthi, S. J. Russell, D. Sontag, D. L. Ong, and A. Kolobov. BLOG: probabilistic models with unknown objects. In *IJCAI*, 2005.
- [22] S. Muggleton and L. D. Raedt. Inductive logic programming: Theory and methods. *J. Log. Program.*, 1994.
- [23] S. Muggleton et al. Stochastic logic programs. *Advances in inductive logic programming*, 1996.
- [24] M. Naik, A. Aiken, and J. Whaley. Effective static race detection for Java. In *PLDI*, 2006.
- [25] F. Niu, C. Ré, A. Doan, and J. W. Shavlik. Tuffy: Scaling up statistical inference in markov logic networks using an RDBMS. In *VLDB*, 2011.
- [26] J. Noessner, M. Niepert, and H. Stuckenschmidt. RockIt: Exploiting parallelism and symmetry for MAP inference in statistical relational models. In *AAAI*, 2013.
- [27] A. V. Nori and R. Sharma. Termination proofs from tests. In *FSE*, 2013.
- [28] H. Oh, H. Yang, and K. Yi. Learning a strategy for adapting a program analysis via bayesian optimisation. In *OOPSLA*, 2015.
- [29] H. Oh, W. Lee, K. Heo, H. Yang, and K. Yi. Selective X-sensitive analysis guided by impact pre-analysis. *ACM TOPLAS*, 2016.
- [30] S. Padhi, R. Sharma, and T. Millstein. Data-driven precondition inference with learned features. In *PLDI*, 2016.
- [31] A. Pettorossi and M. Proietti. Synthesis and transformation of logic programs using unfold/fold proofs. In *The Journal of Logic Programming*, 1999.
- [32] V. Raychev, M. Vechev, and A. Krause. Code completion with statistical language models. In *PLDI*, 2014.
- [33] V. Raychev, M. Vechev, and A. Krause. Predicting program properties from “Big Code”. In *POPL*, 2015.
- [34] M. Richardson and P. Domingos. Markov logic networks. *Machine Learning*, 62(1-2), 2006.
- [35] S. Riedel. Cutting plane map inference for markov logic. In *Intl. Workshop on Statistical Relational Learning*, 2009.
- [36] S. J. Russell. Unifying logic and probability. *Commun. ACM*, 2015.
- [37] R. Sharma, S. Gupta, B. Hariharan, A. Aiken, P. Liang, and A. V. Nori. A data driven approach for algebraic loop invariants. In *ESOP*, 2013.
- [38] R. Sharma, E. Schkufza, B. Churchill, and A. Aiken. Data-driven equivalence checking. In *OOPSLA*, 2013.
- [39] X. Si, X. Zhang, V. Manquinho, M. Janota, A. Ignatiev, and M. Naik. On incremental core-guided maxsat solving. In *CP*, 2016.
- [40] P. Singla and P. Domingos. Discriminative training of markov logic networks. In *AAAI*, 2005.
- [41] Y. Smaragdakis and M. Bravenboer. Using Datalog for fast and easy program analysis. In *Datalog 2.0 Workshop*, 2010.
- [42] A. Solar-Lezama, R. M. Rabbah, R. Bodík, and K. Ebcioğlu. Programming by sketching for bit-streaming programs. In *PLDI*, 2005.
- [43] L. Valiant. A theory of the learnable. *Commun. ACM*, 1984.
- [44] J. Whaley, D. Avots, M. Carbin, and M. Lam. Using Datalog with binary decision diagrams for program analysis. In *APLAS*, 2005.
- [45] X. Zhang, M. Naik, and H. Yang. Finding optimum abstractions in parametric dataflow analysis. In *PLDI*, 2013.
- [46] X. Zhang, R. Mangal, R. Grigore, M. Naik, and H. Yang. On abstraction refinement for program analyses in Datalog. In *PLDI*, 2014.
- [47] X. Zhang, R. Grigore, X. Si, and M. Naik. Effective interactive resolution of static analysis alarms. 2016.
- [48] X. Zhang, R. Mangal, A. V. Nori, and M. Naik. Query-guided maximum satisfiability. In *POPL*, 2016.