COMBINING LOGICAL AND PROBABILISTIC REASONING IN PROGRAM ANALYSIS

A Dissertation
Presented to
The Academic Faculty

By

Xin Zhang

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
School of Computer Science

Georgia Institute of Technology

December 2017

Copyright © Xin Zhang 2017

COMBINING LOGICAL AND PROBABILISTIC REASONING IN PROGRAM ANALYSIS

Approved by:

Dr. Mayur Naik, Advisor
Department of Computer and Information Science
University of Pennsylvania

Dr. Hongseok Yang
Department of Computer Science
Oxford University

Dr. Santosh Pande
School of Computer Science
Georgia Institute of Technology

Dr. William Harris
School of Computer Science
Georgia Institute of Technology

Dr. Aditya Nori
Microsoft Research

Date Approved: August 23, 2017

Knowledge comes, but wisdom lingers.

Alfred, Lord Tennyson

To my parents.

ACKNOWLEDGEMENTS

I am forever in debt to my advisor, Mayur Naik, for his support and guidance throughout my Ph.D. study. It is Mayur who brought me to the wonderful world of research. As his first Ph.D. student, I have received enormous amount of attention from Mayur that other students could only dream of. From topic selection to problem solving, formalization to empirical evaluation, writing to presentation, he has coached me heavily in every aspect that is needed to be a researcher. Mayur's passion and high standards about research will always inspire me to be a better researcher.

Besides Mayur, I was fortunate to be mentored by Hongseok Yang and Aditya Nori. I worked with Hongseok closely in the first half of my Ph.D. study, and learnt the most about programming language theories from him. It always amazes me how Hongseok can draw principles and insights from raw and seemingly hacky ideas. I worked with Aditya closely in the second half of my Ph.D. study and benefited greatly from his crisp feedback. Although we only met for one hour a week, this one hour was often the one hour when I learnt the most of the whole week.

I would like to thank all my collaborators, but especially Radu Grigore, Ravi Mangal, and Xujie Si. They have helped greatly in projects that eventually led to this thesis. I also had a great fun time working with them. I am also grateful to the rest of my collaborators, including Sulekha Kulkarni, Aditya Kamath, Jongse Park, Hadi Esmaeilzadeh, Vasco Manquinho, Mikolas Janota, and Alexey Ignatiev.

Bill Harris and Santosh Pande were kind enough to serve on my thesis committee. They and other folks at Georgia Tech have made GT a wonderful place for graduate study.

Last but not least, I would thank my parents for the unconditional love and support I have received ever since I can remember. Throughout my life, I have always been encouraged by them to pursue things that I am passionate about. When facing challenges in life, I always gain strength knowing that they will be there for me.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SUMMARY

Software is becoming increasingly pervasive and complex. These trends expose masses of users to unintended software failures and deliberate cyber-attacks. A widely adopted solution to enforce software quality is automated program analysis. Existing program analyses are expressed in the form of logical rules that are handcrafted by experts. While such a logic-based approach provides many benefits, it cannot handle uncertainty and lacks 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. The framework consists of a frontend, which automatically integrates probabilities into a logical analysis by synthesizing a system of weighted constraints, and a backend, which is a learning and inference engine for such constraints. 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. We also describe new algorithmic techniques to solve very large instances of weighted constraints that arise not only in our domain but also in other domains such as Big Data analytics and statistical AI.

CHAPTER 1
INTRODUCTION

Software is becoming increasingly pervasive and complex. A pacemaker comprises a hundred thousand lines of code; a mobile application can comprise a few million lines of code; and all the programs in an automobile together consist of up to 100 million lines of code. While programs have become an indispensable part of our daily life, their growing complexity poses a dire challenge for the software industry to build software artifacts that are correct, efficient, secure, and reliable. In particular, traditional methods for ensuring software quality (e.g., testing and code review) require a considerable amount of manual effort and therefore struggle to scale with such increasing software complexity.

One promising solution for enhancing software quality is automated program analysis. Program analyses are algorithms that discover a wide range of useful facts about programs, including proofs, bugs, and specifications. They have achieved remarkable success in different application domains: SLAM [1] is a program analysis tool based on model checking that has been applied widely for verifying safety properties of Windows device drivers; ASTREÉ [2], which is based on abstract interpretation, has been applied to Airbus flight controller software to prove the absence of runtime errors; Coverity [3], which is based on data-flow analysis, has found many bugs in real-world enterprise applications; and more recently, Facebook developed Infer [3], which is based on separation logic, to improve memory safety of mobile applications.

Although the techniques underlying these tools vary, their algorithms are all expressed in the form of logical axioms and inference rules that are handcrafted by experts. This logic-based approach provides many 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 abilities 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 these representations cannot be changed once they are deployed to end-users. However, 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 is therefore prone to repeating past mistakes.

To overcome the drawbacks of the existing logical approach, this thesis proposes 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.

In the rest of this thesis, we demonstrate how such a combined approach can improve the state-of-the-art of important program analysis applications, describe a general recipe for incorporating probabilistic reasoning into existing program analyses that are based on logical reasoning, and present a system for supporting this recipe.

1.1 Motivating Applications

We present an informal overview of 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

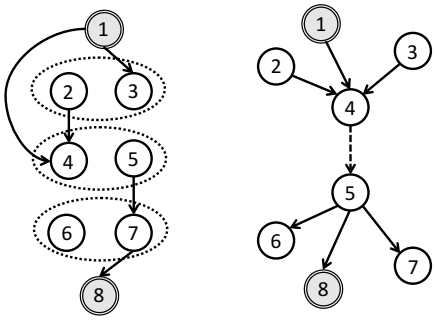


Figure 1.1: Graphs depicting how different applications of our approach enable a program analysis to avoid reporting false information flow from node 1 to node 8 in two programs.

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 1.1. We elide presenting details of this analysis that are not relevant. Chapter 3 discusses each application in more detail.

1.1.1 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 1.1(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, implies 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 that is based on the popular CEGAR (counter-example guided abstraction refinement) technique [4], 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.

1.1.2 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 1.1(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 encoding 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).

1.1.3 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 1.1(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 $\{6, 7, 8\}$.

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)

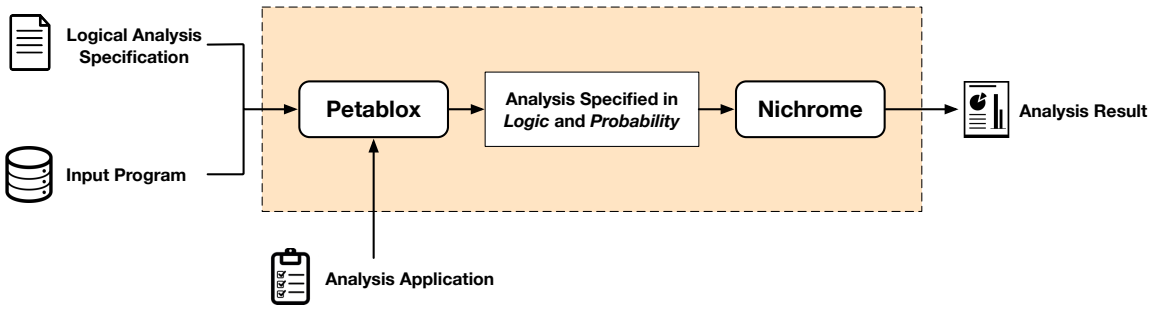


Figure 1.2: Architecture of our system for incorporating probabilistic reasoning in a logical program analysis.

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).

1.2 System Architecture

We need to address two central technical challenges in order to effectively combine logical and probabilistic reasoning in program analysis and thereby enable the three aforementioned applications as well as other emerging applications:

C1. How can we incorporate probabilistic reasoning into an existing logical analysis without requiring excessive effort from the analysis designer? While the probabilistic part provides more expressiveness, the analysis designer is challenged with new design decisions. Specifically, it can be a delicate task to combine logic and probability in a meaningful manner and set suitable parameters for the probabilistic part. Ideally, we should provide an automated way to incorporate probabilistic reasoning

into a conventional logical analysis and thereby avoid changing the analysis design process significantly.

C2. How can we scale the combined analysis to real-world programs? The improvement in expressiveness of the combined analysis comes at the cost of increased complexity. While the conventional logical approach typically requires solving a decision problem, the new combined approach now requires solving an optimization problem or a counting problem. Since the latter is often computationally harder than its counterpart of the former (e.g., in propositional logic, maximum satisfiability and model counting vs. satisfiability), we need novel algorithms to scale the combined approach to large real-world programs.

We address the above two challenges by proposing a system whose high-level architecture is depicted in Figure 1.2. Our system takes a logical program analysis specification and a program as inputs, and outputs program facts that the input analysis intends to discover (e.g., bugs, proofs, or specifications). Besides the two inputs, it is parameterized by the analysis application. The system consists of a frontend and a backend, which address the above two challenges respectively. We describe them briefly below, while Chapter 3 and Chapter 4 include more detailed discussions of both ends respectively.

The frontend, PETABLOX, addresses the first challenge (**C1**) for program analyses specified declaratively in a constraint language. We target constraint-based program analyses for two reasons: first, it allows PETABLOX to leverage many existing benefits of the constraint-based approach [5]; second, it enables PETABLOX to provide a general and automatic mechanism for combining logic and probability by analyzing the analysis constraints. Specifically, PETABLOX requires the input analysis to be specified in Datalog [6], a declarative logic programming language that is widely popular for formulating program analyses [7, 8, 9, 10, 11]. By analyzing the input analysis and program, PETABLOX automatically synthesizes a novel program analysis instance that combines logical and probabilistic reasoning via a system of weighted constraints. This system of weighted constraints

is specified via a Markov Logic Network [12], a declarative language for combining logic and probability from the Artificial Intelligence community. Based on the specified application, PETABLOX formulates the analysis instance differently.

The backend, NICHROME, which is a learning and inference engine for Markov Logic Networks, then solves the synthesized analysis instance and produces the final analysis results. We address the second challenge **(C2)** in NICHROME by applying novel algorithms that exploit domain insights in program analysis. These algorithms enable NICHROME to solve Markov Logic Network instances generated from real-world analyses and programs in a sound, accurate, and scalable manner.

1.3 Thesis Contributions

We summarize the contributions of this thesis below:

1. We propose a new paradigm to program analysis that augments the conventional logic-based approach with probability, which we envision will benefit and enable traditional and emerging applications.

2. We describe a general recipe to incorporate probabilistic reasoning in a conventional logical program analysis by converting a Datalog analysis into a novel analysis instance specified in Markov Logic Networks, a declarative language for combining logic and probability.

3. We present an effective system that implements the proposed paradigm and recipe, which includes a frontend that automates the conversion from Datalog analyses to Markov Logic Networks, and a backend that is an effective learning and inference engine for Markov Logic Networks.

4. We show empirically that the proposed approach significantly improves the effectiveness of program analyses for three important applications: automatic verification, interactive verification, and static bug detection.

1.4 Thesis Organization

The rest of this thesis is organized as follows: Chapter 2 describes the necessary background knowledge, which includes the notions of constraint-based program analysis, Datalog, and Markov Logic Network; Chapter 3 describes how PETABLOX incorporates probability in existing conventional program analyses for emerging applications; Chapter 4 presents novel algorithms applied in NICHROME, which allows solving combined analysis instances in an efficient and accurate manner; Chapter 5 discusses future research directions; finally, Chapter 6 concludes the thesis. We include the proofs to most propositions, lemmas, and theorems in Appendix A except for the ones in Chapter 4.2 as these proofs themselves are among the main contributions of that section.

CHAPTER 2
BACKGROUND

This section describes the concept of constraint-based program analysis, the syntax and semantics of Datalog, and the syntax and semantics of Markov Logic Networks.

2.1 Constraint-Based Program Analysis

Designing a program analysis that works in practice is a challenging task. In theory, any nontrivial analysis problem is undecidable in general [13]; in practice, however, there are concerns related to scalability, imprecisely defined specifications, missing program parts, etc. Due to these constraints, program analysis designers have to make various approximations and assumptions that balance competing aspects like scalability, accuracy, and user effort. As a result, there are various approaches to program analysis, each with their own advantages and drawbacks.

One popular approach is constraint-based program analysis, whose key idea is to divide a program analysis into two stages: *constraint generation* and *constraint resolution*. The former generates constraints from the input program that constitute a declarative specification of the desired information of the program, while the latter then computes the desired information by solving the constraints. The constraint-based approach has several benefits that make it one of the preferred approaches to program analysis [5]:

1. It separates analysis specification from implementation. Constraint generation is the specification of the analysis while constraint resolution is the implementation. Such separation of concerns not only helps organize the analysis but also simplifies understanding it. Specifically, one only needs to inspect constraint generation to reason about correctness and accuracy without understanding the low-level implementation

details of the constraint solver; on the other hand, one only needs to inspect the underlying algorithm of the constraint solver to reason about performance while ignoring the analysis specification. Most importantly, this separation allows analysis writers to focus on the high-level design issues such as formulating the analysis specification and choosing the right approximations and assumptions, without concerning low-level implementation details.

2. Constraints are natural for specifying program analyses. Each constraint is usually local, which means it individually captures a subset of the input program syntax attributes without considering the rest. The conjunctions of local constraints in turn capture global properties of the program.

3. It enables program analyses to leverage the sophisticated implementations of existing constraint solvers. The constraint-based approach has emerged as a popular computing paradigm not only in program analysis, but in all areas of computer science (e.g., hardware design, machine learning, natural language processing at al.), and even in other fields (e.g., mathematical optimization, biology, planning et al.). Motivated by such dire demands, the constraint-solving community has made remarkable strides in both algorithms and engineering of constraint solvers, which can be all leveraged by the constraint-based program analysis.

Because of the above benefits, the constraint-based approach has been widely used to formulate program analyses. Popular constraint problem formulations include boolean satisfiability (SAT), Datalog, satisfiability modulo theories (SMT), and integer linear programming (ILP). Our approach uses Datalog as the constraint language of the input analysis, which we introduce in the next section.

$$
\begin{array}{rcll}
\textit{(program)} & C & ::= & \{c_1, \ldots, c_n\} \\
\textit{(literal)} & l & ::= & r(\alpha_1, \ldots, \alpha_n) \\
\textit{(variable)} & v & \in & \mathbb{V} = \{x, y, \ldots\} \\
\textit{(relation name)} & r & \in & \mathbb{R} = \{\mathtt{a}, \mathtt{b}, \ldots\}
\end{array}
\qquad
\begin{array}{rcll}
\textit{(constraint)} & c & ::= & l_0 \,\text{:-}\, l_1, \ldots, l_n \\
\textit{(argument)} & \alpha & ::= & v \mid d \\
\textit{(constant)} & d & \in & \mathbb{N} = \{0, 1, \ldots\} \\
\textit{(tuple)} & t & \in & \mathbb{T} = \mathbb{R} \times \mathbb{N}^*
\end{array}
$$

Figure 2.1: Datalog syntax.

$$
\begin{array}{rcl}
[\![C]\!] & \in & 2^{\mathbb{T}} \\
[\![c]\!] & \in & 2^{\mathbb{T}} \to 2^{\mathbb{T}} \\
[\![l]\!] & \in & \boldsymbol{\Sigma} \to \mathbb{T}, \text{ where } \boldsymbol{\Sigma} = (\mathbb{N} \cup \mathbb{V}) \to \mathbb{N} \\
[\![C]\!] & = & \mathsf{lfp}\ \lambda T\ .\ T \cup \bigcup_{c \in C} [\![c]\!](T) \\
[\![l_0 \,\text{:-}\, l_1, \ldots, l_n]\!](T) & = & \{[\![l_0]\!](\sigma) \mid (\forall i \in [1, n]\ .\ [\![l_i]\!](\sigma) \in T) \wedge \sigma \in \boldsymbol{\Sigma}\} \\
[\![r(\alpha_1, \ldots, \alpha_n)]\!](\sigma) & = & r(\sigma(\alpha_1), \ldots, \sigma(\alpha_n))
\end{array}
$$

Figure 2.2: Datalog semantics.

2.2 Datalog

Datalog [6] is a declarative logic programming language which originated as a querying language for deductive databases. Compared to the standard querying language SQL, it supports *recursive* queries. It is popular for specifying program analyses due to its declarativity, deductive constraint format, and least fixed point semantics.

Figure 2.1 shows the syntax of Datalog. A Datalog program C is a set of constraints $\{c_1, \ldots, c_n\}$. A constraint c is a deductive rule which consists a head literal l_0 and a body $l_1, \ldots, l_n$ that is a set of literals, which can be empty. Each literal l is a relation name r followed by a list of arguments $\alpha_1, \ldots, \alpha_n$ each of which can be either a variable or a constant. We also call a literal a tuple or a ground literal if its arguments are all constants. Note the standard Datalog syntax includes input tuples which can be changed to vary the output. Without loss of generality, we treat input tuples as constraints with empty bodies.

Figure 2.2 shows the semantics of Datalog. Let $\mathbb{T}$ be the domain of tuples. A Datalog $\mathbf{C}$ program computes a set of output tuples, which is denoted by $[\![C]\!]$. It obtains $[\![C]\!]$ by computing the least fixed point of its constrains. More concretely, starting with an empty set as the initial output T_o, it keeps growing T_o by applying each constraint until T_o no

(*program*)	C	$::=$	$\{c_1, \ldots, c_n\}$	(*constraint*)	c	$::=$	$c_h \mid c_s$
(*hard constraint*)	c_h	$::=$	$l_1 \vee \ldots \vee l_n$	(*soft constraint*)	c_s	$::=$	(c_h, w)
(*literal*)	l	$::=$	$l^+ \mid l^- \in \mathbb{L}$	(*weight*)	w	$\in$	R^+
(*positive literal*)	l^+	$::=$	$r(\alpha_1, \ldots, \alpha_n)$	(*negative literal*)	l^-	$::=$	$\neg\, l^+$
(*relation name*)	r	$\in$	$\mathbb{R} = \{\mathsf{a}, \mathsf{b}, \ldots\}$	(*argument*)	α	$::=$	$v \mid d$
(*variable*)	v	$\in$	$\mathbb{V} = \{x, y, \ldots\}$	(*constant*)	d	$\in$	$\mathbb{N} = \{0, 1, \ldots\}$
(*tuple*)	t	$\in$	$\mathbb{T} = \mathbb{R} \times \mathbb{N}^*$				

Figure 2.3: Markov Logic Network syntax.

longer changes. For a given constraint $c = l_0 \text{ :- } l_1, \ldots, l_n$, it is applied as follows: if there exists a substitution $\sigma \in (\mathbb{N} \cup \mathbb{V}) \to \mathbb{N}$ such that $\{[\![l_1]\!](\sigma), \ldots, [\![l_n]\!](\sigma)\} \subseteq T_o$, then $[\![l_0]\!](\sigma)$ is added to T_o. In the very first iteration, only constraints with empty bodies are triggered. The following standard result will be used tacitly in later arguments.

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

Example. Below is a Datalog program that computes pairwise node reachability in a directed graph:

$$c_1 : \ \mathsf{path}(a, a) \qquad\qquad c_2 : \ \mathsf{path}(a, c) \text{ :- } \mathsf{path}(a, b), \mathsf{edge}(b, c)$$

Besides the above two constraints, the input tuples, which are also part of the constraints, are edge tuples which encode the input graph, while the output tuples are path tuples that encode the reachability information. Constraints c_1 and c_2 capture two axioms about graph reachability respectively: (1) any node can reach itself, and (2) if there is a path from a to b, and there is an edge from b to c, then there is a path from a to c. The program successfully computes the reachability information by evaluating the least fixed point of these two constraints along with the input tuples.

$$
\begin{array}{rcl}
[\![C]\!]_P & \in & 2^{\mathbb{T}} \mapsto [0,1] \\
[\![c]\!]_P & \in & \mathbf{\Sigma} \times \mathbb{T} \mapsto \{-\infty\} \cup R_0^+, \text{ where } \mathbf{\Sigma} = (\mathbb{N} \cup \mathbb{V}) \mapsto \mathbb{N} \\
[\![l]\!] & \in & \mathbf{\Sigma} \mapsto \mathbb{L} \\
[\![C]\!]_P(T) & = & \frac{1}{Z} \exp(W_C(T)), \text{ where } Z = \sum_{T' \subseteq \mathbb{T}} \exp(W_C(T')) \\
W_C(T) & = & \sum_{c \in C, \sigma \in \mathbf{\Sigma}} [\![c]\!]_P(\sigma, T) \\
[\![l_1 \vee \ldots \vee l_n]\!]_P(\sigma, T) & = & \begin{cases} 0, & \text{iff } T \models [\![l_1]\!](\sigma) \vee \ldots \vee [\![l_n]\!](\sigma) \\ -\infty, & \text{otherwise} \end{cases} \\
[\![(l_1 \vee \ldots \vee l_n, w)]\!]_P(\sigma, T) & = & \begin{cases} w, & \text{iff } T \models [\![l_1]\!](\sigma) \vee \ldots \vee [\![l_n]\!](\sigma) \\ 0, & \text{otherwise} \end{cases} \\
T & \models & [\![l_1]\!](\sigma) \vee \ldots \vee [\![l_n]\!](\sigma) \text{ iff } \exists\, i \in [1,n].T \models [\![l_i]\!](\sigma) \\
[\![r(\alpha_1, \ldots, \alpha_n)]\!](\sigma) & = & r(\sigma(\alpha_1), \ldots, \sigma(\alpha_n)) \\
[\![\neg r(\alpha_1, \ldots, \alpha_n)]\!](\sigma) & = & \neg r(\sigma(\alpha_1), \ldots, \sigma(\alpha_n)) \\
T & \models & r(d_1, \ldots, d_n) \text{ iff } r(d_1, \ldots, d_n) \in T \\
T & \models & \neg r(d_1, \ldots, d_n) \text{ iff } r(d_1, \ldots, d_n) \notin T
\end{array}
$$

Figure 2.4: Markov Logic Network semantics.

2.3 Markov Logic Networks

Our system takes a Datalog analysis and synthesizes a novel analysis that combines logic and probability. The new analysis is also constraint-based, and the constraint language we apply is a variant of Markov Logic Networks [12], a declarative language for combining logic and probability. Compared to the original Markov Logic Networks proposed by Richardson and Domingos [12], our variant is different in the following two ways: (1) the logical part is a decidable fragment of the first-order logic instead of the complete first-order logic; and (2) besides constraints with weights, our language also includes constraints without weights. The second difference allows us to directly support hard constraints, which are constraints that cannot be violated. On the other hand, such constraints are typically represented using constraints with very high weights in the original Markov Logic Networks. We next introduce the syntax and semantics of our language in detail.

Figure 2.3 shows the syntax of a Markov Logic Network. A Markov Logic Network

program C is a set of constraints $\{c_1, \ldots, c_n\}$ each of which is either a hard constraint or a soft constraint. Each hard constraint c_h is a disjunction of literals, while each soft constraint c_s is a hard constraint extended with a positive real weight w. A literal l can be either a positive literal or a negative literal: a positive literal l^+ is a relation name r followed by a list of parameters $\alpha_1, \ldots, \alpha_n$ each of which is either a variable or a constant, while a negative literal l^- is a negated positive literal. In the rest of the thesis, we sometimes write constraints in the form of implications (e.g., $\mathsf{r}_1(a) \implies \mathsf{r}_2(a)$ instead of $\neg\mathsf{r}_1(a) \vee \mathsf{r}_2(a)$). Similar to the Datalog syntax, we call a positive literal whose arguments are all constants a tuple. We call constraint c_i an instance or a ground constraint of c if we can obtain c_i by substituting all variables in c with certain constants. We call the set of all instances of c the grounding of c. Similarly we call the union of all instances of the constraints in C the grounding of C. Formally:

$$\mathsf{grounding}(l_1 \vee \ldots \vee l_n) = \{[\![l_1]\!](\sigma) \vee \ldots \vee [\![l_n]\!](\sigma) \mid \sigma \in \Sigma\},$$
$$\mathsf{grounding}(C) = \bigcup_{c \in C} \mathsf{grounding}(c).$$

Different from the Datalog syntax, we parameterize the Markov Logic Network syntax with the domain of constants, which allows us to control the size of the grounding. More concretely, when the domain of constants is some subset $N \subseteq \mathbb{N}$, the domain of substitutions Σ becomes $(N \cup \mathbb{V}) \mapsto N$ and the domain of tuples $\mathbb{T}$ becomes $\mathbb{R} \times N^*$. This further affects the performance of the underlying Markov Logic Network solver (as we shall see in Chapter 3.1). However, unless explicitly specified (only in Chapter 3.1), the domain of constants is the set of all constants $\mathbb{N}$. We introduce a function $\mathsf{constants} : 2^C \mapsto 2^{\mathbb{N}}$ that returns all the constants that appear in constraints C.

Figure 2.4 shows the semantics of a Markov Logic Network. Compared to a Datalog program, which defines a unique output, a Markov Logic Network defines a joint distribution of outputs. Given a set of tuples T, program C returns its probability, which is denoted

by $[\![C]\!]_P(T)$. Specifically, we say T is not a **valid** output if $[\![C]\!]_P(T) = 0$. Intuitively, T is valid iff it does not violate any hard constraint instance, and the more soft constraint instances it satisfies, the higher probability it has. We calculate $[\![C]\!]_P(T)$ by dividing the result of applying the exponential function $\exp$ to the weight of T (denoted by $W_C(T)$) by a normalization factor Z. The normalization factor Z is calculated by adding up the results of applying $\exp$ to the weights of all tuple sets and thereby guarantees that (1) all probabilities are between 0 and 1, and (2) the sum of all probabilities is 1 [1]. For a set of tuples T, we calculate its weight by summarizing its weight over each constraint instance (denoted by $\sum_{c \in C, \sigma \in \mathbf{\Sigma}} [\![c]\!]_P(\sigma, T)$). Given a hard constraint $l_1 \vee \ldots \vee l_n$ and a substitution $\sigma \in (\mathbb{N} \cup \mathbb{V}) \to \mathbb{N}$, the weight of T over the instance $[\![l_1]\!](\sigma) \vee \ldots \vee [\![l_n]\!](\sigma)$ is 0 if T satisfies it and $-\infty$ otherwise. Given a soft constraint $(l_1 \vee \ldots \vee l_n, w)$ and a substitution σ, the weight of T over the instance $([\![l_1]\!](\sigma) \vee \ldots \vee [\![l_n]\!](\sigma), w)$ is w if T satisfies $[\![l_1]\!](\sigma) \vee \ldots \vee [\![l_n]\!](\sigma)$ and 0 otherwise. Note that when the domain of constants is specified as some subset of all constants $N \subseteq \mathbb{N}$, the domain of substitutions Σ changes to $(N \cup \mathbb{V}) \mapsto N$ and the domain of tuples $\mathbb{T}$ becomes $\mathbb{R} \times N^*$.

In all our applications, we are interested in finding an output with the highest probability while satisfying all hard constraint instances, which can be obtained by solving the *maximum a posteriori probability* (MAP) inference problem. We define the MAP inference problem below:

Problem 2 (MAP Inference)**.** Given a Markov Logic Network C, the MAP inference problem is to find a valid set of tuples that maximizes the probability:

$$\mathrm{MAP}(C) = \begin{cases} \mathsf{UNSAT}, & \text{if } \max_T([\![C]\!]_P(T)) = 0, \\ T \text{ such that } T \in \arg\max_T([\![C]\!]_P(T)), & \text{otherwise.} \end{cases}$$

[1] To ensure that Z is a finite number, the domain of constants of C needs to be finite, which is not required for Datalog. Throughout this thesis, we assume $\mathbb{N}$ to be finite except in Chapter 3.1. In that section, however, the domains of constants of all discussed Markov Logic Networks are always some finite subsets of $\mathbb{N}$.

Since Z is a constant and $\exp$ is monotonic, we can rewrite $\arg\max_T(\llbracket C \rrbracket_P(T))$ as:

$$\arg\max_T(\llbracket C \rrbracket_P(T)) = \arg\max_T \frac{1}{Z} \exp(\sum_{c \in C, \sigma \in \mathbf{\Sigma}} \llbracket c \rrbracket_P(\sigma, T)) = \arg\max_T \sum_{c \in C, \sigma \in \mathbf{\Sigma}} \llbracket c \rrbracket_P(\sigma, T).$$

In other words, the MAP inference problem is equivalent to finding a set of output tuples that maximizes the sum of the weights of satisfied ground soft constraints while satisfying all the ground hard constraints. Note that, once we obtain the grounding of C, we can solve this problem directly by casting it as a *maximum satisfiability* (MAXSAT) problem, which in turn can be solved by an off-the-shelf MAXSAT solver.

Example. Consider the same graph reachability example in Chapter 2.2, we can express the problem using the Markov Logic Network below:

$$c_1 : \mathsf{path}(a, a) \quad c_2 : \mathsf{path}(a, b) \wedge \mathsf{edge}(b, c) \implies \mathsf{path}(a, c) \quad c_3 : \neg\mathsf{path}(a, b)\ \mathsf{weight}\ 1.$$

Besides the above three constraints, we also add each input tuple in the original Datalog example as a hard constraint to the program, which we omit for elaboration. Among the three constraints, hard constraints c_1 and c_2 are used to express the same two axioms that their counterparts in the example Datalog program express, while soft constraint c_3 is used to bias towards a minimal model as the MAP solution. As a result, by solving the MAP inference problem of this Markov Logic Network, we obtain the same solution as the previous example Datalog program produces.

CHAPTER 3

APPLICATIONS

This chapter discusses how PETABLOX incorporates probabilistic reasoning in a conventional logic-based program analysis to address key challenges in three prominent applications: automated verification, interactive verification, and static bug detection. For automated verification, it addresses the challenge of selecting an abstraction that balances precision and scalability; for interactive verification, it addresses the challenge of reducing user effort in resolving analysis alarms; and for static bug detection, it addresses the challenge of sifting true alarms from false alarms. These applications were originally discussed in our previous publications [8, 14, 9].

Table 3.1 summarizes the Markov Logic Network encoding for each application. Briefly, while the hard constraints encode the correctness conditions (e.g., soundness), the soft constraints balance various trade-offs of the analysis; by solving the combined analysis instance, we obtain a correct analysis result that strikes the best balance between these trade-offs.

In the rest of this chapter, we discuss each application in detail, including the motivation, our approach (in particular, the underlying Markov Logic Network encoding), the empirical evaluation, and related work.

3.1 Automated Verification

3.1.1 Introduction

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

Table 3.1: Markov Logic Network encodings of different program analysis applications.

Application	Hard Constraints	Soft Constraints	Trade-off
Automated Verification	Analysis Rules $\text{Abstraction}_1 \oplus \ldots \oplus \text{Abstraction}_n$	$\neg$ Result_i weight w_i *where w_i is the award for resolving Result_i* Abstraction_j weight w_j *where w_j is the cost of applying Abstraction_j*	Accuracy vs. Scalability
Interactive Verification	Analysis Rules	$\neg$ Result_i weight w_i *where w_i is the award for resolving Result_i* Cause_j weight w_j *where w_j is the cost of inspecting Cause_j*	Accuracy vs. User Effort
Static Bug Detection	Analysis Rules (Optional)	Analysis Rule_i weight w_i *where w_i is the writer's confidence in Rule_i* Feedback_j weight w_j *where w_j is the user's confidence in Feedback_j*	Writer's Idioms vs. User's Feedback

with efficient data structures and algorithms for the analysis.

One popular approach for addressing this problem is to use Datalog [15, 16, 17, 18]. In this approach, a program analysis only specifies how to generate Datalog constraints from program text. The task of solving the generated constraints is then delegated to an off-the-shelf Datalog constraint solver, such as that underlying BDDBDDB [19], Doop [20], Jedd [21], and Z3's fixpoint engine [22], which in turn relies on efficient symbolic algorithms and data structures, such as Binary Decision Diagrams (BDDs).

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

In this section, we present a new approach for lifting this limitation by automatically finding effective abstractions for program analyses written in Datalog. Our approach is

based on counterexample-guided abstraction refinement (CEGAR), which was developed in the model-checking community and has been applied effectively for software verification with predicate abstraction [4, 23, 1, 24, 25]. A counterexample in Datalog is a derivation of an output tuple from a set of input tuples via Horn-clause inference rules: the rules specify the program analysis, the set of input tuples represents the current program abstraction, and the output tuple represents a (typically undesirable) program property derived by the analysis under that abstraction. The counterexample is spurious if there exists some abstraction under which the property *cannot* be derived by the analysis. The CEGAR problem in our approach is to find such an abstraction from a given family of abstractions.

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

The above result ensures that solving our Markov Logic Network formulation generalizes the cause of verification failure in the current CEGAR iteration to the maximum extent possible, eliminating not only the current abstraction but all other abstractions destined to suffer a similar failure. There is still the problem of deciding which abstraction to try next. We show that the soft constraints in our problem formulation enables us to identify the *cheapest* refined abstraction. Our approach avoids unnecessary refinement by using this abstraction in the next CEGAR iteration.

We have implemented our approach and applied it to two realistic static analyses written in Datalog, a pointer analysis and a typestate analysis, for Java programs. These two analyses differ significantly in aspects such as flow sensitivity (insensitive vs. sensitive),

context sensitivity (cloning-based vs. summary-based), and heap abstraction (weak vs. strong updates), which demonstrates the generality of our approach. On a suite of eight real-world Java benchmark programs, our approach searches a large space of abstractions, ranging from 2^{1k} to 2^{5k} for the pointer analysis and 2^{13k} to 2^{54k} for the typestate analysis, for hundreds of analysis queries considered simultaneously in each program, thereby showing the practicality of our approach.

We summarize the main contributions of our work:

1. We propose a CEGAR-based approach to automatically find effective abstractions for analyses in Datalog. The approach enables Datalog analysis designers to specify high-level knowledge about abstractions while continuing to leverage low-level implementation advances in off-the-shelf Datalog solvers.

2. We solve the CEGAR problem using a Markov Logic Network formulation that has desirable properties of generality, completeness, and optimality: it is independent of the Datalog solver, it fully generalizes the failure of an abstraction, and it computes the cheapest refined abstraction.

3. We show the effectiveness of our approach on two realistic analyses written in Datalog. On a suite of real-world Java benchmark programs, the approach explores a large space of abstractions for a large number of analysis queries simultaneously.

3.1.2 Overview

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

The example program in Figure 3.1 allocates an object in each of methods `f` and `g`, and passes it to methods `id1` and `id2`. The pointer analysis is asked to prove two queries: query `q1` stating that `v6` is not aliased with `v1` at the end of `g`, and query `q2` stating that `v3` is not aliased with `v1` at the end of `f`. Proving `q1` requires a *context-sensitive* analysis

```
1   f() { v1 = new ...;                 7  g() { v4 = new ...;
2         v2 = id1(v1);                 8        v5 = id1(v4);
3         v3 = id2(v2);                 9        v6 = id2(v5);
4     q2: assert(v3 != v1);             10   q1: assert(v6 != v1);
5   }                                   11 }
6   id1(v) { return v; }                12 id2(v) { return v; }
```

Figure 3.1: Example program.

that distinguishes between different calling contexts of methods `id1` and `id2`. Query `q2`, on the other hand, cannot be proven since `v3` is in fact aliased with `v1`.

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

For exposition, we recast this problem as a reachability problem on the graph in Figure 3.2. In that graph, nodes 0, 1, and 2 represent basic blocks of `f`, while nodes 3, 4, and 5 represent basic blocks of `g`. Nodes 6 and 7 represent the bodies of `id1` and `id2` respectively, while nodes $6'$, $6''$, $7'$ and $7''$ are their clones at different call sites. Edges denoting matching calls and returns have the same label. A choice of labels constitutes a valid abstraction of the original program if, for each of a, b, c, and d, either the zero (non-cloned) or the one (cloned) version is chosen. Then, proving query `q1` corresponds to showing that node 5 is unreachable from node 0 under some valid choice of labeled edges, which is the case if edges labeled $\{a_1, b_0, c_1, d_0\}$ are chosen; proving query `q2` corresponds to finding a valid combination of edge labels that makes node 2 unreachable from node 0, but this is impossible.

Our graph reachability problem can be expressed in Datalog as shown in Figure 3.2. A Datalog program consists of a set of input relations, a set of derived relations, and a set of rules that express how to compute the derived relations from the input relations.

Input relations:

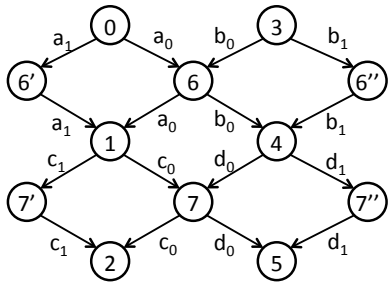
$\mathsf{edge}(i, j, n)$	(edge from node i to node j labeled n)
$\mathsf{abs}(n)$	(edge labeled n is allowed)

Derived relations:

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

Rules: (1): $\mathsf{path}(i, i)$.

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



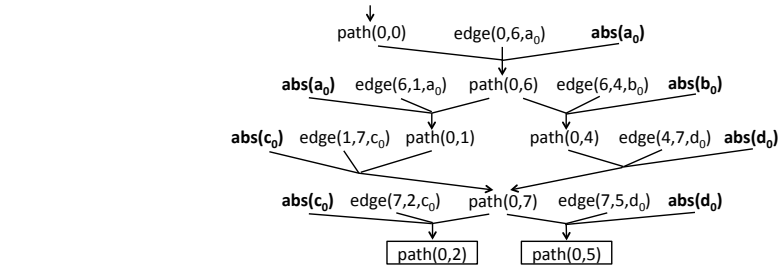
Input tuples:	**Derived tuples:**
$\mathsf{edge}(0, 6, a_0)$	$\mathsf{path}(0, 0)$
$\mathsf{edge}(6, 1, a_0)$	$\mathsf{path}(0, 6)$
$\mathsf{edge}(1, 7, c_0)$	$\mathsf{path}(0, 1)$
...	...
$\mathsf{abs}(a_0)$	**Query tuples:**
$\mathsf{abs}(c_0)$	$\mathsf{path}(0, 5)$
...	$\mathsf{path}(0, 2)$

Figure 3.2: Graph reachability example in Datalog.

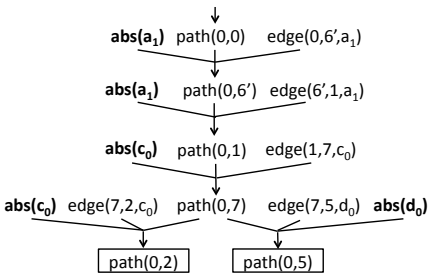
There are two input relations in our example: edge, representing the possible labeled edges in the given graph; and abs, containing labels of edges that may be used in computing graph reachability. Relation abs specifies a program abstraction in our original setting; for instance, $\mathsf{abs} = \{a_1, b_0, c_1, d_0\}$ specifies the abstraction in which only the calls to methods `id1` and `id2` from `f` are inlined.

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

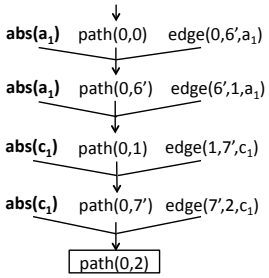
Our two queries `q1` and `q2` correspond to tuples $\mathsf{path}(0, 5)$ and $\mathsf{path}(0, 2)$ respectively. There are in all 16 abstractions, each involving a different choice of the zero/one versions



(a) Using $\mathsf{abs} = \{a_0, b_0, c_0, d_0\}$.



(b) Using $\mathsf{abs} = \{a_1, b_0, c_0, d_0\}$.



(c) Using $\mathsf{abs} = \{a_1, b_0, c_1, d_0\}$.

Figure 3.3: Derivations after different iterations of our approach on our graph reachability example.

Table 3.2: Each iteration (run) eliminates a number of abstractions. Some are eliminated by analyzing the current Datalog run (within run); some are eliminated because of the derivations from the current run interact with derivations from previous runs (across runs).

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

of labels a through d in relation abs. Since we wish to minimize the amount of cloning in the original setting, abstractions with more zero versions of edge labels are cheaper. Our approach, outlined next, efficiently finds the cheapest abstraction $\mathsf{abs} = \{a_1, b_0, c_1, d_0\}$ that proves `q1`, and shows `q2` cannot be proven by any of the 16 abstractions.

Our approach is based on iterative counterexample-guided abstraction refinement. Table 3.2 illustrates its iterations on the graph reachability example. In the first iteration, the cheapest abstraction $\mathsf{abs} = \{a_0, b_0, c_0, d_0\}$ is tried. It corresponds to the case where neither of nodes 6 and 7 is cloned (i.e., a fully context-insensitive analysis). This abstraction fails to

prove both of our queries. Figure 3.3a shows all the possible derivations of the two queries using this abstraction. Each set of edges in this graph, incoming into a node, represents an application of a Datalog rule, with the source nodes denoting the tuples in the body of the rule and the target node denoting the tuple at its head.

The first question we ask is: how do we generalize the failure of the current abstraction to avoid picking another that will suffer a similar failure? Our solution is to exploit a monotonicity property of Datalog: more input tuples can only derive more output tuples. It follows from this property that the maximum generalization of the failure can be achieved if we find *all minimal subsets* of the set of tuples in the current abstraction that suffice to derive queries. From the derivation in Figure 3.3a, we see that these minimal subsets are $\{a_0, b_0, d_0\}$ and $\{a_0, c_0, d_0\}$ for query $\mathsf{path}(0, 5)$, and $\{a_0, c_0\}$ for query $\mathsf{path}(0, 2)$. We thus generalize the current failure to the maximum possible extent, eliminating any abs that is a superset of $\{a_0, b_0, d_0\}$ or $\{a_0, c_0, d_0\}$ for query $\mathsf{path}(0, 5)$ and any abs that is a superset of $\{a_0, c_0\}$ for query $\mathsf{path}(0, 2)$.

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

In the second iteration, our approach uses this abstraction, and again fails to prove both queries. But this time the derivation, shown in Figure 3.3b, is different from that in the first iteration. This time, we eliminate any abs that is a superset of $\{a_1, c_0, d_0\}$ for query $\mathsf{path}(0, 5)$, and any abs that is a superset of $\{a_1, c_0\}$ for query $\mathsf{path}(0, 2)$. The cheapest of the remaining abstractions, not eliminated for both the queries, is $\mathsf{abs} = \{a_1, b_0, c_1, d_0\}$, which corresponds to cloning methods `id1` and `id2` in `f` (but not in `g`).

Using this abstraction in the third iteration, our approach succeeds in proving query $\mathsf{path}(0, 5)$, but still fails to prove query $\mathsf{path}(0, 2)$. As seen in the derivation in Figure 3.3c,

this time $\{a_1, c_1\}$ is the minimal failure subset for query $\mathsf{path}(0, 2)$ and we eliminate any abs that is its superset.

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

$$\mathsf{path}(0, 6) \text{ :- } \mathsf{path}(0, 0),\ \mathsf{edge}(0, 6, a_0),\ \mathsf{abs}(a_0)$$
$$\mathsf{path}(0, 1) \text{ :- } \mathsf{path}(0, 6),\ \mathsf{edge}(6, 1, a_0),\ \mathsf{abs}(a_0)$$
$$\mathsf{path}(0, 7') \text{ :- } \mathsf{path}(0, 1),\ \mathsf{edge}(1, 7', c_1), \mathsf{abs}(c_1)$$
$$\mathsf{path}(0, 2) \text{ :- } \mathsf{path}(0, 7'), \mathsf{edge}(7', 2, c_1), \mathsf{abs}(c_1)$$

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

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

Hard constraints:	**Soft constraints:**
$\mathsf{path}(i, i)$	$\mathsf{abs}(a_0)$ weight 1
$\mathsf{path}(i, k) \wedge \mathsf{edge}(k, j, n) \wedge \mathsf{abs}(n) \implies \mathsf{path}(i, j)$	$\mathsf{abs}(b_0)$ weight 1
$\mathsf{edge}(0, 6, a_0)$	$\mathsf{abs}(c_0)$ weight 1
$\mathsf{edge}(6, 1, a_0)$	$\mathsf{abs}(d_0)$ weight 1
$\mathsf{edge}(1, 7, c_0)$	$\neg\mathsf{path}(0, 5)$ weight 5
...	$\neg\mathsf{path}(0, 2)$ weight 5

Figure 3.4: Formula from the Datalog run's result in the first iteration.

Encoding as a Markov Logic Network. In the graph reachability example, our approach searched a space of 16 different abstractions for each of two queries. In practice, we seek to apply our approach to real-world programs and analyses, where the space of abstractions is 2^x for x of the order of tens of thousands, for each of hundreds of queries. To handle such large spaces efficiently, our frontend PETABLOX formulates a Markov Logic Network from the output of the Datalog computation in each iteration, and solves it using our backend solver NICHROME to obtain the abstraction to try in the next iteration. For our example, Figure 3.4 shows the Markov Logic Network PETABLOX constructs at the end of the first iteration. It has two kinds of constraints: hard constraints, whose instances *must* be satisfied, and soft constraints, each of which is associated with a weight and whose instances may be left unsatisfied. We seek to find a solution that maximizes the sum of the weights of satisfied soft constraint instances, which can be obtained by solving the MAP inference problem of the constructed formula.

Briefly, the formula has three parts. The first part is hard constraints that encode the derivation. Concretely, the first two hard constraints corresponds to the two rules in the original Datalog program, while the rest hard constraints correspond to non-abstraction input tuples. The instances of the rule constraints and the non-abstraction input tuple constraints together capture the derivation of the Datalog program.

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

when it contains a one label.

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

The formula constructed in each subsequent iteration conjoins those from all previous iterations with the formula encoding the derivation of the current iteration. We also add additional hard constraints to encode the space of valid abstractions. For instance, $\mathsf{abs}(a_0) \vee \mathsf{abs}(a_1)$ and $\neg\mathsf{abs}(a_0) \vee \neg\mathsf{abs}(a_1)$ are added to the formula constructed at the end of the second iteration to denote that one must choose to either clone `id1` at the call in `f` or not, and cannot choose both simultaneously.

For efficiency concerns, we limit the domain of constants to ones that appear in the output tuples of the Datalog runs. Given the space of valid abstractions can be very large and even infinite, such treatment is essential to the scalability of the underlying Markov Logic Network solver.

3.1.3 Parametric Dataflow Analyses

Datalog is a logic programming language that is capable of naturally expressing many static analyses [19, 20]. In this section, we review this use of Datalog, especially for developing ***parametric*** static analyses. Such an analysis takes (an encoding of) the program to analyze, a set of queries, and a setting of parameters that dictate the degree of program

abstraction. The analysis then outputs queries that it could successfully verify using the chosen abstraction. Our goal is to develop an efficient algorithm for automatically adjusting the setting of these abstraction parameters for a given program and set of queries. We formally state this problem in this subsection and present our CEGAR-based solution in the subsequent subsection.

3.1.3.1 Abstractions and Queries

Datalog programs that implement parametric static analyses contain three types of constraints: (1) those that encode the abstract semantics of the programming language, (2) those that encode the program being analyzed, and (3) those that determine the degree of program abstraction used by the abstract semantics.

Example. Our graph reachability example (Figure 3.2) is modeled after a parametric pointer analysis, and is specified using these three types of constraints. The two rules in the figure describe inference steps for deriving tuples of the path relation. These rules apply for all graphs, and they are constraints of type (1). Input tuples of the edge relation encode information about a specific graph, and are of type (2). The remaining input tuples of the abs relation specify the amount of cloning and control the degree of abstraction used in the analysis; they are thus constraints of type (3).

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

[1]To join with C, we implicitly cast a set of tuples A as a set of constraints each of which is a unit clause.

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

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

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

Example. In our graph reachability example, the family $\mathbf{A}$ of valid abstractions consists of sets $\{\mathsf{abs}(a_i), \mathsf{abs}(b_j), \mathsf{abs}(c_k), \mathsf{abs}(d_l)\}$ for all $i, j, k, l \in \{0, 1\}$. They describe 16 options of cloning nodes in the graph. The set of queries is $Q = \{\mathsf{path}(0, 5), \mathsf{path}(0, 2)\}$.

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

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

Some analyses have a most precise abstraction $A_\top$, which can rule out most queries. This abstraction, however, is often impractical, in the sense that computing $[\![C \cup A_\top]\!]$ requires too much time or space. The efficiency preorder captures the notion of abstraction efficiency:

$A_1 \preceq A_2$ denotes that A_1 is at most as efficient as A_2. Often, the two preorders point in opposite directions.

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

3.1.3.2 Problem Statement

Our aim is to solve the following problem:

Definition 3 (Datalog Analysis Problem). Suppose we are given an analysis C, a set $Q \subseteq \mathbf{Q}$ of queries, and an abstraction family $(\mathbf{A}, \sqsubseteq, \preceq)$. Compute the set $\mathcal{R}(\mathbf{A}, Q)$ of queries that can be ruled out by some valid abstraction.

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

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

Example. What we have described for our running example provides the instance $(C, Q, (\mathbf{A}, \sqsubseteq, \preceq))$ of the Datalog Analysis problem. Recall that $Q = \{\mathsf{path}(0, 2), \mathsf{path}(0, 5)\}$. As we explained in Chapter 3.1.2, among these two queries, only $\mathsf{path}(0, 5)$ can be ruled out by some abstraction. A cheapest such abstraction according to the efficiency order $\preceq$ is $A = \{\mathsf{abs}(a_1), \mathsf{abs}(b_0), \mathsf{abs}(c_1), \mathsf{abs}(d_0)\}$, which clones two nodes, while the most expensive one is $B = \{\mathsf{abs}(a_1), \mathsf{abs}(b_1), \mathsf{abs}(c_1), \mathsf{abs}(d_1)\}$ with four clones. Hence, the answer $\mathcal{R}(\mathbf{A}, Q)$ for this problem is $\{\mathsf{path}(0, 5)\}$, and our goal is to arrive at this answer in a small number of refinement iterations, while mostly trying a cheap abstraction in each iteration, such as the abstraction A rather than B.

3.1.4 Algorithm

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

We begin by presenting the first idea (Chapter 3.1.4.1): how Datalog runs are encoded in sets of hard constraints, and what properties this encoding has. In particular, we observe that conjoining the encoding of multiple Datalog runs gives an under-approximation for the set of unviable abstractions (Theorem 5). This observation motivates the overall structure of our CEGAR-based solution to the Datalog analysis problem, which we describe next (Chapter 3.1.4.2). The algorithm relies on a subroutine for choosing the next abstraction. While arguing for the correctness of the algorithm, we formalize the requirements for this subroutine: it should choose a cheap abstraction not yet known to be unviable. We finish by

describing the second idea (Chapter 3.1.4.3), how choosing a good abstraction is essentially a MAP inference problem, thus completing the description of our solution.

3.1.4.1 From Datalog Derivations to Hard Constraints

In the CEGAR algorithm, we iteratively call a Datalog solver. It is desirable to do so as few times as possible, so we wish to eliminate as many unviable abstractions as possible without calling the solver. To do so, we need to rely on more information than the binary answer of a Datalog run, on whether an abstraction derives or rules out a query. Intuitively, there is more information, waiting to be exploited, in *how* a query is derived. Theorem 4 shows that by recording the Datalog run for an abstraction A as a set of hard constraints it is possible to partly predict what Datalog will do for other abstractions that share tuples with A. Perhaps less intuitively, Theorem 5 shows that it is sound to mix (parts of) derivations seen for different runs of Datalog. Thus, in some situations we can predict that an abstraction A_1 will derive a certain query by combining tuple dependencies observed in runs for two other abstractions A_2 and A_3.

Our method of producing a set of hard constraints from the result of running analysis C with abstraction A is straightforward: we simply take all the constraints in C and limit the domain of constants to the ones that are present in $[\![C \cup A]\!]$, which we refer to as $\mathsf{constants}([\![C \cup A]\!])$. While the set of constructed constraints do not change across iterations, the information that they capture about the Datalog runs increases as the domain of constants grows. We grow the domain of constants lazily as the MAP inference problem of Markov Logic Networks is computationally challenging and the domain of constants can be very large and even infinite (when the space of valid abstractions is infinite). However, we show that such a set of constraints still contain sufficient information about existing derivations. The following theorem states that such a set of constraints allows us to predict what Datalog would do for those abstractions $A' \subseteq A$.

Theorem 4. *Given a set of Datalog constraints C and a set of tuples A, let A' be a subset of*

A. Then the Datalog run result $[\![C \cup A']\!]$ is fully determined by the Markov Logic Network C where the domain of constants is $\mathsf{constants}([\![C \cup A]\!])$*, as follows:*

$$t \in [\![C \cup A']\!] \quad \iff \quad \forall T.(T = \mathrm{MAP}(C \cup A')) \implies t \in T,$$

where $\mathsf{constants}([\![C \cup A]\!])$ *is the domain of constants of Markov Logic Network $C \cup A'$.*

One interesting instantiation of the theorem is the case that A is a valid abstraction. To see the importance of this case, consider the first iteration of the running example (Chapter 3.1.2), where A is $\{\mathsf{abs}(a_0), \mathsf{abs}(b_0), \mathsf{abs}(c_0), \mathsf{abs}(d_0)\}$. The theorem says that it is possible to predict exactly what Datalog would do for subsets of A. If such a subset derives a query then, by monotonicity (Proposition 1), so do all its supersets. In other words, we can give lower bounds for which queries do other abstractions derive. (All other abstractions are supersets of subsets of A.)

When the analysis C is run multiple times with different abstractions, the results $A_1, \ldots,$ A_n of these runs lead to the same set of hard constraints with different constant domains $\mathsf{constants}([\![C \cup A_1]\!]), \ldots, \mathsf{constants}([\![C \cup A_n]\!])$. The next theorem points out the benefit of considering these formulas together, as illustrated in Chapter 3.1.2. It implies that by conjoining these formulas, we can mix derivations from different runs, and identify more unviable abstractions.

Theorem 5. *Let $A_1, \ldots, A_n$ be sets of tuples. For all $A' \subseteq \bigcup_{i \in [1,n]} A_i$,*

$$\forall T.(T = \mathrm{MAP}(C \cup A')) \implies t \in T \quad \implies \quad t \in [\![C \cup A']\!],$$

where $\bigcup_{i \in [1,n]} \mathsf{constants}([\![C \cup A_i]\!])$ *is the domain of constants of Markov Logic Network $C \cup A'$.*

Algorithm 1 CEGAR-based Datalog analysis.

1: **INPUT:** Queries Q
2: **OUTPUT:** A partition (R, I) of Q, where R contains queries that have been ruled out and I queries impossible to rule out.
3: **var** $R := \emptyset, T := \emptyset$
4: **loop**
5: $\quad A := \mathsf{choose}(T, C, Q \setminus R)$
6: $\quad$ **if** $(A = \mathsf{impossible})$ **return** $(R, Q \setminus R)$
7: $\quad T' := [\![C \cup A']\!]$
8: $\quad R := R \cup (Q \setminus T')$
9: $\quad T := T \cup T'$
10: **end loop**

3.1.4.2 The Algorithm

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

The call $\mathsf{choose}(T, C, Q')$ evaluates to $\mathsf{impossible}$ only if all queries in Q' are impossible to rule out, according to the information encoded in C where the domain of constants is $\mathsf{constants}(T)$. Thus, the algorithm terminates only if all queries that can be ruled out have been ruled out. Conversely, choose never returns an abstraction whose analysis was previously recorded in C and T. Intuitively, C with $\mathsf{constants}(T)$ as the domain of constants represents the set of abstractions known to be unviable for the remaining set of queries $Q' = (Q \setminus R)$. Formally, this notion is captured by the concretization function γ, whose definition is justified by Theorem 5. Let $\mathbf{C}$ be the domain of constraints, we have

$$\gamma \in \mathcal{P}(\mathbb{T}) \times \mathcal{P}(\mathbf{C}) \times \mathcal{P}(\mathbf{Q}) \to \mathcal{P}(\mathbf{A})$$

$$\gamma(T, C, Q') \stackrel{\Delta}{=} \big\{\, A \in \mathbf{A} \;\big|\; \forall T'.(T' = \mathrm{MAP}(C \cup (A \cap T))) \implies Q' \subseteq T', \text{ where the domain of constants is } \mathsf{constants}(T) \,\big\}.$$

The condition that $\forall T.(T = \mathrm{MAP}(C \cup (A \cap T))) \implies Q' \subseteq T$ appears complicated, but

it is just a formal way to say that all queries in Q' are derivable from A using derivations encoded in C with $\mathsf{constants}(T)$ as the domain of constants. Hence, $\gamma(T, C, Q')$ contains the abstractions known to be unviable with respect to Q'; thus $\mathbf{A} \setminus \gamma(T, C, Q')$ is the set of valid abstractions that, according to C and T, might be able to rule out some queries in Q'. The function $\mathsf{choose}(T, C, Q')$ chooses an abstraction from $\mathbf{A} \setminus \gamma(T, C, Q')$, if this set is not empty.

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

Theorem 6. *If* $\mathsf{choose}(T, C, Q')$ *evaluates to an element of the set* $\mathbf{A} \setminus \gamma(T, C, Q')$ *whenever such an element exists, and to* impossible *otherwise, then Algorithm 1 is partially correct: it returns* (R, I) *such that* $R = \mathcal{R}(\mathbf{A}, Q)$ *and* $I = Q \setminus R$*. In addition, if* $\mathbf{A}$ *is finite, then Algorithm 1 terminates.*

The next section gives one definition of the function choose that satisfies the requirements of Theorem 6, thus completing the description of a correct algorithm. The definition of choose from Chapter 3.1.4.3 makes use of the efficiency preorder $\preceq$, such that the resulting algorithm is not only correct, but also efficient. Our implementation also makes use of the precision preorder $\sqsubseteq$ to further improve efficiency. The main idea is to constrain the sequence of used abstractions to be ascending with respect to $\sqsubseteq$. For this to be correct, however, the abstraction family must satisfy an additional condition: for any two abstractions A_1 and A_2 there must exist another abstraction A such that $A \sqsupseteq A_1$ and $A \sqsupseteq A_2$. The proofs are available in Appendix A.2.

3.1.4.3 Choosing Good Abstractions via Mixed Hard and Soft Constraints

The requirement that function $\mathsf{choose}(T, C, Q')$ should satisfy was laid down in the previous section: it should return an element of $\mathbf{A} \setminus \gamma(T, C, Q')$, or say impossible if no

such element exists. In this subsection we describe how to choose the cheapest element, according to the preorder $\preceq$. The type of function choose is

$$\mathsf{choose} \ \in \ \mathcal{P}(\mathbb{T}) \times \mathcal{P}(\mathbf{C}) \times \mathcal{P}(\mathbf{Q}) \ \rightarrow \ \mathbf{A} \uplus \{\mathsf{impossible}\}$$

The function choose is essentially a reduction to the MAP problem of a Markov Logic Network that extends the hard constraints C with additional soft constraints. Before describing the formulation in general, let us examine an example.

Example. Consider an analysis with parameters p_1, p_2, ..., p_n, each taking a value from $\{1, 2, \ldots, k\}$. The set of valid abstractions is $\mathbf{A} = \{1, 2, \ldots, n\} \rightarrow \{1, 2, \ldots, k\}$. We use $[p_i = j]$ to refer to the Datalog tuple that encodes p_i has value j. Let the queries be $\{q_1, q_2, q_3\}$, the Datalog program be C, and suppose we have tried abstractions $\bigwedge_i p_i = 1$ and $\bigwedge_i p_i = 2$ and none of the queries has been resolved. We construct the hard constraint such that (1) only valid abstractions are considered, (2) abstractions known to be unviable with respect to $\{q_1, q_2, q_3\}$ are avoided. For efficiency, we limit the domain of constants to the ones that appear in the existing derived tuples.

$$\psi_0 \ = \ \delta_{\mathbf{A}} \cup C \cup \{ \neg q_1 \vee \neg q_2 \vee \neg q_3 \}$$

The formula $\delta_{\mathbf{A}}$ encodes the space of valid abstractions which can be very large and even infinite. We introduce a new relation $[p_i > j]$ to represent the set of parameter settings where $p_i > j$. It serves two purposes: (1) it helps compactly represent the large number of unexplored abstractions without expanding the domain of constants with ones that do not appear in existing derived tuples, and (2) it allows $\delta_{\mathbf{A}}$ to grow linearly with the maximum p_i that the Datalog runs have tried so far rather than quadratically as a natively encoding

will do. We define $\delta_{\mathbf{A}}$ as $\bigcup_{1 \leq i \leq n} \delta^i_{\mathbf{A}}$ where

$$\begin{aligned}\delta^i_{\mathbf{A}} \stackrel{\Delta}{=} & \{\ [p_i = 1] \vee [p_i = 2] \vee [p_i > 2]\ \} \\ & \cup \{\ \neg([p_i > j] \wedge [p_i = j]) \mid 1 \leq j \leq 2\ \} \\ & \cup \{\ ([p_i > j] \vee [p_i = j]) \Leftrightarrow [p_i > j - 1] \mid j = 2\ \}\big)\end{aligned}$$

Formula $\delta^i_{\mathbf{A}}$ says that p_i can either take exactly one of the values that the Datalog runs have explored so far ($p_i = 1$ or $p_i = 2$) or an unexplored value ($p_i > 2$). When $p_i > 2$ is chosen, it is up to the designer to choose which concrete value to try next, as long as the chosen values form a cheapest abstraction among the valid ones that have not been shown as unviable. For example, if a larger p_i yields a more expensive analysis, the designer would choose $p_i = 3$ as the parameter. We use $\Gamma \in \mathcal{P}(\mathbb{T}) \uplus \{\mathsf{UNSAT}\} \mapsto \mathcal{P}(\mathbf{A}) \uplus \{\mathsf{impossible}\}$ where $\Gamma(\mathsf{UNSAT}) = \mathsf{impossible}$ to denote the user-defined function that converts the MAP inference result into an abstraction to try next.

It remains to construct the soft constraints. The first category of soft constraints give estimations of the costs of different abstractions. To a large extent, this is done based on knowledge about the efficiency characteristics of a particular analysis, and thus is left to the designer of the analysis. For example, if the designer knows from experience that the analysis tends to take longer as $\sum p_i$ increases, then they could include two kinds of soft constraints: (1) $[p_i = j]$ with weight $k-j$, for each $i \in [1, n]$ and $j \in [1, 2]$, and (2) $[p_i > j]$ with weight $k - j - 1$ for each $i \in [1, n]$ and $j = 2$. The former encodes the costs incurred by the parameter values that have been explored while the latter gives a lower bound to the costs incurred by the parameter values that have not been explored so far. We use $k - j - 1$ as the weight for $[p_i > j]$ as $p_i = j + 1$ would yield the cheapest setting among values in $p_i > j$.

The remaining soft constraints that we include are independent of the particular analysis, and they express that abstractions should be preferred when they could potentially help

with more queries. In this example, the extra soft constraints are $\neg q_1$, $\neg q_2$, $\neg q_3$, all with some large weight w. Suppose an abstraction A_1 is known to imply q_1, and an abstraction A_2 is known to imply both q_1 and q_2. Then A_1 is preferred over A_2 because of the last three soft constraints. Note that an abstraction is not considered at all only if it is known to imply all three queries, because of the hard constraint.

In general, choose is defined as follows:

$$\mathsf{choose}(T, C, Q) \triangleq \Gamma\big(\mathrm{MAP}\big(\delta_{\mathbf{A}} \cup C \cup \alpha(Q) \cup \eta_{\mathbf{A}} \cup \beta_{\mathbf{A}}(Q)\big)\big),$$

where the domain of constants of the constructed Markov Logic Network is $\mathsf{constants}(T)$. The boolean formula $\delta_{\mathbf{A}}$ encodes the set $\mathbf{A}$ of valid abstractions, the soft constraints $\eta_{\mathbf{A}}$ encode the efficiency preorder $\preceq$, and function Γ converts a MAP solution to an abstraction. Formally, we require $\delta_{\mathbf{A}}$, $\eta_{\mathbf{A}}$, and Γ to satisfy the conditions

$$T' = \mathrm{MAP}(\delta_{\mathbf{A}}) \quad \Longrightarrow \Gamma(T') \in \mathbf{A} \tag{1}$$

$$\forall A' \in \mathbf{A}.\exists T' = \mathrm{MAP}(\delta_{\mathbf{A}}).A' \preceq \Gamma(T') \wedge (\Gamma(T') \cap T = A' \cap T) \tag{2}$$

$$\Gamma(T_1) \preceq \Gamma(T_2) \quad \Longleftrightarrow W_{\eta_{\mathbf{A}}}(T_1) \leq W_{\eta_{\mathbf{A}}}(T_2) \tag{3}$$

where both T_1 and T_2 are MAP solutions to $\delta_{\mathbf{A}}$.

Condition (1) specifies that Γ always convert a map solution of $\delta_{\mathbf{A}}$ to a valid abstraction. Condition (2) specifies that for any given valid abstraction A', $\delta_{\mathbf{A}}$ encodes an abstraction that share the same set of abstraction tuples in T and is not more expensive; this condition ensures that if there exists a valid abstraction that avoids problematic derivations (that is, the ones that lead to queries) captured by C and T, then $\delta_{\mathbf{A}}$ encodes an abstraction that avoids the same derivations and is not more expensive. Condition (3) specifies that $\eta_{\mathbf{A}}$ captures the costs of different abstractions.

Table 3.3: Benchmark characteristics. All numbers are computed using a 0-CFA call-graph analysis.

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

Finally, the hard constraint $\alpha(Q)$ and the soft constraints $\beta_{\mathbf{A}}(Q)$ are

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

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

Discussion. The function $\mathsf{choose}(A, C, Q)$ reasons about a possibly very large set $\gamma(A, C, Q)$ of abstractions known to be unviable, and it does so by using the compact representation C where $\mathbb{N} = \mathbb{N}(C \cup A)$ of previous Datalog runs, together with several helper formulas, such as $\delta_{\mathbf{A}}$ and $\eta_{\mathbf{A}}$. Moreover, the reduction to a Markov Logic Network is natural and involves almost no transformation of the formulas involved.

Our algorithm provides a general algorithm to find a viable abstraction that satisfies a given metric. One only needs to replace the soft constraints η_A to extend it to other metrics rather than analysis costs. For example, to speed up the convergence of the overall algorithm, one can factor in the probability that a given abstraction can resolve a certain query [26].

3.1.5 Empirical Evaluation

In this section, we empirically evaluate our approach on real-world analyses and programs. The experimental setup is described in Chapter 3.1.5.1 and the evaluation results are discussed in Chapter 3.1.5.2.

3.1.5.1 Evaluation Setup

We evaluate our approach on two static analyses written in Datalog, a pointer analysis and a typestate analysis, for Java programs. We study the results of applying our approach to each of these analyses on eight Java benchmark programs described in Table 3.3. We analyzed the bytecode of the programs including the libraries that they use. The programs are from Ashes Suite and DaCapo Suite.

We implemented our approach using NICHROME and an open-source Datalog solver without any modification. Both our analyses are expressed and solved using `bddbddb` [15], a BDD-based Datalog solver. We use NICHROME to solve Markov Logic Networks generated by this algorithm. All our experiments were done using Oracle HotSpot JVM 1.6 on a Linux machine with 128GB memory and 3.0GHz processors. We next describe our two analyses in more detail.

Pointer Analysis. Our pointer analysis is flow-insensitive but context-sensitive, based on k-object-sensitivity [27]. It computes a call graph simultaneously with points-to results, because the precision of the call graph and points-to results is inter-dependent due to dynamic dispatching in OO languages like Java.

The precision and efficiency of this analysis depends heavily on how many distinctions it makes between different calling contexts. In Java, the value of the receiver object **this** provides context at runtime. But a static analysis cannot track concrete objects. Instead, static analyses typically track the allocation site of objects. In general, in object-sensitive analyses the abstract execution context is an *allocation string* $h_1, \ldots, h_k$. The site h_1 is where the receiver object was instantiated. The allocation string $h_2, \ldots, h_k$ is the context of h_1, defined recursively. Typically, all contexts are truncated to the same length k.

In our setting, an abstraction A enables finer control on how contexts are truncated: The context for allocation site h is truncated to length $A(h)$. Thus, truncation length may vary from one allocation site to another. This finer control allows us to better balance precision and efficiency. Abstractions are ordered as follows:

$$\text{(precision)} \quad A_1 \sqsubseteq A_2 \iff \forall h : \; A_1(h) \leq A_2(h)$$
$$\text{(efficiency)} \quad A_1 \preceq A_2 \iff \Sigma_h A_1(h) \geq \Sigma_h A_2(h)$$

These definitions reflect the intuition that making more distinctions between contexts increases precision but is more expensive.

Typestate Analysis. Our typestate analysis is based on that by Fink et al. [28]. It differs in three major ways from the pointer analysis described above. First, it is fully flow-sensitive, whereas the pointer analysis is fully flow-insensitive. Second, it is fully context-sensitive, using procedure summaries instead of cloning, and therefore capable of precise reasoning for programs with arbitrary call chain depth, including recursive ones. It is based on the tabulation algorithm [29] that we expressed in Datalog. Third, it performs both may- and must-alias reasoning; in particular, it can do strong updates, whereas our pointer analysis only does weak updates. These differences between our two analyses highlight the versatility of our approach.

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

Table 3.4: Results showing statistics of queries, abstractions, and iterations of our approach (**CURRENT**) and the baseline approaches (**BASELINE**) on the pointer analysis.

	queries			abstraction size		
	total	resolved		final	max.	iterations
		CURRENT	**BASELINE**			
toba-s	7	7	0	170	17,820	10
javasrc-p	46	46	0	470	18,450	13
weblech	5	5	2	140	30,950	10
hedc	47	47	6	730	29,480	18
antlr	143	143	5	970	29,170	15
luindex	138	138	67	1,160	40,550	26
lusearch	322	322	29	1,460	39,360	17
schroeder-m	51	51	25	450	58,260	15

as the abstraction. Different abstractions, however, do affect the precision and efficiency of the analysis, and are ordered as follows:

$$\text{(precision)} \quad A_1 \sqsubseteq A_2 \iff A_1 \subseteq A_2$$
$$\text{(efficiency)} \quad A_1 \preceq A_2 \iff |A_1| \geq |A_2|$$

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

Using the typestate analysis client, we compare our refinement approach to a scalable CEGAR-based approach for finding optimal abstractions proposed by Zhang et al. [30]. A similar comparison is not possible for the pointer analysis client since the work by Zhang et al. cannot handle non-disjunctive analyses. Instead, we compare the precision and scalability of our approach on the pointer analysis client with an optimized Datalog-based implementation of k-object-sensitive pointer analysis that uses $k = 4$ for all allocation sites in the program. Using a higher k value caused this baseline analysis to timeout on our larger benchmarks.

3.1.5.2 Evaluation Results

Table 3.4 and Table 3.5 summarize the results of our experiments. It shows the numbers of queries, abstractions, and iterations of our approach (**CURRENT**) and the baseline ap-

Table 3.5: Results showing statistics of queries, abstractions, and iterations of our approach (**CURRENT**) and the baseline approaches (**BASELINE**) on the typestate analysis.

	queries			abstraction size		
	total	resolved		final	max.	iterations
		CURRENT	**BASELINE**			
toba-s	543	543	62	14,781	15	159
javasrc-p	159	159	89	13,653	14	92
weblech	13	13	33	25,781	14	16
hedc	24	24	14	23,622	7	10
antlr	77	77	66	24,815	12	45
luindex	26	248	248	79	33,835	16
lusearch	45	45	74	33,526	13	52
schroeder-m	194	194	71	54,741	9	49

proaches (**BASELINE**) for each analysis and benchmark.

The 'total' column under queries shows the number of queries posed by the analysis on each benchmark. For the pointer analysis, each query corresponds to proving that a certain dynamically dispatching call site in the benchmark is monomorphic; i.e., it has a single target method. We excluded queries that could be proven by a context-insensitive pointer analysis. For the typestate analysis, each query corresponds to a typestate assertion. We tracked typestate properties for the objects from the same set of classes as used by Fink et al. [28] in their evaluation.

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

Table 3.4 and Table 3.5 give the abstraction size, which is an estimate of how costly it is to run an abstraction. An abstraction is considered to be more efficient when its size is smaller. But the *size of abstraction* A is defined differently for the two analyses: for pointer analysis, it is $\sum_h A(h)$; for typestate analysis, it is $|A|$. The 'max.' column shows the maximum size of an abstraction for the given program. For the pointer analysis, the maximum size corresponds to 10-object-sensitive analysis. for the typestate analysis, the maximum size corresponds to tracking all access paths. Even on the smallest benchmark, these costly analyses ran out of memory, emphasizing the need for our CEGAR approach. The 'final' column shows the size of the abstraction used in the last iteration. In all cases the size of the final abstraction is less than 5% of the maximum size.

The 'iterations' column shows the total number of iterations until all queries were solved. These numbers show that our approach is capable of exploring a huge space of abstractions for a large number of queries *simultaneously*, in under a few iterations. In comparison, the baseline approach (**BASELINE**) of Zhang et al. invokes the typestate client analysis far more frequently because it refines each query individually. For example, the baseline approach took 159 iterations to finish the typestate analysis on `toba-s`, while our approach only needed 15 iterations. Since the baseline for the pointer analysis client is not a refinement-based approach, it invokes the client analysis just once and is not comparable with our approach.

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

Performance of Datalog solver. Table 3.6 shows statistics of the running time of the Datalog solver in different iterations of our approach. These statistics include the minimum, maximum, and average running time over all iterations for a given analysis and benchmark. The numbers in Table 3.6 indicate that the abstractions chosen by our approach are small enough to allow the analyses to scale. For `schroeder-m`, one of our largest benchmarks, the change in running time from the slowest to the fastest run is only 2X for both client

Table 3.6: Running time (in seconds) of the Datalog solver in each iteration.

	pointer analysis				typestate analysis		
	BASELINE	min.	max.	avg.	min.	max.	avg.
toba-s	11	5	7	6	49	82	68.1
javasrc-p	29	7	11	9	76	152	120.8
weblech	2,574	44	54	47.5	121	172	146.6
hedc	5,058	21	37	27.9	52	58	54.3
antlr	3,723	30	55	39.3	193	325	264.8
luindex	913	59	84	76.4	311	512	426.7
lusearch	7,040	59	85	72.7	238	437	343.9
schroeder-m	23,038	192	428	289.6	1,778	2,681	2,304.6

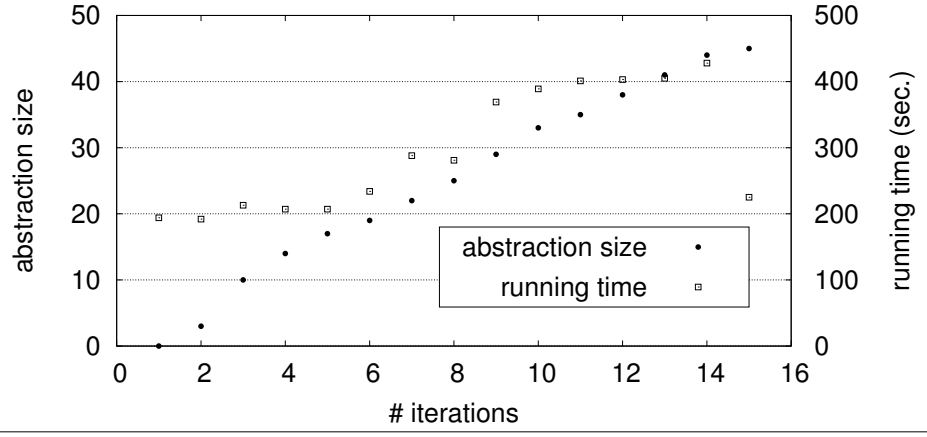


Figure 3.5: Running time of the Datalog solver and abstraction size for *pointer analysis* on `schroeder-m` in each iteration.

analyses. This further indicates that our approach is able to resolve all posed queries simultaneously before the sizes of the chosen abstractions start affecting the scalability of the client Datalog analyses. In contrast, the baseline k-object-sensitive analysis could only scale upto $k = 4$ on our larger benchmarks. Even with $k = 4$, the Datalog solver ran for over six hours on our largest benchmark when using the baseline approach. With our approach, on the other hand, the longest single run of the Datalog solver for the pointer analysis client was only seven minutes.

Figures 3.5 and 3.6 show the change in abstraction size and the analysis running time across iterations for the pointer and typestate analysis, respectively, applied on `schroeder-m`. There is a clear correlation between the growth in abstraction size and the increase in the running times. For both analyses, since our approach only chooses the cheapest viable ab-

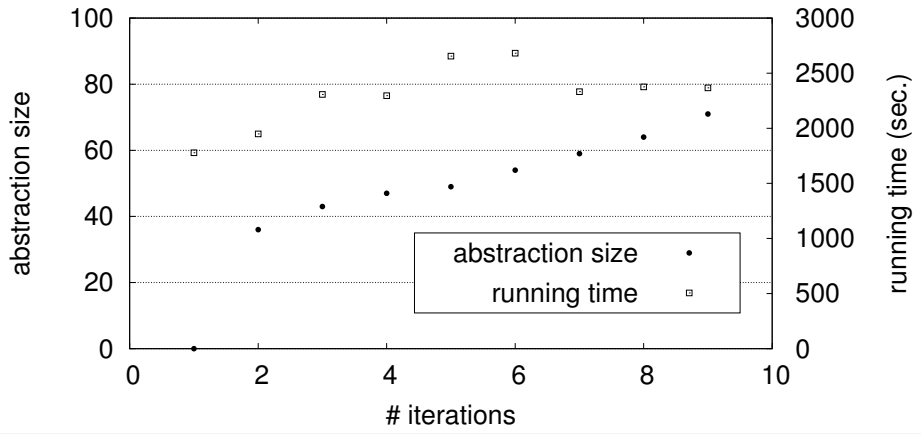


Figure 3.6: Running time of the Datalog solver and abstraction size for *typestate analysis* on `schroeder-m` in each iteration.

Table 3.7: Running time (in seconds) of the Markov Logic Network solver in each iteration.

	pointer analysis			typestate analysis		
	min.	max.	avg.	min.	max.	avg.
toba-s	2	7	3.1	1	6	3.1
javasrc-p	<1	4	1.6	2	19	6.4
weblech	5	11	6.7	3	8	5.3
hedc	1	23	3.7	1	2	1.7
antlr	11	44	24.1	5	27	13.25
luindex	8	48	16.3	6	26	14.7
lusearch	7	62	23.9	6	29	15.9
schroeder-m	34	257	114	37	308	138.6

straction in each iteration, the abstraction size grows almost linearly, as expected. Further, for typestate analysis, an increase in abstraction size typically results in an almost linear growth in the number of abstract states tracked. Consequently, the linear growth in the running time for the typestate analysis is also expected behavior. However, for the pointer analysis, typically, the number of distinct calling contexts grows exponentially with the increase in abstraction size. The linear curve for the running time in Figure 3.5 indicates that the abstractions chosen by our approach are small enough to limit this exponential growth.

Performance of Markov Logic Network solver. Table 3.7 shows statistics of the running time of the Markov Logic Network solver in different iterations of our approach. The metrics reported are the same as those for the Datalog solver. Although the performance of Markov Logic Network solvers is not completely deterministic, it is largely affected

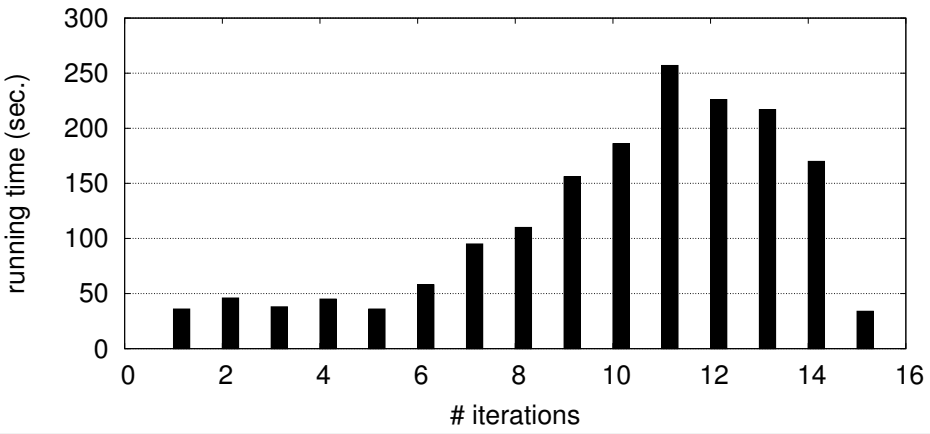


Figure 3.7: Running time of the Markov Logic Network solver for *pointer analysis* on `schroeder-m` in each iteration.

by two factors, (1) the size of the grounding of the instance posed to the solver, and (2) the structure of the constraints in this grounding. For both analyses, as seen previously, the abstraction size increases with the number of iterations while the number of unresolved queries decreases. Growth in abstraction size increases the complexity of the client Datalog analyses, causing an increase in the number of constraints in the grounding of the Markov Logic Network instance generated. On the other hand, fewer queries tends to simplify the structure of the constraints to be solved.

Figure 3.7 shows the running time of the Markov Logic Network solver across all iterations for the pointer analysis applied to our largest benchmark `schroeder-m`. Initially, the solver running time shows an increasing trend but this reverses towards the end. We believe that the two conflicting factors of size and structure of the constraints are at play here. While the complexity of the constraints increases initially due to the growing size of their grounding, after a certain iteration, the number of unresolved queries becomes small enough to suitably simplify the structure of the constraints and overwhelm the effect of growing grounding size. For the remaining benchmarks and analyses, we observed a similar trend, which we omit for the sake of brevity.

3.1.6 Related Work

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

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

CEGAR-based Model Checking. CEGAR was originally proposed to enable model checkers to scale to even larger state-spaces than those possible by symbolic approaches such as BDDs [4]. Our motivation for using CEGAR, in contrast, is to enable designers of analyses in Datalog to express flexible abstractions. Moreover, our notions of counterexamples and refined abstractions differ radically from those in model checkers.

Our work is most closely related to recent work on synthesizing software verifiers from proof rules for safety and liveness properties in the form of Horn-like clauses [31, 32, 33]. Their approach is also CEGAR-based but differs in two key ways: (1) they can identify internal nodes of derivations where information gets lost due to the current abstraction, which they subsequently refine, whereas we focus on leaf nodes of derivations; and (2) they use CEGAR to solve difficult Horn constraints formulated even on infinite domains, whereas we use CEGAR for finding a better abstraction, which is then used to generate new Horn constraints. As such, their approach is more expressive and flexible, but ours appears to scale better.

Zhang et al. [30] propose a CEGAR-based approach for efficiently finding an optimal abstraction in a parametric program analysis. Our approach improves on Zhang et al. in three aspects. First, their counterexample generation requires a parametric static analysis

to be disjunctive (which implies path-sensitivity), whereas any analysis written in Datalog, including non-disjunctive ones, can be handled by our approach. As a result, their approach is not applicable to the pointer analysis in Chapter 3.1.5. Second, their approach relies on a nontrivial backward analysis for analyzing a counterexample and selecting a next abstraction to try, but this backward analysis is not generic and should be designed for each parametric analysis. Our approach, on the other hand, uses a generic algorithm based on Markov Logic Networks for the counterexample analysis and the abstraction selection, which only requires users to define the cost model of abstractions. Conversely, [30] converges faster for certain problems. Finally, the approach in [30] cannot mix counterexamples across iterations to generate new counterexamples for free, a feature that is present in our approach as illustrated in Section 3.1.2.

Program Analysis Using Datalog. Recent years have witnessed a surge of interest in using Datalog for program analysis (see Chapter 3.1.1). Datalog solvers have simultaneously advanced, using either symbolic representations of relations such as BDDs (e.g., BDDBDDB [19] and Jedd [21]) or even explicit representations (e.g., Doop [20]). More recently the popular Z3 SMT solver has been extended to compute least fixpoints of constraints expressed in Datalog [22]. Our CEGAR approach is independent of the underlying Datalog solver and leverages these advances.

Liang et al. [34] propose a cause-effect dependence analysis technique for analyses in Datalog. The technique identifies input tuples that definitely do not affect output tuples. More specifically, it computes the transitive closure of all derivations of an output tuple to identify an over-approximation of the set of input tuples needed in any derivation (e.g., $\{t_1, t_2, t_3\}$). In contrast, our approach identifies the exact set of input tuples needed in each of even exponentially many derivations (e.g., $\{\{t_1\}, \{t_2, t_3\}\}$). Thus, in our example, their approach prunes abstractions that contain $\{t_1, t_2, t_3\}$, whereas ours also prunes those that only contain $\{t_1\}$ or $\{t_2, t_3\}$.

3.1.7 Conclusion

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

3.2 Interactive Verification

3.2.1 Introduction

Automated static analyses make a number of approximations. These approximations are necessary because the static analysis problem is undecidable in general [13]. However, they result in many false alarms in practice, which in turn imposes a steep burden on users of the analyses.

A common pragmatic approach to reduce false alarms involves applying heuristics to suppress their root causes. For instance, such a heuristic may ignore a certain code construct that can lead to a high false alarm rate [3]. While effective in alleviating user burden, however, such heuristics potentially render the analysis results unsound.

In this paper, we propose a novel methodology that synergistically combines a sound but imprecise analysis with precise but unsound heuristics, through user interaction. Our key idea is that, instead of directly applying a given heuristic in a manner that may unsoundly suppress false alarms, the combined approach poses questions to the user about the truth of root causes that are targeted by the heuristic. If the user confirms them, only then is the heuristic applied to eliminate the false alarms, with the user's knowledge.

To be effective, however, the combined approach must accomplish two key goals: *gen-*

eralization and *prioritization*. We describe each of these goals and how we realize them.

Generalization. The number of questions posed to the user by our approach should be much smaller compared to the number of false alarms that are eliminated. Otherwise, the effort in answering those questions could outweigh the effort in resolving the alarms directly. To realize this objective for analyses that produce many alarms, we observe that most alarms are often symptoms of a relatively small set of common root causes. For example, a method that is falsely deemed reachable can result in many false alarms in the body of that method. Inspecting this single root cause is easier than inspecting multiple alarms.

Prioritization. Since the number of false alarms resulting from different root causes may vary, the user might wish to only answer questions with relatively high payoffs. Our approach realizes this objective by interacting with the user in an iterative manner instead of posing all questions at once. In each iteration, the questions asked are chosen such that the expected payoff is maximized. Note that the user may be asked multiple questions per iteration because single questions may be insufficient to resolve any of the remaining alarms. Asking questions in order of decreasing payoff allows the user to stop answering them when diminishing returns set in. Finally, the iterative process allows incorporating the user's answers to past questions in making better choices of future questions, and thereby further amplify the reduction in user effort.

We formulate the problem to be solved in each iteration of our approach as a non-linear optimization problem called the *optimum root set problem*. This problem aims at finding a set of questions that maximizes the benefit-cost ratio in terms of the number of false alarms expected to be eliminated and the number of questions posed. We propose an efficient solution to this problem by reducing it to a sequence of Markov Logic Networks. Each Markov Logic Network encodes the dependencies between analysis facts, and weighs the benefits and costs of questioning different sets of root causes. Since our objective function is non-linear, we solve a sequence of Markov Logic Networks that performs a binary search

between the upper bound and lower bound of the benefit-cost ratio.

Our approach automatically generates the Markov Logic Network formulation for any analysis specified in a constraint language. Specifically, we target Datalog, a declarative logic programming language that is widely used in formulating program analyses [7, 8, 10, 9, 20, 19]. Our constraint-based approach also allows incorporating orthogonal techniques to reduce user effort, such as *alarm clustering* techniques that express dependencies between different alarms using constraints, possibly in a different abstract domain [35, 36].

We have implemented our approach in a tool called URSA and evaluate it on a static datarace analysis using a suite of 8 Java programs comprising 41-194 KLOC each. URSA eliminates 74% of the false alarms per benchmark with an average payoff of 12× per question. Moreover, URSA effectively prioritizes questions with high payoffs. Further, based on data collected from 40 Java programmers, we observe that the average time to resolve a root cause is only half of that to resolve an alarm. Together, these results show that our approach achieves significant reduction in user effort.

We summarize our contributions below:

1. We propose a new paradigm to synergistically combine a sound but imprecise analysis with precise but unsound heuristics, through user interaction.

2. We present an interactive algorithm that implements the paradigm. In each iteration, it solves the optimum root set problem which finds a set of questions with the highest expected payoff to pose to the user.

3. We present an efficient solution to the optimum root set problem using Markov Logic Networks for a general class of constraint-based analyses.

4. We empirically show that our approach eliminates a majority of the false alarms by asking a few questions only. Moreover, it effectively prioritizes questions with high payoffs.

```
1  public class FTPServer implements Runnable{
2    private ServerSocket serverSocket;
3    private List conList;
4
5    public void main(String args[]){
6      ...
7      startButton.addActionListener(
8      new ActionListener(){
9        public void actionPerformed(...){
10         new Thread(this).start();
11       }
12     });
13     stopButton.addActionListener(
14     new ActionListener() {
15       public void actionPerformed(...){
16         stop();
17       }
18     });
19   }
20
21   public void run(){
22     ...
23     while (runner != null){
24       Socket soc = serverSocket.accept();
25       connection = new RequestHandler(soc);
26       conList.add(connection);
27       new Thread(connection).start();
28     }
29     ...
30   }
31
32   private void stop(){
33     ...
34     for (RequestHandler con : conList)
35       con.close();
36     serverSocket.close();
37   }
38 }
```

```
39 public class RequestHandler implements Runnable{
40   private FtpRequestImpl request;
41   private Socket controlSocket;
42   ...
43
44   public RequestionHandler(Socket socket){
45     ...
46     controlSocket = socket;
47     request = new FtpRequestImpl();
48     request.setClientAddress(socket.getInetAddress());
49     ...
50   }
51
52   public void run(){
53     ...
54     // log client information
55      clientAddr = request.getRemoteAddress();
56      ...
57      input = controlSocket.getInputStream();
58      reader = new BufferedReader(new
            InputStreamReader(input));
59      while (!isConnectionClosed){
60        ...
61        commandLine = reader.readLine();
62        // parse and check permission
63        request.parse(commandLine);
64        // execute command
65        service(request);
66      }
67   }
68
69   public synchronized void close(){
70     ...
71     controlSocket.close();
72     controlSocket = null;
73     request.clear();
74     request = null;
75   }
76 }
```

Figure 3.8: Example Java program extracted from Apache FTP Server.

3.2.2 Overview

We illustrate our approach by applying a static race detection tool to the multi-threaded Java program shown in Figure 3.8, which is extracted from the open-source program Apache FTP server [37]. It starts by constructing a GUI in the `main` method that allows the administrator to control the status of the server. In particular, it creates a `startButton` and a `stopButton`, and registers the corresponding callbacks (on line 7-18) for switching the server on/off. When the administrator clicks the `startButton`, a thread starts (on line 10) to execute the `run` method of class `FTPServer`, which handles connections in a loop (on line 23-28). We refer to this thread as the *Server* thread. In each iteration of the loop, a fresh `RequestHandler` object is created to handle the incoming connection (on line 25). Then, it is added to field `conList` of `FTPServer`, which allows tracking all connections. Finally, a thread is started asynchronously to execute the `run` method of class `RequestHandler`.

Input Relations:

$\mathsf{access}(p, o)$	(program point p accesses some field of abstract object o)
$\mathsf{alias}(p_1, p_2)$	(program points p_1 and p_2 access same memory location)
$\mathsf{escape}(o)$	(abstract object o is thread-shared)
$\mathsf{parallel}(p_1, p_2)$	(program points p_1 and p_2 can be reached by different threads simultaneously)
$\mathsf{unguarded}(p_1, p_2)$	(program points p_1 and p_2 are not guarded by any common lock)

Output Relations:

$\mathsf{shared}(p)$	(program point p accesses thread-shared memory location)
$\mathsf{race}(p_1, p_2)$	(datarace between program points p_1 and p_2)

Rules:

$$\mathsf{shared}(p) \text{ :- } \mathsf{access}(p, o), \mathsf{escape}(o). \quad (1)$$

$$\mathsf{race}(p_1, p_2) \text{ :- } \mathsf{shared}(p_1), \ \mathsf{shared}(p_2), \mathsf{alias}(p_1, p_2), \mathsf{parallel}(p_1, p_2), \mathsf{unguarded}(p_1, p_2). \quad (2)$$

Figure 3.9: Simplified static datarace analysis in Datalog.

We refer to this thread as the *Worker* thread. We next inspect the `RequestHandler` class more closely. It has multiple fields that can be accessed from different threads and potentially cause dataraces. For brevity, we only discuss accesses to fields `request` and `controlSocket`, which are marked in bold in Figure 3.8. Both fields are initialized in the constructor of `RequestHandler` which is invoked by the Server thread on line 25. Then, they are accessed in the `run` method by the Worker thread. Finally, they are accessed and set to `null` in the `close` method which is invoked to clean up the state of `RequestHandler` when the current connection is about to be closed. The `close` method can be either invoked in the `service` method by the Worker thread when the thread finishes processing the connection, or be invoked by the Java GUI thread in the `stop` method (on line 35) when the administrator clicks the `stopButton` to shut down the entire server by invoking `stop` (on line 16). The accesses in `close` are guarded by a synchronization block (on line 69-75) to prevent dataraces between them.

The program has multiple harmful race conditions: one on `controlSocket` between line 72 and line 57, and three on `request` between line 74 and line 55, 63, and 65. More concretely, these two fields can be accessed simultaneously by the Worker thread in `run` and the Java GUI thread in `close`. The race conditions can be fixed by guarding lines 55-66 with a synchronization block that holds a lock on the `this` object.

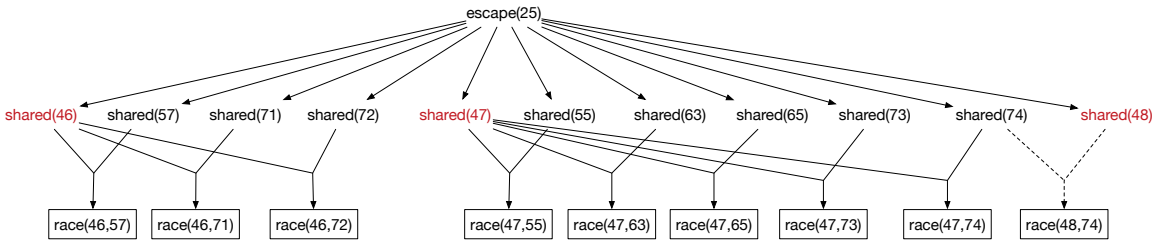


Figure 3.10: Derivation of dataraces in example program.

We next describe using a static analysis to detect these race conditions. In our implementation, we use the static datarace analysis from Chord [38], which is context- and flow-sensitive, and combines a thread-escape analysis, a may-happen-in-parallel analysis, and a lockset analysis. Here, for ease of exposition, we use a much simpler version of that analysis, shown in Figure 3.9. It comprises two logical inference rules in Datalog. Rule (1) states that the instruction at program point p accesses a thread-shared memory location if the instruction accesses a field of an object o, and o is thread-shared. Since these properties are undecidable, the rule over-approximates them using *abstract objects* o, such as object allocation sites. Rule (2) uses the thread-escape information computed by Rule (1) along with several input relations to over-approximate dataraces: instructions at p_1 and p_2, at least one of which is a write, may race if a) each of them may access a thread-shared memory location, b) both of them may access the same memory location, c) they may be reached from different threads simultaneously, and d) they are not guarded by any common lock. All input relations are in fact calculated by the analysis itself but we treat them as inputs for brevity.

Since the analysis is over-approximating, it not only successfully captures the four real dataraces but also reports nine false alarms. The derivation of these false alarms is shown by the graph in Figure 3.10. We use line numbers of the program to identify program points p and object allocation sites o. Each hyperedge in the graph is an instantiation of an analysis rule. In the graph, we elide tuples from all input relations except the ones in escape. For example, the dotted hyperedge is an instantiation of Rule (2): it derives $\mathsf{race}(48, 74)$, a race

between lines 48 and 74, from facts shared(48), shared(74), alias$(48, 74)$, parallel$(48, 74)$, and unguarded$(48, 74)$ which in turn are recursively derived or input facts.

Note that the above analysis does not report any datarace between accesses in the constructor of `RequestHandler`. This is because, by excluding tuples such as parallel$(46, 46)$, the parallel relation captures the fact that the constructor can be only invoked by the Server thread. Likewise, it does not report any datarace between accesses in the `close` method as it is able to reason about locks via the unguarded relation. However, there are still nine false alarms among the 13 reported alarms. To find the four real races, the user must inspect all 13 alarms in the worst case.

To reduce the false alarm rate, Chord incorporates various heuristics, one of which is particularly effective in reducing false alarms in our scenario. This heuristic can be turned on by running Chord with option `chord.datarace.exclude.init=true`, which causes Chord to ignore all races that involve at least one instruction in an object constructor. Internally, *the heuristic suppresses all* shared *tuples whose instruction lies in an object constructor*. Intuitively, most of such memory accesses are on the object being constructed which typically stays thread-local until the constructor returns. Indeed, in our scenario, shared(46), shared(47), and shared(48) are all spurious (as the object only becomes thread-shared on line 27) and lead to nine false alarms. However, applying the heuristic can render the analysis result unsound. For instance, the object under construction can become thread-shared inside the constructor by being assigned to the field of a thread-shared object or by starting a new thread on it. Applying the heuristic can suppress true alarms that are derived from shared tuples related to such facts.

We present a new methodology and tool, URSA, to address this problem. Instead of directly applying the above heuristic in a manner that may introduce false negatives, URSA poses questions to the user about root causes that are targeted by the heuristic. In our example, such causes are the shared tuples in the constructor of `RequestHandler`. If the user confirms such a tuple as spurious, only then it is suppressed in the analysis, thereby

eliminating all false alarms resulting from it.

URSA has two features that make it effective. First, it can eliminate a large number of false alarms by asking a few questions. The key insight is that most alarms are often symptoms of a relatively small set of common root causes. For instance, false alarms $\mathsf{race}(47, 55)$, $\mathsf{race}(47, 63)$, $\mathsf{race}(47, 65)$, $\mathsf{race}(47, 73)$, and $\mathsf{race}(47, 74)$ are all derived due to $\mathsf{shared}(47)$. Inspecting this single root cause is easier than inspecting all four alarms.

Second, URSA interacts with the user in an iterative manner, and prioritizes questions with high payoffs in earlier iterations. This allows the user to only answer questions with high payoffs and stop the interaction when the gain in alarms resolved has diminished compared to the effort in answering further questions.

URSA realizes the above two features by solving an optimization problem called the optimum root set problem, in each iteration. Recall that Chord's heuristic identified as potentially spurious the tuples $\mathsf{shared}(46)$, $\mathsf{shared}(47)$, $\mathsf{shared}(48)$; these appear in red in the derivation graph. We wish to ask the user if these tuples are indeed spurious, but not in an arbitrary order, because a few answers to well-chosen questions may be sufficient. We wish to find a small non-empty subset of those tuples, possibly just one tuple, which could rule out many false alarms. We call such a subset of tuples a *root set*, as the tuples in it are root causes for the potential false alarms, according to the heuristic. Each root set has an expected gain (how many false alarms it could rule out) and a cost (its size, which is how many questions the user will be asked). *The optimum root set problem is to find a root set that maximizes the expected gain per unit cost.* We refer to the gain per unit cost as the *payoff*. We next describe each iteration of URSA on our example. Here, we assume the user always answers the question correctly. Later we discuss the case where the user occasionally gives incorrect answers (Chapter 3.2.4.6 and Chapter 3.2.6.2).

Iteration 1. URSA picks $\{\mathsf{shared}(47)\}$ as the optimum root set since it has an expected payoff of 5, which is the highest among those of all root sets. Other root sets like $\{\mathsf{shared}(46), \mathsf{shared}(47)\}$ may resolve more alarms but end up with lower expected payoffs

due to larger numbers of required questions.

Based on the computed root set, URSA poses a single question about $\mathsf{shared}(47)$ to the user: "Does line 47 access any thread-shared memory location? If the answer is no, five races will be suppressed as false alarms." Here URSA only asks one question, but there are other cases in which it may ask multiple questions in an iteration depending on the size of the computed optimum root set. Besides the root set, we also present the corresponding number of false alarms expected to be resolved. This allows the user to decide whether to answer the questions by weighing the gains in reducing alarms against the costs in answering them.

Suppose the user decides to continue and answers no. Doing so labels $\mathsf{shared}(47)$ as false, which in turn resolves five race alarms $\mathsf{race}(47, 55)$, $\mathsf{race}(47, 63)$, $\mathsf{race}(47, 65)$, $\mathsf{race}(47, 73)$, and $\mathsf{race}(47, 74)$, highlighting the ability of URSA to generalize user input. Next, URSA repeats the above process, but this time taking into account the fact that $\mathsf{shared}(47)$ is labeled as false.

Iteration 2. In this iteration, we have two optimum root set candidates: $\{\mathsf{shared}(46)\}$ with an expected payoff of 3 and $\{\mathsf{shared}(48)\}$ with an expected payoff of 1. While resolving $\{\mathsf{shared}(46)\}$ is expected to eliminate $\{\mathsf{race}(46, 57), \mathsf{race}(46, 71), \mathsf{race}(46, 72)\}$, resolving $\{\mathsf{shared}(48)\}$ can only eliminate $\{\mathsf{race}(48, 74)\}$.

URSA picks $\{\mathsf{shared}(46)\}$ as the optimum root set and poses the following question to the user: "Does line 46 access any thread-shared memory location? If the answer is no, three races will be suppressed as false alarms." Compared to the previous iteration, the expected payoff of the optimum root set has reduced, highlighting URSA's ability to prioritize questions with high payoffs.

We assume the user answers no, which labels $\mathsf{shared}(46)$ as false. As a result, race alarms $\mathsf{race}(46, 57)$, $\mathsf{race}(46, 71)$, and $\mathsf{race}(46, 72)$ are eliminated. At the end of this iteration, eight out of the 13 alarms have been eliminated. Moreover, only one false alarm $\mathsf{race}(48, 74)$ remains.

$$a \in \mathbf{A} \subseteq \mathbb{T} \text{ (alarms)} \qquad q \in \mathbf{Q} \subseteq \mathbb{T} \text{ (potential causes)} \qquad f \in \mathbf{F} \subseteq \mathbf{Q} \text{ (causes)}$$

(a) Auxiliary definitions and notations.

$$\begin{aligned} [\![C]\!]_{\mathbf{F}} &:= \mathsf{lfp}\ SF^{\mathbf{F}}_{C} \\ SF^{\mathbf{F}}_{C}(T) &:= S_C(T) \setminus \mathbf{F} \\ S_C(T) &:= T \cup \{\, t \mid t \in s_c(T) \text{ and } c \in C \,\} \\ s_{l_0 \,:\!-\, l_1,\ldots,l_n}(T) &:= \{\, \sigma(l_0) \mid \sigma(l_1) \in T, \ldots, \sigma(l_n) \in T \,\} \end{aligned}$$

(b) Augmented semantics of Datalog.

Figure 3.11: Syntax and semantics of Datalog with causes.

Iteration 3. In this iteration, there is only one root set $\{\mathsf{shared}(48)\}$ which has an expected payoff of 1. Similar to the previous two iterations, URSA poses a single question about $\mathsf{shared}(48)$ to the user along with the number of false alarms expected to be suppressed. At this point, the user may prefer to resolve the remaining five alarms manually, as the expected payoff is too low and very few alarms remain unresolved. URSA terminates if the user makes this decision. Otherwise, the user proceeds to answer the question regarding $\mathsf{shared}(48)$ and therefore resolve the only remaining false alarm $\mathsf{race}(48, 74)$. At this point, URSA terminates as all three tuples targeted by the heuristic have been answered.

In summary, URSA resolves, in a sound manner, 8 (or 9) out of 9 false alarms by only asking two (or three) questions. Moreover, it prioritizes questions with high payoffs in earlier iterations.

3.2.3 The Optimum Root Set Problem

In this section, we define the Optimum Root Set problem, abbreviated ORS, in the context of static analyses expressed in Datalog.

3.2.3.1 Declarative Static Analysis

We say that an analysis is sound when it over-approximates the concrete semantics. We augment the semantics of Datalog in a way that allows us to control the over-approximation.

Figure 3.11(a) introduces three special subsets of tuples: alarms, potential causes, and

causes. An alarm corresponds to a bug report. A potential cause is a tuple that is identified by a heuristic as possibly spurious. A cause is a tuple that is identified by an oracle (e.g., a human user) as indeed spurious.

Figure 3.11(b) gives the augmented semantics of Datalog, which allows controlling the over-approximation with a set of causes $\mathbf{F}$. It is similar to the standard Datalog semantics except that in each iteration it removes tuples in causes $\mathbf{F}$ from derived tuples T (denoted by function $SF_C^{\mathbf{F}}$). Since a cause in $\mathbf{F}$ is never derived, it is not used in deriving other tuples. In other words, the augmented semantics do not only suppress the causes but also spurious tuples that can be derived from them.

3.2.3.2 Problem Statement

We assume the following are fixed: a set C of Datalog constraints, a set $\mathbf{A}$ of alarms, a set $\mathbf{Q}$ of potential causes, and a set $\mathbf{F}$ of confirmed causes. Informally, our goal is to grow $\mathbf{F}$ such that $[\![C]\!]_{\mathbf{F}}$ contains fewer alarms. To make this precise, we introduce a few terms. Given two sets $T_1 \supseteq T_2$ of tuples, we define ***the gain of*** T_2 ***relative to*** T_1, as the number of alarms that are in T_1 but not in T_2:

$$gain(T_1, T_2) := \left|(T_1 \setminus T_2) \cap \mathbf{A}\right|$$

A ***root set*** $\mathbf{R}$ is a non-empty subset of potential but unconfirmed causes: $\mathbf{R} \subseteq \mathbf{Q} \setminus \mathbf{F}$. Intuitively, the name is justified because we will aim to put in $\mathbf{R}$ the root causes of the false alarms. The potential causes in $\mathbf{R}$ are those we will investigate, and for that we have to pay a ***cost*** $|\mathbf{R}|$. The ***potential gain of*** $\mathbf{R}$ is the gain of $[\![C]\!]_{\mathbf{F}\cup\mathbf{R}}$ relative to $[\![C]\!]_{\mathbf{F}}$. The ***expected payoff of*** $\mathbf{R}$ is the potential gain per unit cost. (The ***actual*** payoff depends on which potential causes in $\mathbf{R}$ will be confirmed.) With these definitions, we can now formally introduce the ORS problem.

Problem 7 (**O**ptimum **R**oot **S**et)**.** Given a set C of constraints, a set $\mathbf{Q}$ of potential causes, a

set $\mathbf{F}$ of confirmed causes, and a set $\mathbf{A}$ of alarms, find a root set $\mathbf{R} \subseteq \mathbf{Q} \setminus \mathbf{F}$ that maximizes $gain\big([\![C]\!]_{\mathbf{F}}, [\![C]\!]_{\mathbf{F}\cup\mathbf{R}}\big)/|\mathbf{R}|$, which is the expected payoff.

3.2.3.3 Monotonicity

The definitions from the previous two subsections (Chapter 3.2.3.1 and Chapter 3.2.3.2) immediately imply some monotonicity properties. We list these here, because we refer to them in later sections.

Lemma 8. *If* $T_1 \subseteq T_2$ *and* $\mathbf{F}_1 \supseteq \mathbf{F}_2$*, then* $SF_C^{\mathbf{F}_1}(T_1) \subseteq SF_C^{\mathbf{F}_2}(T_2)$*. In particular, if* $\mathbf{F}_1 \supseteq \mathbf{F}_2$*, then* $[\![C]\!]_{\mathbf{F}_1} \subseteq [\![C]\!]_{\mathbf{F}_2}$*.*

Lemma 9. *If* $T_1 \subseteq T_1'$ *and* $T_2' \subseteq T_2$*, then* $gain(T_1', T_2') \geq gain(T_1, T_2)$*. Also,* $gain(T, T) = 0$ *for all* T*.*

3.2.3.4 NP-Completeness

Problem 7 is an optimization problem which can be turned into a decision problem in the usual way, by providing a threshold for the objective. The question becomes whether there exists a root set $\mathbf{R}$ that can achieve a given gain. This decision problem is clearly in NP: the set $\mathbf{R}$ is a suitable polynomial certificate.

The problem is also NP-hard, which we can show by a reduction from the minimum vertex cover problem. Given a graph $G = (V, E)$, a subset $U \subseteq V$ of vertices is said to be a ***vertex cover*** when it intersects all edges $\{i, j\} \in E$. The minimum vertex cover problem, which asks for a vertex cover of minimum size, is well-known to be NP-complete. We reduce it to the ORS problem as follows. We represent the graph G with three relations $\mathbf{R} = \{\texttt{vertex}, \texttt{edge}, \texttt{alarm}\}$. Without loss of generality, let V be the set $\{0, \ldots, n-1\}$, for some n. Thus, we identify vertices with Datalog constants. For each edge $\{i, j\} \in E$

we include two ground constraints:

$$\texttt{edge}(i, j) \texttt{ :- } \texttt{vertex}(i), \texttt{vertex}(j)$$

$$\texttt{alarm}() \texttt{ :- } \texttt{edge}(i, j)$$

We set $\mathbf{F} = \emptyset$, $\mathbf{A} := \{\texttt{alarm}()\}$, and $\mathbf{Q} := \{\, \texttt{vertex}(i) \mid i \in V \,\}$. Since $\mathbf{A}$ has size 1, the gain can only be 0 or 1. To maximize the ORS objective, the gain has to be 1. Further, the size of the root set $\mathbf{R}$ has to be minimized. But, one can easily see that a root set leads to a gain of 1 if and only if it corresponds to a vertex cover.

3.2.4 Interactive Analysis

Our interactive analysis combines three ingredients: (a) a static analysis; (b) a heuristic; and (c) an oracle. We require the static analysis to be implemented in Datalog, which lets us track dependencies. The requirements on the heuristic and the oracle are mild: given a set of Datalog tuples they must pick some subset. These are all the requirements. Suppose there is a *ground truth* that returns the truth of each analysis fact based on the concrete semantics. It helps to think of the three ingredients intuitively as follows:

(a) the static analysis is *sound*, *imprecise* with respect to the ground truth and *fast*;

(b) the heuristic is *unsound*, *precise* with respect to the ground truth and *fast*; and

(c) the oracle *agrees with* the ground truth and is *slow*.

Technically, we show a *soundness preservation* result (Theorem 13): if the given oracle agrees with the ground truth and the static analysis over-approximates it, then our combined analysis also over-approximates the ground truth. Soundness preservation holds with any heuristic. When a heuristic agrees with the ground truth, we say that it is *ideal*; when it almost agrees with the ground truth, we say that it is *precise*. Even though which heuristic one uses does not matter for soundness, we expect that more precise heuristics lead to better

Algorithm 2 Interactive Resolution of Alarms

INPUT: constraints C, and potential alarms $\mathbf{A}$
OUTPUT: remaining alarms

$\mathbf{Q} := \mathsf{Heuristic}()$
$\mathbf{F} := \emptyset$
while $\mathbf{Q} \neq \mathbf{F}$ **do**
  $\mathbf{R} := \mathsf{OptimumRootSet}(C, \mathbf{A}, \mathbf{Q}, \mathbf{F})$
  $Y, N := \mathsf{Decide}(\mathbf{R})$
  $\mathbf{Q} := \mathbf{Q} \setminus Y$
  $\mathbf{F} := \mathbf{Q} \setminus [\![C]\!]_{\mathbf{F} \cup N}$
end while
return $\mathbf{A} \cap [\![C]\!]_{\mathbf{F}}$

performance. We explore speed and precision later, through experiments (Chapter 3.2.6). Also later, we discuss what happens in practice when the static analysis is unsound or the oracle does not agree with the ground truth (Chapter 3.2.4.6).

The high level intuition is as follows. The heuristic is fast. But, since the heuristic might be unsound, we need to consult an oracle before relying on what the heuristic reports. However, consulting the oracle is an expensive operation, so we would like to not do it often. Here is where the static analysis specified in Datalog helps. On one hand, by analyzing the Datalog derivation, we can choose good questions to ask the oracle. On the other hand, after we obtain information from the oracle, we can propagate that information through the Datalog derivation, effectively finding answers to questions we have not yet asked.

We begin by showing how the static analysis, the heuristic, and the oracle fit together (Chapter 3.2.4.1), and why the combination preserves soundness Chapter 3.2.4.2). A key ingredient in our algorithm is solving the ORS problem. We do this solving a sequence of Markov Logic Networks (Chapter 3.2.4.3). Each Markov logic Network encodes the problem whether, given a Datalog derivation, a certain payoff is feasible (Chapter 3.2.4.4 and Section 3.2.4.5). Finally, we close with a short discussion (Chapter 3.2.4.6).

3.2.4.1 Main Algorithm

Algorithm 2 brings together our key ingredients:

(a) the set C of constraints and the set $\mathbf{A}$ of potential alarms represent the static analysis implemented in Datalog;

(b) Heuristic is the heuristic; and

(c) Decide is the oracle.

The key to combining these ingredients into an analysis that is fast, sound, and precise is the procedure OptimumRootSet, which solves Problem 7.

From now on, the set C of constraints and the set $\mathbf{A}$ of potential alarms should be thought of as immutable global variables: they do not change, and they will be available in subroutines even if not explicitly passed as arguments. In contrast, the set $\mathbf{Q}$ of potential causes and the set $\mathbf{F}$ of confirmed causes do change in each iteration, while maintaining the invariant $\mathbf{F} \subseteq \mathbf{Q}$. Initially, the invariant holds because no cause has been confirmed, $\mathbf{F} = \emptyset$.

In each iteration, we invoke the oracle for a set of potential causes that are not yet confirmed: we call $\mathsf{Decide}(\mathbf{R})$ for some non-empty $\mathbf{R} \subseteq \mathbf{Q} \setminus \mathbf{F}$. The result we get back is a partition (Y, N) of $\mathbf{R}$. In Y we have tuples that are in fact true, and therefore are not causes for false alarms; in N we have tuples that are in fact false, and therefore are causes for false alarms. We remove tuples Y from the set $\mathbf{Q}$ of potential causes (line 6); we insert tuples N into the set $\mathbf{F}$ of confirmed causes, and we also insert all other potential causes that may have been blocked by N (line 7).

At the end, we return the remaining alarms $\mathbf{A} \cap [\![C]\!]_{\mathbf{F}}$.

3.2.4.2 Soundness

For correctness, we require that the oracle is always right. In particular, the answers given by the oracle should be consistent: if asked repeatedly, the oracle should give the same

answer about a tuple. Formally, we require that there exists a partition of all tuples $\mathbb{T}$ into two subsets True and False such that

$$\mathsf{Decide}(T) \;=\; (T \cap \mathsf{True}, T \cap \mathsf{False}) \qquad\qquad \text{for all } T \subseteq \mathbb{T} \tag{3.1}$$

We refer to the set True as the ***ground truth***. We say that the analysis C is ***sound*** when

$$\mathbf{F} \subseteq \mathsf{False} \quad \text{implies} \quad [\![C]\!]_{\mathbf{F}} \supseteq \mathsf{True} \tag{3.2}$$

That is, a sound analysis is one that over-approximates, as long as we do not explicitly block a tuple in the ground truth.

Lemma 10. *A sound analysis C can derive the ground truth; that is,* $\mathsf{True} = [\![C]\!]_{\mathsf{False}}$.

The correctness of Algorithm 2 relies on the analysis C being sound and also on making progress in each iteration, which we ensure by requiring

$$\emptyset \;\subset\; \mathsf{OptimumRootSet}(C, \mathbf{A}, \mathbf{Q}, \mathbf{F}) \;\subseteq\; \mathbf{Q} \setminus \mathbf{F} \tag{3.3}$$

Lemma 11. *In Algorithm 2, suppose that* $\mathsf{Heuristic}$ *returns all tuples* $\mathbb{T}$*. Also, assume* (3.1)*,* (3.2)*, and* (3.3)*. Then, Algorithm 2 returns the true alarms* $\mathbf{A} \cap \mathsf{True}$*.*

Proof sketch. The key invariant is that

$$\mathbf{F} \;\subseteq\; \mathsf{False} \;\subseteq\; \mathbf{Q} \tag{3.4}$$

The invariant is established by setting $\mathbf{F} := \emptyset$ and $\mathbf{Q} := \mathbb{T}$. By (3.1), we know that $Y \subseteq \mathsf{True}$ and $N \subseteq \mathsf{False}$, on line 5. It follows that the invariant is preserved by removing Y from $\mathbf{Q}$, on line 6. It also follows that $\mathbf{F} \cup N \subseteq \mathsf{False}$ and, by (3.2), that $[\![C]\!]_{\mathbf{F} \cup N} \supseteq \mathsf{True}$. So, the invariant is also maintained by line 7. We conclude that (3.4) is indeed an invariant.

Because $\mathbf{R}$ is nonempty, the loop terminates (details in appendix). When it does, we have $\mathbf{F} = \mathbf{Q}$. Together with the invariant (3.4), we obtain that $\mathbf{F} = \mathsf{False}$. By Lemma 10, it follows that Algorithm 2 returns $\mathbf{A} \cap \mathsf{True}$. □

We can relax the requirement that Heuristic returns $\mathbb{T}$ because of the following result:

Lemma 12. *Consider two heuristics which compute, respectively, the sets $\mathbf{Q}_1$ and $\mathbf{Q}_2$ of potential causes. Let A_1 and A_2 be the corresponding sets of alarms computed by 2. If $\mathbf{Q}_1 \subseteq \mathbf{Q}_2$, then $A_1 \supseteq A_2$.*

Proof sketch. Use the same argument as in the proof of Lemma 11, but replace the invariant (3.4) with

$$\mathbf{F} \;\subseteq\; \mathbf{Q}_0 \cap \mathsf{False} \;\subseteq\; \mathbf{Q}$$

where $\mathbf{Q}_0 \in \{\mathbf{Q}_1, \mathbf{Q}_2\}$ is the initial value of $\mathbf{Q}$. □

From Lemmas 11 and 12, we conclude that 2 is sound, for all heuristics.

Theorem 13. *Assume that there is a ground truth, the analysis C is sound, and* OptimumRootSet *makes progress; that is, assume* (3.1)*,* (3.2)*, and* (3.3)*. Then, 2 terminates and it returns at least the true alarms $\mathbf{A} \cap \mathsf{True}$.*

Observe that there is no requirement on Heuristic. If Heuristic always returns $\emptyset$, then our method degenerates to simply using the imprecise but fast static analysis implemented in Datalog. If Heuristic always returns $\mathbb{T}$ and the oracle is a precise analysis, then our method degenerates to an iterative combination between the imprecise and the precise analyses. Usually, we would use a heuristic that has demonstrated its effectiveness in practice, even though it may lack theoretical guarantees for soundness. In our approach, soundness is inherited from the static analysis specified in Datalog and from the oracle. If the oracle is unsound, perhaps because the user makes mistakes, then so is our analysis. Thus, Theorem 13 is a *relative* soundness result.

Algorithm 3 OptimumRootSet

INPUT: constraints C, potential alarms $\mathbf{A}$, potential causes $\mathbf{Q}$, causes $\mathbf{F}$
OUTPUT: root set $\mathbf{R}$ with maximum payoff

```
ratios := Sorted({ a/b | a ∈ {0,...,|A|} and b ∈ {1,...,|Q \ F|} })
i := 0    k := |ratios|        {ratios is an array indexed from 0, and |ratios| is its length}
while i + 1 < k do
    j := ⌊(i + k)/2⌋
    if IsFeasible(ratios[j], C, A, Q, F) then
        i := j
    else
        k := j
    end if
end while
return FeasibleSet(ratios[i], C, A, Q, F)
```

3.2.4.3 Finding an Optimum Root Set

We want to find a root set $\mathbf{R}$ that maximizes expected payoff

$$\frac{gain\big([\![C]\!]_{\mathbf{F}}, [\![C]\!]_{\mathbf{F}\cup\mathbf{R}}\big)}{|\mathbf{R}|} = \frac{\big|\big([\![C]\!]_{\mathbf{F}} \setminus [\![C]\!]_{\mathbf{F}\cup\mathbf{R}}\big) \cap \mathbf{A}\big|}{|\mathbf{R}|}$$

This expression is nonlinear, so it is not obvious how to maximize it by using a solver for linear programs. Our solution is to do a binary search on the payoff values, as seen in Algorithm 3. Given a gain a, a cost b, an analysis of constraints C and alarms $\mathbf{A}$, potential causes $\mathbf{Q}$, and causes $\mathbf{F}$, we find out if a payoff a/b is feasible by calling $\mathsf{IsFeasible}(a/b, C, \mathbf{A}, \mathbf{Q}, \mathbf{F})$, which we will implement by solving a Markov Logic Network (Chapter 3.2.4.5). Similarly, we implement $\mathsf{FeasibleSet}(a/b, C, \mathbf{A}, \mathbf{Q}, \mathbf{F})$ using a Markov Logic Network solver, to find a root set $\mathbf{R} \subseteq \mathbf{Q} \setminus \mathbf{F}$ with payoff $\geq a/b$. We require, as a precondition, that $\mathbf{Q} \setminus \mathbf{F}$ is nonempty. The array $ratios$ contains all possible payoff values, sorted.

3.2.4.4 From Augmented Datalog to Markov Logic Network

Let us begin by encoding the augmented Datalog semantics (Figure 3.11(b)) into Markov Logic Networks: Given $\mathbf{F}$, we want the Markov Logic Network solver to compute $[\![C]\!]_{\mathbf{F}}$.

Standard Datalog semantics correspond to the case $\mathbf{F} = \emptyset$. So, to find $[\![C]\!]_{\emptyset}$ and all

rule instances $t_0 \text{ :- } t_1, \ldots, t_n$, we start by running a Datalog solver once. We set $\mathbb{T} := [\![C]\!]_\emptyset$. Knowing the result of the standard Datalog solver, the task is to construct a Markov Logic Network that would compute $[\![C]\!]_{\mathbf{F}}$ for an arbitrary $\mathbf{F}$. For each relation $r \in \mathbb{R}$, we introduce new relations X_r, Y_r, Z_r. Consequently, we introduce new tuples x_t, y_t, z_t for each tuple t. Let $\mathbf{X}$ be the union of X_r's computed by the Markov Logic Network solver, and we define $\mathbf{Y}$ and $\mathbf{Z}$ similarly. We will construct our Markov logic Network such that $\mathbf{X} = [\![C]\!]_{\mathbf{F}}$, $\mathbf{Y} = S_C(\mathbf{X})$, and $\mathbf{Z} = \mathbf{F}$. We encode $\mathbf{Z} = \mathbf{F}$ by having hard constraints

$$\begin{aligned} z_t \qquad & \text{for each } t \in \mathbf{F} \\ \neg z_t \qquad & \text{for each } t \notin \mathbf{F} \end{aligned}$$

We encode $\mathbf{Y} \supseteq S_C(\mathbf{X})$ by having constraints

$$X_1 \wedge \ldots \wedge X_n \implies Y_0 \qquad \text{for each } l_0 \text{ :- } l_1, \ldots, l_n$$

where X_i corresponds to the relation in l_i. Note we use the same list of arguments in X_i as the ones used in l_i, which we elide for elaboration.

Finally, we encode $\mathbf{X} \supseteq \mathbf{Y} \setminus \mathbf{Z}$ by

$$Y_r \wedge \neg Z_r \implies X_r \qquad \text{for each } r \in \mathbb{R}.$$

Observe that $\mathbf{X} \supseteq \mathbf{Y} \setminus \mathbf{Z}$ together with $\mathbf{Y} \supseteq S_C(\mathbf{X})$ imply that $\mathbf{X} \supseteq SF_C^{\mathbf{Z}}(\mathbf{X})$; that is, $\mathbf{X}$ is a post fixed point of $SF_C^{\mathbf{Z}}$. By Lemma 8 and the Knaster–Tarski theorem, the least post fixed point of $SF_C^{\mathbf{Z}}$ coincides with its least fixed point. Thus, to guarantee that $\mathbf{X} = [\![C]\!]_{\mathbf{F}}$, it only remains to add soft constraints $\{\neg X_r \text{ weight } 1 \mid r \in \mathbb{R}\}$ to minimize $|\mathbf{X}|$. Note $\mathbf{X} = [\![C]\!]_{\mathbf{F}}$ implies $\mathbf{X} = SF_C^{\mathbf{Z}}(\mathbf{X})$, which together with $\mathbf{X} \supseteq \mathbf{Y} \setminus \mathbf{Z} \supseteq SF_C^{\mathbf{Z}}(\mathbf{X})$ imply that $\mathbf{Y} = S_C(\mathbf{X})$.

We use the Markov Logic Network solver as follows. Given a set $\mathbf{F}$, we create con-

$$X_1 \wedge \ldots \wedge X_n \implies Y_0 \qquad \text{for each } l_0 \text{ :- } l_1, \ldots, l_n \tag{3.1}$$

$$Y_r \wedge \neg Z_r \implies X_r \qquad \text{for each } r \in \mathbb{R} \tag{3.2}$$

$$\neg z_t \qquad \text{for } t \notin \mathbf{Q} \tag{3.3}$$

$$z_t \qquad \text{for } t \in \mathbf{F} \tag{3.4}$$

$$\bigvee_{t \in \mathbf{Q} \setminus \mathbf{F}} z_t \qquad \text{(requires } |\mathbf{R}| > 0\text{)} \tag{3.5}$$

$$b \sum_{t \in \mathbf{A} \cap [\![C]\!]_{\mathbf{F}}} (1 - x_t) - a \sum_{t \in \mathbf{Q} \setminus \mathbf{F}} z_t \geq 0 \qquad \text{(requires payoff} \geq a/b\text{)} \tag{3.6}$$

Figure 3.12: Implementing IsFeasible and FeasibleSet by solving a Markov Logic Network. All x_t, y_t, z_t are tuples except that in 3.6, they are variables taking values in $\{0, 1\}$ which represent whether the corresponding tuples are derived.

straints as above. Then we run the Markov Logic Network solver, which will return a set of tuples. The desired $[\![C]\!]_{\mathbf{F}}$ is encoded in the derived tuples $\{x_t\}_{t \in \mathbb{T}}$.

3.2.4.5 Feasible Payoffs

In this section, we adjust the Markov Logic Network encoding of augmented Datalog so that we can implement the subroutines IsFeasible and FeasibleSet used in Algorithm 3. Given sets $\mathbf{A}, \mathbf{Q}, \mathbf{F}$ and a rational payoff a/b, we want to decide whether there exists a nonempty root set $\mathbf{R} \subseteq \mathbf{Q} \setminus \mathbf{F}$ that achieves the payoff (IsFeasible), and find an example of such a set (FeasibleSet). In other words, we want to find a set $\mathbf{R}$ such that

$$b \cdot \left| \left([\![C]\!]_{\mathbf{F}} \setminus [\![C]\!]_{\mathbf{F} \cup \mathbf{R}} \right) \cap \mathbf{A} \right| - a \cdot |\mathbf{R}| \geq 0$$

and $|\mathbf{R}| > 0$. The resulting encoding appears in Figure 3.12. As before, we introduce new relations X_r, X_r, X_r and new tuples x_t, y_t, z_t, but with slightly different meanings: The constraints are set up such that $\mathbf{X} \supseteq [\![C]\!]_{\mathbf{Z}}$ and $\mathbf{Z} = \mathbf{F} \cup \mathbf{R}$. As before (Chapter 3.2.4.4), constraints (1)-(4) in Figure 3.12 encode $\mathbf{Y} \supseteq S_C(\mathbf{X})$ and $\mathbf{X} \supseteq \mathbf{Y} \setminus \mathbf{Z}$, which imply that $\mathbf{X}$ is a post fixed point of $SF_C^{\mathbf{Z}}$. We dropped the optimization objective that ensured we compute least fixed points, which is why $\mathbf{X} \supseteq [\![C]\!]_{\mathbf{Z}}$ rather than $\mathbf{X} = [\![C]\!]_{\mathbf{Z}}$. However, using Lemma 9, we can show that there exists a post fixed point of $SF_C^{\mathbf{Z}}$ that leads to the

payoff a/b (that is, satisfies constraints (5) and (6) in Figure 3.12) if and only if the least fixed point of $SF_C^{\mathbf{Z}}$ leads to the payoff. So, the minimization of $|\mathbf{X}|$ is unnecessary. Note, for elaboration, we write the payoff constraint (6) as a linear constraint. It can be encoded as Markov Logic Network constraints in a way akin to standard approaches for converting linear constraints to SAT constraints [39].

Example. We show the IsFeasible encoding for the second iteration of the example from Chapter 3.2.2.

We first show the encoding that corresponds to constraints (1) in Figure 3.12. Recall the analysis rules from Figure 3.9, we can encode them using the following hard constraints:

$$X_{\mathsf{access}}(p, o) \wedge X_{\mathsf{escape}}(o) \implies Y_{\mathsf{shared}}(p)$$

$$X_{\mathsf{shared}}(p_1) \wedge X_{\mathsf{shared}}(p_2) \wedge X_{\mathsf{alias}}(p_1, p_2) \wedge X_{\mathsf{parallel}}(p_1, p_2) \wedge X_{\mathsf{unguarded}}(p_1, p_2) \implies Y_{\mathsf{race}}(p_1, p_2).$$

We also model the inputs with unit hard constraints such as $X_{\mathsf{escape}}(25)$.

Then, for every relation r in Figure 3.9, we add $Y_r \wedge \neg Z_r \implies X_r$ (e.g., $Y_{\mathsf{shared}}(p) \wedge \neg Z_{\mathsf{shared}}(p) \implies X_{\mathsf{shared}}(p)$), which correspond to constrains (2) in Figure 3.12.

The potential causes returned by the heuristic are $\mathbf{Q} = \{\mathsf{shared}(46), \mathsf{shared}(47), \mathsf{shared}(48)\}$. At the beginning of the second iteration one cause had been confirmed, $\mathbf{F} = \{\mathsf{shared}(47)\}$. We have z_t for $t \in \mathbf{F}$, and $\neg z_t$ for $t \notin \mathbf{Q}$, which correspond to constraints (4) and (3) in Figure 3.12 respectively.

For constraints (5), we have $Z_{\mathsf{shared}}(6) \vee Z_{\mathsf{shared}}(9)$.

Finally,

$$b \cdot (1 - X_{\mathsf{race}}(46, 57) + 1 - X_{\mathsf{race}}(46, 71) + 1 - X_{\mathsf{race}}(46, 72) + 1 - X_{\mathsf{race}}(48, 74)) - a \cdot (Z_{\mathsf{shared}}(46) + Z_{\mathsf{shared}}(48)) \geq 0$$

where the X tuples range over the four unresolved alarms at the beginning of the second iteration. □

Let us see what we need to write down the constraints from Figure 3.12. First, we need the result of running a standard Datalog solver: the set $[\![C]\!]_\emptyset$, and the corresponding constraints C. We obtain these by running the Datalog solver once, in the beginning. Second, we need the sets $\mathbf{A}$, $\mathbf{Q}$, and $\mathbf{F}$. The set $\mathbf{A}$ is fixed, since it only depends on the analysis. Sets $\mathbf{Q}$ and $\mathbf{F}$ do change, but they are sent as arguments to IsFeasible and FeasibleSet. Third, we need the ratio a/b, which is also sent as an argument. Fourth, we need the set $[\![C]\!]_\mathbf{F}$; we can compute it as described earlier (Chapter 3.2.4.4).

If the Markov Logic Network solver finds the instance to be feasible, then it gives us a set of output tuples, including tuples in $\mathbf{Z}$. To implement FeasibleSet, we compute $\mathbf{R}$ as $\mathbf{Z} \setminus \mathbf{F}$.

3.2.4.6 Discussion

First, we discuss a few alternatives for the payoff, for the termination condition, and for the oracle. Then, we discuss the issue of soundness from the point of view of the mismatch between theory and practice. Finally, we discuss how our approach could be applied in a non-Datalog setting.

Payoff The algorithm presented above uses $|\mathbf{R}|$ as the cost measure. It might be, however, that some potential causes are more difficult to investigate than others. If the user provides us with an expected cost of investigating each potential cause, then we can adapt our algorithm, in the obvious way, so that it prefers calling Decide on cheap potential causes.

In situations when multiple root sets have the same maximum payoff, we may want to prefer one with minimum size. Intuitively, small root sets let us gather information from the oracle quickly. To encode a preference for smaller root sets, we can extend the constraints in Figure 3.12 with soft constraints $\{\ \neg z_t\ \ \mathsf{weight}\ 1 \mid t \in \mathbf{Q} \setminus \mathbf{F}\ \}$.

Termination Algorithm 2 terminates when all potential causes have been investigated; that is, when $\mathbf{Q} = \mathbf{F}$. Another option is to look at the current value of the expected payoff. Suppose Algorithm 3 finds an expected payoff ≤ 1. This means that we expect that it will *not* be easier to investigate potential causes rather than investigate the alarms themselves. So, we could decide to terminate, as we do in our experiments (Chapter 3.2.6). Finally, instead of looking at the expected payoff computed by the Markov Logic Network solver, we could track the actual payoff for each iteration. Then, we could stop if the average payoff in the last few iterations is less than some threshold.

Oracle. By default, the oracle Decide is a human user. However, we can also use a precise yet expensive analysis as the oracle, which enables an alternative use case of our approach. This use case focuses on balancing the overall precision and scalability of combining the base analysis and the oracle analysis, rather than reducing user effort in resolving alarms. Our approach allows the oracle analysis to focus on only the potential causes that are relevant to the alarms, especially the ones with high expected payoffs. For example, the end user might find it too long a time to apply the oracle analysis to resolve all potential causes. By applying our approach, they can use the oracle analysis to only answer the potential causes with high payoffs and resolve the rest alarms via other methods (e.g., manual inspection). Appendix B includes a more detailed discussion of this use case.

Soundness. In practice, most static analyses are unsound [40]. In addition, in our setting, the user may answer incorrectly. We discuss each of these issues below.

Many program analyses are designed to be sound in theory but are unsound in practice due to engineering compromises. If certain language features were handled in a sound manner, then the analysis would be unscalable or exceedingly imprecise. For example, in Java, such features include reflection, dynamic class loading, native code, and exceptions. If we start from an unsound static analysis, then our interactive analysis is also unsound. However, Theorem 13 gives evidence that we do not introduce new sources of unsoundness.

More importantly, our approach is still effective in suppressing false alarms and therefore reduces user effort, as we demonstrate through experiments (Chapter 3.2.6).

In theory, the oracle never makes mistakes; in practice, users do make mistakes, of two kinds: they may label a true tuple as spurious, and they may label a spurious tuple as true. If a true tuple is labeled as spurious, then the interactive analysis becomes unsound. However, if the user answers $x\%$ of questions incorrectly, then we expect that the fraction of false negatives is not far from $x\%$. Our experiments show that, even better, the fraction of false negatives tends to be less than $x\%$. A consequence of this observation is that, if potential causes are easier to inspect than alarms, then our approach will decrease the chances that real bugs are missed.

If a spurious tuple is labeled as true, then the interactive analysis may ask more questions. It is also possible that fewer false alarms are filtered out: the user could make the mistake on the only remaining question with expected payoff > 1, which in turn would cause the interaction to terminate earlier. (See the previous discussion on termination.)

Later, we analyze both kinds of mistakes quantitatively (Chapter 3.2.6.2 and Table 3.9).

Non-Datalog Analyses. We focus on program analyses implemented in Datalog for two reasons: (1) it is easy to capture provenance information for such analyses; and (2) there is a growing trend towards specifying program analyses in Datalog [20, 10, 11, 8, 9]. However, not all analyses are implemented in Datalog; for example, see [3, 41, 42]. In principle, it is possible to apply our approach to any program analysis. To do so, the analysis designer would need to figure out how to capture provenance information of an analysis' execution. This might not be an easy task, but it is a one-time effort.

3.2.5 Instance Analyses

We demonstrate our approach on the datarace analysis from Chord, a static datarace detector for Java programs. To show the generality and versatility of our approach, Appendix B

describes its instantiation on a pointer analysis for the alternative use case, where the oracle is a precise yet expensive static analysis. Next, we briefly describe the datarace analysis, its notions of alarms and causes, and our implementation of the procedure Heuristic.

The datarace analysis is a context- and flow-sensitive analysis introduced by [43]. It comprises 30 rules, 18 input relations, and 18 output relations. It combines a thread-escape analysis, a may-happen-in-parallel analysis, and a lockset analysis. It reports a datarace between each pair of instructions that access the same thread-shared object, are reachable by different threads in parallel, and are not guarded by a common lock.

While the alarms are the datarace reports, the potential causes, which are identified by Heuristic, could be any set of tuples in theory. However, we found it useful to focus our heuristics on two relations: `shared` and `parallel`, which we observe to often contain common root causes of false alarms. The `shared` relation contains instructions that may access thread-shared objects; the `parallel` relation contains instruction pairs that may be reachable in parallel. We call the set of all tuples from these two relations the ***universe of potential causes***.

We provide four different Heuristic instantiations, which are shown in Figure 3.13. Instantiations `static_optimistic` and `static_pessimistic` contain static rules that reflect analysis designers' intuition. Instantiation `dynamic` leverages a dynamic analysis to identify analysis facts that are likely spurious. Finally, instantiation `aggregated` combines the previous three instantiations using a decision tree. We next describe each instantiation in detail.

Instantiation `static_optimistic` encodes a heuristic applied in the implementation by [43], which treats `shared` tuples whose associated access occurs in an object constructor as spurious. Moreover, it includes `parallel` tuples related to instructions in `Thread.run()` and `Runnable.run()` in the potential causes as they are often falsely derived due to a context-insensitive call-graph. Instantiation `static_pessimistic` is similar to `static_optimistic` except it aggressively classifies all `shared` tuples as false,

$$
\begin{aligned}
\texttt{static\_optimistic}() = {} & \{\texttt{shared}(i) \mid \text{instruction } i \text{ is in a constructor}\} \cup \\
& \{\texttt{parallel}(i, t_1, t_2) \mid \text{instruction } i \text{ is in } \texttt{java.lang.Thread.run}() \text{ or} \\
& \qquad \texttt{java.lang.Runnable.run}())\} \\
\texttt{static\_pessimistic}() = {} & \{\texttt{shared}(i) \mid i \text{ is an instruction}\} \cup \\
& \{\texttt{parallel}(i, t_1, t_2) \mid \text{instruction } i \text{ is in } \texttt{java.lang.Thread.run}() \text{ or} \\
& \qquad \texttt{java.lang.Runnable.run}())\} \\
\texttt{dynamic}() = {} & \{\texttt{shared}(i) \mid \text{Instruction } i \text{ is executed and only accesses thread-local} \\
& \qquad \text{objects during the runs}\} \cup \\
& \{\texttt{parallel}(i, t_1, t_2) \mid \text{whenever thread } t_1 \text{ executes instruction } i \text{ in} \\
& \qquad \text{the run, } t_2 \text{ is not running}\} \\
\texttt{aggregated}() = {} & \texttt{decisionTree}(\texttt{dynamic}, \texttt{static\_optimistic}, \texttt{static\_pessimistic})
\end{aligned}
$$

Figure 3.13: Heuristic instantiations for the datarace analysis.

capturing the intuition that most accesses are thread-local.

When applying our approach, one might lack intuitions like the above ones to effectively identify potential causes. In this case, they can leverage the power of testing, akin to work on discovering likely invariants [44]: if an analysis fact is not observed consistently across different runs, then it is very likely to be false. We provide instantiation `dynamic` to capture this intuition. It leverages a dynamic analysis and returns tuples whose associated program points are reached but associated program facts are not observed across runs.

Having multiple Heuristic instantiations is not unusual; we gain the benefits of each of them using a combined instantiation `aggregated`, which decides whether to classify each given tuple as a potential cause by considering the results of all the individual instantiations. Instantiation `aggregated` realizes this idea by using a decision tree that aggregates the results of the other instantiations. We obtain such a decision tree by training it on benchmarks where the tuples in the universe of potential causes are fully labeled.

3.2.6 Empirical Evaluation

This section evaluates the effectiveness of our approach by applying it to the datarace analysis on a suite of 8 Java programs. In addition, Appendix B discusses the evaluation results for the use case where the oracle is a precise yet expensive analysis, by applying our approach to the pointer analysis on the same benchmark suite.

Table 3.8: Benchmark characteristics. Column $|\mathbf{A}|$ shows the numbers of alarms. Column $|\mathbf{Q}_U|$ shows the sizes of the universes of potential causes, where k stands for thousands. All the reported numbers except for $|\mathbf{A}|$ and $|\mathbf{Q}_U|$ are computed using a 0-CFA call-graph analysis.

	# Classes		# Methods		Bytecode (KB)		Source (KLOC)		$\|\mathbf{A}\|$			$\|\mathbf{Q}_U\|$
	app	total	app	total	app	total	app	total	false	total	false%	
raytracer	18	87	74	283	5.1	18	1.8	41.4	226	411	55%	5.3k
montecarlo	18	114	115	442	5.2	23	3.5	50.8	37	38	97.4%	4.4k
sor	6	100	12	431	1.8	30	0.6	52.5	64	64	100%	940
elevator	5	188	24	899	2.3	52	0.6	88	100	100	100%	1.4k
jspider	113	391	422	1572	17.7	74.6	6.7	106	214	264	81.1%	82k
hedc	44	353	230	2,134	16	140	6.1	128	317	378	83.9%	38k
ftp	119	527	608	2,705	36.5	142	18.2	162	594	787	75.5%	131k
weblech	11	576	78	3,326	6	208	12	194	6	13	46.2%	6.2k

3.2.6.1 Evaluation Setup

We implemented our approach in a tool called URSA for analyses specified in Datalog that target Java programs. We use Chord [38] as the Java analysis framework and bddbddb [19] as the Datalog solver. All experiments were done using Oracle HotSpot JVM 1.6 on a Linux machine with 64GB memory and 2.7GHz processors.

Table 3.8 shows the characteristics of each benchmark. In particular, it shows the numbers of alarms and tuples in the universes of potential causes. Note that the size of the universe of potential causes is one to two orders of magnitude higher than the alarms for each benchmark, highlighting the large search space of our problem.

We next describe how we instantiate the main algorithm (Algorithm 2) of URSA by describing our implementations of Heuristic, Decide, and the termination check.

Heuristic**.** We evaluate all Heuristic instantiations described in Chapter 3.2.5, plus an `ideal` one that answers according to the ground truth. We define `aggregated` by

$$\texttt{aggregated}() := \{\, t \in Q_U \mid f(t \in \texttt{static\_optimistic}(), t \in \texttt{static\_pessimistic}(), t \in \texttt{dynamic}())\}$$

where f is a ternary Boolean function represented by a decision tree. We learn this decision tree, using the C4.5 algorithm [45], on the four smaller benchmarks. Each data point

corresponds to a tuple in the universes of potential causes and we obtain the expected output by invoking the Decide procedure, whose implementation is described immediately after the current discussion on Heuristic. The result of learning is $f(x, y, z) = x \wedge z$, which leads to

$$\texttt{aggregated}() = \texttt{static\_optimistic}() \cap \texttt{dynamic}().$$

Thus, `aggregated` only classifies a tuple as spurious when both `static_optimistic` and `dynamic` do so. We find `aggregated` to be the best instantiation overall in terms of numbers of questions asked and alarms resolved by URSA.

To evaluate the effectiveness of various heuristics, we also define the heuristic $\texttt{ideal}() := Q_U \cap \mathsf{True}$. This heuristic requires precomputing the ground truth, which one would not do for using URSA normally. As ground truth, we use consensus between answers from many users. Note that the interactive analysis will treat `ideal` as it does with any of the other heuristics, cross-checking its answers against the oracle.

Decide**.** In practice, Decide is implemented by a real user or an expensive analysis that provides answers to questions posed by URSA in an online manner. To evaluate our approach uniformly under various settings, however, we obtained answers to all the alarms and potential causes offline. To obtain such answers, we hired a group of 44 Java developers on UpWork [46], a freelancer platform. For improved confidence in the answers, we required them to have prior experience with concurrent programming in Java; moreover, we filtered out 4 users who gave incorrect answers to three hidden diagnostic questions, resulting in 40 valid participants. Finally, to reduce the answering effort, we applied a dynamic analysis and marked facts observed to hold in concrete runs as true, and required answers only for the unresolved ones.

Termination. As described in Chapter 3.2.4.6, we decide to terminate the algorithm when the expected payoff is ≤ 1. Intuitively, there is no root cause left that could explain

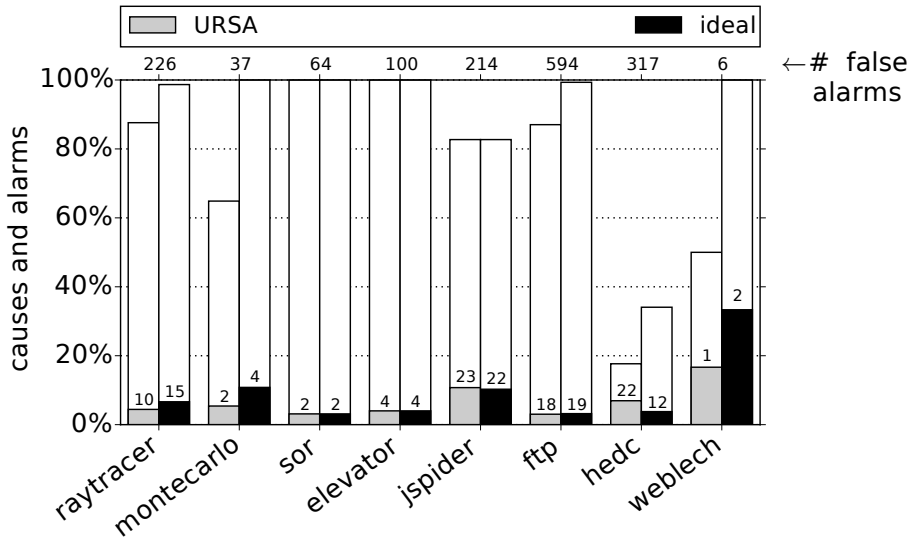


Figure 3.14: Number of questions asked over total number of false alarms (denoted by the lower dark bars) and percentage of false alarms resolved (denoted by the upper light bars) by URSA. Note that URSA terminates when the expected payoff is ≤ 1, which indicates that the user should stop looking at potential causes and focus on the remaining alarms.

more than one alarm. Thus, the user may find it more effective to inspect the remaining reports directly or use other techniques to further reduce the false alarms.

3.2.6.2 Evaluation Results

Our evaluation addresses the following five questions:

1. *Generalization:* How effectively does URSA identify the fewest queries that eliminate the most false alarms?
2. *Prioritization:* How effectively does URSA prioritize those queries that eliminate the most false alarms?
3. *User time for causes vs. alarms:* How much time do users spend inspecting a potential cause versus a potential alarm?
4. *Impact of incorrect user answers:* How do incorrect user answers affect the effectiveness of URSA in terms of precision, soundness, and user effort?
5. *Scalability:* Does the optimization procedure of URSA scale to large programs?

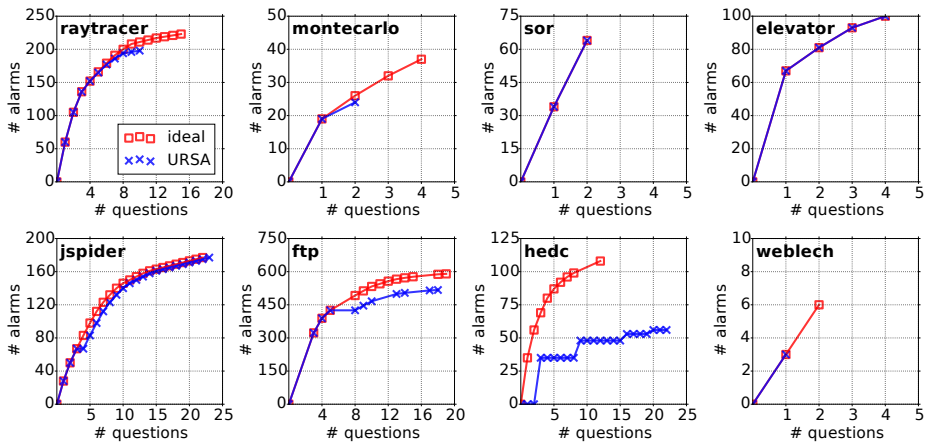


Figure 3.15: Number of questions asked and number of false alarms resolved by URSA in each iteration.

6. *Effect of different heuristics:* What is the impact of different heuristics on generalization and prioritization?

We report all averages using arithmetic means except average payoffs and average speedups, which are calculated using geometric means.

Generalization results. Figure 3.14 shows the generalization results of URSA with `aggregated`, the best available Heuristic instantiation. For comparison, we also include the `ideal` heuristic. For each benchmark under both settings, we show the percentage of resolved false alarms (the upper light bars) and the number of asked questions over the total number of false alarms (the lower dark bars). The figure also shows the absolute numbers of asked questions (denoted by the numbers over dark bars) and the numbers of false alarms produced by the input static analysis (denoted by the numbers at the top).

URSA is able to eliminate 73.7% of the false alarms with an average payoff of $12\times$ per question. On `ftp`, the benchmark with the most false alarms, the gain is as high as 87% and the average payoff increases to $29\times$. Note that URSA does not resolve all alarms as it terminates when the expected payoff becomes 1 or smaller, which means there are no common root causes for the remaining alarms. These results show that most of the false alarms can indeed be eliminated by inspecting only a few common root causes.

URSA eliminates most of the false alarms for all benchmarks except `hedc`, where only 17.7% of the false alarms are eliminated. In fact, even under the ideal setting, only 34% of the false alarms can be eliminated. Closer inspection revealed that most alarms in `hedc` indeed do not share common root causes. However, the questions asked comprise only 7% of the false alarms. This shows that even when there is scarce room to generalize, URSA does not ask unnecessary questions.

In the ideal case, URSA eliminates an additional 15.6% of false alarms on average, which modestly improves over the results with `aggregated`. We thus conclude the `aggregated` instantiation is effective in identifying common root causes of false alarms.

Prioritization results. Figure 3.15 shows the prioritization results of URSA. In the plots, each iteration of Algorithm 2 has a corresponding point, which represents the number of false alarms eliminated (y-axis) and the number of questions (x-axis) asked so far. As before, we compare the results of URSA to the ideal case, which has a perfect heuristic.

We observe that a few causes often yield most of the benefits. For instance, three causes can resolve 323 out of 594 false alarms on `ftp`, yielding a payoff of $108\times$. URSA successfully identifies these causes with high payoffs and poses them to the user in the earliest few iterations. In fact, for the first three iterations on most benchmarks, URSA asks exactly the same set of questions as the ideal setting. The results of these two settings only differ in later iterations, where the payoff becomes relatively low. We also notice that there can be multiple causes in the set that gives the highest benefit (for instance, the aforementioned `ftp` results). The reason is that there can be multiple derivations to each alarm. If we naively search the potential causes by fixing the number of questions in advance, we can miss such causes. URSA, on the other hand, successfully finds them by solving the optimal root set problem iteratively.

The fact that URSA is able to prioritize causes with high payoffs allows the user to stop interacting after a few iterations and still get most of the benefits. URSA terminates either when the expected payoff drops to 1 or when there are no more questions to ask. But in

practice, the user might choose to stop the interaction even earlier. For instance, the user might be satisfied with the current result or she may find limited payoff in answering more questions.

We study these causes with high payoffs more closely. For our datarace analysis, the two main categories of causes are (i) spurious thread-shared memory access (`shared`) tuples in object constructors of classes that extend the `java.lang.Thread` class or implement the `java.lang.Runnable` interface, and (ii) spurious may-happen-in-parallel (`parallel`) tuples in the `run` methods of similar classes. The objects whose constructors contain spurious (`shared`) tuples are mostly created in loops where a new thread is created and executes the `run` methods of these objects in each iteration. The datarace analysis is unable to distinguish the objects created in different iterations and considers them all as thread-shared after the first iteration. This leads to many false datarace alarms between the main thread which invokes the constructors and the threads created in the loops which also access the objects by executing their `run` methods. The spurious `parallel` tuples are produced due to the context-insensitive call-graphs which mismatch the `run` methods containing them to `Thread.start` invocations that cannot actually reach these `run` methods. This in turn leads to many false alarms between multiple threads.

User time for causes vs. alarms. While URSA can significantly reduce the number of alarms that a user must inspect, it comes at the expense of the user inspecting causes. We measured the time consumed by each programmer when labeling individual causes and alarms for the datarace analysis. Our measurements show that it takes 578 seconds on average for a user to inspect a datarace alarm but only 265 seconds on average to inspect a cause. This is because reasoning about an alarm often requires reasoning about multiple causes and other program facts that can derive it. To precisely quantify the reduction in user effort, a more rigorous user study is needed, which is beyond the scope of the current paper. However, the massive reduction in the number of alarms that a user needs to inspect and

Table 3.9: Results of URSA on `ftp` with noise in Decide. The baseline analysis produces 193 true alarms and 594 false alarms. We run each setting for 30 times and take the averages.

% Noise	# Resolved False Alarms	% Resolved False Alarms	# False Negatives	% Retained True Alarms	# Questions	Payoff
0%	517.0	87.0%	0.0	100.0%	18.0	28.7
1%	516.4	86.9%	0.0	100.0%	18.1	28.6
5%	515.4	86.8%	4.9	97.4%	18.2	28.4
10%	505.0	85.0%	9.2	95.2%	19.4	26.3

the fact that a cause is on average much less expensive to inspect shows URSA's potential to significantly reduce overall user effort.

In practice, there might be cases where the causes are much more expensive to inspect than alarms. However, as discussed in Chapter 3.2.4.6, as long as the costs can be quantified in some way, we can naturally encode them in our optimization problem. As a result, URSA will be able to find the set of questions that maximizes the payoff in user effort.

Impact of incorrect user answers. As we discussed earlier (Chapter 3.2.4.6), users can make mistakes. We analyze the impact of mistakes quantitatively by injecting random noise in Decide: we flip its answer to each tuple with probability $x\%$. We set x to 1, 5, 10, and apply URSA on `ftp`, the benchmark with the most alarms. We run the experiment under each setting for 30 times and calculate the averages of all statistics. Table 3.9 summarizes the results in terms of precision, soundness, and user effort. We also show the results of URSA without noise as a comparison.

Columns 2 and 3 show, respectively, the number and the percentage of false alarms that are resolved by URSA with noise. When the amount of noise increases, both statistics drop. This is due to Decide incorrectly marking certain spurious tuples posed by URSA as true. These tuples might well be the only tuples that can resolve certain alarms and are with payoff > 1. However, we also notice that the drop is modest: a noise of 10% leads to a drop in resolved false alarms of 2% only.

Columns 4 and 5 show, respectively, the number of false negatives and the percentage of retained true alarms. URSA can introduce false negatives when a noisy Decide incorrectly

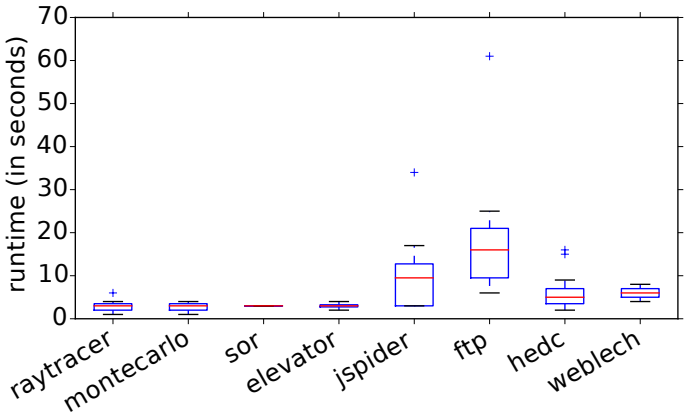


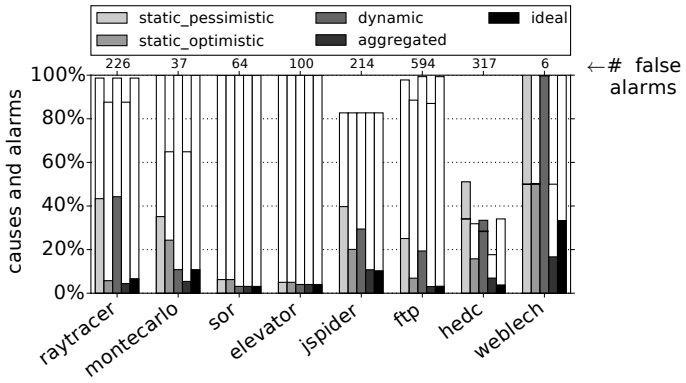
Figure 3.16: Time consumed by URSA in each iteration.

marks a true tuple as spurious. The number of false negatives grows as the amount of noise increases, but the growth is not significant: a noise of 10% leads to missing 4.8% true alarms only. This shows that URSA does not amplify user errors. It is especially true for datarace analysis, given that its alarms are harder to inspect compared to the root causes.

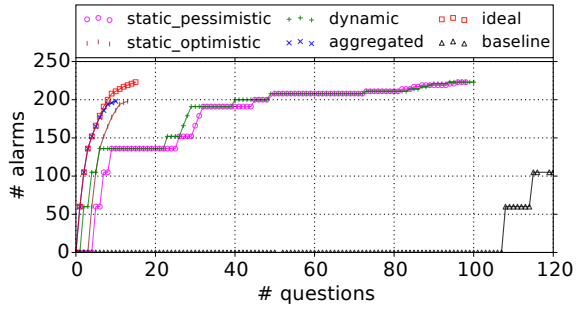
Columns 6 and 7 show, respectively, the number of questions asked by URSA and the payoff of answering these questions. As expected, more noise leads to more questions and smaller payoffs. But, the effect is modest: a noise of 10% increases the number of questions by 7.7% and decreases the payoff by 8.4%.

In summary, URSA is resilient to user mistakes.

Scalability results. Figure 3.16 shows the distributions of the time consumed by URSA in each iteration. As the figure shows, almost all iterations finish within half a minute. After closer inspection, we find that most of the time is spent in the Markov Logic Network solver. While each invocation only takes less than 5 seconds, one iteration consists of up to 20 invocations. We note that while the user inspects causes (which takes 265 seconds on average), URSA would have time to speculate what the answers will be, and prepare the next set of questions accordingly. There are two cases in which URSA spent more than half a minute: the first iterations of `jspider` and `ftp`. This iteration happens after the static analysis runs and before the user interaction starts. So, from the point of view of the user,



(a) Overall.



(b) Across iterations on `raytracer`.

Figure 3.17: Number of questions asked and number of false alarms eliminated by URSA with different Heuristic instantiations.

it is the same as having a slightly longer static analysis time. Moreover, on the smaller four benchmarks, each iteration of URSA takes only a few seconds. Such fast response times ensure URSA's usability in an interactive setting.

Effect of different heuristics. The results of running the datarace analysis under each heuristic (Chapter 3.2.5) are shown in Figure 3.17: (a) generalization and (b) prioritization.

Generalization. We begin with a basic sanity check. By Lemma 12, if a heuristic returns a subset of potential causes, then it cannot resolve more false alarms. Thus, `aggregated` can never resolve more false alarms than `static_optimistic` or `dynamic`, and `static_optimistic` can never resolve more false alarms than `static_pessimistic`. This is indeed the case, as we can see from the white bars in Figure 3.17(a).

We make two main observations: (i) `dynamic` has wide variation in performance, and

(ii) `aggregated` has the best average performance. As before, we measure generalization performance by payoff. The variation in payoff under the `dynamic` heuristic can be explained by a variation in test coverage: `dynamic` performs well for `montecarlo` and `elevator` (which have good test coverage), and poorly for `raytracer` and `hedc` (which have poor test coverage). The good average payoff of `aggregated` shows that aggressive heuristics guide our approach towards asking good questions.

Prioritization. Figure 3.17(b) shows the prioritization result for each instantiation on `raytracer`. For comparison, we also show the result with a baseline Heuristic instantiation which returns *all* tuples in the `shared` and `parallel` relations as potential causes. Instantiation `aggregated` yields the best result: the first six questions posed under it are same as those under the ideal case, and answering these questions resolves 177 out of 226 false alarms. The other instantiations perform slightly worse but still prioritize the top three questions in terms of payoff in the first 10 questions. On the other hand, the baseline fails to resolve any alarms for the first 107 iterations, highlighting the effectiveness of using any of our Heuristic instantiations over using none of them.

3.2.7 Related Work

We survey techniques for reducing user effort in inspecting static analysis alarms. Then, we explain the relation with techniques for combining analyses.

Models for interactive analysis. Different usage models have been proposed to counter the impact of approximations [47, 48, 9] or missing specifications [49, 50] on the accuracy of static analyses.

In the approaches proposed by [48, 9], user annotations on inspected alarms are generalized to re-rank or re-classify the remaining alarms. These approaches are probabilistic, and do not guarantee soundness. Also, users inspect alarm reports rather than intermediate causes.

The technique proposed by [47] asks users to resolve intermediate causes that are guaranteed to resolve individual alarms soundly. It aims to minimize user effort needed to resolve a single alarm. In contrast, our approach aims to minimize user effort to resolve all alarms considered together, making it more suitable for static analyses that report large numbers of alarms with shared underlying causes of imprecision.

Instead of approximations, the approaches proposed by [50, 49] target specifications for missing program parts that result in misclassified alarms. In particular, they infer candidate specifications that are needed to verify a given assertion, and present them to users for validation. These techniques are applicable to analyses based on standard graph reachability [49] or CFL reachability [50], while we target analyses specified in Datalog.

Just-in-time analysis aims at low-hanging fruit: it first presents alarms that can be found quickly, are unlikely to be false positives, and involve only code close to where the programmer edits. While the programmer responds to these alarms, the analysis searches for more alarms, which are harder to find. A just-in-time analysis exploits user input only in a limited sense: it begins by looking at the code in the programmer's working set. [51] show how to transform an analysis based on the IFDS/IDE framework into a just-in-time analysis. By comparison, our approach targets Datalog (rather than IFDS/IDE), requires a delay in the beginning to run the underlying analysis, but then fully exploits user input and provenance information to minimize the number of alarms presented to the user.

The Ivy model checker takes user guidance to infer inductive invariants [52]. Our approach is to start with an over-approximation and ask for user help to prune it down towards the ground truth; the Ivy approach is to guess a separating frontier between reachable and error states and ask for user help in refining this guess both downwards and upwards, until no transition crosses the frontier. Ivy targets small, modeling languages, rather than full-blown programming languages like Java.

Solvers for interactive analysis. Various solving techniques have been developed to incorporate user feedback into analyses or to ask relevant questions to analysis users. Abductive inference, used by [47, 49], involves computing minimum satisfying assignments to SMT formulae [47] or minimum-size prime implicants of Boolean formulae [49]. The maximum satisfiability problem used by [9] involves solving a system of mixed hard and soft constraints. While the hard constraints encode soundness conditions, the soft constraints encode various objectives such as the costs of different user questions, likelihood of different outcomes, and confidence in different hypotheses. These problems are NP-hard; in contrast, the CFL reachability algorithm used by [50] is polynomial time.

Report ranking and clustering. A common approach for reducing a user's effort in inspecting static analysis alarms is to rank them by their likelihood of being true [53, 54, 55]. In the work by [53], a statistical post-analysis is presented that computes the probability of each alarm being true. Z-ranking [54] employs a simple statistical model based on semantic inconsistency detection [56] to rank those alarms most likely to be true errors over those that are least likely. In the work by [55], a post-processing technique also based on semantic inconsistency detection is proposed to de-prioritize reporting alarms in modular verifiers that are caused by overly demonic environments.

Report clustering techniques [35, 36, 48] group related alarms together to reduce user effort in inspecting redundant alarms. These techniques either compute logical dependencies between alarms [35, 36], or employ a probabilistic model for finding correlated alarms [48]. Unlike these techniques, our approach correlates alarms by identifying their common root causes, which are obtained by analyzing the abstract semantics of the analysis at hand.

Our approach is complementary to the above techniques; information such as the truth likelihood of alarms and logical dependencies between alarms can be exploited to further reduce user effort in our approach.

Spectrum-based fault localization. There is a large body of work on fault localization, especially spectrum-based techniques [57, 58, 59, 60, 61] that seek to pin-point the root causes of failing tests in terms of program behaviors that differ in passing tests, and thereby reduce programmers' debugging effort. Analogously, our approach aims to identify the root causes of alarms, albeit with two crucial differences: our approach is not aware upfront whether each alarm is true or false, unlike in the case of fault localization where each test is known to be failing or passing; secondly, our approach operates on an abstract program semantics used by the given static analysis, whereas fault localization techniques operate on the concrete program semantics.

Interactive Program Optimization. IOpt [62, 63] is an interactive program optimizer. While it targets program performance rather than correctness, it also uses benefit-cost ratios to rank the questions, where the benefit is the expected program speedup and the cost is the estimated user effort in answering the question. However, the underlying techniques to compute such ratios are radically different: while IOpt's approach is based on heuristics, our approach solves a rigorous optimization problem that is generated from the analysis semantics.

Combining multiple analyses. Our method allows a user and an analysis to work together interactively. It is possible to replace the user with another analysis, thus obtaining a method for combining two analyses. We note that the converse is often false: If one starts with a method for combining analyses, then it is usually impossible to replace one of the analyses with the user. The reason is that humans need an interface with low bandwidth. In our case, they get simple yes/no questions. But, if we look at the approach proposed by [8], which is closest to our work on a technical level, then we see a tight integration between analyses, in which most state is shared, and it is even difficult to say where the interface between analyses lies. There is no user that would be happy to be presented with the entire internal state of an analysis and asked to perform an update on it. Designing a

fruitful method for combining analyses becomes significantly more difficult if one adds the constraint that the interaction must be low bandwidth, and such a constraint is necessary if a user is to be involved.

Low-bandwidth methods for combining analyses may prove useful even in the absence of a user because they do not require analyses to be alike. This is speculation, but we can point to a similar situation that is reality: The Nelson–Oppen method for combining decision procedures is wildly successful because it requires only equalities to be communicated [64].

Existing methods for combining analyses do not have the low-bandwidth property. In previous works by [65, 66, 67], a fast pre-analysis is used to infer a precise and efficient abstraction for a parametric analysis. This pre-analysis can be either an over-approximating static analysis [66] or an under-approximating dynamic analysis [65, 67].

3.2.8 Conclusion

We presented an interactive approach to resolve static analysis alarms. The approach encompasses a new methodology that synergistically combines a sound but imprecise analysis with precise but unsound heuristics. In each iteration, it solves the optimum root set problem which finds a set of questions with the highest expected payoff to pose to the user. We presented an efficient solution to this problem based on Markov Logic Networks for a general class of constraint-based analyses. We demonstrated the effectiveness of our approach in practice at eliminating a majority of false alarms by asking only a few questions, and at prioritizing questions with high payoffs.

3.3 Static Bug Detection

3.3.1 Introduction

Program analysis tools often make approximations. These approximations are a necessary evil as the program analysis problem is undecidable in general. There are also several

specific factors that drive various assumptions and approximations: program behaviors that the analysis intends to check may be impossible to define precisely (e.g., what constitutes a security vulnerability), computing exact answers may be prohibitively costly (e.g., worst-case exponential in the size of the analyzed program), parts of the analyzed program may be missing or opaque to the analysis (e.g., if the program is a library), and so on. As a result, program analysis tools often produce false positives (or false bugs) and false negatives (or missed bugs), which are absolutely undesirable to users. Users today, however, lack the means to guide such tools towards what they believe to be "interesting" analysis results, and away from "uninteresting" ones.

This section presents a new approach to *user-guided program analysis*. It shifts decisions about the kind and degree of approximations to apply in an analysis from the *analysis writer* to the *analysis user*. The user conveys such decisions in a natural fashion, by giving feedback about which analysis results she likes or dislikes, and re-running the analysis.

Our approach is a radical departure from existing approaches, allowing users to control both the precision and scalability of the analysis. It offers a different, and potentially more useful, notion of precision—one from the standpoint of the analysis user instead of the analysis writer. It also allows the user to control scalability, as the user's feedback enables tailoring the analysis to the precision needs of the analysis user instead of catering to the broader precision objectives of the analysis writer.

Our approach and tool called EUGENE satisfies three useful goals: (i) *expressiveness*: it is applicable to a variety of analyses, (ii) *automation*: it does not require unreasonable effort by analysis writers or analysis users, and (iii) *precision* and *scalability*: it reports interesting analysis results from a user's perspective and it handles real-world programs. EUGENE achieves each of these goals as described next.

Expressiveness. An analysis in EUGENE is expressed as a Markov Logic Network, which is a set of logic inference rules with optional weights (Chapter 3.3.3). In the absence of weights, rules become "hard rules", and the analysis reduces to a conventional one where a

solution that satisfies all hard rules is desired. Weighted rules, on the other hand, constitute "soft rules" that generalize a conventional analysis in two ways: they enable to express different degrees of approximation, and they enable to incorporate feedback from the analysis user that may be at odds with the assumptions of the analysis writer. The desired solution of the resulting analysis is one that satisfies all hard rules and maximizes the weight of satisfied soft rules. Such a solution amounts to respecting all indisputable conditions of the analysis writer, while maximizing precision preferences of the analysis user.

Automation. EUGENE takes as input analysis rules from the analysis writer, and automatically learns their weights using an *offline learning algorithm* (Chapter 3.3.4.2). EUGENE also requires analysis users to specify which analysis results they like or dislike, and automatically generalizes this feedback using an *online inference algorithm* (Chapter 3.3.4.1). The analysis rules (hard and soft) together with the feedback from the user (modeled as soft rules) forms a probabilistic constraint system that EUGENE solves efficiently.

Precision and **Scalability.** EUGENE maintains precision by ensuring *integrity* and *optimality* in solving the rules without sacrificing scalability. Integrity (i.e., satisfying hard rules) amounts to respecting indisputable conditions of the analysis. Optimality (i.e., maximally satisfying soft rules) amounts to generalizing user feedback effectively. Together these aspects ensure precision. Satisfying all hard rules and maximizing the weight of satisfied soft rules corresponds to the aforementioned MAP inference problem of Markov Logic Networks. EUGENE leverages our learning and inference engine, NICHROME, to solve Markov Logic Networks in a manner that is integral, optimal, and scalable.

We demonstrate the precision and scalability of EUGENE on two analyses, namely, datarace detection, and monomorphic call site inference, applied to a suite of seven Java programs of size 131–198 KLOC. We also report upon a user study involving nine users who employ EUGENE to guide an information-flow analysis on three Java micro-benchmarks. In these experiments, EUGENE significantly reduces misclassified reports upon providing limited amounts of feedback.

```
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();
18       m_request = null;
19       m_writer.close();
20       m_writer = null;
21       m_reader.close();
22       m_reader = null;
23       m_controlSocket.close();
24       m_controlSocket = null;
25    }
26 }
```

Figure 3.18: Java code snippet of Apache FTP server.

In summary, our work makes the following contributions:

1. We present a new approach to user-guided program analysis that shifts decisions about approximations in an analysis from the analysis writer to the analysis users, allowing users to tailor its precision and cost to their needs.

2. We formulate our approach in terms of solving a combination of hard rules and soft rules, which enables leveraging off-the-shelf solvers for weight learning and inference that scale without sacrificing integrity or optimality.

3. We show the effectiveness of our approach on diverse analyses applied to a suite of real-world programs. The approach significantly reduces the number of misclassified reports by using only a modest amount of user feedback.

3.3.2 Overview

Similar to the interactive verification example described in Chapter 3.2.2, we illustrate our approach using the example of applying the static race detection tool Chord [38] to a real-world multi-threaded Java program, Apache FTP server [37]. However, for elaboration,

Analysis Relations:

$\mathsf{next}(p_1, p_2)$	(program point p_1 is immediate successor of program point p_2)
$\mathsf{parallel}(p_1, p_2)$	(different threads may reach program points p_1 and p_2 in parallel)
$\mathsf{alias}(p_1, p_2)$	(instructions at program points p_1 and p_2 may access the same memory location, and constitute a possible datarace)
$\mathsf{unguarded}(p_1, p_2)$	(no common lock guards program points p_1 and p_2)
$\mathsf{race}(p_1, p_2)$	(datarace may occur between different threads while executing the instructions at program points p_1 and p_2)

Analysis Rules:

$$\mathsf{parallel}(p_3, p_2) \text{ :- } \mathsf{parallel}(p_1, p_2),\ \mathsf{next}(p_3, p_1). \quad (1)$$

$$\mathsf{parallel}(p_2, p_1) \text{ :- } \mathsf{parallel}(p_1, p_2). \quad (2)$$

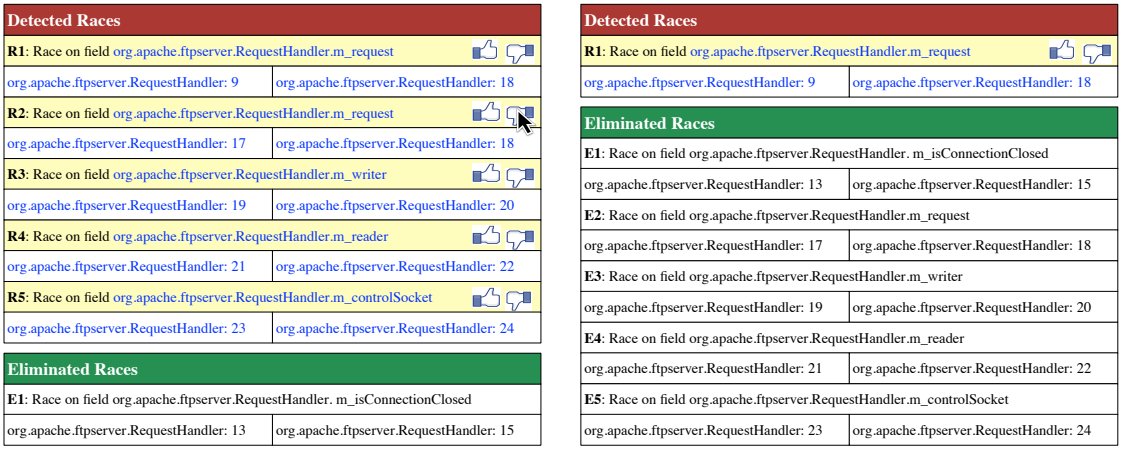
$$\mathsf{race}(p_1, p_2) \text{ :- } \mathsf{parallel}(p_1, p_2),\ \mathsf{alias}(p_1, p_2),\ \mathsf{unguarded}(p_1, p_2). \quad (3)$$

Figure 3.19: Simplified race detection analysis.

the example in this subsection looks slightly different, as the approach we will present improves a program analysis in a different manner and is complementary to the previous approach.

Figure 3.18 shows a code snippet 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 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.

We motivate our approach by contrasting the goals and capabilities of a *writer of an*



(a) Before feedback. (b) After feedback.

Figure 3.20: 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.

analysis, such as the race detection analysis in Chord, with those of a *user of the analysis*, such as a developer of the Apache FTP server.

The role of the analysis writer. The designer or writer of a static analysis tool, say Alice, strives to develop an analysis that is precise yet scales to real-world programs, and is widely applicable to a large variety of programs. In the case of Chord, this translates into a race detection analysis that is context-sensitive but path-insensitive. This is a common design choice for balancing precision and scalability of static analyses. The analysis in Chord is expressed using Datalog, a declarative logic programming language, and Figure 3.19 shows a simplified subset of the logical inference rules used by Chord. The actual analysis implementation uses a larger set of more elaborate rules but the rules shown here suffice for the discussion. 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 Alice deems to be the most important for capturing dataraces in Java programs. For example, Rule (1) in Figure 3.19 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 symmetric. Via Rule (3), Alice 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.

The role of the analysis user. The user of a static analysis tool, say Bob, ideally wants the tool to produce exact (i.e., sound and complete) results on his program. This allows him to spend his time on fixing the bugs in the program instead of classifying the reports generated by the tool as spurious or real. In our example, suppose that Bob runs Chord on the Apache FTP server program in Figure 3.18. Based on the rules in Figure 3.19, Chord produces the list of datarace reports shown in Figure 3.20(a). Reports R1–R5 are identified as potential dataraces in the program, whereas for report E1, Chord detects that the accesses to `m_isConnectionClosed` on lines 13 and 15 are guarded by a common lock, and therefore do not constitute a datarace. Typically, the analysis user Bob is well-acquainted with the program being analyzed, but not with the details of underlying analysis itself. In this case, given his familiarity with the program, it is relatively easy for Bob to conclude that the code from line 17–24 in the body of the `close()` method can never be executed by multiple threads in parallel, and thus reports R2–R5 are spurious.

The mismatch between analysis writers and users. The design decisions of the analysis writer Alice have a direct impact on the precision and scalability of the analysis. The datarace analysis in Chord is imprecise for various theoretical and usability reasons.

First, the analysis must scale to large programs. For this reason, it is designed to be path-insensitive and over-approximates the possible thread interleavings. To eliminate spurious reports R2–R5, the analysis would need to only consider feasible thread interleavings by accurately tracking control-flow dependencies across threads. However, such precise analyses do not scale to programs of the size of Apache FTP server, which comprises 130 KLOC.

Scalability concerns aside, relations such as alias are necessarily inexact as the corresponding property is undecidable for Java programs. Chord over-approximates this property by using a context-sensitive but flow-insensitive pointer analysis, resulting in spurious pairs (p_1, p_2) in this relation, which in turn are reported as spurious dataraces.

Third, the analysis writer may lack sufficient information to design a precise analysis, because the program behaviors that the analysis intends to check may be vague or ambiguous. For example, in the case of datarace analysis, real dataraces can be *benign* in that they do not affect the program's correctness [68]. Classifying such reports typically requires knowledge about the program being analyzed.

Fourth, the program specification can be incomplete. For instance, the race in report R1 above could be harmful but impossible to trigger due to timing reasons extrinsic to Apache FTP server, such as the hardware environment.

In short, while the analysis writer Alice can influence the design of the analysis, she cannot foresee every usage scenario or program-specific tweaks that might improve the analysis. Conversely, analysis user Bob is acquainted with the program under analysis, and can classify the analysis reports as spurious or real. But he lacks the tools or expertise to suppress the spurious bugs by modifying the underlying analysis based on his intuition and program knowledge.

Closing the gap between analysis writers and users. Our user-guided approach aims to empower the analysis user Bob to adjust the underlying analysis as per his demands without involving the analysis writer Alice. Our system, EUGENE, achieves this by automatically incorporating user feedback into the analysis. The user provides feedback in a natural fashion, by "liking" or "disliking" a subset of the analysis reports, and re-running the analysis. For example, when presented with the datarace reports in Figure 3.20(a), Bob might start inspecting from the first report. This report is valid and Bob might choose to either like or ignore this report. Liking a report conveys that Bob accepts the reported bug as a real one and would like the analysis to generate more similar reports, thereby reinforcing

the behavior of the underlying analysis that led to the generation of this report. However, suppose that Bob ignores the first report, but indicates that he dislikes the second report by clicking on the corresponding icon. Re-running Chord after providing this feedback produces the reports shown in Figure 3.20(b). While the true report R1 is generated in this run as well, all the remaining spurious reports are eliminated. This highlights a key strength of our approach: EUGENE not only incorporates user feedback, but it also generalizes the feedback to other similar results of the analysis. Reports R2–R5 are correlated and are spurious for the same root cause: the code from line 17–24 in the body of the `close()` method can never be executed by multiple threads in parallel. Bob's feedback on report R2 conveys to the underlying analysis that lines 17 and 18 cannot execute in parallel. EUGENE is able to generalize this feedback automatically and conclude that none of the lines from 17–24 can execute in parallel.

In the following subsections, we describe the underlying details of EUGENE that allow it to incorporate user feedback and generalize it automatically to other reports.

3.3.3 Analysis Specification

EUGENE uses a constraint-based approach wherein analyses are written in a declarative constraint language. Constraint languages have been widely adopted to specify a broad range of analyses. The declarative nature of such languages allows analysis writers to focus on the high-level analysis logic while delegating low-level implementation details to off-the-shelf constraint solvers. In particular, Datalog, a logic programming language, is widely used in such approaches. Datalog has been shown to suffice for expressing a variety of analyses, including pointer and call-graph analyses [15, 16, 18, 7], concurrency analyses [43, 69], security analyses [70, 71], and reflection analysis [72].

Existing constraint-based approaches allow specifying only *hard rules* where an acceptable solution is one that satisfies all the rules. However, this is insufficient for incorporating feedback from the analysis user that may be at odds with the assumptions of the analysis

Input tuples:

$\mathsf{next}(18,17)$ $\mathsf{alias}(18,17)$ $\mathsf{guarded}(13,13)$
$\mathsf{next}(19,18)$ $\mathsf{alias}(20,19)$ $\mathsf{guarded}(15,15)$
$\mathsf{next}(20,19)$ $\mathsf{alias}(22,21)$ $\mathsf{guarded}(15,13)$ ...

Ground formula:

$$
\begin{array}{ll}
w_1 : (\neg\mathsf{parallel}(17,17) \vee \neg\mathsf{next}(18,17) \vee \mathsf{parallel}(18,17)) & \wedge \\
\quad (\neg\mathsf{parallel}(18,17) \vee \mathsf{parallel}(17,18)) & \wedge \\
w_1 : (\neg\mathsf{parallel}(17,18) \vee \neg\mathsf{next}(18,17) \vee \mathsf{parallel}(18,18)) & \wedge \\
w_1 : (\neg\mathsf{parallel}(18,18) \vee \neg\mathsf{next}(19,18) \vee \mathsf{parallel}(19,18)) & \wedge \\
\quad (\neg\mathsf{parallel}(19,18) \vee \mathsf{parallel}(18,19)) & \wedge \\
w_1 : (\neg\mathsf{parallel}(18,19) \vee \neg\mathsf{next}(19,18) \vee \mathsf{parallel}(19,19)) & \wedge \\
w_1 : (\neg\mathsf{parallel}(19,19) \vee \neg\mathsf{next}(20,19) \vee \mathsf{parallel}(20,19)) & \wedge \\
\left(\begin{array}{lll} \neg\mathsf{parallel}(18,17) & \vee & \neg\mathsf{alias}(18,17) \quad \vee \\ \neg\mathsf{unguarded}(18,17) & \vee & \mathsf{race}(18,17) \end{array}\right) & \wedge \\
\left(\begin{array}{lll} \neg\mathsf{parallel}(20,19) & \vee & \neg\mathsf{alias}(20,19) \quad \vee \\ \neg\mathsf{unguarded}(20,19) & \vee & \mathsf{race}(20,19) \end{array}\right) & \wedge \\
\boxed{w_2 : \neg\mathsf{race}(18,17)} & \wedge \ldots
\end{array}
$$

Output tuples (before feedback):

$\mathsf{parallel}(18,9)$ $\mathsf{parallel}(20,19)$ $\mathsf{race}(18,9)$ $\mathsf{race}(20,19)$ ...
$\mathsf{parallel}(18,17)$ $\mathsf{parallel}(22,21)$ $\mathsf{race}(18,17)$ $\mathsf{race}(22,21)$

Output tuples (after feedback):

$\mathsf{parallel}(18,9)$ $\mathsf{race}(18,9)$ ...

Figure 3.21: Probabilistic analysis example.

writer. To enable the flexibility of having conflicting constraints, it is necessary to allow *soft rules* that an acceptable solution can violate. Our user-guided approach is based on Markov Logic Networks that extend Datalog rules with weights. We refer to analyses specified in this extended language as *probabilistic analyses*. The semantics of these analyses is encoded as MAP inference problems of corresponding Markov Logic Networks.

Example. Recall the syntax and semantics of Markov Logic Networks (Chapter 2.3), we now describe how EUGENE works on the race detection example from Chapter 3.3.2. Figure 3.21 shows a subset of the input and output facts as well as a snippet of the ground formula constructed for the example. The input tuples are derived from the analyzed program (Apache FTP server) and comprise the next, alias, and unguarded relations. In all these relations, the domain of program points is represented by the corresponding line number in the code. Note that all the analysis rules expressed in Figure 3.19 are hard rules

since existing tools like Chord do not accept soft rules. However, we assume that when this analysis is fed to EUGENE, rule (1) is specified to be soft by analysis writer Alice, which captures the fact that the parallel relation is imprecise. EUGENE automatically learns the weight of this rule to be w_1 by applying the learning engine of NICHROME to training data (see Chapter 3.3.4.2 for details).

To understand how EUGENE generalizes user feedback, we inspect the ground formula generated from the probabilistic datarace analysis. Given these input tuples and rules, the ground formula is generated by grounding the analysis rules, and a snippet of the constructed ground formula is shown in Figure 3.21. Recall the definition of the MAP inference problem: it is to find a set of output tuples that maximizes the sum of the weights of satisfied ground soft rules while satisfying all ground hard rules. Ignoring the clause enclosed in the box, solving this ground formula (without the boxed clause) yields output tuples, a subset of which is shown under "Output tuples (before feedback)" in Figure 3.21. The output includes multiple spurious races like $\mathsf{race}(18, 17)$, $\mathsf{race}(20, 19)$, and $\mathsf{race}(22, 21)$.

As described in Chapter 3.3.2, when analysis user Bob provides feedback that $\mathsf{race}(18, 17)$ is spurious, EUGENE suppresses all spurious races while retaining the real race $\mathsf{race}(18, 9)$. EUGENE achieves this by incorporating the user feedback itself as a soft rule, represented by the boxed clause $\neg\mathsf{race}(18, 17)$ in Figure 3.21. The weight for such user feedback is also learned during the training phase. Assuming the weight w_2 of the feedback clause is higher than the weight w_1 of rule (1)—a reasonable choice that emphasizes Bob's preferences over Alice's assumptions—the MAP problem semantics ensures that the solver prefers violating rule (1) over violating the feedback clause. When the ground formula (with the boxed clause) in Figure 3.21 is then solved, the output solution violates the clause $w_1 : (\neg\mathsf{parallel}(17, 17) \vee \neg\mathsf{next}(18, 17) \vee \mathsf{parallel}(18, 17))$ and does not produce tuples $\mathsf{parallel}(18, 17)$ and $\mathsf{race}(18, 17)$ in the output. Further, all the tuples that are dependent on $\mathsf{parallel}(18, 17)$ are not produced either.[2] This implies that tuples like $\mathsf{parallel}(19, 18)$,

[2]This is due to implicit soft rules that negate each output relation, such as $w_0 : \neg\mathsf{parallel}(p_1, p_2)$ where $w_0 < w_1$, in order to obtain the least solution.

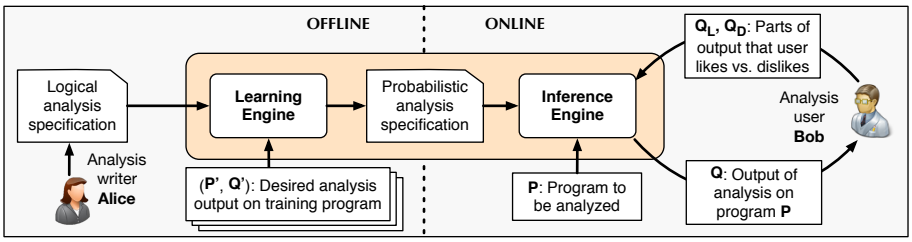


Figure 3.22: Workflow of the EUGENE system for user-guided program analysis.

$\mathsf{parallel}(20, 19)$, $\mathsf{parallel}(22, 21)$ are not produced, and therefore $\mathsf{race}(20, 19)$ and $\mathsf{race}(22, 21)$ are also suppressed. Thus, EUGENE is able to generalize user feedback. The degree of generalization depends on the quality of the weights assigned or learned for the soft rules.

3.3.4 The EUGENE System

This section describes our system EUGENE for user-guided program analysis. Its workflow, shown in Figure 3.22, comprises an online inference stage and an offline learning stage.

In the online stage, EUGENE takes the probabilistic analysis specification together with a program P that an analysis user Bob wishes to analyze. The inference engine of NICHROME (Chapter 4), uses these inputs to produce the analysis output Q. Further, the online stage allows Bob to provide feedback on the produced output Q. In particular, Bob can indicate the output queries he likes (Q_{L}) or dislikes (Q_{D}), and invoke the inference engine with Q_{L} and Q_{D} as additional inputs. The inference engine incorporates Bob's feedback as additional soft rules in the probabilistic analysis specification used for producing the new result Q. This interaction continues until Bob is satisfied with the analysis output.

The accuracy of the produced results in the online stage is sensitive to the weights assigned to the soft rules. Manually assigning weights is not only inefficient, but in most cases it is also infeasible since weight assignment needs analysis of data. Therefore, EUGENE provides an offline stage that automatically learns the weights of soft rules in the probabilistic analysis specification. In the offline stage, EUGENE takes a logical analysis specification from analysis writer Alice and training data in the form of a set of input programs

Algorithm 4 `Inference`: Online component of EUGENE.

1: **PARAM** (w_l, w_d): Weights of liked and disliked queries.
2: **INPUT:** $C = C_h \cup C_s$: Probabilistic analysis, where C_h are the hard rules and C_s are the soft rules.
3: **INPUT:** P: Program to analyze.
4: **OUTPUT:** Q: Final output of user-guided analysis.
5: $Q_{\mathsf{L}} := \emptyset$; $Q_{\mathsf{D}} := \emptyset$
6: $C'_h := C_h \cup P$
7: **repeat**
8: $\quad C'_s := C_s \cup \{(t, w_l) \mid t \in Q_{\mathsf{L}}\} \cup \{(\neg t, w_d) \mid t \in Q_{\mathsf{D}}\}$
9: $\quad Q := \mathrm{MAP}(C'_h \cup C'_s)$
10: $\quad Q_{\mathsf{L}} :=$ `PositiveUserFeedback`(Q)
11: $\quad Q_{\mathsf{D}} :=$ `NegativeUserFeedback`(Q)
12: **until** $Q_{\mathsf{L}} \cup Q_{\mathsf{D}} = \emptyset$

and desired analysis output on these programs. These inputs are fed to the learning engine of NICHROME (Chapter 4). The logical analysis specification includes hard rules as well as rules marked as soft whose weights need to be learnt. The learning engine infers these weights to produce the probabilistic analysis specification. The learning engine ensures that the learnt weights maximize the likelihood of the training data with respect to the probabilistic analysis specification.

3.3.4.1 Online Component of EUGENE*: Inference*

Algorithm 4 describes the online component `Inference` of EUGENE, which leverages the inference engine of NICHROME. `Inference` takes as input, a probabilistic analysis C (with learnt weights), and the program P to be analyzed. First, in line 6, the algorithm augments the hard and soft rules C_h, C_s of the analysis with the inputs P, Q_{L}, Q_{D} to the analysis, to obtain an extended set of rules $C'_h \cup C'_s$ (lines 6 and 8). Notice that the user feedback Q_{L} (liked queries) and Q_{D} (disliked queries) are incorporated as soft rules in the extended rule set. Each liked query feedback is assigned the fixed weight w_l, while each disliked query feedback is assigned weight w_d (line 8). Weights w_l and w_d are learnt in the offline stage and fed as parameters to Algorithm 4. Instead of using fixed weights for the user feedback, two other options are: (a) treating user feedback as hard rules, and

Algorithm 5 `Learning`: Offline component of EUGENE.

1: **INPUT:** C: Initial probabilistic analysis.
2: **INPUT:** Q: Desired analysis output.
3: **OUTPUT:** C': Probabilistic analysis with learnt weights.
4: $C' = \text{learn}(C, Q)$

(b) allowing a different weight for each query feedback. Option (a) does not account for users being wrong, leaving no room for the inference engine to ignore the feedback if necessary. Option (b) is too fine-grained, requiring learning separate weights for each query. We therefore take a middle ground between these two extremes.

Next, in line 9, the algorithm invokes the inference engine of NICHROME with the extended set of rules. Note that EUGENE treats the solver as a black-box and any suitable solver suffices. The solver produces a solution Q that satisfies all the hard rules in the extended set, while maximizing the weight of satisfied soft rules. The solution Q is then presented to Bob who can give his feedback by liking or disliking the queries (lines 10–11). The sets of liked and disliked queries, Q_{L} and Q_{D}, are used to further augment the soft rules C_s of the analysis. This loop (lines 7–12) continues until no further feedback is provided by Bob.

3.3.4.2 Offline Component of EUGENE*: Learning*

Algorithm 5 describes the offline component `Learning` of EUGENE, which leverages the learning engine of NICHROME. `Learning` takes a probabilistic analysis C with arbitrary weights, a set of programs $\mathcal{P}$ and the desired analysis output $Q \subseteq \mathbb{T}$ as input, and outputs a probabilistic analysis C' with learnt weights. Without loss of generality, we assume that $\mathcal{P}$ is encoded as a set of hard rules and is part of C. `Learning` computes the output analysis C' by invoking the learning engine of NICHROME (denoted by learn) with the input analysis C and the desired analysis output Q.

3.3.5 Empirical Evaluation

We implemented EUGENE atop Chord [38], an extensible program analysis framework for Java bytecode that supports writing analyses in Datalog. In our evaluation, we investigate the following research questions:

- **RQ1:** Does using EUGENE improve analysis precision for practical analyses applied to real-world programs? How much feedback is needed for the same, and how does the amount of provided feedback affect the precision?
- **RQ2:** Does EUGENE scale to large programs? Does the amount of feedback influence the scalability?
- **RQ3:** How feasible is it for users to inspect analysis output and provide useful feedback to EUGENE?

3.3.5.1 Evaluation Setup

We performed two different studies with EUGENE: a control study and a user study.

First, to evaluate the precision and scalability of EUGENE, we performed a *control study* using two realistic analyses expressed in Datalog applied to seven Java benchmark programs. The goal of this study is to thoroughly investigate the performance of EUGENE in realistic scenarios and with varying amounts of feedback. To practically enable the evaluation of EUGENE over a large number of a data-points in the $(benchmark, analysis, \#feedback)$ space, this study uses a more precise analysis, instead of a human user, as an oracle for generating the feedback to be provided. This study helps us evaluate **RQ1** and **RQ2**.

Second, to evaluate the practical usability of EUGENE when human analysis users are in the loop, we conducted a *user study* with nine users who employed EUGENE to guide an information-flow analysis on three benchmark programs. In contrast to the first study, the human users provide the feedback in this case. This study helps us evaluate **RQ3**.

Table 3.10: Statistics of our probabilistic analyses.

	rules	input relations	output relations
datarace	30	18	18
polysite	76	50	42
infoflow	76	52	42

All experiments were run using Oracle HotSpot JVM 1.6.0 on a Linux server with 64GB RAM and 3.0GHz processors.

Clients. Our two analyses in the first study (Table 3.10) are datarace detection (*datarace*) and monomorphic call site inference (*polysite*), while we use an information-flow (*infoflow*) analysis for the user study. Each of these analyses is sound, and composed of other analyses written in Datalog. For example, *datarace* includes a thread-escape analysis and a may-happen-in-parallel analysis, while *polysite* and *infoflow* include a pointer analysis and a call-graph analysis. The pointer analysis used here is a flow/context-insensitive, field-sensitive, Andersen-style analysis using allocation site heap abstraction [73]. The *datarace* analysis is from [43], while the *polysite* analysis has been used in previous works [8, 74, 75] to evaluate pointer analyses. The *infoflow* analysis only tracks explicit information flow similar to the analysis described in [76]. For scalability reasons, all these analyses are context-, object-, and flow-insensitive, which is the main source of false positives reported by them.

Benchmarks. The benchmarks for the first study (upper seven rows of Table 3.11) are 131–198 KLOC in size, and include programs from the DaCapo suite [77] (`antlr`, `avrora`, `luindex`, `lusearch`) and from past works that address our two analysis problems.

The benchmarks for the user study (bottom three rows of Table 3.3) are 0.6–4.2 KLOC in size, and are drawn from Securibench Micro [78], a micro-benchmark suite designed to exercise different parts of a static information-flow analyzer.

Methodology. We describe the methodology for the offline (learning) and online (inference) stages of EUGENE.

Offline stage. We first converted the above three logical analyses into probabilistic analyses using the offline training stage of EUGENE. To avoid selection bias, we used a set of small benchmarks for training instead of those in Table 3.11. Specifically, we used `elevator` and `tsp` (100 KLOC each) from [79]. While the training benchmarks are smaller and fewer than the testing benchmarks, they are sizable, realistic, and disjoint from those in the evaluation, demonstrating the practicality of our training component. Besides the sample programs, the training component of EUGENE also requires the expected output of the analyses on these sample programs. Since the main source of false positives in our analyses is the lack of context- and object-sensitivity, we used context- and object-sensitive versions of these analyses as oracles for generating the expected output. Specifically, we used k-object-sensitive versions [27] with cloning depth $k=4$. Note that these oracle analyses used for generating the training data comprise their own approximations (for example, flow-insensitivity), and thus do not produce the absolute ground truth. Using better training data would only imply that the weights learnt by EUGENE are more reflective of the ground truth, leading to more precise results.

Online stage. We describe the methodology for the online stage separately for the control study and the user study.

Control study methodology. To perform the control study, we started by running the inference stage of EUGENE on our probabilistic analyses (*datarace* and *polysite*) with no feedback to generate the initial set of reports for each benchmark. Next, we simulated the process of providing feedback by: (i) randomly choosing a subset of the initial set of reports, (ii) classifying each of the reports in the chosen subset as spurious or real, and (iii) re-running the inference stage of EUGENE on the probabilistic analyses with the labeled reports in the chosen subset as feedback. To classify the reports as spurious or real, we

Table 3.11: Benchmark statistics. Columns "total" and "app" are with and without JDK library code.

	# classes		# methods		bytecode (KB)		source (KLOC)	
	app	total	app	total	app	total	app	total
`antlr`	111	350	1,150	2,370	128	186	29	131
`avrora`	1,158	1,544	4,234	6,247	222	325	64	193
`ftp`	93	414	471	2,206	29	118	13	130
`hedc`	44	353	230	2,134	16	140	6	153
`luindex`	206	619	1,390	3,732	102	235	39	190
`lusearch`	219	640	1,399	3,923	94	250	40	198
`weblech`	11	576	78	3,326	6	208	12	194
`secbench1`	4	5	10	13	0.3	0.3	0.08	0.6
`secbench2`	3	4	9	12	0.2	0.2	0.07	0.6
`secbench3`	2	17	4	46	0.3	1.25	0.06	4.2

used the results of k-object-sensitive versions of our client analyses as ground truth. In other words, if a report in the chosen subset is also generated by the precise version of the analysis, it is classified as a real report, otherwise it is labeled as a spurious report. For each $(benchmark, analysis)$ pair, we generated random subsets that contain 5%, 10%, 15%, and 20% of the initial reports. This allows us to study the effect of varying amounts of feedback on EUGENE's performance. Moreover, EUGENE can be sensitive to not just the amount of feedback, but also to the actual reports chosen for feedback. To mitigate this effect, for a given $(benchmark, analysis)$ pair, and a given feedback subset size, we ran EUGENE thrice using different random subsets of the given size in each run. Randomly choosing feedback ensures that we conservatively estimate the performance of EUGENE. Finally, we evaluated the quality of the inference by comparing the output of EUGENE with the output generated by the k-object-sensitive versions of our analyses with $k=4$.

User study methodology. For the user study, we engaged nine users, all graduate students in computer science, to run EUGENE on *infoflow* analysis. Each user was assigned two benchmarks from the set of {`secbench1`, `secbench2`, `secbench3`}, such that each of these benchmarks was assigned to six users in total. The users interacted with EUGENE by first running it without any feedback so as to produce the initial set of reports. The users then analyzed these produced reports, and were asked to provide any eight reports

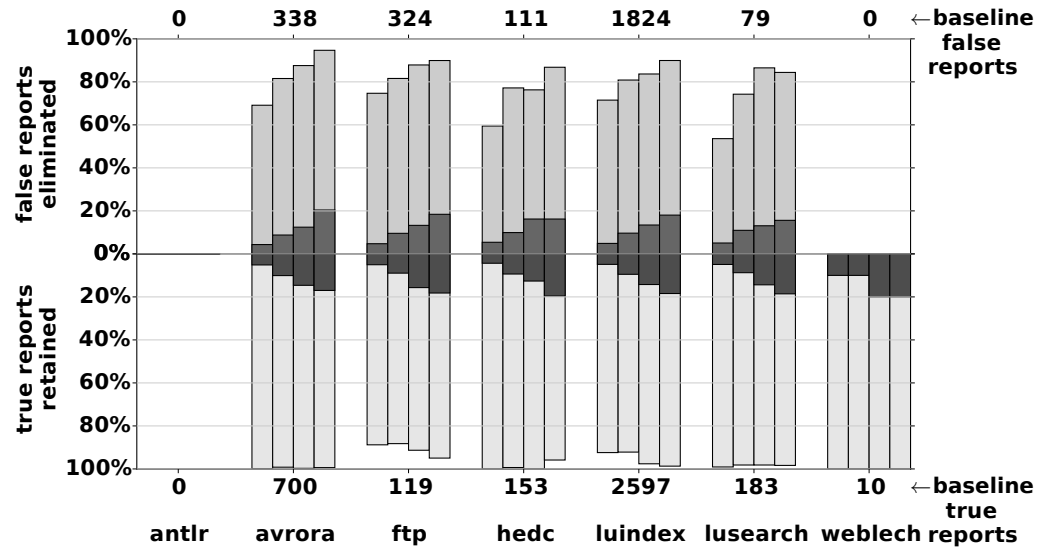


Figure 3.23: Results of EUGENE on *datarace* analysis.

with their corresponding label (spurious or real) as feedback. Also, for each benchmark, we recorded the time spent by each user in analyzing the reports and generating the feedback. Next, EUGENE was run with the provided feedback, and the produced output was compared with manually generated ground truth for each of the benchmarks.

We next describe the results of evaluating EUGENE's precision (**RQ1**), scalability (**RQ2**), and usability (**RQ3**).

3.3.5.2 Evaluation Results

Precision of EUGENE. The analysis results of our control study under varying amounts of feedback are shown in Figures 3.23 and 3.24. In these figures, "baseline false reports" and "baseline true reports" are the numbers of false and true reports produced when EUGENE is run without any feedback. The light colored bars above and below the x-axis indicate the % of false reports eliminated and the % of true reports retained, respectively, when the % of feedback indicated by the corresponding dark colored bars is provided. For each benchmark, the feedback percentages increase from left to right, i.e., 5% to 20%. Ideally, we want all the false reports to be eliminated and all the true reports to be retained, which would be indicated by the light color bars extending to 100% on both sides.

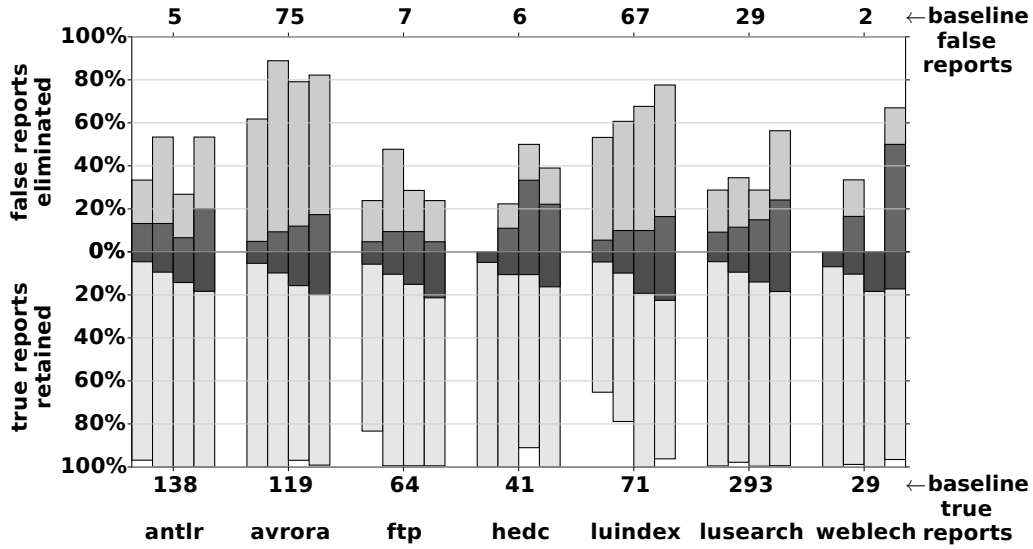


Figure 3.24: Results of EUGENE on *polysite* analysis.

Even without any feedback, our probabilistic analyses are already fairly precise and sophisticated, and eliminate all except the non-trivial false reports. Despite this, EUGENE helps eliminate a significant number of such hard-to-refute reports. On average 70% of the false reports are eliminated across all our experiments with 20% feedback. Likewise importantly, on average 98% of the true reports are retained when 20% feedback is provided. Also, note that with 5% feedback the percentage of false reports eliminated falls to 44% on average, while that of true reports retained is 94%. A finer-grained look at the results for individual benchmarks and analyses reveals that in many cases, increasing feedback only leads to modest gains.

We next discuss the precision of EUGENE for each of our probabilistic analyses. For DataraceAnalysis, with 20% feedback, an average of 89% of the false reports are eliminated while an average of 98% of the true reports are retained. Further, with 5% feedback the averages are 66% for false reports eliminated and 97% for true reports retained. Although the precision of EUGENE increases with more feedback in this case, the gains are relatively modest. Note that given the large number of initial reports generated for `luindex` and `avrora` (4421 and 1038 respectively), it is somewhat impractical to expect analysis users to provide up to 20% feedback. Consequently, we re-run EUGENE for these benchmarks

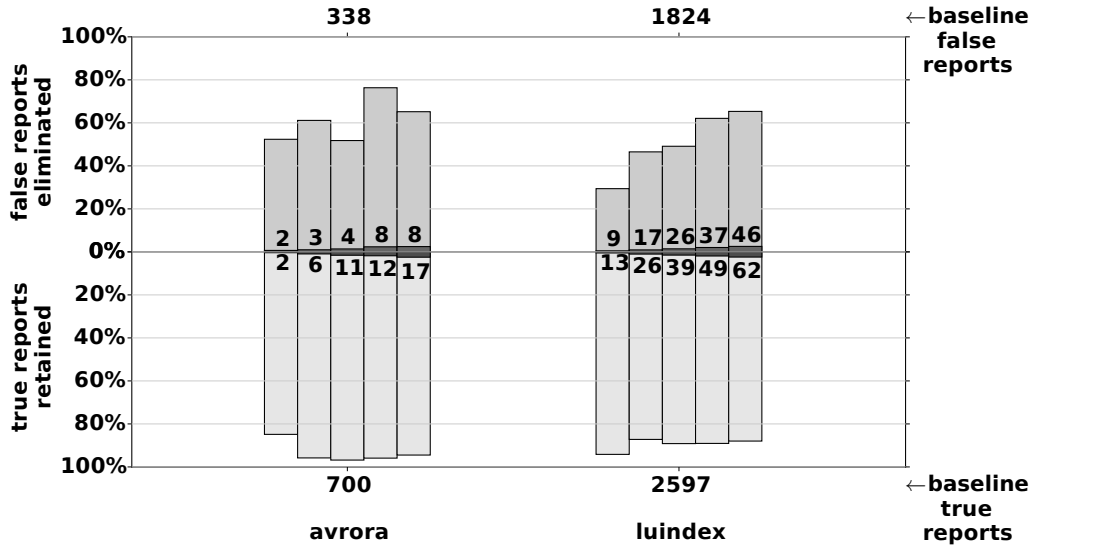


Figure 3.25: Results of EUGENE on *datarace* analysis with feedback (0.5%,1%,1.5%,2%, 2.5%).

with 0.5%, 1%, 1.5%, 2% and 2.5% feedback. The results are shown in Figure 3.25. Interestingly, we observe that for `luindex`, with only 2% feedback on the false reports and 1.9% feedback on true reports, EUGENE eliminates 62% of false reports and retains 89% of the true reports. Similarly for `avrora`, with only 2.3% feedback on the false reports and 1.8% feedback on true reports, EUGENE eliminates 76% of false reports and retains 96% of the true reports. These numbers indicate that, for the *datarace* client, EUGENE is able to generalize even with a very limited amount of user feedback.

For *polysite*, with 20% feedback, an average of 57% of the false reports are eliminated and 99% of the true reports are retained, while with 5% feedback, 29% of the false reports are eliminated and 92% of the true reports are retained. There are two important things to notice here. First, the number of eliminated false reports does not always grow monotonically with more feedback. The reason is that EUGENE is sensitive to the reports chosen for feedback, but in each run, we randomly choose the reports to provide feedback on. Though the precision numbers here are averaged over three runs for a given feedback amount, the randomness in choosing feedback still seeps into our results. Second, EUGENE tends to do a better job at generalizing the feedback for the larger benchmarks compared to the smaller

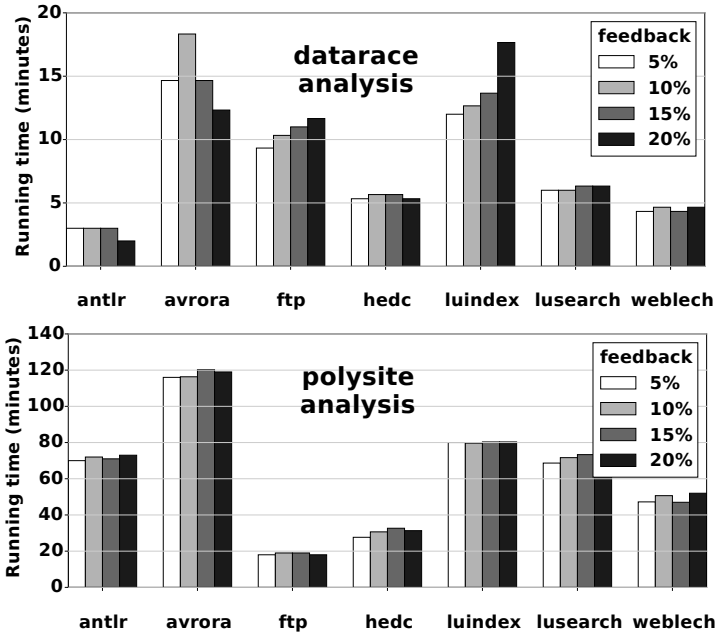


Figure 3.26: Running time of EUGENE.

ones. We suspect the primary reason for this is the fact that smaller benchmarks tend to have a higher percentage of bugs with unique root causes, and thereby a smaller number of bugs are attributable to each unique cause. Consequently, the scope for generalization of the user feedback is reduced.

Answer to **RQ1**: EUGENE significantly reduces false reports with only modest feedback, while retaining the vast majority of true reports. Though increasing feedback leads to more precise results in general, for many cases, the gain in precision due to additional feedback is modest.

Scalability of EUGENE. The performance of EUGENE for our control study, in terms of the inference engine running time, is shown in Figure 3.26. For each $(benchmark, analysis, \#feedback)$ configuration, the running time shown is an average over the three runs of the corresponding configuration. We observe two major trends from this figure. First, as expected, the running time is dependent on the size of the benchmark and the complexity of the analysis. For both the analyses in the control study, EUGENE takes the longest time for `avrora`, our largest benchmark. Also, for each of our benchmarks, the *datarace* analysis, with fewer rules, needs shorter time. Recollect that EUGENE uses an off-the-shelf solver

for solving the constraints of probabilistic analysis, and thus the performance of the inference engine largely depends on the performance of the underlying solver. The running time of all such solvers depends on the number of ground clauses that are generated, and this number in turn depends on the size of the input program and complexity of the analysis.

Second, the amount of feedback does not significant affect running time. Incorporating the feedback only requires adding the liked/disliked queries as soft rules, and thus does not significantly alter the underlying set of constraints.

Finally, the fact that EUGENE spends up to 120 minutes (*polysite* analysis on `avrora` with 15% feedback) might seem disconcerting. But note that this represents the time spent by the *system* rather than the *user*, in computing the new results after incorporating the user feedback. Since EUGENE uses the underlying solver as a black-box, any improvement in solver technology directly translates into improved performance of EUGENE. Given the variety of solvers that already exist [80, 81, 82, 83, 84, 85], and the ongoing active research in this area, we expect the running times to improve further.

Answer to **RQ2**: EUGENE effectively scales to large programs up to a few hundred KLOC, and its scalability will only improve with advances in underlying solver technology. Additionally, the amount of feedback has no significant effect on the scalability of EUGENE.

Usability of EUGENE Next we evaluate the results of our user study conducted using EUGENE. The usage model for EUGENE assumes that analysis users are familiar with the kind of reports produced by the analysis as well as with the program under analysis. To ensure familiarity with reports produced by *infoflow* analysis, we informed all our users about the expected outcomes of a precise *infoflow* analysis in general. However, familiarity with the program under analysis is harder to achieve and typically requires the user to have spent time developing or fixing the program. To address this issue, we choose relatively smaller benchmarks in our study that users can understand without too much effort or expertise. The users in this study were not informed about the internal working of either

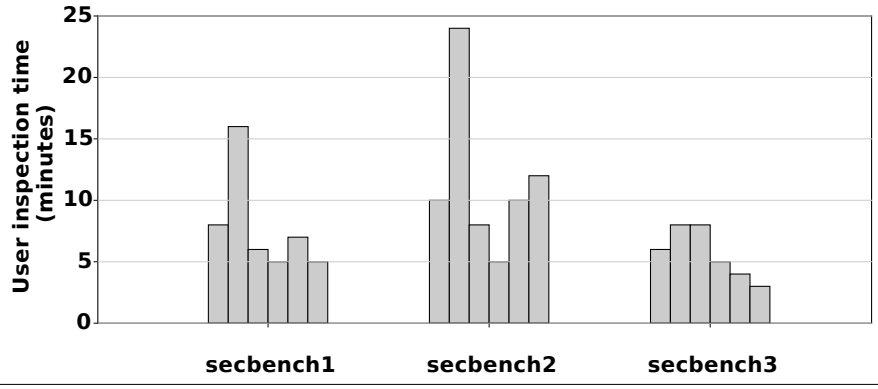


Figure 3.27: Time spent by each user in inspecting reports of *infoflow* analysis and providing feedback.

EUGENE or the *infoflow* analysis.

The two main questions that we evaluate here are: (i) the ease with which users are able to analyze the reported results and provide feedback, and (ii) the quality of the user feedback. To answer the first question, we record the time spent by each user in analyzing the *infoflow* reports and providing the feedback for each benchmark. Recall that we ask each user to provide eight reports as feedback, labeled either spurious or real. Figure 3.27 shows the time spent by each user on analyzing the reports and providing feedback. We observe that the average time spent by the users is only 8 minutes on `secbench1`, 11.5 minutes on `secbench2`, and 5.5 minutes on `secbench3`. These numbers show that the users are able to inspect the analysis output and provide feedback to EUGENE with relative ease on these benchmarks.

To evaluate the quality of the user provided feedback, we consider the precision of EUGENE when it is run on the probabilistic version of *infoflow* analysis with the feedback. Figure 3.28 shows the false bugs eliminated and the true bugs retained by EUGENE for each user and benchmark. This figure is similar in format to Figures 3.23 and 3.24. However, for each benchmark, instead of different bars representing different amounts of feedback, the different bars here represent different users, with feedback amount fixed at eight reports. The varying behavior of EUGENE on these benchmarks highlights the strengths and limits of our approach.

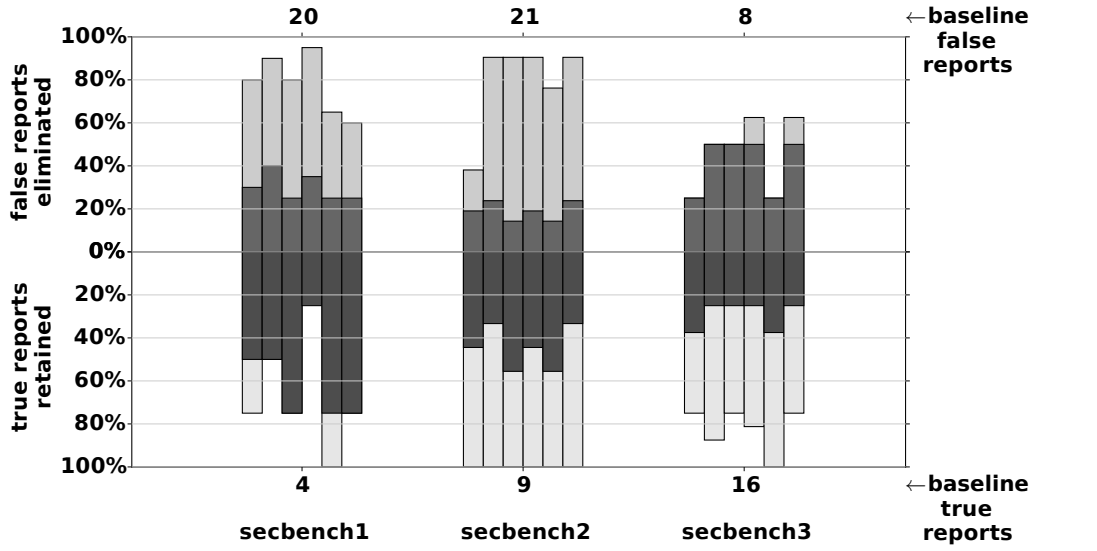


Figure 3.28: Results of EUGENE on *infoflow* analysis with real user feedback. Each bar maps to a user.

For `secbench1`, an average of 78% of the false reports are eliminated and 62.5% of the true reports are retained. The important thing to note here is that the number of true reports retained is sensitive to the user feedback. With the right feedback, all the true reports are retained (5^{th} bar). However, in the case where the user only chooses to provide one true feedback report (4^{th} bar), EUGENE fails to retain most of the true reports.

For `secbench2`, an average of 79% of the false reports are eliminated and 100% of the true reports are retained. The reason EUGENE does well here is that `secbench2` has multiple large clusters of reports with the same root cause. User feedback on any report in such clusters generalizes to other reports in the cluster. This highlights the fact that EUGENE tends to produce more precise results when there are larger clusters of reports with the same root cause.

For `secbench3`, an average of 46% of the false reports are eliminated while 82% of the true reports are retained. First, notice that this benchmark produces only eight false reports. We traced the relatively poor performance of EUGENE in generalizing the feedback on false reports to limiting the analysis user's interaction with the system to liking or disliking the results. This does not suffice for `secbench3` because, to effectively suppress the false

reports in this case, the user must add new analysis rules. We intend to explore this richer interaction model in future work.

Finally, we observed that for all the benchmarks in this study, the labels provided by the users to the feedback reports matched with the ground truth. While this is not unexpected, it is important to note that EUGENE is robust even under incorrectly labeled feedback, and can produce precise answers if a majority of the feedback is correctly labeled.

Answer to **RQ3**: It is feasible for users to inspect analysis output and provide feedback to EUGENE since they only needed an average of 8 minutes for this activity in our user study. Further, in general, EUGENE produce precise results with this user feedback, leading to the conclusion that it is not unreasonable to expect useful feedback from users.

Limitations of EUGENE EUGENE requires analyses to be specified using the Datalog-based language described in Chapter 2.2. Additionally, the program to be analyzed itself has to be encoded as a set of ground facts. This choice is motivated by the fact that a growing number of program analysis tools including bddbddb [19], Chord [38], Doop [20], LLVM [86], Soot [21], and Z3 [87] support specifying analyses and programs in Datalog.

The offline (learning) component of EUGENE requires the analysis designer to specify which analysis rules must be soft. Existing analyses employ various approximations such as path-, flow-, and context-insensitivity; in our experience, rules encoding such approximations are good candidates for soft rules. Further, the learning component requires suitable training data in the form of desired analysis output. We expect such training data to be either annotated by the user, or generated by running a precise but unscalable version of the same analysis on small sample programs. Learning using partial or noisy training data is an interesting future direction that we plan to explore.

3.3.6 Related Work

Our work is related to past work on classifying error reports and applications of probabilistic reasoning in program analysis.

Dillig et al. [47] propose a user-guided approach to classify reports of analyses as errors or non-errors. They use abductive inference to compute small, relevant queries to pose to a user that capture exactly the information needed to discharge or validate an error. Their approach does not incorporate user feedback into the analysis specification and generalize it to other reports. Blackshear and Lahiri [55] propose a post-processing framework to prioritize alarms produced by a static verifier based on semantic reasoning of the program. Statistical error ranking techniques [54, 48, 53] employ statistical methods and heuristics to rank errors reported by an underlying static analysis. Non-statistical clustering techniques correlate error reports based on a root-cause analysis [36, 35]. Our technique, on the other hand, makes the underlying analysis itself probabilistic.

Recent years have seen many applications of probabilistic reasoning to analysis problems. In particular, specification inference techniques based on probabilistic inference [88, 89, 90] can be formulated as Markov Logic Networks (as defined in Chapter 2.3). Another connection between user-guided program analysis and specification inference is that user feedback can be looked upon as an iterative method by means of which the analysis user communicates a specification to the program analysis tool. Finally, the inferred specifications can themselves be employed as soft rules in our system.

3.4 Conclusion

We presented a user-guided approach to program analysis that shifts decisions about the kind and degree of approximations to apply in analyses from analysis writers to analysis users. Our approach enables users to interact with the analysis by providing feedback on a portion of the results produced by the analysis, and automatically uses the feedback to guide the analysis approximations to the user's preferences. We implemented our approach in a system EUGENE and evaluated it on real users, analyses, and programs. We showed that EUGENE greatly reduces misclassified reports even with limited user feedback.

CHAPTER 4

SOLVER TECHNIQUES

This chapter discusses our backend, NICHROME, a learning and inference system for Markov Logic Networks. In order to solve the problems generated by aforementioned applications in a efficient and accurate manner, NICHROME exploits various domain insights that are not limited to our applications. We first give an overview of our learning and inference algorithms, then discuss how the algorithms exploit these insights in detail. In particular, most of the discussion will focus on the inference algorithm as the learning algorithm is eventually reduced to a sequence of invocations to the inference algorithm. The algorithms discussed in this chapter were originally described in our previous publications [85, 91].

Weight Learning. NICHROME supports weight learning assuming the structure of the constraints is given. Example applications include learning weights that represent designers' confidence for constraints from an existing Datalog analysis in the static bug detection application (Section 3.3). We adapted the gradient descent algorithm described in [92] for weight learning which reduces the learning problem into a sequence of MAP inference problems.

Algorithm 6 outlines our algorithm. The inputs are a Markov Logic Network C which consists of hard constraints C_h and soft constraints C_s. The output is a Markov Logic Network C'. C' has the same structure as C has but the weights in soft constraints C_s are updated with learnt weights. Our algorithm calculates the weights such that the expected MAP solution T becomes the output of $\mathrm{MAP}(C')$.

As a first step, in line 5, our algorithm assigns initial weights to all the soft constraints. The initial weight w' of a constraint $(c_h, w') \in C'_s$ is computed as a log of the ratio of the number of instances of c_h satisfied by the desired output T (denoted by

Algorithm 6 Weight learning algorithm for Markov Logic Networks.

1: **PARAM** α: rate of change of weight of soft rules.
2: **INPUT:** A Markov Logic Network $C = C_h \cup C_s$, where C_h are the hard constraints and C_s are the soft constraints.
3: **INPUT:** T, expected output tuples.
4: **OUTPUT:** C', a Markov Logic Network with the same structure as C has but with learnt weights.
5: $C'_s := \{ \ (c_h, w') \mid \exists w.(c_h, w) \in C_s \wedge w' = \log(n_1/n_2) \ \}$ where $n_1 = |\texttt{Satisfactions}(c_h, T)|$, and $n_2 = |\texttt{Violations}(c_h, T)|$.
6: **repeat**
7: $C' := C_h \cup C'_s$
8: $T' = \mathrm{MAP}(C')$
9: $C_s := C'_s$
10: $C'_s := \{ \ (c_h, w') \mid \exists w.(c_h, w) \in C_s \wedge w' = w + \alpha \times (n_1 - n_2) \ \}$ where $n_1 = |\texttt{Violations}(c_h, T')|$ and $n_2 = |\texttt{Violations}(c_h, T)|$.
11: **until** $C'_s = C_s$

$|\texttt{Satisfactions}(c_h, T)|$) to the number of instances of c_h violated by T (denoted by $|\texttt{Violations}(c_h, T)|$). In other words, the initial weight captures the log odds of a rule being true in the training data. Note that, in the case $|\texttt{Violations}(h, T)| = 0$, it is substituted by a suitably small value [92]. We formally define `Satisfactions` and `Violations` below:

$$\texttt{Satisfactions}(l_1 \vee \ldots \vee l_n, T) := \{ \ [\![l_1]\!](\sigma) \vee \ldots \vee [\![l_n]\!](\sigma) \mid \sigma \in \Sigma \wedge T \models [\![l_1]\!](\sigma) \vee \ldots \vee [\![l_n]\!](\sigma) \ \}$$

$$\texttt{Violations}(l_1 \vee \ldots \vee l_n, T) := \{ \ [\![l_1]\!](\sigma) \vee \ldots \vee [\![l_n]\!](\sigma) \mid \sigma \in \Sigma \wedge T \not\models [\![l_1]\!](\sigma) \vee \ldots \vee [\![l_n]\!](\sigma) \ \}$$

Next, in line 8, we perform MAP inference on the Markov Logic Network C' (defined in line 8) with the initialized weights. This produces a solution T', which then used to update the weights of the soft constraints. The weights are updated according to the formulae in lines 10. The basic intuition for updating weights is as follows: weights learnt by the learning algorithm must be such that $\mathrm{MAP}(C')$ is as close to the desired output T as possible. Towards that end, if the current output T' produces more violations for a rule than

the desired output, it implies that the rule needs to be strengthened and its weight should be increased. On the other hand, if the current output T' produces fewer violations for a rule then T, the rule needs to be weakened and its weight should be reduced. The formula in the algorithm has exactly the same effect as described here. Moreover, the rate of change of weights can be controlled by an input parameter α. The learning process continues iteratively until the learnt weights do not change. In practice, the learning process can be terminated after a fixed number of iterations, or when the difference in weights between successive iterations does not change significantly.

Since the above algorithm eventually reduces the learning problem into a series of MAP inference problems, the key component that decides the effectiveness of NICHROME is the inference algorithm. As a result, most of our technical innovations are on the inference algorithm, which we will illustrate in detail in the rest of the chapter.

MAP Inference. Before illustrating our approach, we first describe the properties of an ideal inference solver and identify key challenges in implementing such an ideal solver. Ideally, an inference solver should satisfy the following three criteria:

Soundness. We say an inference solver is *sound* if any solution it produces does not violate any hard constraint in the input formula. Soundness is necessary as it ensures correctness properties of the applications built upon the inference solver. For instance, applying an unsound solver on the formulae generated by the automated verification application (Section 3.1) can produce unviable abstractions that repeat the same mistakes and even invalid abstractions. As a result, the application may not terminate even when the space of valid abstractions is finite. As another example, applying an unsound solver in the static bug detection application can lead to solutions that violate analysis constraints that should definitely hold based on the analysis designers' knowledge. This in turn can greatly degrade the quality of the results (i.e., lead to a lot of false positives or false negatives).

Optimality. We say an inference solver is *optimal* if any solution it produces maximizes the sum of the weights of the satisfied soft constraints while respecting the soundness property. Optimality is important as it impacts the quality of the up-level application. For instance, returning suboptimal answers in the automated verification application will result in non-minimal abstractions, which can still resolve the desired queries but may result in program analyses with unnecessarily high costs. And these analyses might not even terminate if the resources are limited. As another example, a non-optimal solver can cause the user to spend extra effort in inspecting computed root causes in the interactive verification setting. In worst case, the user may end up spending up more time inspecting these root causes than inspecting the final alarms directly.

Scalability. Finally, a practical solver should be able to solve large Markov Logic Network large instances generated from real-world applications. This is especially important to our setting as the instances generated by our applications are more demanding compared to past applications of Markov Logic Networks, which we will discuss later.

Since Markov Logic Networks have emerged as a promising model for combining logic and probability, researchers from different communities have developed various MAP inference solvers [80, 81, 84, 83, 22]. However, none of them satisfies all three criteria. Tuffy [80] and Alchemy [81] are probably the most well-known Markov Logic Network solvers from the Artificial Intelligence community. They enforce neither soundness nor optimality. They do not enforce soundness as unlike problems in program reasoning, problems in Artificial Intelligence typically do not require rigorous formal guarantees. Moreover, while they scale well at problems in the Artificial Intelligence domain, they have difficulties terminating on our problems, whose characteristics are radically different. Markov thebeast [84] and Rockit [83] are two solvers that are developed by Artificial Intelligence researchers later. While both enforce soundness and optimality, they also do not scale on our problems. Finally, Z3 [22] is the most successful constraint solver from the program reasoning community. While it is possible to use Z3's MaxSMT engine to solve Markov Logic Networks

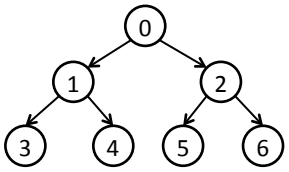
indirectly by converting the instances into MaxSMT instances, the approach does not scale well while being able to enforce soundness and optimality.

Before introducing our approach, we first discuss key challenges in designing a sound, optimal, and scalable MAP inference solver. At a high level, all the algorithms employed by the aforementioned solvers can be divided into two phases. The first phase is *grounding*, where the input Markov Logic Network is reduced to a weighted propositional formula by instantiating all variables with constants in the domain. The generated formulae are instances of *Maximum Satisfiability* (or MAXSAT), an optimization of the well-known boolean satisfiability problem. Specifically, we use one of its variants, Partial Weighted MAXSAT. Then the generated MAXSAT instance is solved by a MAXSAT solver in the *solving* phase. Both phases are challenging to scale: for the grounding phase, if we naively replace variables with all constants in the domain, we can easily get a MAXSAT formula comprising over 10^{30} clauses, which is intractable for any existing MAXSAT solver; for the solving phase, the MAXSAT problem is a combinatorial optimization problem, which is known for being computationally challenging.

We next describe two approaches that address the challenges in the above two phases respectively. Briefly, the first approach enables effective grounding by exploiting two observations: 1) solutions are typically "**sparse**", meaning that most of the tuples in the domain will not be derived in the output, and 2) a majority of the constraints in the formula are **Horn** as they come from a Datalog analysis. The second approach enables effective MAXSAT solving by exploiting the insight that in all our applications, we are only interested in part of the result that is about a few tuples – intuitively, if we only care about these few tuples, we do not always have to reason about the complete formula to get a correct partial solution. This is in line with **locality**, a property that is universal to program analysis and has been successfully leveraged by query-driven and demand-driven program analysis for better efficiency and precision. In the rest of the chapter, we describe these two approaches in detail.

$c_1 :\ \mathsf{path}(a, a) \quad c_2 :\ \mathsf{path}(a, b) \wedge \mathsf{edge}(b, c) \implies \mathsf{path}(a, c) \quad c_3 :\ \neg\mathsf{path}(a, b)\ \mathsf{weight}\ 1$

Figure 4.1: Graph reachability in Markov Logic Network.



Input facts:

$\mathsf{edge}(0, 1)$ $\mathsf{edge}(0, 2)$
$\mathsf{edge}(1, 3)$ $\mathsf{edge}(1, 4)$
$\mathsf{edge}(2, 5)$ $\mathsf{edge}(2, 6)$

Output facts:

$\mathsf{path}(0, 0)$ $\mathsf{path}(0, 1)$
$\mathsf{path}(0, 2)$ $\mathsf{path}(0, 3)$
$\mathsf{path}(0, 4)$ $\mathsf{path}(0, 5)$
...

Figure 4.2: Example graph reachability input and solution.

4.1 Iterative Lazy-Eager Grounding

4.1.1 Introduction

We first show informally why a naive grounding strategy which eagerly replace variables with all constants will produce intractable formulae. Consider the Markov Logic Network encoding the graph reachability problem in Figure 4.1 and Figure 4.2. Eagerly grounding constraint c_2 entails instantiating a, b, c over all nodes in the graph, producing N^3 ground constraints, where N is the number of nodes. Hence, the number of ground constraints quickly becomes intractable as the size of the input graph grows.

In order to generate tractable MaxSat formulae, several techniques have been proposed to lazily ground constraints [80, 81, 82, 83, 84]. For the scale of problems we consider, however, these techniques are either too lazy and converge very slowly, or too eager and produce instances that are beyond the reach of sound MaxSat solvers (i.e., solvers that do not produce solutions that violate hard clauses). For instance, a recent technique [82] grounds hard constraints too lazily and soft constraints too eagerly. Specifically, for the graph reachability example, this technique takes L iterations to lazily ground hard constraint c_2 (where L is the length of the longest path in the input graph), and generates N^2 constraints upfront by eagerly grounding soft constraint c_3.

In this section, we propose an iterative eager-lazy algorithm that strikes a balance between eager grounding and lazy grounding. Our key underlying idea comprises two com-

plementary optimizations: eagerly exploiting proofs and lazily refuting counterexamples.

To eagerly exploit proofs, our algorithm uses an efficient procedure to upfront ground constraints that will necessarily be grounded during the iterative process. As a concrete instance of this procedure, we use a Datalog solver, which efficiently computes the least solution of a set of recursive Horn constraints [1]. In practice, most constraints in many inference tasks are Horn, allowing to effectively leverage a Datalog solver. For instance, both hard constraints c_1 and c_2 in our graph reachability example are Horn. Our algorithm therefore applies a Datalog solver to efficiently ground them upfront. On the example graph in Figure 4.2, this produces 7 ground instances of c_1 and only 10 instances of c_2.

To lazily refute counterexamples, our algorithm uses an efficient procedure to check for violations of constraints by the MAXSAT solution to the set of ground constraints in each iteration, terminating the iterative process in the absence of violations. We use a Datalog solver as a concrete instance of this procedure as well. Existing lazy techniques, such as Cutting Plane Inference (CPI) [84] and SoftCegar [82], can be viewed as special cases of our algorithm in that they only lazily refute counterexamples. For this purpose, CPI uses a relational database query engine and SoftCegar uses a satisfiability modulo theories or SMT theorem prover [22], whereas we use a Datalog solver; engineering differences aside, the key motivation underlying all three approaches is the same: to use a separate procedure that is efficient at refuting counterexamples. Our main technical insight is to apply this idea analogously to eagerly exploit proofs. Our resulting strategy of guiding the grounding process based on both proofs and counterexamples gains the benefits of both eager and lazy grounding without suffering from the disadvantages of either.

We apply our algorithm to enable sound and efficient inference for large problem instances not only from the aforementioned program analysis applications, but also from *information retrieval* application, which concern discovering relevant information from large, unstructured data collections.

[1] A Horn constraint is a disjunction of literals with at most one positive (i.e., unnegated) literal.

The graph reachability example illustrates important aspects of inference tasks in both these domains. For program analysis, the reader can refer to the pointer analysis example described in Section 3.1.2. In the information retrieval domain, an example task is the entity resolution problem for removing duplicate entries in a citation database [93]. In this problem, a hard constraint similar to transitivity constraint c_2 in Figure 4.1 encodes an axiom about the equivalence of citations:

$$sameBib(v_1, v_2) \wedge sameBib(v_2, v_3) \implies sameBib(v_1, v_3)$$

We evaluate our algorithm on three benchmarks with three different input datasets generated from real-world information retrieval and program analysis applications. Our empirical evaluation shows that our approach achieves significant improvement over three state-of-art approaches, CPI [84], RockIt [83], and Tuffy [80], in running time as well as the quality of the solution.

4.1.2 The IPR Algorithm

We propose an efficient iterative algorithm IPR (Inference via Proof and Refutation) for solving Markov Logic Networks. The algorithm has the following key features:

1. **Eager proof exploitation.** IPR eagerly explores the logical structure of the relational constraints to generate an initial grounding, which has the effect of speeding up the convergence of the algorithm. When the relational constraints are in the form of Horn constraints, we show that such an initial grounding is optimal (Theorem 15).

2. **Lazy counterexample refutation.** After solving the constraints in the initial grounding, IPR applies a refutation-based technique to refine the solution: it lazily grounds the constraints that are violated by the current solution, and solves the accumulated grounded constraints in an iterative manner.

Algorithm 7 IPR: the eager-lazy algorithm.

INPUT: $C = C_h \cup C_s$, a Markov Logic Network where C_h are the hard constraints and C_s are the soft constraints.
OUTPUT: $T \subseteq \mathbb{T}$, solution (assumes $\mathrm{MAP}(C) \neq \mathsf{UNSAT}$).
ϕ := any $\bigcup_{i=1}^{n} \{\rho_i\}$ such that $\forall i \in [1,n].\exists l_1 \vee \ldots \vee l_m \in C_h.\exists \sigma \in (\mathbb{N} \cup \mathbb{V}) \mapsto \mathbb{N}.\rho_i = [\![l_1]\!](\sigma) \vee \ldots \vee [\![l_n]\!](\sigma)$
ψ := any $\bigcup_{i=1}^{n} \{(\rho_i, w_i)\}$ such that $\forall i \in [1,n].\exists (l_1 \vee \ldots \vee l_m, w) \in C_s.w = w_i \wedge (\exists \sigma \in (\mathbb{N} \cup \mathbb{V}) \mapsto \mathbb{N}.\rho_i = [\![l_1]\!](\sigma) \vee \ldots \vee [\![l_n]\!](\sigma))$
$T := \emptyset$; $w := 0$
while true do
  $\phi' := \bigcup_{c_h \in C_h} \texttt{Violations}(c_h, T)$
  $\psi' := \bigcup_{(c_h,w) \in C_s} \{\, (\rho, w) \mid \rho \in \texttt{Violations}(c_h, T) \,\}$
  $(\phi, \psi) := (\phi \cup \phi', \psi \cup \psi')$
  $T' :=$ MAXSAT (ϕ, ψ)
  $w' :=$ WEIGHT (ψ, T')
  if $(w' = w \wedge \phi' = \emptyset)$ **then return** T
  $T := T'$; $w := w'$
end while

3. **Termination with soundness and optimality.** IPR performs a termination check that guarantees the soundness and optimality of the solution if an exact MAXSAT solver is used. Moreover, this check is more precise than the termination checks in existing refutation-based algorithms [84, 82], therefore leading to faster convergence.

Before presenting the algorithm, we first introduce two functions MAXSAT and WEIGHT to abstract the interface of a MAXSAT solver, which allows us to focus our discussion on the grounding subproblem of the MAP inference problem. We will discuss the MAXSAT problem definition and our MAXSAT solver implementation in detail in the next section. To define these two functions, We introduce a symbol ρ to represent a ground hard constraint. That is

$$\rho ::= \bigvee_{i \in [1,n]} t_i \vee \bigvee_{j \in 1,m} \neg t_j.$$

Let ϕ be a set of ground hard constraints $\{\rho_i \mid 0 < i < n\}$ and ψ be a set of ground soft

constraints $\{(\rho_i, w_i) \mid 0 < i < m\}$, then we have

$$\textsc{Weight}(\psi, T) = \sum_{(\rho,w)\in\psi'} w, \text{ where } \psi' = \{(\rho, w) \mid (\rho, w) \in psi \wedge T \models \rho\},$$

$$\textsc{MaxSat}(\phi, \psi) = \begin{cases} \mathsf{UNSAT} & \text{if } \forall T.\exists \rho \in \phi.T \not\models \rho, \\ T \text{ such that } T \in \arg\max_{T'\in\mathbf{T}} \textsc{Weight}(\psi, T') & \text{otherwise.} \\ \quad \text{where } \mathbf{T} = \{T' \mid \forall \rho \in \phi.T' \models \rho\} & \end{cases}$$

Intuitively, WEIGHT returns the sum of the weights of ground soft constraints which are satisfied by a given solution, while MAXSAT returns a set of tuples which maximizes WEIGHT and satisfies all ground hard constraints.

Now we introduce the IPR algorithm. IPR (Algorithm 7) takes a Markov Logic Network C as input and produces a set of tuples T as output. The input C is divided into hard constraints C_h and soft constraints C_s. For elaboration, we assume the hard constraints C_h as satisfiable, allowing us to elide showing UNSAT as a possible alternative to output T. We next explain each component of IPR separately.

Eager proof exploitation. To start with, IPR computes an initial set of ground hard constraints ϕ and ground soft constraints ψ by exploiting the logical structure of the constraints (line 3–4 of Algorithm 7). The sets ϕ and ψ can be arbitrary subsets of the ground hard constraints and ground soft constraints in the full grounding. When ϕ and ψ are both empty, the behavior of IPR defaults to lazy approaches like CPI [84] (Definition 14).

As a concrete instance of eager proof exploitation, when a subset of the relational constraints have a recursive Horn form, IPR applies a Datalog solver to efficiently compute the least solution of this set of recursive Horn constraints, and find a relevant subset of clauses to be grounded upfront. In particular, when all the hard constraints are Horn, IPR prescribes a recipe for generating an optimal initial set of grounded constraints.

Theorem 15 shows that for hard relational constraints in Horn form, lazy approaches like CPI ground at least as many ground hard constraints as the number of true ground

facts in the least solution of such Horn constraints. Also, and more importantly, we show there exists a strategy that can upfront discover the set of all these necessary ground hard constraints and guarantee that no more ground constraints, besides those in the initial set, will be grounded. In practice, eager proof exploitation is employed for both hard and soft Horn constraints. Theorem 15 guarantees that if upfront grounding is applied to ground hard Horn constraints, then only relevant constraints are grounded. While this guarantee does not apply to upfront grounding of soft Horn constraints, we observe empirically that most such ground constraints are relevant. Moreover, as Section 4.1.3 shows, using this strategy for initial grounding allows the iterative process to terminate in far fewer iterations while ensuring that each iteration does approximately as much work as before (without initial grounding).

Definition 14. (Lazy Algorithm) Algorithm `Lazy` is an instance of IPR with $\phi = \psi = \emptyset$ as the initial grounding.

Theorem 15 (**Optimal initial grounding for Horn constraints).** *If a Markov Logic Network comprises a set of hard constraints C_h, each of which is a Horn constraint $\bigwedge_{i=1}^{n} l_i \implies l_0$, whose least solution is desired:*

$$T = \mathsf{lfp}\ \lambda T'.\ T' \cup \{\ [\![l_0]\!](\sigma) \mid (\textstyle\bigwedge_{i=1}^{n} l_i \implies l_0) \in \mathbf{C}_h \\ \wedge\ \forall i \in [1,n].[\![l_i]\!](\sigma) \in T' \wedge \sigma \in \Sigma\ \},$$

then for such a system, (a) $\texttt{Lazy}(C_h, \emptyset)$ grounds at least $|T|$ constraints, and (b) $\textsc{cegar}(C_h, \emptyset)$ with the initial grounding ϕ does not ground any more constraints where

$$\phi = \bigcup\{\ \bigvee_{i=1}^{n} \neg[\![l_i]\!](\sigma) \vee [\![l_0]\!](\sigma) \mid (\bigwedge_{i=1}^{n} l_i \implies l_0) \in C_h \wedge \forall i \in [0,n].[\![l_i]\!](\sigma) \in T \wedge \sigma \in \Sigma\ \}.$$

Lazy counterexample refutation. After generating the initial grounding, IPR iteratively grounds more constraints and refines the solution by refutation (lines 6–14 of Algorithm 7).

In each iteration, the algorithm keeps track of the previous solution T, and the weight w of the solution T. Initially, the solution is empty with weight zero (line 5).

In line 7, IPR computes all the violations of the hard constraints for the previous solution T using `Violations` (defined at the beginning of this chapter). Similarly, in line 8, the set of ground soft constraints ψ' violated by the previous solution T is computed. In line 9, both sets of violations ϕ' and ψ' are added to the corresponding sets of ground hard constraints ϕ and ground soft constraints ψ respectively. The intuition for adding violated hard ϕ' to the set ϕ is straightforward—the set of ground hard constraints ϕ is not sufficient to prevent the MAXSAT procedure from producing a solution T that violates the set of hard constraints H. The intuition for ground soft constraints is similar—since the goal of MAXSAT is to maximize the sum of the weights of satisfied soft constraints in C_s, and all weights in Markov Logic Networks are non-negative, any violation of a ground soft constraint possibly leads to a sub-optimal solution which could have been avoided if the violated constraint was present in the set of ground soft constraints ψ.

In line 10, this updated set ϕ of ground hard constraints and set ψ of ground soft constraints are fed to the MAXSAT procedure to produce a new solution T' and its corresponding weight w'. At this point, in line 12, the algorithm checks if the terminating condition is satisfied by the solution T'.

Termination check. IPR terminates when no hard constraints are violated by current solution ($\phi' = \emptyset$) and the current objective cannot be improved by adding more ground soft constraints ($w' = w$). This termination check improves upon that in previous works [84, 82], and speeds up the convergence of the algorithm in practice. Theorem 16 shows that our `CEGAR` algorithm always terminates with a sound and optimum solution if the underlying MAXSAT solver is exact.

Theorem 16 (Soundness and Optimality of IPR). *For any Markov Logic Network $C_h \cup C_s$ where hard constraints C_h is satisfiable, `CEGAR`(C) produces a sound and optimal solution.*

Table 4.1: Clauses in the initial grounding and additional constraints grounded in each iteration of IPR for graph reachability example.

Initial		
$\neg\mathsf{path}(0,0) \vee \neg\mathsf{edge}(0,1) \vee \mathsf{path}(0,1)$	$\neg\mathsf{path}(0,0) \vee \neg\mathsf{edge}(0,2) \vee \mathsf{path}(0,2)$	$\mathsf{path}(0,0)$
$\neg\mathsf{path}(1,1) \vee \neg\mathsf{edge}(1,3) \vee \mathsf{path}(1,3)$	$\neg\mathsf{path}(1,1) \vee \neg\mathsf{edge}(1,4) \vee \mathsf{path}(1,4)$	$\mathsf{path}(1,1)$
$\neg\mathsf{path}(2,2) \vee \neg\mathsf{edge}(2,5) \vee \mathsf{path}(2,5)$	$\neg\mathsf{path}(2,2) \vee \neg\mathsf{edge}(2,6) \vee \mathsf{path}(2,6)$	$\mathsf{path}(2,2)$
$\neg\mathsf{path}(0,1) \vee \neg\mathsf{edge}(1,3) \vee \mathsf{path}(0,3)$	$\neg\mathsf{path}(0,1) \vee \neg\mathsf{edge}(1,4) \vee \mathsf{path}(0,4)$	$\mathsf{path}(3,3)$
$\neg\mathsf{path}(0,2) \vee \neg\mathsf{edge}(2,5) \vee \mathsf{path}(0,5)$	$\neg\mathsf{path}(0,2) \vee \neg\mathsf{edge}(2,6) \vee \mathsf{path}(0,6)$	$\mathsf{path}(4,4)$
$\mathsf{path}(5,5)$	$\mathsf{path}(6,6)$	

Iteration 1		
$\neg\mathsf{path}(0,0)$ weight 1	$\neg\mathsf{path}(1,1)$ weight 1	$\neg\mathsf{path}(2,2)$ weight 1
$\neg\mathsf{path}(3,3)$ weight 1	$\neg\mathsf{path}(4,4)$ weight 1	$\neg\mathsf{path}(5,5)$ weight 1
$\neg\mathsf{path}(6,6)$ weight 1	$\neg\mathsf{path}(0,1)$ weight 1	$\neg\mathsf{path}(0,2)$ weight 1
$\neg\mathsf{path}(1,3)$ weight 1	$\neg\mathsf{path}(1,4)$ weight 1	$\neg\mathsf{path}(2,5)$ weight 1
$\neg\mathsf{path}(2,6)$ weight 1	$\neg\mathsf{path}(0,3)$ weight 1	$\neg\mathsf{path}(0,4)$ weight 1
$\neg\mathsf{path}(0,5)$ weight 1	$\neg\mathsf{path}(0,6)$ weight 1	

Example. The IPR algorithm takes two iterations and grounds 17 ground hard constraints and 17 ground soft constraints to solve the graph reachability example in Figures 4.1 and 4.2. Table 4.1 shows the constraints in the initial grounding computed using a Datalog solver and additional constraints grounded in each iteration of CEGAR. IPR grounds no additional constraints in Iteration 2. Therefore, the corresponding table is omitted in Table 4.1.

On the other hand, an eager approach with full grounding needs to ground 392 hard constraints and 49 soft constraints, which is 12× of the number of constraints grounded in CEGAR. Moreover, the eager approach generates ground constraints such as $\neg\mathsf{path}(0,1) \vee \neg\mathsf{edge}(1,5) \vee \mathsf{path}(0,5)$ and $\neg\mathsf{path}(1,4) \vee \neg\mathsf{edge}(4,2) \vee p(1,2)$, that are trivially satisfied given the input edge relation.

A lazy approach with an empty initial grounding grounds the same number of hard constraints and soft constraints as CEGAR. However, it takes 5 iterations to terminate, which is 2.5× of the number of iterations needed by CEGAR.

The results on the graph reachability example show that, IPR combines the benefits of the eager approach and the lazy approach while avoiding their drawbacks. □

Table 4.2: Statistics of application constraints and datasets.

	# relations	# rules	# EDB tuples			# clauses in full grounding		
			E1	E2	E3	E1	E2	E3
PA	94	89	1,274	3.9×10^6	1.1×10^7	2×10^{10}	1.6×10^{28}	1.2×10^{30}
AR	14	24	607	3,595	7,010	1.7×10^8	1.1×10^9	2.4×10^{10}
RC	5	17	1,656	3,190	7,766	7.9×10^{11}	1.2×10^{13}	3.8×10^{14}

Table 4.3: Results of evaluating CPI, IPR1, ROCKIT, IPR2, and TUFFY, on three benchmark applications. CPI and IPR1 use LBX as the underlying solver, while ROCKIT and IPR2 use GUROBI. In all experiments, we used a memory limit of 64GB and a time limit of 24 hours. Timed out experiments (denoted '–') exceeded either of these limits.

	EDB	# iterations				total time (m = min., s = sec.)					# ground constraints (K = thousand, M = million)					solution cost (K = thousand, M = million)				
		CPI	IPR1	ROCKIT	IPR2	CPI	IPR1	ROCKIT	IPR2	TUFFY	CPI	IPR1	ROCKIT	IPR2	TUFFY	CPI	IPR1	ROCKIT	IPR2	TUFFY
PA	E1	23	6	19	3	29s	28s	53s	13s	6m56s	0.6K	0.6K	0.6K	0.6K	0.6K	0	0	0	0	743
	E2	171	8	185	3	235m	23m	286m	150m	–	3M	3.2M	2.9M	3.2M	–	46	46	35	35	–
	E3	–	11	–	–	–	114m	–	–	–	–	13M	–	–	–	–	345	–	–	–
AR	E1	6	5	4	3	8s	8s	4s	8s	2m25s	9.8K	9.8K	27K	83K	589K	6.4K	6.4K	5.8K	6.1K	7K
	E2	6	6	7	7	34m	36m	37s	42s	–	2.3M	2.3M	0.4M	0.4M	–	0.4M	0.4M	0.39M	0.39M	–
	E3	6	6	7	7	141m	124m	1m45s	2m1s	–	8M	8M	1.4M	1.4M	–	0.68M	0.68M	0.67M	0.67M	–
RC	E1	17	6	5	4	3m5s	1m4s	6s	6s	11s	0.5M	0.3M	55K	68K	28K	5.7K	5.7K	5.7K	5.7K	160K
	E2	8	8	5	3	6m19s	3m41s	9s	10s	2m28s	2.4M	1.3M	0.11M	0.17M	64K	10.7K	10.7K	10.6K	10.6K	42.7K
	E3	17	13	20	20	150m	46m20s	2m35s	2m54s	180m	14M	4.5M	0.5M	1.2M	0.35M	25.1K	25.1K	24.9K	24.9K	0.25M

4.1.3 Empirical Evaluation

In this section, we evaluate CEGAR and compare it with three state-of-the-art approaches, CPI [84], ROCKIT [83], and TUFFY [80], on three different benchmarks with three different-sized inputs per benchmark.

We implemented CEGAR in roughly 10,000 lines of Java. To compute the initial ground constraints, we use bddbddb [15], a Datalog solver. The same solver is used to identify grounded constraints that are violated by a solution. For our evaluation, we use two different instances of CEGAR, referred as IPR1 and IPR2, that vary in the underlying solver used for solving the constraints. For IPR1, we use LBX [94] as the underlying WPMS solver which guarantees soundness of the solution (i.e., does not violate any hard clauses). Though LBX does not guarantee the optimality of the solution, in practice, we find the cost of the solution computed by LBX is close to that of the solution computed by an exact solver. For IPR2, we use GUROBI as the underlying solver. GUROBI is an integer linear program (ILP) solver which guarantees soundness of the solution. Additionally, it guarantees that the cost of the generated solution is within a limited bound from that of the optimal

solution. We incorporate it in our approach by replacing the call to MAXSAT (line 10 of Algorithm 7) with a call to an ILP encoder followed by a call to GUROBI. The ILP encoder translates the WPMS problem to an equivalent ILP formulation.

All experiments were done using Oracle HotSpot JVM 1.6.0 on a Linux server with 64GB RAM and 3.0GHz processors. The three benchmarks are as follows:

Program Analysis (PA): We choose static bug detection as a representative for the previously discussed program analysis applications. The datasets for this application are derived from a pointer analysis on three real-world Java programs ranging in size from 1.4K to 190K lines of code.

Advisor Recommendation (*AR*): This is an advisor recommendation system to aid new graduate students in finding good PhD advisors. The datasets for this application were generated from the AI Genealogy Project (http://aigp.eecs.umich.edu) and from DBLP (http://dblp.uni-trier.de).

Relational Classification (*RC*): In this application, papers from the Cora dataset [95] are classified into different categories based on the authors and their main area of research.

Table 4.2 shows statistics of our three benchmarks (PA, *AR*, *RC*) and the corresponding EDBs used in our evaluation.

We compare IPR1 and IPR2 with three state-of-the-art techniques, CPI, ROCKIT, and TUFFY.

TUFFY employs a non-iterative approach which is composed of two steps: first, it generates an initial set of grounded constraints that is expected to be a superset of all the required grounded constraints; next, TUFFY uses a highly efficient but approximate WPMS solver to solve these grounded constraints. In our evaluation, we use the TUFFY executable available from its website http://i.stanford.edu/hazy/hazy/tuffy/.

CPI is a fully lazy iterative approach, which refutes counterexamples in a way similar to CEGAR. However, CPI does not employ proof exploitation and applies a more conservative termination check. In order to ensure a fair comparison between CEGAR and CPI, we

implement CPI in our framework by incorporating the above two differences and use LBX as the underlying solver.

ROCKIT is also a fully lazy iterative approach similar to CPI. Additionally, like CPI, ROCKIT does not employ proof exploitation and its termination check is as conservative as CPI. The main innovation of ROCKIT is a clever ILP encoding for solving the underlying constraints. This reduces the time per iteration for solving, but does not necessarily reduce the number of iterations. In our evaluation, we use the ROCKIT executable available from its website https://code.google.com/p/rockit/.

The primary innovation of ROCKIT is complementary to our approach. In fact, to ensure a fair comparison between CEGAR and ROCKIT, in IPR2 we use the same ILP encoding as used by ROCKIT. This combined approach yields the benefits of both ROCKIT and our approach.

Table 4.3 summarizes the results of running CPI, IPR1, ROCKIT, IPR2, and TUFFY on our benchmarks. IPR1 significantly outperforms CPI in terms of running time. Similarly, IPR2 outperforms ROCKIT in running time and the number of iterations needed while TUFFY has the worst performance. IPR1 terminates under all nine experiment settings, while IPR2, CPI and ROCKIT terminate under eight settings and TUFFY only terminates under five settings. In terms of the quality of the solution, IPR1 and CPI produce solutions with similar costs under all settings. IPR2 and ROCKIT produce solutions with slightly lower costs as they employ an ILP solver that guarantees a solution whose cost is within a fixed bounded from that of the optimal solution. TUFFY produces solutions with significantly higher costs. We next study the results for each benchmark more closely.

Program Analysis (PA): For PA, we first compare IPR1 with CPI. IPR1 significantly outperforms CPI on larger datasets, with CPI not terminating on E3 even after 24 hours while IPR1 terminates in under two hours. This is because most of the relational constraints in PA are Horn, allowing IPR1 to effectively perform eager proof exploitation, and ground relevant clauses upfront. This is also reflected in the reduced number of iterations for IPR1

compared to CPI. IPR2, that also uses eager proof exploitation, similarly outperforms ROCKIT on E1 and E2. However, both IPR2 and ROCKIT fail to terminate on E3. This indicates that the underlying type of solver plays an important role in the performance of these approaches. For PA, the WPMS solver employed by IPR1 is better suited to the problem compared to the ILP solver employed by IPR2. TUFFY performs the worst out of all the approaches, only terminating on the smallest dataset E1. Even for E1, the cost of the final solution generated by TUFFY is significantly higher compared to the other approaches. More acutely, TUFFY violates *ten ground hard constraints* on E1 which is absolutely unacceptable for program analysis benchmarks.

Advisor Recommendation (*AR*): On *AR*, IPR1, IPR2, CPI and ROCKIT terminate on all three datasets and produce similar results while TUFFY only terminates on the smallest dataset. IPR1 and CPI have comparable performance on *AR* as a fully lazy approach suffices to solve the relational constraint system efficiently. Similarly, IPR2 and ROCKIT have similar performance. However, both IPR2 and ROCKIT significantly outperform IPR1 and CPI in terms of the running time although they need similar number of iterations. This indicates that for *AR*, the smart ILP encoding leads to fewer constraints and also that the ILP solver is better suited than the WPMS solver, leading to a lower running time per iteration. Note that, all theses approaches except TUFFY terminate within seven iterations on all three datasets. On the other hand, TUFFY times out on the two larger datasets without passing its grounding phase, reflecting the need for lazily grounding the constraints.

Relational Classification (*RC*): All the approaches terminate on *RC* for all the three datasets. However, IPR1 outperforms CPI and TUFFY significantly in terms of runtime on the largest dataset. On the other hand, IPR2 and ROCKIT outperform IPR1. This is again due to the faster solving time per iteration enabled by the smart ILP encoding and the use of ILP solver. Unlike other benchmarks, the running time of TUFFY is comparable to the other approaches. However, the costs of its solutions are on average 14$\times$ more than the costs of the solutions produced by the other approaches.

4.1.4 Related Work

A large body of work exists to solve weighted relational constraints in a lazy manner. Lazy inference techniques [93, 96] rely on the observation that most ground facts in the final solution to a relational inference problem have default values (a value appearing much more often than the others). These techniques start by assuming a default value for all ground facts, and gradually ground clauses as the values of facts are determined to be changed for a better objective. Such techniques apply a loose termination check, and therefore do not guarantee soundness nor optimality of the final solution. Iterative ground-and-solve approaches such as CPI [84] and RockIt [83] solve constraints lazily by iteratively grounding only those constraints that are violated by the current solution. Compared to lazy inference techniques, they apply a conservative termination check which guarantees the soundness and optimality of the solution. Compared to our approach, all of the above techniques either need more iterations to converge or have to terminate prematurely with potentially unsound and suboptimal solutions.

Tuffy [80] applies a non-iterative technique which is divided into a grounding phase and a solving phase. In the grounding phase, Tuffy grounds a set of constraints that it deems relevant to the solution of the inference problem. In practice, this set is often imprecise, which either hinders the scalability of the overall process (when the set is too large), or degrades the quality of the solution (when the set is too small). Soft-CEGAR [82] grounds the soft constraints eagerly and solves the hard constraints in a lazy manner. It uses a SMT solver to efficiently find the hard constraints violated by a solution. Lifted inference techniques [97, 98, 99] use approaches from first-order logic, like variable elimination, to simplify the system of relational constraints. Such techniques can be used in conjunction with various iterative techniques including our approach for solving such constraints.

4.1.5 Conclusion

We presented a new technique for grounding Markov Logic Networks. Existing approaches either ground the constraints too eagerly, which produces intractably large propositional instances to WPMS solvers, or they ground the constraints too lazily, which prohibitively slows down the convergence of the overall process. Our approach strikes a balance between these two extremes by applying two complementary optimizations: eagerly exploiting proofs and lazily refuting counterexamples. Our empirical evaluation showed that our technique achieves significant improvement over two existing techniques in both performance and quality of the solution.

4.2 Query-Guided Maximum Satisfiability

4.2.1 Introduction

The maximum satisfiability or MAXSAT problem [100] is an optimization extension of the well-known satisfiability or SAT problem [101, 102]. A MAXSAT formula consists of a set of conventional SAT or hard clauses together with a set of soft or weighted clauses. The solution to a MAXSAT formula is an assignment to its variables that satisfies all the hard clauses, and maximizes the sum of the weights of satisfied soft clauses. A number of interesting problems that span across a diverse set of areas such as program reasoning [8, 9, 103, 104], information retrieval [105, 106, 80, 81], databases [107, 108], circuit design [109], bioinformatics [110, 111], planning and scheduling [112, 113, 114], and many others can be naturally encoded as MAXSAT formulae. Many of these problems specify hard constraints that encode various soundness conditions, and soft constraints that specify objectives to optimize.

MAXSAT solvers have made remarkable strides in performance over the last decade [115, 116, 117, 118, 119, 120, 121, 122, 123]. This in turn has motivated even more demanding and emerging applications to be formulated using MAXSAT. Real-world in-

stances of many of these problems, however, result in very large MAXSAT formulae [124]. These formulae, comprising millions of clauses or more, are beyond the scope of existing MAXSAT solvers. Fortunately, for many of these problems, one is interested in a small set of *queries* that constitute a very small fraction of the entire MAXSAT solution. For instance, in program analysis, a query could be analysis information for a particular variable in the program—intuitively, one would expect the computational cost for answering a small set of queries to be much smaller than the cost of computing analysis information for all program variables. In the MAXSAT setting, the notion of a query translates to the value of a specific variable in a MAXSAT solution. Given a MAXSAT formula φ and a set of queries $\mathcal{Q}$, one obvious method for answering queries in $\mathcal{Q}$ is to compute the MAXSAT solution to φ and project it to variables in $\mathcal{Q}$. Needless to say, especially for very large MAXSAT formulae, this is a non-scalable solution. Therefore, it is interesting to ask the following question: "*Given a* MAXSAT *formula φ and a set of queries $\mathcal{Q}$, is it possible to answer $\mathcal{Q}$ by only computing information relevant to $\mathcal{Q}$?*". We call this question the *query-guided maximum satisfiability* or Q-MAXSAT problem $(\varphi, \mathcal{Q})$.

Our main technical insight is that a Q-MAXSAT instance $(\varphi, \mathcal{Q})$ can be solved by computing a MAXSAT solution of a small subset of the clauses in φ. The main challenge, however, is how to efficiently determine whether the answers to $\mathcal{Q}$ indeed correspond to a MAXSAT solution of φ. We propose an iterative algorithm for solving a Q-MAXSAT instance $(\varphi, \mathcal{Q})$ (Algorithm 8 in Section 4.2.4). In each iteration, the algorithm computes a MAXSAT solution to a subset of clauses in φ that are relevant to $\mathcal{Q}$. We also define an algorithm (Algorithm 9 in Section 4.2.4.1) that efficiently checks whether the current assignment to variables in $\mathcal{Q}$ corresponds to a MAXSAT solution to φ. In particular, our algorithm constructs a small set of clauses that succinctly summarize the effect of the clauses in φ that are not considered by our algorithm, and then uses it to overestimate the gap between the optimum objective value of φ under the current assignment and the optimum objective value of φ.

We have implemented our approach in a tool called PILOT and applied it to 19 large MAXSAT instances ranging in size from 100 thousand to 22 million clauses generated from real-world problems in program analysis and information retrieval. Our empirical evaluation shows that PILOT achieves significant improvements in runtime and memory over conventional MAXSAT solvers: on these instances, PILOT used only 285 MB of memory on average and terminated in 107 seconds on average. In contrast, conventional MAXSAT solvers timed out for eight of the instances.

In summary, the main contributions of this paper are as follows:

1. We introduce and formalize a new optimization problem called Q-MAXSAT. In contrast to traditional MAXSAT, where one is interested in an assignment to all variables, for Q-MAXSAT, we are interested only in an assignment to a subset of variables, called queries.

2. We propose an iterative algorithm for Q-MAXSAT that has the desirable property of being able to efficiently check whether an assignment to queries is an optimal solution to Q-MAXSAT.

3. We present empirical results showing the effectiveness of our approach by applying PILOT to several Q-MAXSAT instances generated from real-world problems in program analysis and information retrieval.

4.2.2 Example

We illustrate the Q-MAXSAT instance and our solution with the help of an example. Figure 4.3 represents a large MAXSAT formula φ in conjunctive form. Each variable v_i in φ is represented as a node in the graph labeled by its subscript i. Each clause is a disjunction of literals with a positive weight. Nodes marked as **T** (or **F**) indicate soft clauses of the form $(100, v_i)$ (or $(100, \neg v_i)$) while each edge from node i to a node j represents a soft clause of the form $(5, \neg v_i \vee v_j)$.

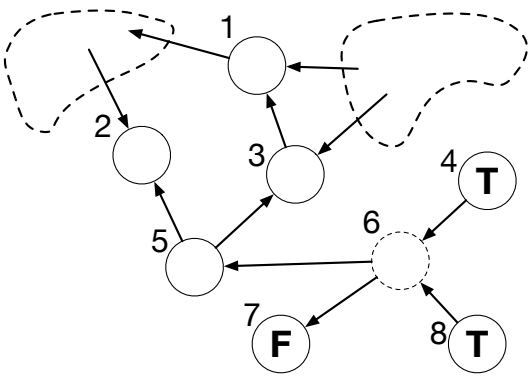


Figure 4.3: Graph representation of a large MAXSAT formula φ.

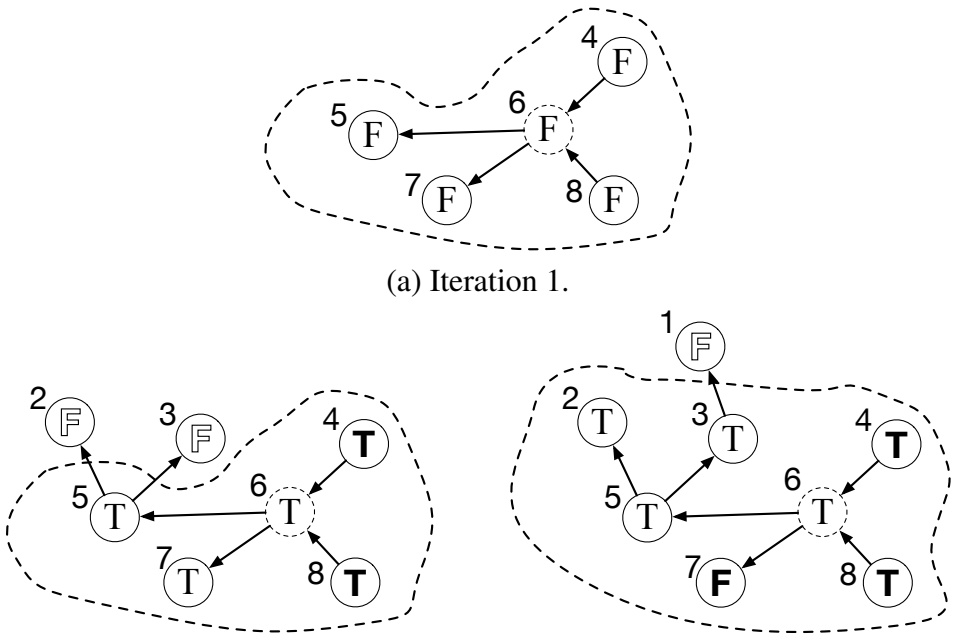
Suppose we are interested in the assignment to the query v_6 in φ (shown by the dashed node in Figure 4.3). Then, the Q-MAXSAT instance that we want to solve is $(\varphi, \{v_6\})$. A solution to this Q-MAXSAT instance is a partial model which maps v_6 to $true$ or $false$ such that there is a completion of this partial model maximizing the objective value of φ.

A naive approach to solve this problem is to directly feed φ into a MAXSAT solver and extract the assignment to v_6 from the solution. However, this approach is highly inefficient due to the large size of practical instances.

Our approach exploits the fact that the Q-MAXSAT instance only requires assignment to specific query variables, and solves the problem lazily in an iterative manner. The high level strategy of our approach (see Chapter 4.2.4 for details) is to only solve a subset of clauses relevant to the query in each iteration, and terminate when the current solution can be proven to be the solution to the Q-MAXSAT instance. In particular, our approach proceeds in the following four steps.

Initialize. Given a Q-MAXSAT instance, our query-guided approach constructs an initial set of clauses that includes all the clauses containing a query variable. We refer to these relevant clauses as the *workset* of our algorithm.

Solve. The next step is to solve the MAXSAT instance induced by the workset. Our approach uses an existing off-the-shelf MAXSAT solver for this purpose. This produces a partial model of the original instance.



(a) Iteration 1.

(b) Iteration 2.

(c) Iteration 3.

Figure 4.4: Graph representation of each iteration in our algorithm when it solves the Q-MAXSAT instance $(\varphi, \{v_6\})$.

Check. The key step in our approach is to check whether the current partial model can be completed to a model of the original MAXSAT instance. Since the workset only includes a small subset of clauses from the complete instance, the challenge here is to summarize the effect of the unexplored clauses on the assignment to the query variables. We propose a novel technique for performing this check efficiently without actually considering all the unexplored clauses (see Chapter 4.2.4.1 for formal details).

Expand. If the check in the previous step fails, it indicates that we need to grow our workset of relevant clauses. Based on the check failure, in this step, our approach identifies the set of clauses to be added to the workset for the next iteration.

As long as the Q-MAXSAT instance is finite, our iterative approach is guaranteed to terminate since it only grows the workset in each iteration.

We next describe how our approach solves the Q-MAXSAT instance $(\varphi, \{v_6\})$. To resolve v_6, our approach initially constructs the workset φ' that includes all the clauses containing v_6. We represent φ' by the subgraph contained within the dotted area in Fig-

ure 4.4(a). By invoking a MAXSAT solver on φ', we get a partial model $\alpha_{\varphi'} = [v_4 \mapsto \mathit{false}, v_5 \mapsto \mathit{false}, v_6 \mapsto \mathit{false}, v_7 \mapsto \mathit{false}, v_8 \mapsto \mathit{false}]$ as shown in Figure 4.4(a), with an objective value of 20. Our approach next checks if the current partial model found is a solution to the Q-MAXSAT instance $(\varphi, \{v_6\})$. It constructs a set of clauses ψ which succinctly summarizes the effect of the clauses that are not present in the workset. We refer to this set as the *summarization set*, and use the following expression to overestimate the gap between the optimum objective value of φ under the current partial model and the optimum objective value of φ:

$$\max_{\alpha} eval(\varphi' \cup \psi, \alpha) - \max_{\alpha} eval(\varphi', \alpha),$$

where $\max_\alpha eval(\varphi' \cup \psi, \alpha)$ and $\max_\alpha eval(\varphi', \alpha)$ are the optimum objective values of $\varphi' \cup \psi$ and φ', respectively.

To construct such a summarization set, our insight is that the clauses that are not present in the workset can only affect the query assignment via clauses sharing variables with the workset. We call such clauses as the *frontier clauses*. Furthermore, if a frontier clause is already satisfied by the current partial model, expanding the workset with such a clause cannot further improve the partial model. We now try to construct a summarization set ψ by taking all frontier clauses not satisfied by $\alpha_{\varphi'}$. As a result, our algorithm produces $\psi = \{(100, v_4), (100, v_8)\}$. The check comparing the optimum objective values of φ' and $\varphi' \cup \psi$ fails in this case. In particular, by solving $\varphi' \cup \psi$, we get a partial model $\alpha_{\varphi' \cup \psi} = [v_4 \mapsto \mathit{true}, v_5 \mapsto \mathit{true}, v_6 \mapsto \mathit{true}, v_7 \mapsto \mathit{true}, v_8 \mapsto \mathit{true}]$ with 220 as the objective value, which is greater than the optimum objective value of φ'. As a consequence, our approach expands the workset φ' with $(100, v_4)$ and $(100, v_8)$ and proceeds to the next iteration. We include these two clauses as these are not satisfied by $\alpha_{\varphi'}$ in the last iteration, but satisfied by $\alpha_{\varphi' \cup \psi}$, which indicates they are likely responsible for the failure of the previous check.

In iteration 2, invoking a MAXSAT solver on the new workset, φ', produces $\alpha_{\varphi'} = [v_4 \mapsto \mathit{true}, v_5 \mapsto \mathit{true}, v_6 \mapsto \mathit{true}, v_7 \mapsto \mathit{true}, v_8 \mapsto \mathit{true}]$, as shown in Figure 4.4(b), with 220 as the objective value. The violated frontier clauses in this case are $(100, \neg v_7)$, $(5, \neg v_5 \vee v_2)$, and $(5, \neg v_5 \vee v_3)$. However, if this set of violated frontier clauses were to be used as the summarization set ψ, the newly added variables v_2 and v_3 will cause the check comparing the optimum objective values of φ' and $\varphi' \cup \psi$ to trivially fail. To address this problem, and further improve the precision of our check, we modify the summarization set as $\psi = \{(100, \neg v_7), (5, \neg v_5), (5, \neg v_5)\}$ by removing v_2 and v_3 and strengthening the corresponding clauses. In this case, it is equivalent to setting v_2 and v_3 to false (marked as $\mathbb{F}$ in the graph). Intuitively, by setting these variables to false, we are overestimating the effects of the unexplored clauses, by assuming these frontier clauses will not be satisfied by unexplored variables like v_2 and v_3. By solving $\varphi' \cup \psi$, we get a partial model $\alpha_{\varphi' \cup \psi} = [v_4 \mapsto \mathit{true}, v_5 \mapsto \mathit{false}, v_6 \mapsto \mathit{true}, v_7 \mapsto \mathit{false}, v_8 \mapsto \mathit{true}]$ with 320 as the objective value, which is greater than the optimum objective value of φ'.

In iteration 3, as shown in Figure 4.4(c), our approach expands the workset φ' with $(100, \neg v_7)$, $(5, \neg v_5 \vee v_2)$ and $(5, \neg v_5 \vee v_3)$. By invoking a MAXSAT solver on φ', we produce $\alpha_{\varphi'} = [v_2 \mapsto \mathit{true}, v_3 \mapsto \mathit{true}, v_4 \mapsto \mathit{true}, v_5 \mapsto \mathit{true}, v_6 \mapsto \mathit{true}, v_7 \mapsto \mathit{false}, v_8 \mapsto \mathit{true}]$ with an objective value of 325. We omit the edges representing frontiers clauses that are already satisfied by $\alpha_{\varphi'}$ in Figure 4.4(c). To check the whether current partial model is a solution to the Q-MAXSAT instance, we construct strengthened summarization set $\psi = \{(5, \neg v_3)\}$ from the frontier clause $(5, \neg v_3 \vee v_1)$. By invoking MAXSAT on $\varphi' \cup \psi$, we get an optimum objective value of 325 which is the same as that of φ'. As a result, our algorithm terminates and extracts $[v_6 \mapsto \mathit{true}]$ as the solution to the Q-MAXSAT instance.

Despite the fact that many clauses and variables can reach v_6 in the graph (i.e, they might affect the assignment to the query), our approach successfully resolves v_6 in three iterations and only explores a very small, local subset of the graph, while successfully summarizing the effects of the unexplored clauses.

$$
\begin{array}{rrcl}
\textit{(variable)} & v & \in & \mathcal{V} \\
\textit{(clause)} & c & ::= & \bigvee_{i=1}^{n} v_i \vee \bigvee_{j=1}^{m} \neg v'_j \mid 1 \mid 0 \\
\textit{(weight)} & w & \in & \mathbb{R}^+ \\
\textit{(soft clause)} & s & ::= & (w, c) \\
\textit{(formula)} & \varphi & ::= & \{s_1, ..., s_n\} \\
\textit{(model)} & \alpha & \in & \mathcal{V} \rightarrow \{0, 1\}
\end{array}
$$

$$
\mathit{fst} = \lambda(w, c).w, \quad \mathit{snd} = \lambda(w, c).c
$$

$$
\forall \alpha : \alpha \models 1, \quad \forall \alpha : \alpha \not\models 0
$$

$$
\begin{array}{rcl}
\alpha \models \bigvee_{i=1}^{n} v_i \vee \bigvee_{j=1}^{m} \neg v'_j & \Leftrightarrow & \exists i : \alpha(v_i) = 1 \text{ or } \exists j : \alpha(v'_j) = 0 \\
\mathit{eval}(\{s_1, ..., s_n\}, \alpha) & = & \Sigma_i^n \{\mathit{fst}(s_i) \mid \alpha \models \mathit{snd}(s_i)\} \\
\mathit{MaxSAT}(\varphi) & = & \arg\max_\alpha \mathit{eval}(\varphi, \alpha)
\end{array}
$$

Figure 4.5: Syntax and interpretation of MAXSAT formulae.

4.2.3 The Q-MAXSAT Problem

First, we describe the standard MAXSAT problem [100]. The syntax of a MAXSAT formula is shown in Figure 4.5. A MAXSAT formula φ consists of a set of *soft clauses*. Each soft clause $s = (w, c)$ is a pair that consists of a positive weight $w \in \mathbb{R}^+$, and a clause c that is a disjunctive form over a set of variables $\mathcal{V}$ [2]. We use 1 to denote true and 0 to denote false. Given an assignment $\alpha : \mathcal{V} \rightarrow \{0, 1\}$ to each variable in a MAXSAT formula φ, we use $\mathit{eval}(\varphi, \alpha)$ to denote the sum of the weights of the soft clauses in the formula that are satisfied by the assignment. We call this sum the *objective value* of the formula under that assignment. The space of *models* $\mathit{MaxSAT}(\varphi)$ of a MAXSAT formula φ is the set of all assignments that maximize the objective value of φ.

The query-guided maximum satisfiability or Q-MAXSAT problem is an extension to the MAXSAT problem, that augments the MAXSAT formula with a set of *queries* $\mathcal{Q} \subseteq \mathcal{V}$. In contrast to the MAXSAT problem, where the objective is to find an assignment to all variables $\mathcal{V}$, Q-MAXSAT only aims to find a *partial model* $\alpha_\mathcal{Q} : \mathcal{Q} \rightarrow \{0, 1\}$. In particular, given a MAXSAT formula φ and a set of queries $\mathcal{Q}$, the Q-MAXSAT problem seeks a partial model $\alpha_\mathcal{Q} : \mathcal{Q} \rightarrow \{0, 1\}$ for φ, that is, an assignment to the variables in $\mathcal{Q}$ such that there

[2] Without loss of generality, we assume that MAXSAT formulae contain only soft clauses. Every hard clause can be converted into a soft clause with a sufficiently high weight.

exists a completion $\alpha : \mathcal{V} \rightarrow \{0,1\}$ of $\alpha_{\mathcal{Q}}$ that is a model of the MAXSAT formula φ. Formally:

Definition 17 (Q-MAXSAT **problem).** Given a MAXSAT formula φ, and a set of queries $\mathcal{Q} \subseteq \mathcal{V}$, a model of the Q-MAXSAT instance $(\varphi, \mathcal{Q})$ is a partial model $\alpha_{\mathcal{Q}} : \mathcal{Q} \rightarrow \{0,1\}$ such that

$$\exists \alpha \in \mathit{MaxSAT}(\varphi).(\forall v \in \mathcal{Q}.\alpha_{Q}(v) = \alpha(v)).$$

Example. Let $(\varphi, \mathcal{Q})$ where $\varphi = \{(5, \neg a \vee b), (5, \neg b \vee c), (5, \neg c \vee d), (5, \neg d)\}$ and $\mathcal{Q} = \{a\}$ be a Q-MAXSAT instance. A model of this instance is given by $\alpha_{\mathcal{Q}} = [a \mapsto 0]$. Indeed, there is a completion $\alpha = [a \mapsto 0, b \mapsto 0, c \mapsto 0, d \mapsto 0]$ that belongs to $\mathit{MaxSAT}(\varphi)$ (in other words, α maximizes the objective value of φ, which is equal to 20), and agrees with $\alpha_{\mathcal{Q}}$ on the variable a (that is, $\alpha(a) = \alpha_{\mathcal{Q}}(a) = 0$). □

Hereafter, we use $\mathit{Var}(\varphi)$ to denote the set of variables occurring in MAXSAT formula φ. For a set of variables $\mathcal{U} \subseteq \mathcal{V}$, we denote the partial model $\alpha : \mathcal{U} \rightarrow \{0,1\}$ by $\alpha_{\mathcal{U}}$. Also, we use α_{φ} as shorthand for $\alpha_{\mathit{Var}(\varphi)}$. Throughout the paper, we assume that for $\mathit{eval}(\varphi, \alpha)$, the assignment α is well-defined for all variables in $\mathit{Var}(\varphi)$.

4.2.4 Solving a Q-MAXSAT Instance

Algorithm 8 describes our technique for solving the Q-MAXSAT problem. It takes as input a Q-MAXSAT instance $(\varphi, \mathcal{Q})$, and returns a solution to $(\varphi, \mathcal{Q})$ as output. The main idea in the algorithm is to iteratively identify and solve a subset of clauses in φ that are relevant to the set of queries $\mathcal{Q}$, which we refer to as the *workset* of our algorithm.

The algorithm starts by invoking function INIT (line 3) which returns an initial set of clauses $\varphi' \subseteq \varphi$. Specifically, φ' is the set of clauses in φ which contain variables in the query set $\mathcal{Q}$.

In each iteration (lines 4–12), Algorithm 8 first computes a model $\alpha_{\varphi'}$ of the MAXSAT formula φ' (line 5). Next, in line 6, the function CHECK checks whether the model $\alpha_{\varphi'}$

Algorithm 8

1: **INPUT:** A Q-MAXSAT instance $(\varphi, \mathcal{Q})$, where φ is a MAXSAT formula, and $\mathcal{Q}$ is a set of queries.
2: **OUTPUT:** A solution to the Q-MAXSAT instance $(\varphi, \mathcal{Q})$.
3: $\varphi' := \text{INIT}(\varphi, \mathcal{Q})$
4: **while true do**
5: $\quad \alpha_{\varphi'} := \text{MAXSAT}(\varphi')$
6: $\quad \varphi'' := \text{CHECK}((\varphi, \mathcal{Q}), \varphi', \alpha_{\varphi'})$, where $\varphi'' \subseteq \varphi \setminus \varphi'$
7: $\quad$ **if** $\varphi'' = \emptyset$ **then**
8: $\quad\quad$ **return** $\lambda v.\alpha_{\varphi'}(v)$, where $v \in \mathcal{Q}$
9: $\quad$ **else**
10: $\quad\quad \varphi' := \varphi' \cup \varphi''$
11: $\quad$ **end if**
12: **end while**

Algorithm 9 $\text{CHECK}((\varphi, \mathcal{Q}), \varphi', \alpha_{\varphi'})$

1: **INPUT:** A Q-MAXSAT instance $(\varphi, \mathcal{Q})$, a MAXSAT formula $\varphi' \subseteq \varphi$, and a model $\alpha_{\varphi'} \in MaxSAT(\varphi')$.
2: **OUTPUT:** A set of clauses $\varphi'' \subseteq \varphi \setminus \varphi'$.
3: $\psi := \text{APPROX}(\varphi, \varphi', \alpha_{\varphi'})$
4: $\varphi'' := \varphi' \cup \psi$
5: $\alpha_{\varphi''} := \text{MAXSAT}(\varphi'')$
6: **if** $eval(\varphi'', \alpha_{\varphi''}) - eval(\varphi', \alpha_{\varphi'}) = 0$ **then**
7: $\quad$ **return** $\emptyset$
8: **else**
9: $\quad \varphi_s := \{(w, c) \mid (w, c) \in \psi \wedge \alpha_{\varphi''} \models c\}$
10: $\quad$ **return** $\text{REFINE}(\varphi_s, \varphi, \varphi', \alpha_{\varphi'})$
11: **end if**

is sufficient to compute a model of the Q-MAXSAT instance $(\varphi, \mathcal{Q})$. If $\alpha_{\varphi'}$ is sufficient, then the algorithm returns a model which is $\alpha_{\varphi'}$ restricted to the variables in $\mathcal{Q}$ (line 8). Otherwise, CHECK returns a set of clauses φ'' that is added to the set φ' (line 10). It is easy to see that Algorithm 8 always terminates if φ is finite.

The main and interesting challenge is the implementation of the CHECK function so that it is sound (that is, when CHECK returns $\emptyset$, then the model $\alpha_{\varphi'}$ restricted to the variables in $\mathcal{Q}$ is indeed a model of the Q-MAXSAT instance), yet efficient.

4.2.4.1 Implementing an Efficient CHECK *Function*

Algorithm 9 describes our implementation of the CHECK function. The input to the

CHECK function is a Q-MAXSAT instance $(\varphi, \mathcal{Q})$, a MAXSAT formula $\varphi' \subseteq \varphi$ as described in Algorithm 8, and a model $\alpha_{\varphi'} \in MaxSAT(\varphi')$. The output is a set of clauses φ'' that are required to be added to φ' so that the resulting MAXSAT formula $\varphi' \cup \varphi''$ is solved in the next iteration of Algorithm 8. If φ'' is empty, then this means that Algorithm 8 can stop and return the appropriate model (as described in line 8 of Algorithm 8).

CHECK starts by calling the function APPROX (line 3) which takes φ, φ' and $\alpha_{\varphi'}$ as inputs, and returns a new set of clauses ψ, which we refer to as the *summarization set*. APPROX analyzes clauses in $\varphi \setminus \varphi'$, and returns a much smaller formula ψ which allows us to overestimate the gap between the optimum objective value under current partial model $\alpha_{\varphi'}$ and the optimum objective value of φ. In line 5, CHECK computes a model $\alpha_{\varphi''}$ of the MAXSAT formula $\varphi'' = \varphi' \cup \psi$. Next, in line 6, CHECK compares the objective value of φ' with respect to $\alpha_{\varphi'}$ and the objective value of φ'' with respect to $\alpha_{\varphi''}$. If these objective values are equal, CHECK concludes that the partial assignment for the queries in $\alpha_{\varphi'}$ is a model to the Q-MAXSAT problem and returns an empty set (line 7). Otherwise, it computes φ_s in line 9 which is the set of clauses satisfied by $\alpha_{\varphi''}$ in ψ. Finally, in line 10, CHECK returns the set of clauses to be added to the MAXSAT formula φ'. This is computed by REFINE, which takes φ_s, φ, φ', and $\alpha_{\varphi'}$ as input. Essentially, REFINE identifies the clauses in $\varphi \setminus \varphi'$ which are likely responsible for failing the check in line 6, and uses them to expand the MAXSAT formula φ'.

Optimality check via APPROX**.** The core step of CHECK is line 6, which uses $eval(\varphi'', \alpha_{\varphi''}) - eval(\varphi', \alpha_{\varphi'})$ to overestimate the gap between the optimum objective value under current partial model $\alpha_{\varphi'}$ and the optimum objective value of φ. The key idea here is to apply APPROX to generate a *small* set of clauses ψ which succinctly summarizes the effect of the unexplored clauses $\varphi \setminus \varphi'$. We next describe the specification of the APPROX function, and formally prove the soundness of the optimality check in line 6.

Given a set of variables $\mathcal{U} \subseteq Var(\varphi)$, and an assignment $\alpha_{\mathcal{U}}$, we define a *substitution*

operation $\varphi[\alpha_{\mathcal{U}}]$ which simplifies φ by replacing all occurrences of variables in $\mathcal{U}$ with their corresponding values in assignment $\alpha_{\mathcal{U}}$. Formally,

$$\begin{aligned}
\{s_1, ..., s_n\}[\alpha_{\mathcal{U}}] &= \{s_1[\alpha_{\mathcal{U}}], ..., s_n[\alpha_{\mathcal{U}}]\} \\
(w, c)[\alpha_{\mathcal{U}}] &= (w, c[\alpha_{\mathcal{U}}]) \\
1[\alpha_{\mathcal{U}}] &= 1 \\
0[\alpha_{\mathcal{U}}] &= 0
\end{aligned}$$

$$(\textstyle\bigvee_{i=1}^{n} v_i \vee \bigvee_{j=1}^{m} \neg v'_j)[\alpha_{\mathcal{U}}] \quad =$$

$$\begin{cases}
1, & \text{if } \alpha_{\mathcal{U}} \models \bigvee_{i=1}^{n} v_i \vee \bigvee_{j=1}^{m} \neg v'_j, \\
0, & \text{if } \{v_1, .., v_n\} \subseteq \mathcal{U} \wedge \{v'_1, ..., v'_m\} \subseteq \mathcal{U} \wedge \\
& \quad \alpha_{\mathcal{U}} \not\models \bigvee_{i=1}^{n} v_i \vee \bigvee_{j=1}^{m} \neg v'_j, \\
\bigvee_{v \in \{v_1,...,v_n\} \setminus \mathcal{U}} v \vee \bigvee_{v' \in \{v'_1,...,v'_n\} \setminus \mathcal{U}} \neg v', & \text{otherwise.}
\end{cases}$$

Definition 18 (Summarizing unexplored clauses). Given two MAXSAT formulae φ and φ' such that $\varphi' \subseteq \varphi$, and $\alpha_{\varphi'} \in MaxSAT(\varphi')$, we say $\psi = \text{APPROX}(\varphi, \varphi', \alpha_{\varphi'})$ *summarizes* the effect of $\varphi \setminus \varphi'$ with respect to $\alpha_{\varphi'}$ if and only if:

$$\begin{aligned}
\max_\alpha eval(\varphi' \cup \psi, \alpha) \;\geq\; & \max_\alpha eval(\varphi, \alpha) - \\
& \max_\alpha eval((\varphi \setminus \varphi')[\alpha_{\varphi'}], \alpha)
\end{aligned}$$

We state two useful facts about $eval$ before proving soundness.

Proposition 19. Let φ and φ' be two MAXSAT formulae such that $\varphi \cap \varphi' = \emptyset$. Then, $\forall \alpha. eval(\varphi \cup \varphi', \alpha) = eval(\varphi, \alpha) + eval(\varphi', \alpha)$.

Proposition 20. Let φ and φ' be two MAXSAT formulae. Then,

$$\begin{aligned}
\max_\alpha\ eval(\varphi, \alpha) + \max_\alpha\ eval(\varphi', \alpha) \geq \\
\max_\alpha\ (eval(\varphi, \alpha) + eval(\varphi', \alpha)).
\end{aligned}$$

The theorem below states the soundness of the optimality check performed in line 6 in CHECK.

Theorem 21 (Soundness of optimality check). *Given a* Q-MAXSAT *instance* $(\varphi, \mathcal{Q})$*, a* MAXSAT *formula* $\varphi' \subseteq \varphi$ *s.t.* $Vars(\varphi') \supseteq \mathcal{Q}$*, a model* $\alpha_{\varphi'} \in MaxSAT(\varphi')$*, and* $\psi = \text{APPROX}(\varphi, \varphi', \alpha_{\varphi'})$ *such that the following condition holds:*

$$\max_{\alpha} eval(\varphi' \cup \psi, \alpha) = \max_{\alpha} eval(\varphi', \alpha).$$

Then, $\lambda v.\alpha_{\varphi'}(v)$*, where* $v \in \mathcal{Q}$ *is a model of the* Q-MAXSAT *instance* $(\varphi, \mathcal{Q})$.

Proof. Let $\alpha_{\mathcal{Q}} = \lambda v.\alpha_{\varphi'}(v)$, where $v \in \mathcal{Q}$. Let $\varphi'' = \varphi[\alpha_{\varphi'}]$. Construct a completion α_{φ} of $\alpha_{\mathcal{Q}}$ as follows:

$$\alpha_{\varphi} = \alpha_{\varphi'} \uplus \alpha_{\varphi''}, \text{ where } \alpha_{\varphi''} \in MaxSAT(\varphi'')$$

It suffices to show that $\alpha_{\varphi} \in MaxSAT(\varphi)$, that is, $eval(\varphi, \alpha_{\varphi}) = \max_{\alpha}\ eval(\varphi, \alpha)$. We have:

$$\begin{aligned}
eval(\varphi, \alpha_{\varphi}) &= eval(\varphi, \alpha_{\varphi'} \uplus \alpha_{\varphi''}) \\
&= eval(\varphi[\alpha_{\varphi'}], \alpha_{\varphi''}) \\
&= \max_{\alpha} eval(\varphi[\alpha_{\varphi'}], \alpha) \\
&\qquad \text{[since } \varphi'' = \varphi[\alpha_{\varphi'}] \text{ and } \alpha_{\varphi''} \in MaxSAT(\varphi'')\text{]} \\
&= \max_{\alpha} eval(\varphi'[\alpha_{\varphi'}] \cup (\varphi \setminus \varphi')[\alpha_{\varphi'}], \alpha) \\
&= \max_{\alpha} (eval(\varphi'[\alpha_{\varphi'}], \alpha) + eval((\varphi \setminus \varphi')[\alpha_{\varphi'}], \alpha)) \ \text{[from Prop. 19]} \\
&= \max_{\alpha} (eval(\varphi', \alpha_{\varphi'}) + eval((\varphi \setminus \varphi')[\alpha_{\varphi'}], \alpha)) \\
&\qquad \text{[since } \alpha_{\varphi'} \in MaxSAT(\varphi')\text{]} \\
&= eval(\varphi', \alpha_{\varphi'}) + \max_{\alpha} eval((\varphi \setminus \varphi')[\alpha_{\varphi'}], \alpha) \\
&\geq eval(\varphi', \alpha_{\varphi'}) + \max_{\alpha} eval(\varphi, \alpha) - \max_{\alpha} eval(\varphi' \cup \psi, \alpha) \\
&\qquad \text{[since } \psi = \text{APPROX}(\varphi, \varphi', \alpha_{\varphi'}), \text{see Defn. 18]}
\end{aligned}$$

$$= \max_{\alpha} eval(\varphi', \alpha) + \max_{\alpha} eval(\varphi, \alpha) - \max_{\alpha} eval(\varphi' \cup \psi, \alpha)$$

[since $\alpha_{\varphi'} \in MaxSAT(\varphi')$]

$$= \max_{\alpha} eval(\varphi' \cup \psi, \alpha) + \max_{\alpha} eval(\varphi, \alpha) - \max_{\alpha} eval(\varphi' \cup \psi, \alpha)$$

[from the condition defined in the theorem statement]

$$= \max_{\alpha} eval(\varphi, \alpha)$$ □

Discussion. There are number of possibilities for the APPROX function that satisfy the specification in Definition 18. The quality of such an APPROX function can be measured by two criteria: the efficiency and the precision of the optimality check (lines 3 and 6 in Algorithm 9).

Given that $eval$ can be efficiently computed, the cost of the optimality check primarily depends on the cost of computing ψ via APPROX, and the cost of invoking the MAXSAT solver on $\varphi' \cup \psi$. Since MAXSAT is known to be a computationally hard problem, a simple ψ returned by APPROX can significantly speedup the optimality check.

On the other hand, a precise optimality check can significantly reduce the number of iterations of Algorithm 8. We define a partial order $\preceq$ on APPROX functions that compares the precision of the optimality check via them. We say APPROX_1 is more precise than APPROX_2 (denoted by $\text{APPROX}_2 \preceq \text{APPROX}_1$), if for any given MAXSAT formulae φ, $\varphi' \subseteq \varphi$, and $\alpha_{\varphi'} \in MaxSAT(\varphi')$, the optimum objective value of $\varphi' \cup \text{APPROX}_1(\varphi, \varphi', \alpha_{\varphi'})$ is no larger than that of $\varphi' \cup \text{APPROX}_2(\varphi, \varphi', \alpha_{\varphi'})$. More formally:

$$\text{APPROX}_2 \preceq \text{APPROX}_1 \Leftrightarrow \forall \varphi, \varphi' \subseteq \varphi, \alpha_{\varphi'} \in MaxSAT(\varphi') :$$
$$\max_{\alpha} eval(\varphi' \cup \psi_1, \alpha) \leq \max_{\alpha} eval(\varphi' \cup \psi_2, \alpha),$$
$$\text{where } \psi_1 = \text{APPROX}_1(\varphi, \varphi', \alpha_{\varphi'}), \psi_2 = \text{APPROX}_2(\varphi, \varphi', \alpha_{\varphi'}).$$

In Section 4.2.4.2 we introduce three different APPROX functions with increasing order of precision. While the more precise APPROX operators reduce the number of iterations in

our algorithm, they are more expensive to compute.

Expanding relevant clauses via REFINE**.** The REFINE function finds a new set of clauses that are relevant to the queries, and adds them to the workset when the optimality check fails. To guarantee the termination of our approach, when the optimality check fails, REFINE should always return a nonempty set. We describe the details of various REFINE functions in Section 4.2.4.2.

4.2.4.2 Efficient Optimality Check via Distinct APPROX *Functions*

We introduce three different APPROX functions and their corresponding REFINE functions to construct efficient CHECK functions. These three functions are the ID-APPROX function, the π-APPROX function, and the γ-APPROX function. Each function is constructed by extending the previous one, and their precision order is:

$$\text{ID-APPROX} \preceq \pi\text{-APPROX} \preceq \gamma\text{-APPROX}.$$

The cost of executing each of these APPROX functions also increases with precision. After defining each function and proving that it satisfies the specification of an APPROX function, we discuss the efficiency and the precision of the optimality check using it.

I The ID-APPROX Function

The ID-APPROX function is based on the following observation: for a Q-MAXSAT instance, the clauses not explored by Algorithm 8 can only affect the assignments to the queries via the clauses sharing variables with the workset. We refer to such unexplored clauses as *frontier clauses*. If all the frontier clauses are satisfied by the current partial model, or they cannot further improve the objective value of the workset, we can construct a model of the Q-MAXSAT instance from the current partial model. Based on these observations, the ID-APPROX function constructs the summarization set by adding all the

frontier clauses that are not satisfied by the current partial model.

To define ID-APPROX, we first define what it means to say that a clause is satisfied by a partial model. A clause is satisfied by a partial model over a set of variables $\mathcal{U}$, if it is satisfied by all completions under that partial model. In other words:

$$\alpha_{\mathcal{U}} \models \bigvee_{i=1}^{n} v_i \vee \bigvee_{j=1}^{m} \neg v'_j \Leftrightarrow (\exists i : (v_i \in \mathcal{U}) \wedge \alpha_{\mathcal{U}}(v_i) = 1) \text{ or } (\exists j : (v'_j \in \mathcal{U}) \wedge \alpha_{\mathcal{U}}(v'_j) = 0).$$

Definition 22 (ID-APPROX)**.** Given formulae φ, $\varphi' \subseteq \varphi$, and a partial model $\alpha_{\varphi'} \in MaxSAT(\varphi')$, ID-APPROX is defined as follows:

$$\text{ID-APPROX}(\varphi, \varphi', \alpha_{\varphi'}) = \{(w, c) \mid (w, c) \in (\varphi \setminus \varphi') \wedge Var(\{(w, c)\}) \cap Var(\varphi') \neq \emptyset \wedge \alpha_{\varphi'} \not\models c)\}.$$

The corresponding REFINE function is:

$$\text{ID-REFINE}(\varphi_s, \varphi, \varphi', \alpha_{\varphi'}) = \varphi_s.$$

Example. Let $(\varphi, \mathcal{Q})$ be a Q-MAXSAT instance, where $\varphi = \{(2, x), (5, \neg x \vee y), (5, y \vee z), (1, \neg y)\}$ and $\mathcal{Q} = \{x, y\}$. Assume that the workset is $\varphi' = \{(2, x), (5, \neg x \vee y)\}$. By invoking a MAXSAT solver on φ', we get a model $\alpha_{\mathcal{Q}} = [x \mapsto 1, y \mapsto 1]$. Both clauses in $\varphi \setminus \varphi'$ contain y, where $\varphi \setminus \varphi' = \{(5, y \vee z), (1, \neg y)\}$. Since $(1, \neg y)$ is not satisfied by $\alpha_{\mathcal{Q}}$, ID-APPROX$(\varphi, \varphi', \alpha_{\varphi'})$ produces $\psi = \{(1, \neg y)\}$. As the optimum objective values of both φ' and $\varphi' \cup \psi$ are 7, we conclude $[x \mapsto 1, y \mapsto 1]$ is a model of the given Q-MAXSAT instance. Indeed, its completion $[x \mapsto 1, y \mapsto 1, z \mapsto 0]$ is a model of the MAXSAT formula φ. □

Theorem 23 (**Soundness of** ID-APPROX)**.** ID-APPROX$(\varphi, \varphi', \alpha_{\varphi'})$ *summarizes the effect of* $\varphi \setminus \varphi'$ *with respect to* $\alpha_{\varphi'}$.

Proof. Let ψ=ID-APPROX$(\varphi, \varphi', \alpha_{\varphi'})$. We show that ID-APPROX satisfies the specifica-

tion of an APPROX function in Definition 18 by proving the inequality below:

$$\max_{\alpha}\ eval(\varphi' \cup \psi, \alpha) + \max_{\alpha}\ eval(\varphi \setminus \varphi'[\alpha_{\varphi'}], \alpha) \geq \max_{\alpha}\ eval(\varphi, \alpha)$$

We use φ_1 to represent the set of frontier clauses, and φ_2 to represent the rest of the clauses in $\varphi \setminus \varphi'$:

$$\varphi_1 = \{(w, c) \in (\varphi \setminus \varphi') \mid Var(\{(w, c)\}) \cap Var(\varphi') \neq \emptyset\}$$
$$\varphi_2 = \{(w, c) \in (\varphi \setminus \varphi') \mid Var(\{(w, c)\}) \cap Var(\varphi') = \emptyset\}$$

We further split the set of frontier clauses φ_1 into two sets:

$$\varphi_1' = \{(w, c) \in \varphi_1 \mid \alpha_{\varphi'} \not\models c\}, \quad \varphi_1'' = \{(w, c) \in \varphi_1 \mid \alpha_{\varphi'} \models c\}$$

φ_1' is effectively what is returned by ID-APPROX$(\varphi, \varphi', \alpha_{\varphi'})$. We first prove the following claim:

$$\forall \alpha. eval(\varphi_1''[\alpha_{\varphi'}] \cup \varphi_2, \alpha) \geq eval(\varphi_1'' \cup \varphi_2, \alpha) \tag{4.1}$$

We have:

$eval(\varphi_1''[\alpha_{\varphi'}] \cup \varphi_2, \alpha)$

$$= eval(\varphi_1''[\alpha_{\varphi'}], \alpha) + eval(\varphi_2, \alpha) \text{ [from Prop. 19]}$$
$$= \max_{\alpha} eval(\varphi_1'', \alpha) + eval(\varphi_2, \alpha) \text{ [since } \forall (w, c) \in \varphi_1''. \alpha_{\varphi'} \models c]$$
$$\geq eval(\varphi_1'', \alpha) + eval(\varphi_2, \alpha)$$
$$= eval(\varphi_1'' \cup \varphi_2, \alpha) \qquad \text{[from Prop. 19]}$$

Now we prove the theorem. We have:

$$
\begin{aligned}
&\max_{\alpha}\ eval(\varphi' \cup \psi, \alpha) + \max_{\alpha}\ eval(\varphi \setminus \varphi'[\alpha_{\varphi'}], \alpha) \\
&\quad = \max_{\alpha} eval(\varphi' \cup \varphi'_1, \alpha) + \max_{\alpha} eval(\varphi \setminus \varphi'[\alpha_{\varphi'}], \alpha) \\
&\qquad\qquad [\text{since } \psi = \text{ID-APPROX}(\varphi, \varphi', \alpha_{\varphi'}), \text{see Defn. 22}] \\
&\quad = \max_{\alpha} eval(\varphi' \cup \varphi'_1, \alpha) + \\
&\qquad \max_{\alpha} eval(\varphi'_1[\alpha_{\varphi'}] \cup \varphi''_1[\alpha_{\varphi'}] \cup \varphi_2[\alpha_{\varphi'}], \alpha) \\
&\quad \geq \max_{\alpha} eval(\varphi' \cup \varphi'_1, \alpha) + \max_{\alpha} eval(\varphi''_1[\alpha_{\varphi'}] \cup \varphi_2[\alpha_{\varphi'}], \alpha) \\
&\qquad\qquad [\text{since } \forall \varphi, \varphi'. \max_{\alpha} eval(\varphi \cup \varphi') \geq \max_{\alpha} eval(\varphi)] \\
&\quad = \max_{\alpha} eval(\varphi' \cup \varphi'_1, \alpha) + \max_{\alpha} eval(\varphi''_1[\alpha_{\varphi'}] \cup \varphi_2, \alpha) \\
&\qquad\qquad [\text{since } Var(\varphi') \cap Var(\varphi_2) = \emptyset] \\
&\quad \geq \max_{\alpha} eval(\varphi' \cup \varphi'_1, \alpha) + \max_{\alpha} eval(\varphi''_1 \cup \varphi_2, \alpha) \ \ [\text{from (1)}] \\
&\quad \geq \max_{\alpha}(eval(\varphi' \cup \varphi'_1, \alpha) + eval(\varphi''_1 \cup \varphi_2, \alpha)) \quad [\text{from Prop. 20}] \\
&\quad = \max_{\alpha}\ eval(\varphi' \cup \varphi'_1 \cup \varphi''_1 \cup \varphi_2, \alpha) \qquad\qquad [\text{from Prop. 19}] \\
&\quad = \max_{\alpha}\ eval(\varphi, \alpha) \qquad\qquad \square
\end{aligned}
$$

Discussion. The ID-APPROX function effectively exploits the observation that, in a potentially very large Q-MAXSAT instance, the unexplored part of the formula can only impact the assignments to the queries via the frontier clauses. In practice, the set of frontier clauses is usually much smaller compared to the set of all unexplored clauses, resulting in an efficient invocation of the MAXSAT solver in the optimality check. Moreover, ID-APPROX is cheap to compute as it can be implemented via a linear scan of the unexplored clauses. However, the precision of ID-APPROX may not be satisfactory, causing the optimality check to be overly conservative. One such scenario is that, if the clauses in the summarization set generated from ID-APPROX contain any variable that is not used in any clause in

the workset, then the check will fail. To overcome this limitation, we next introduce the π-APPROX function.

II The π-APPROX Function

π-APPROX improves the precision of ID-APPROX by exploiting the following observation: if the frontier clauses violated by the current partial model have relatively low weights, even though they may contain new variables, it is very likely the case that we can resolve the queries with the workset. To overcome the limitation of ID-APPROX, π-APPROX generates the summarization set by applying a strengthening function on the frontier clauses violated by the current partial model.

We define the strengthening function *retain* below.

Definition 24 (*retain*)**.** We define $\mathit{retain}(c, \mathcal{V})$ as follows:

$$
\begin{aligned}
\mathit{retain}(1, \mathcal{V}) &= 1 \\
\mathit{retain}(0, \mathcal{V}) &= 0 \\
\mathit{retain}(\textstyle\bigvee_{i=1}^{n} v_i \vee \bigvee_{j=1}^{m} \neg v'_j, \mathcal{V}) &= \\
\text{let } \mathcal{V}_1 = \mathcal{V} \cap \{v_1, .., v_n\} &\text{ and } \mathcal{V}_2 = \mathcal{V} \cap \{v'_1, ..., v'_m\} \\
\text{in } \begin{pmatrix} 0 & \text{if } \mathcal{V}_1 = \mathcal{V}_2 = \emptyset \\ \bigvee_{u \in \mathcal{V}_1} u \vee \bigvee_{u' \in \mathcal{V}_2} \neg u' & \text{otherwise} \end{pmatrix}
\end{aligned}
$$

Definition 25 (π-APPROX)**.** Given a formula φ, a formula $\varphi' \subseteq \varphi$, and $\alpha_{\varphi'} \in \mathit{MaxSAT}(\varphi')$, π-APPROX is defined as follows:

$$
\begin{aligned}
&\pi\text{-APPROX}(\varphi, \varphi', \alpha_{\varphi'}) = \{\ (w, \mathit{retain}(c,\ \mathit{Var}(\varphi'))) \mid \\
&(w, c) \in \varphi \setminus \varphi' \wedge\ \ \mathit{Var}(\{(w, c)\}) \cap \mathit{Var}(\varphi') \neq \emptyset \wedge \alpha_{\varphi'} \not\models c)\ \}
\end{aligned}
$$

The corresponding REFINE function is:

$$\pi\text{-REFINE}(\varphi_s, \varphi, \varphi', \alpha_{\varphi'}) = \{(w, c) \in \varphi \setminus \varphi' \mid (w, \textit{retain}(c, \mathit{Var}(\varphi'))) \in \varphi_s\}$$

Example. Let $(\varphi, \mathcal{Q})$ where $\varphi = \{(5, x), (5, \neg x \vee y), (1, \neg y \vee z), (5, \neg z)\}$ and $Q = \{x, y\}$ be a Q-MAXSAT instance. Assume that the workset is $\varphi' = \{(5, x), (5, \neg x \vee y)\}$. By invoking a MAXSAT solver on φ', we get $\alpha_{\mathcal{Q}} = [x \mapsto 1, y \mapsto 1]$ with the objective value of φ' being 10. The only clause in $\varphi \setminus \varphi'$ that uses x or y is $(1, \neg y \vee z)$, which is not satisfied by $\alpha_{\mathcal{Q}}$. By applying π-APPROX, we generate the summarization set $\psi = \{(1, \neg y)\}$. By invoking a MAXSAT solver on $\varphi' \cup \psi$, we find its optimum objective value to be 10, which is the same as that of φ'. Thus, we conclude that $\alpha_{\mathcal{Q}}$ is a solution of Q-MAXSAT instance $(\varphi, \{x, y\})$. Indeed, model $[x \mapsto 1, y \mapsto 1, z \mapsto 0]$ which is a completion of α_Q, is a solution of the MAXSAT formula φ. On the other hand, the optimality check using the ID-APPROX function will return $\psi' = \{(1, \neg y \vee z)\}$ as the summarization set. By invoking the MAXSAT solver on $\phi' \cup \psi'$, we get an optimum objective value of 11 with $[x \mapsto 1, y \mapsto 1, z \mapsto 1]$. As a result, the optimality check with ID-APPROX fails here because of the presence of z in the summarization set. □

To prove that π-APPROX satisfies the specification of APPROX in Definition 18, we first introduce a decomposition function.

Definition 26 (π-DECOMP)**.** Given a formula φ and set of variables $\mathcal{V} \subseteq \mathit{Var}(\varphi)$, let $\mathcal{V}' = Var(\varphi) \setminus \mathcal{V}$. Then, we define π-DECOMP$(\varphi, \mathcal{V}) = (\varphi_1, \varphi_2)$ such that:

$$\begin{aligned}
\varphi_1 =& \{\ (w, \textit{retain}(c, \mathcal{V})) \mid (w, c) \in \varphi \wedge \mathit{Var}(\{(w, c)\}) \cap \mathcal{V} \neq \emptyset\ \} \\
\varphi_2 =& \{\ (w, \textit{retain}(c, \mathcal{V}')) \mid (w, c) \in \varphi \wedge \\
& \qquad (\mathit{Var}(\{(w, c)\}) \cap \mathcal{V}' \neq \emptyset \vee c = 1 \vee c = 0)\ \}.
\end{aligned}$$

Lemma 27. *Let* π-DECOMP$(\varphi, \mathcal{V}) = (\varphi_1, \varphi_2)$*, where* $\mathcal{V} \subseteq Var(\varphi)$*. We have* $eval(\varphi_1 \cup \varphi_2, \alpha) \geq eval(\varphi, \alpha)$ *for all* α*.*

Proof. To simplify the proof, we first remove the clauses with no variables from both φ and $\varphi_1 \cup \varphi_2$. We use the following two sets to represent such clauses:

$$\varphi_3 = \{(w, c) \in \varphi \mid c = 1\}, \quad \varphi_4 = \{(w, c) \in \varphi \mid c = 0\}.$$

We also define the following two sets:

$$\varphi_3' = \{(w, c) \in \varphi_2 \mid c = 1\}, \quad \varphi_4' = \{(w, c) \in \varphi_2 \mid c = 0\}.$$

Based on the definition of φ_2, we have $\varphi_3 = \varphi_3'$ and $\varphi_4 = \varphi_4'$. Therefore, the inequality in the lemma can be rewritten as:

$$eval(\varphi_1 \cup \varphi_2 \setminus (\varphi_3' \cup \varphi_4'), \alpha) \geq eval(\varphi \setminus (\varphi_3 \cup \varphi_4), \alpha).$$

From this point on, we assume $\varphi = \varphi \setminus (\varphi_3 \cup \varphi_4)$ and $\varphi_2 = \varphi_2 \setminus (\varphi_3' \cup \varphi_4')$. Let $\mathcal{V}' = Var(\varphi) \setminus \mathcal{V}$. We prove the lemma by showing that for any $s \in \varphi$, where $s = (w, c)$ and α,

$$\begin{aligned} &eval(\{(w, \textit{retain}(c, \mathcal{V}))\}, \alpha) + eval(\{(w, \textit{retain}(c, \mathcal{V}'))\}, \alpha) \\ &\quad \geq eval(\{(w, c\}, \alpha) \end{aligned} \tag{1}$$

Let $S = \{(w, c)\}$, $S_1 = \{(w, \textit{retain}(c, \mathcal{V}))\}$, $S_2 = \{(w, \textit{retain}(c, \mathcal{V}'))\}$. We prove the above claim with respect to the three different cases:

1. If $Var(S) \cap \mathcal{V} = \emptyset$, then we have $S_1 = \{(w, 0)\}$, $S_2 = S$. Since $\forall w, \alpha \,.\, eval(\{w, 0\}, \alpha) = 0$, we have $\forall \alpha \,.\, eval(S_1, \alpha) + eval(S_2, \alpha) = eval(S, \alpha)$.
2. If $Var(S) \cap \mathcal{V}' = \emptyset$, then we have $S_1 = S$, $S_2 = \{(w, 0)\}$. Similar to Case 1, we have $\forall \alpha \,.\, eval(S_1, \alpha) + eval(S_2, \alpha) = eval(S, \alpha)$.
3. If $Var(S) \cap \mathcal{V} \neq \emptyset$ and $Var(S) \cap \mathcal{V}' \neq \emptyset$, we prove the case by converting S, S_1 and

S_2 into their equivalent pseudo-Boolean functions. A pseudo-Boolean function f is a multi-linear polynomial, which maps the assignment of a set of Boolean variables to a real number:

$$f(v_1, ..., v_n) \ = \ a_0 + \sum_{i=1}^{m} (a_i \prod_{j=1}^{p} v_j), \quad \text{where } a_i \in \mathbb{R}^+.$$

Any MAXSAT formula can be converted into a pseudo-Boolean function. The conversion $\langle \cdot \rangle$ is as follows:

$$\begin{aligned} \langle \{s_1, ..., s_n\} \rangle &= \textstyle\sum_{i=1}^{n} (fst(s_i) - fst(s_i) \langle snd(s_i) \rangle) \\ \langle \textstyle\bigvee_{i=1}^{n} v_i \vee \bigvee_{j=1}^{m} \neg v'_j \rangle &= \textstyle\prod_{i=1}^{n} (1 - v_i) \ * \ \prod_{j=1}^{m} v'_j \\ \langle 1 \rangle &= 0 \\ \langle 0 \rangle &= 1 \end{aligned}$$

For example, the MAXSAT formula $\{(5, \neg w \vee x), (3, y \vee z)\}$ is converted into function $5 - 5w(1 - x) + 3 - 3(1 - y)(1 - z)$. We use $eval(f, \alpha)$ to denote the evaluation of pseudo-Boolean function f under model α. From the conversion, we can conclude that $\forall \varphi, \alpha \ . \ eval(\varphi, \alpha) = eval(\langle \varphi \rangle, \alpha)$.

We now prove the claim (1) under the third case above. We rewrite $c = \bigvee_{i=1}^{n} v_i \vee \bigvee_{j=1}^{m} \neg v'_j$ as below:

$$c = \textstyle\bigvee_{x \in \mathcal{V}_1} x \vee \bigvee_{x' \in \mathcal{V}_2} \neg x' \vee \bigvee_{y \in \mathcal{V}_3} y \vee \bigvee_{y' \in \mathcal{V}_4} \neg y'$$

where $\mathcal{V}_1 = \mathcal{V} \cap \{v_1, .., v_n\}$, $\mathcal{V}_2 = \mathcal{V} \cap \{v'_1, .., v'_m\}$, $\mathcal{V}_3 = \mathcal{V}' \cap \{v_1, .., v_n\}$ and $\mathcal{V}_4 = \mathcal{V}' \cap \{v'_1, .., v'_m\}$.

Let $c_1 = \bigvee_{x \in \mathcal{V}_1} x \vee \bigvee_{x' \in \mathcal{V}_2} \neg x'$ and $c_2 = \bigvee_{y \in \mathcal{V}_3} y \vee \bigvee_{y' \in \mathcal{V}_4} \neg y'$. Then, we have S_1=$\{(w, c_1)\}$ and S_2=$\{(w, c_2)\}$. It suffices to prove that $\langle S_1 \rangle + \langle S_2 \rangle - \langle S \rangle \geq 0$. We

have:

$$
\begin{aligned}
&\langle S_1\rangle + \langle S_2\rangle - \langle S\rangle \\
&= w - w \prod_{x\in\mathcal{V}_1}(1-x) \prod_{x'\in\mathcal{V}_2} x' + w - w \prod_{y\in\mathcal{V}_3}(1-y) \prod_{y'\in\mathcal{V}_4} y' \\
&\quad -(w - w \prod_{x\in\mathcal{V}_1}(1-x) \prod_{x'\in\mathcal{V}_2} x' \prod_{y\in\mathcal{V}_3}(1-y) \prod_{y'\in\mathcal{V}_4} y') \\
&= w(1 - \prod_{x\in\mathcal{V}_1}(1-x) \prod_{x'\in\mathcal{V}_2} x') - \\
&\quad w \prod_{y\in\mathcal{V}_3}(1-y) \prod_{y'\in\mathcal{V}_4} y'(1 - \prod_{x\in\mathcal{V}_1}(1-x) \prod_{x'\in\mathcal{V}_2} x') \\
&= w(1 - \prod_{x\in\mathcal{V}_1}(1-x) \prod_{x'\in\mathcal{V}_2} x')(1 - \prod_{y\in\mathcal{V}_3}(1-y) \prod_{y'\in\mathcal{V}_4} y')
\end{aligned}
$$

Since all variables are Boolean variables, we have

$$
\begin{aligned}
&1 - \textstyle\prod_{x\in\mathcal{V}_1}(1-x) * \prod_{x'\in\mathcal{V}_2} x' \geq 0 \text{ and} \\
&1 - \textstyle\prod_{y\in\mathcal{V}_3}(1-y) * \prod_{y'\in\mathcal{V}_4} y' \geq 0.
\end{aligned}
$$

□

From Lemma 27, Propositions 19 and 20, we can also conclude that

$$
\begin{aligned}
&\max_\alpha eval(\varphi_1, \alpha) + \max_\alpha eval(\varphi_2, \alpha) \\
&\geq \max_\alpha eval(\varphi_1 \cup \varphi_2, \alpha) \geq \max_\alpha eval(\varphi, \alpha)
\end{aligned}
$$

where π-DECOMP$(\varphi, \mathcal{V}) = (\varphi_1, \varphi_2)$.

We now use the lemma to prove that π-APPROX is a APPROX function as defined in Definition 18.

Theorem 28 (Soundness of π-APPROX). *π-APPROX$(\varphi, \varphi', \alpha_{\varphi'})$ summarizes the effect of $\varphi \setminus \varphi'$ with respect to $\alpha_{\varphi'}$.*

Proof. Let $\psi = \pi$-APPROX$(\varphi, \varphi', \alpha_{\varphi'})$. We show that π-APPROX satisfies the specification

of an APPROX function in Definition 18 by proving the inequality below:

$$\max_{\alpha} eval(\varphi' \cup \psi, \alpha) + \max_{\alpha} eval(\varphi \setminus \varphi'[\alpha_{\varphi'}], \alpha) \geq \max_{\alpha} eval(\varphi, \alpha)$$

Without loss of generality, we assume that φ does not contain any clause (w, c) where $c = 0$ or $c = 1$. Otherwise, we can rewrite the inequality above by removing these clauses from φ, φ', and $\varphi \setminus \varphi'$.

As in the soundness proof for ID-APPROX, we define φ_1,φ_2, φ'_1, and φ''_1 as follows:

$$\begin{aligned}
\varphi_1 &= \{(w, c) \in (\varphi \setminus \varphi') \mid Var(\{(w, c)\}) \cap Var(\varphi') \neq \emptyset\},\\
\varphi_2 &= \{(w, c) \in (\varphi \setminus \varphi') \mid Var(\{(w, c)\}) \cap Var(\varphi') = \emptyset\},\\
\varphi'_1 &= \{(w, c) \in \varphi_1 \mid \alpha_{\varphi'} \not\models c\},\\
\varphi''_1 &= \{(w, c) \in \varphi_1 \mid \alpha_{\varphi'} \models c\}.
\end{aligned}$$

We have: $\max_{\alpha} eval(\varphi' \cup \psi, \alpha) + \max_{\alpha} eval((\varphi \setminus \varphi')[\alpha_{\varphi'}], \alpha)$

$$\begin{aligned}
&= \max_{\alpha} eval(\varphi' \cup \psi, \alpha) +\\
&\quad \max_{\alpha} eval(\varphi'_1[\alpha_{\varphi'}] \cup \varphi''_1[\alpha_{\varphi'}] \cup \varphi_2[\alpha_{\varphi'}], \alpha)\\
&= \max_{\alpha} eval(\varphi' \cup \psi, \alpha) + \max_{\alpha}(eval(\varphi''_1[\alpha_{\varphi'}], \alpha) +\\
&\quad eval(\varphi'_1[\alpha_{\varphi'}] \cup \varphi_2[\alpha_{\varphi'}], \alpha)) \quad \text{[from Prop. 19]}\\
&= \max_{\alpha} eval(\varphi' \cup \psi, \alpha) + \max_{\alpha} eval(\varphi''_1, \alpha) +\\
&\quad \max_{\alpha} eval(\varphi'_1[\alpha_{\varphi'}] \cup \varphi_2[\alpha_{\varphi'}], \alpha) \ \text{[since } \forall (w, c) \in \varphi''_1.\alpha_{\varphi'} \models c]\\
&= \max_{\alpha} eval(\varphi' \cup \psi, \alpha) + \max_{\alpha} eval(\varphi''_1, \alpha) +\\
&\quad \max_{\alpha} eval(\varphi'_1[\alpha_{\varphi'}] \cup \varphi_2, \alpha) \ \text{[since } Var(\varphi') \cap Var(\varphi_2) = \emptyset]
\end{aligned}$$

Next, we show that π-DECOMP$(\varphi' \cup \varphi'_1 \cup \varphi_2, \mathit{Var}(\varphi')) = (\varphi' \cup \psi, \varphi'_1[\alpha_{\varphi'}] \cup \varphi_2)$. Let $\mathcal{V}_1 = \mathit{Var}(\varphi')$ and $\mathcal{V}_2 = \mathit{Var}(\varphi) \setminus \mathcal{V}_1$. For a given clause $(w, c) \in \varphi' \cup \varphi'_1 \cup \varphi_2$, we show the result of $(w, retain(c, \mathcal{V}_1))$ and $(w, retain(c, \mathcal{V}_2))$ under the following cases:

1. When $(w, c) \in \varphi'$, $\mathit{retain}(c, \mathcal{V}) = c$ and $\mathit{retain}(c, \mathcal{V}') = 0$. Hence we have $\varphi' = \{(w, \mathit{retain}(c, \mathcal{V})) \mid (w, c) \in \varphi' \wedge \mathit{retain}(c, \mathcal{V}) \neq 0\}$
2. When $(w, c) \in \varphi_2$, $\mathit{retain}(c, \mathcal{V}) = 0$ and $\mathit{retain}(c, \mathcal{V}') = c$. Hence we have $\varphi_2 = \{(w, \mathit{retain}(c, \mathcal{V}')) \mid (w, c) \in \varphi_2 \wedge \mathit{retain}(c, \mathcal{V}') \neq 0\}$
3. When $(w, c) \in \varphi'_1$, based on the definition of ψ and φ'_1, we have $\psi = \{(w, \mathit{retain}(c, \mathcal{V})) \mid (w, c) \in \varphi'_1 \wedge \mathit{retain}(c, \mathcal{V}) \neq 0\}$ and $\varphi'_1[\alpha_{\varphi'}] = \{(w, \mathit{retain}(c, \mathcal{V}')) \mid (w, c) \in \varphi'_1 \wedge \mathit{retain}(c, \mathcal{V}') \neq 0\}$.

   Therefore, π-DECOMP$(\varphi' \cup \varphi'_1 \cup \varphi_2, \mathit{Var}(\varphi')) = (\varphi' \cup \psi, \varphi'_1[\alpha_{\varphi'}] \cup \varphi_2)$.

   By applying Lemma 27, we can derive

$$\max_\alpha eval(\varphi' \cup \psi, \alpha) + \max_\alpha eval(\varphi_1[\alpha_{\varphi'}] \cup \varphi_2, \alpha) \geq \max_\alpha eval(\varphi' \cup \varphi'_1 \cup \varphi_2, \alpha). \tag{4.1}$$

Thus, we can prove the theorem as follows:

$\max_\alpha eval(\varphi' \cup \psi, \alpha) + \max_\alpha eval(\varphi \setminus \varphi'[\alpha_{\varphi'}], \alpha)$

$$\begin{aligned}
&= \max_\alpha eval(\varphi' \cup \psi, \alpha) + \max_\alpha eval(\varphi''_1, \alpha) + \\
&\quad \max_\alpha eval(\varphi'_1[\alpha_{\varphi'}] \cup \varphi_2, \alpha) \\
&\geq \max_\alpha eval(\varphi' \cup \varphi'_1 \cup \varphi_2, \alpha) + \max_\alpha eval(\varphi''_1, \alpha) \ \text{[from (1)]} \\
&\geq \max_\alpha [eval(\varphi' \cup \varphi'_1 \cup \varphi_2, \alpha) + eval(\varphi''_1, \alpha)] \ \text{[from Prop. 20]} \\
&\geq \max_\alpha eval(\varphi' \cup \varphi'_1 \cup \varphi''_1 \cup \varphi_2, \alpha) \ \text{[from Prop. 19]} \\
&= \max_\alpha eval(\varphi, \alpha)
\end{aligned}$$

□

Discussion. Similar to ID-APPROX, π-APPROX generates the summarization set from the frontier clauses that are not satisfied by the current solution. Consequently, the performance of the MAXSAT invocation in the optimality check is similar for both π-APPROX and ID-APPROX. π-APPROX further improves the precision of the check by applying the *retain* operator to strengthen the clauses generated. The *retain* operator does incur modest overheads when computing π-APPROX. In practice, however, we find that the additional precision provided by π-APPROX function improves the overall performance of the algorithm by terminating the iterative process earlier.

III The γ-APPROX Function

While both the ID-APPROX function and the π-APPROX function only consider information on the frontier clauses, γ-APPROX further improves precision by considering information from non-frontier clauses. Similar to the π-APPROX function, γ-APPROX also generates the summarization set by applying the *retain* function on the frontier clauses violated by the current partial model. The key improvement is that γ-APPROX tries to reduce the weights of generated clauses by exploiting information from the non-frontier clauses.

Before defining the γ-APPROX function, we first introduce some definitions:

$$PV(\bigvee_{i=1}^{n} v_i \vee \bigvee_{i=1}^{m} \neg v'_i)=\{v_1, ..., v_n\},\ \ PV(0)=PV(1)=\emptyset$$
$$NV(\bigvee_{i=1}^{n} v_i \vee \bigvee_{i=1}^{m} \neg v'_i)=\{v'_1, ..., v'_m\},\ NV(0)=NV(1)=\emptyset$$

$$link(v, u, \varphi) \Leftrightarrow \exists (w, c) \in \varphi : v \in NV(c) \wedge u \in PV(c)$$
$$tReachable(v, \varphi) = \{v\} \cup \{u \mid (v, u) \in R^+\}, \text{where } R = \{(v, u) \mid link(v, u, \varphi)\}$$
$$fReachable(v, \varphi) = \{v\} \cup \{u \mid v \in tReachable(u, \varphi)\}$$
$$tPenalty(v, \varphi) = \sum\{w \mid (w, \bigvee_{i=1}^{m} \neg v'_i) \in \varphi \ \wedge\ \{v'_1, ..., v'_m\} \cap tReachable(v, \varphi) \neq \emptyset\}$$
$$fPenalty(v, \varphi) = \sum\{w \mid (w, \bigvee_{i=1}^{n} v_i) \in \varphi \ \wedge\ \{v_1, ..., v_n\} \cap fReachable(v, \varphi) \neq \emptyset\}$$

Intuitively, $tPenalty(v, \varphi)$ overestimates the penalty incurred on the objective value of

φ by setting variable v to $true$, while $fPenalty(v, \varphi)$ overestimates the penalty incurred on the objective value of φ by setting variable v to $false$.

We next introduce the γ-APPROX approximation function.

Definition 29 (γ-APPROX)**.** Given φ, $\varphi' \subseteq \varphi$, and a partial model $\alpha_{\varphi'} \in MaxSAT(\varphi')$, γ-APPROX is defined as below:

$$\gamma\text{-APPROX}(\varphi, \varphi', \alpha_{\varphi'}) = \{(w', c') \mid (w, c) \in \varphi \setminus \varphi' \wedge \\ Var(\{(w, c)\}) \cap Var(\varphi') \neq \emptyset \wedge \alpha_{\varphi'} \not\models c \wedge c' = \\ \textit{retain}(c, Var(\varphi')) \wedge w' = reduce(w, c, \varphi, \varphi', \alpha_{\varphi'}).\}$$

We next define the $reduce$ function. Let $c'' = \textit{retain}(c, Var(\varphi) \setminus Var(\varphi'))$. Then, function $reduce$ is defined as follows:

1. If $c'' = 0$, $reduce(w, c, \varphi, \varphi', \alpha_{\varphi'}) = w$.

2. If $PV(c'') \neq \emptyset$ and $NV(c'') = \emptyset$, $reduce(w, c, \varphi, \varphi', \alpha_{\varphi'}) = min(w,\ min_{v \in PV(c'')} tPenalty(v, (\varphi \setminus \varphi')[\alpha_{\varphi'}]))$.

3. If $PV(c'') = \emptyset$ and $NV(c'') \neq \emptyset$, $reduce(w, c, \varphi, \varphi', \alpha_{\varphi'}) = min(w,\ min_{v \in NV(c'')} fPenalty(v, (\varphi \setminus \varphi')[\alpha_{\varphi'}]))$.

4. If $PV(c'') \neq \emptyset$ and $NV(c'') \neq \emptyset$, $reduce(w, c, \varphi, \varphi', \alpha_{\varphi'}) = min(min(w,\ min_{v \in PV(c'')} tPenalty(v, (\varphi \setminus \varphi')[\alpha_{\varphi'}])), min_{v \in NV(c'')} fPenalty(v, (\varphi \setminus \varphi')[\alpha_{\varphi'}]))$.

The corresponding REFINE function is:

$$\gamma\text{-REFINE}(\varphi_s, \varphi, \varphi', \alpha_{\varphi'}) = \{(w, c) \mid (w, c) \in \varphi \setminus \varphi' \wedge \\ (reduce(w, c, \varphi, \varphi', \alpha_{\varphi'}), \textit{retain}(c, Var(\varphi'))) \in \varphi_s\}$$

Example. Let $(\varphi, \mathcal{Q})$ where $\varphi = \{(5, x), (5, \neg x \vee y), (100, \neg y \vee z), (1, \neg z)\}$ and $\mathcal{Q} = \{x, y\}$ be a Q-MAXSAT instance. Suppose that the workset is $\varphi' = \{(5, x), (5, \neg x \vee y)\}$.

By invoking a MAXSAT solver on φ', we get $\alpha_{\mathcal{Q}} = [x \mapsto 1, y \mapsto 1]$ with the objective value of φ' being 10. The only clause in $\varphi \setminus \varphi'$ that uses x or y is $(100, \neg y \vee z)$, which is not satisfied by $\alpha_{\mathcal{Q}}$. By applying the *retain* operator, γ-APPROX strengthens $(100, \neg y \vee z)$ into $(100, \neg y)$. It further transforms it into $(1, \neg y)$ via $reduce$. By comparing the optimum objective value of φ' and $\varphi' \cup \{(1, \neg y)\}$, CHECK concludes that $\varphi_{\mathcal{Q}}$ is a solution of the Q-MAXSAT instance. Indeed, $[x \mapsto 1, y \mapsto 1, z \mapsto 1]$ which is a completion of $\alpha_{\varphi'}$ is a solution of the MAXSAT formula φ. On the other hand, applying π-APPROX will fail the check due to the high weight of the clause generated via *retain*.

We explain how $reduce$ works on this example. We first generate $c'' = retain(\neg y \vee z, \{x, y, z\} \setminus \{x, y\}) = z$. Then, we compute $(\varphi \setminus \varphi')[\alpha_{\mathcal{Q}}] = \{(100, \neg y \vee z), (1, \neg z)\}[[x \mapsto 1, y \mapsto 1]] = \{(100, z), (1, \neg z)\}$. As evident, setting z to 1 only violates $(1, \neg z)$, incurring a penalty of 1 on the objective value of $(\varphi \setminus \varphi')[\alpha_{\mathcal{Q}}]$. As a result, $tPenalty(z, (\varphi \setminus \varphi')[\alpha_{\mathcal{Q}}])$ = 1, which is lower than the weight of the summarization clause $(100, \neg y)$. Hence, $reduce$ returns 1 as the new weight for the summarization clause. □

To prove that γ-APPROX satisfies the specification of APPROX in Definition 18, we first prove two lemmas.

Lemma 30. *Given $(w, c) \in \varphi$, $v \in PV(c)$, and $tPenalty(v, \varphi) < w$, we construct φ' as follows:*

$$\varphi' = \varphi \setminus \{(w, c)\} \cup \{(w', c)\}, \textit{where } w' = tPenalty(v, \varphi).$$

Then we have

$$\max_{\alpha} eval(\varphi, \alpha) = \max_{\alpha'} eval(\varphi', \alpha') + w - w'.$$

Proof. We first prove a claim:

$$\exists \alpha : (eval(\varphi, \alpha) = \max_{\alpha'} eval(\varphi, \alpha')) \wedge \alpha \models c.$$

We prove this by contradiction. Suppose we can only find a model α that maximizes the

objective value of φ, such that $\alpha \not\models c$. We can construct a model α^1 in the following way:

$$\alpha^1(u) = \begin{cases} \alpha(u) & \text{if } u \notin tReachable(v, \varphi), \\ 1 & \text{otherwise.} \end{cases}$$

Based on the definition of $tPenalty$, we have

$$eval(\varphi, \alpha^1) \geq eval(\varphi, \alpha) - tPenalty(v, \varphi) + w.$$

Since $w \geq tPenalty(v, \varphi)$, α^1 yields no worse objective value than α. Since $\alpha^1 \models c$ as $v \in PV(c)$ and $\alpha^1(v) = 1$, we proved the claim.

Similarly, we can show

$$\exists \alpha' : (eval(\varphi', \alpha') = \max_{\alpha''} eval(\varphi', \alpha'')) \wedge \alpha' \models c.$$

Based on the two claims, we can find $\alpha \models c$ and $\alpha' \models c$ such that

$$\max_{\alpha^1} eval(\varphi, \alpha^1) = eval(\varphi, \alpha) = w + eval(\varphi \setminus \{(w, c)\}, \alpha),$$

$$\max_{\alpha^2} eval(\varphi', \alpha^2) = eval(\varphi', \alpha') = w' + eval(\varphi' \setminus \{(w', c)\}, \alpha').$$

We next show $eval(\varphi \setminus \{(w, c)\}, \alpha) = eval(\varphi' \setminus \{(w', c)\}, \alpha')$ by contradiction. Assuming $eval(\varphi \setminus \{(w, c)\}, \alpha) > eval(\varphi' \setminus \{(w', c)\}, \alpha')$, we will have $eval(\varphi', \alpha) > eval(\varphi', \alpha')$. This is because $eval(\varphi', \alpha) = eval(\varphi' \setminus \{(w', c)\}, \alpha) + w'$ and $eval(\varphi' \setminus \{(w', c)\}, \alpha) = eval(\varphi \setminus \{(w, c)\}, \alpha)$. This contradicts with the fact that $\max_{\alpha^2} eval(\varphi', \alpha^2) = eval(\varphi', \alpha')$. Thus, we conclude $eval(\varphi \setminus \{(w, c)\}, \alpha) \leq eval(\varphi' \setminus \{(w', c)\}, \alpha')$. Similarly,we can show $eval(\varphi \setminus \{(w, c)\}, \alpha) \geq eval(\varphi' \setminus \{(w', c)\}, \alpha')$.

Given $eval(\varphi \setminus \{(w,c)\}, \alpha) = eval(\varphi' \setminus \{(w',c)\}, \alpha')$, we have:

$$\begin{aligned}
&\max_{\alpha^1} eval(\varphi, \alpha^1) = eval(\varphi, \alpha) = w + eval(\varphi \setminus \{(w,c)\}, \alpha) \\
&= w + eval(\varphi' \setminus \{(w',c)\}, \alpha') = eval(\varphi', \alpha') + w - w' \\
&= \max_{\alpha^2} \ eval(\varphi', \alpha^2) + w - w'.
\end{aligned}$$

□

Lemma 31. *Given* $(w,c) \in \varphi$, $v \in NV(c)$ *and* $fPenalty(v,\varphi) < w$, *we construct* φ' *as follows:*

$$\varphi' = \varphi \setminus \{(w,c)\} \cup \{(w',c)\}, \textit{where } w' = fPenalty(v,\varphi).$$

Then we have

$$\max_{\alpha} eval(\varphi, \alpha) = \max_{\alpha'} \ eval(\varphi', \alpha') + w - w'.$$

Proof. Analogous to the proof to Lemma 30; we omit the details. □

We now prove that γ-APPROX satisfies the specification of APPROX in Definition 18.

Theorem 32 (γ-APPROX)**.** γ-APPROX$(\varphi, \varphi', \alpha_{\varphi'})$ *summarizes the effect of* $\varphi \setminus \varphi'$ *with respect to* $\alpha_{\varphi'}$.

Proof. Let $\psi = \gamma$-APPROX$(\varphi, \varphi', \alpha_{\varphi'})$. We show that γ-APPROX satisfies the specification of an APPROX function in Definition 18 by proving the inequality below:

$$\max_{\alpha} \ eval(\varphi' \cup \psi, \alpha) + \max_{\alpha} \ eval(\varphi \setminus \varphi'[\alpha_{\varphi'}], \alpha) \geq \max_{\alpha} \ eval(\varphi, \alpha)$$

We define φ_1 and φ_2 as follows:

$$\begin{aligned}
\varphi_1 &= \{(w,c) \in (\varphi \setminus \varphi') \mid Var(\{(w,c)\}) \cap Var(\varphi') \neq \emptyset\}, \\
\varphi_2 &= \{(w,c) \in (\varphi \setminus \varphi') \mid Var(\{(w,c)\}) \cap Var(\varphi') = \emptyset\}.
\end{aligned}$$

φ_1 is the set of frontier clauses and φ_2 contains rest of the clauses in $\varphi \setminus \varphi'$. We further split φ_1 into the following disjoint sets:

$$\begin{aligned}
\varphi_1^1 &= \{(w,c) \in \varphi_1 \mid \alpha_{\varphi'} \models c\},\\
\varphi_1^2 &= \{(w,c) \in \varphi_1 \mid \alpha_{\varphi'} \not\models c \ \wedge \mathit{reduce}(w,c,\varphi,\varphi',\alpha_{\varphi'}) = w\},\\
\varphi_1^3 &= \{(w,c) \in \varphi_1 \mid \alpha_{\varphi'} \not\models c \wedge w' < w \wedge\\
&\qquad\qquad \exists v \in PV(c) : w' = tPenalty(v, \varphi \setminus \varphi'[\alpha_{\varphi'}]) \wedge\\
&\qquad\qquad \mathit{reduce}(w,c,\varphi,\varphi',\alpha_{\varphi'}) = w'\},\\
\varphi_1^4 &= \varphi_1 \setminus (\ \varphi_1^1 \cup \varphi_1^2 \cup \varphi_1^3).
\end{aligned}$$

Effectively, φ_1^2, φ_1^3, and φ_1^4 contain all the clauses being strengthened in γ-APPROX: φ_1^2 contains all the clauses whose weights are not reduced; φ_1^3 contains all the clauses whose weights are reduced through positive literals in them; φ_1^4 contains all the clauses whose weights are reduced through negative literals in them.

Further, we define $\hat{\varphi_1^3}$ and $\hat{\varphi_1^4}$ as below:

$$\begin{aligned}
\hat{\varphi_1^3} &= \{(w',c) \mid (w,c) \in \varphi_1^3 \wedge \exists v \in PV(c) : w' =\\
&\quad tPenalty(v, \varphi \setminus \varphi'[\alpha_{\varphi'}]) \wedge \mathit{reduce}(w,c,\varphi,\varphi',\alpha_{\varphi'}) = w'\},\\
\hat{\varphi_1^4} &= \{(w',c) \mid (w,c) \in \varphi_1^4 \wedge \exists v \in NV(c) : w' =\\
&\quad fPenalty(v, \varphi \setminus \varphi'[\alpha_{\varphi'}]) \wedge \mathit{reduce}(w,c,\varphi,\varphi',\alpha_{\varphi'}) = w'\}.
\end{aligned}$$

Using Lemmas 30 and 31, we prove the following claim (1):

$$\begin{aligned}
&\max_{\alpha} eval(\varphi \setminus \varphi'[\alpha_{\varphi'}], \alpha)\\
&= \max_{\alpha} eval(\varphi_1^1[\alpha_{\varphi'}] \cup \varphi_1^2[\alpha_{\varphi'}] \cup \varphi_1^3[\alpha_{\varphi'}] \cup \varphi_1^4[\alpha_{\varphi'}] \cup \varphi_2[\alpha_{\varphi'}], \alpha)\\
&\geq \max_{\alpha} eval(\varphi_1^1[\alpha_{\varphi'}] \cup \varphi_1^2[\alpha_{\varphi'}] \cup \hat{\varphi_1^3}[\alpha_{\varphi'}] \cup \hat{\varphi_1^4}[\alpha_{\varphi'}] \cup \varphi_2[\alpha_{\varphi'}], \alpha)\\
&+ \sum\{w \mid (w,c) \in \varphi_1^3 \cup \varphi_1^4\} - \sum\{w' \mid (w',c) \in \hat{\varphi_1^3} \cup \hat{\varphi_1^4}\}
\end{aligned}$$

Similar to the soundness proof for π-APPROX, we show:

$$\pi\text{-DECOMP}(\varphi' \cup \varphi_1^2 \cup \hat{\varphi_1^3} \cup \hat{\varphi_1^4} \cup \varphi_2,\, Var(\varphi')) =$$
$$(\varphi' \cup \psi, \varphi_1^2[\alpha_{\varphi'}] \cup \hat{\varphi_1^3}[\alpha_{\varphi'}] \cup \hat{\varphi_1^4}[\alpha_{\varphi'}] \cup \varphi_2[\alpha_{\varphi'}]).$$

Following Lemma 27, we can derive the following claim (2):

$$\begin{aligned}
&\max_\alpha eval(\varphi' \cup \psi, \alpha) + \\
&\max_\alpha eval(\varphi_1^2[\alpha_{\varphi'}] \cup \hat{\varphi_1^3}[\alpha_{\varphi'}] \cup \hat{\varphi_1^4}[\alpha_{\varphi'}] \cup \varphi_2[\alpha_{\varphi'}], \alpha) \\
\geq\ &\max_\alpha eval(\varphi' \cup \varphi_1^2 \cup \hat{\varphi_1^3} \cup \hat{\varphi_1^4} \cup \varphi_2, \alpha).
\end{aligned}$$

By contradiction we can show the following claim (3) holds:

$$\begin{aligned}
&\forall \varphi, (w, c) \in \varphi, w' \leq w.\, \max_\alpha eval(\varphi, \alpha) \leq \\
&\max_\alpha eval(\varphi \setminus \{(w, c)\} \cup \{(w', c)\}, \alpha) + w - w'.
\end{aligned}$$

Combining Claim (1), (2), and (3), we have

$$\begin{aligned}
&\max_\alpha eval(\varphi' \cup \psi, \alpha) + \max_\alpha eval(\varphi \setminus \varphi'[\alpha_{\varphi'}], \alpha) \\
\geq\ &\max_\alpha eval(\varphi' \cup \psi, \alpha) + \max_\alpha eval(\varphi_1^1[\alpha_{\varphi'}] \cup \varphi_1^2[\alpha_{\varphi'}] \cup \hat{\varphi_1^3}[\alpha_{\varphi'}] \cup \varphi_1^4[\hat{\alpha}_{\varphi'}] \cup \varphi_2[\alpha_{\varphi'}], \alpha) \\
&+ \sum\{w \mid (w, c) \in \varphi_1^3 \cup \varphi_1^4\} - \sum\{w' \mid (w', c) \in \hat{\varphi_1^3} \cup \hat{\varphi_1^4}\} \\
\geq\ &\max_\alpha eval(\varphi' \cup \varphi_1^2 \cup \hat{\varphi_1^3} \cup \hat{\varphi_1^4} \cup \varphi_2, \alpha) + \max_\alpha eval(\varphi_1^1, \alpha) \\
&+ \sum\{w \mid (w, c) \in \varphi_1^3 \cup \varphi_1^4\} - \sum\{w' \mid (w', c) \in \hat{\varphi_1^3} \cup \hat{\varphi_1^4}\} \\
\geq\ &\max_\alpha eval(\varphi' \cup \varphi_1^1 \cup \varphi_1^2 \cup \hat{\varphi_1^3} \cup \hat{\varphi_1^4} \cup \varphi_2, \alpha) \\
&+ \sum\{w \mid (w, c) \in \varphi_1^3 \cup \varphi_1^4\} - \sum\{w' \mid (w', c) \in \hat{\varphi_1^3} \cup \hat{\varphi_1^4}\} \\
\geq\ &\max_\alpha eval(\varphi, \alpha).
\end{aligned}$$

□

Discussion. γ-APPROX is the most precise among the three functions as it is the only one that considers the effects of non-frontier clauses. This improved precision comes at the cost of computing additional information from the non-frontier clauses. Such information can be computed in polynomial time via graph reachability algorithms. In practice, we find this overhead to be negligible compared to the performance boost for the overall approach. Therefore, in the empirical evaluation, we use γ-APPROX and its related REFINE function in our implementation.

4.2.5 Empirical Evaluation

This section evaluates our approach on Q-MAXSAT instances generated from several real-world problems in program analysis and information retrieval.

4.2.5.1 Evaluation Setup

We implemented our algorithm for Q-MAXSAT in a tool PILOT. In all our experiments, we used the MiFuMaX solver [125] as the underlying MAXSAT solver, although PILOT allows using any off-the-shelf MAXSAT solver. We also study the effect of different such solvers on the overall performance of PILOT.

All our experiments were performed on a Linux machine with 8 GB RAM and a 3.0 GHz processor. We limited each invocation of the MAXSAT solver to 3 GB RAM and one hour of CPU time. Next, we describe the details of the Q-MAXSAT instances that we considered for our evaluation.

Instances from program analysis. These are instances generated from the aforementioned static bug detection application on two fundamental static analyses for sequential and concurrent programs: a pointer analysis and a datarace analysis. Both analyses are expressed in a framework that combines conventional rules specified by analysis writers with feedback on false alarms provided by analysis users [9]. The framework produces

Table 4.4: Characteristics of the benchmark programs . Columns "total" and "app" are with and without counting JDK library code, respectively.

benchmark	brief description	# classes		# methods		bytecode (KB)		source (KLOC)	
		app	total	app	total	app	total	app	total
`ftp`	Apache FTP server	93	414	471	2,206	29	118	13	130
`hedc`	web crawler from ETH	44	353	230	2,134	16	140	6	153
`weblech`	website download/mirror tool	11	576	78	3,326	6	208	12	194
`antlr`	generates parsers and lexical analyzers	111	350	1,150	2,370	128	186	29	131
`avrora`	AVR microcontroller simulator	1,158	1,544	4,234	6,247	222	325	64	193
`chart`	plots graphs and render them as PDF	192	750	1,516	4,661	102	306	54	268
`luindex`	document indexing tool	206	619	1,390	3,732	102	235	39	190
`lusearch`	text searching tool	219	640	1,399	3,923	94	250	40	198
`xalan`	XML to HTML transforming tool	423	897	3,247	6,044	188	352	129	285

MAXSAT instances whose hard clauses express soundness conditions of the analysis, while soft clauses specify false alarms identified by users. The goal is to automatically generalize user feedback to other false alarms produced by the analysis on the same input program.

The pointer analysis is a flow-insensitive, context-insensitive, and field-sensitive pointer analysis with allocation site heap abstraction [73]. Each query for this analysis is a Boolean variable that represents whether an unsafe downcast identified by the analysis prior to feedback is a true positive or not.

The datarace analysis is a combination of four analyses described in [43]: call-graph, may-alias, thread-escape, and lockset analyses. A query for this analysis is a Boolean variable that represents whether a datarace reported by the analysis prior to feedback is a true positive or not.

We generated Q-MAXSAT instances by running these two analyses on nine Java benchmark programs. Table 4.4 shows the characteristics of these programs. Except for the first three smaller programs in the table, all the other programs are from the DaCapo suite [77]. Table 4.5 shows the numbers of queries, variables, and clauses for the Q-MAXSAT instances corresponding to the above two analyses on these benchmark programs.

Instances from information retrieval. These are instances generated from problems in information retrieval. In particular, we consider problems in relational learning where the goal is to infer new relationships that are likely present in the data based on certain rules. Relational solvers such as Alchemy [81] and Tuffy [80] solve such problems by solving

Table 4.5: Number of queries, variables, and clauses in the MAXSAT instances generated by running the datarace analysis and the pointer analysis on each benchmark program. The datarace analysis has no queries on `antlr`and `chart` as they are sequential programs.

	datarace analysis			pointer analysis		
	# queries	# variables	# clauses	# queries	# variables	# clauses
`ftp`	338	1.2M	1.4M	55	2.3M	3M
`hedc`	203	0.8M	0.9M	36	3.8M	4.8M
`weblech`	7	0.5M	0.9M	25	5.8M	8.4M
`antlr`	0	-	-	113	8.7M	13M
`avrora`	803	0.7M	1.5M	151	11.7M	16.3M
`chart`	0	-	-	94	16M	22.3M
`luindex`	3,444	0.6M	1.1M	109	8.5M	11.9M
`lusearch`	206	0.5M	1M	248	7.8M	10.9M
`xalan`	11,410	2.6M	4.9M	754	12.4M	18.7M

a system of weighted constraints generated from the data and the rules. The weight of each clause represents the confidence in each inference rule. We consider Q-MAXSAT instances generated from three standard relational learning applications, described next, whose datasets are publicly available [126, 127, 95].

The first application is an advisor recommendation system (*AR*), which recommends advisors for first year graduate students. The dataset for this application is generated from the AI genealogy project [126] and DBLP [127]. The query specifies whether a professor is a suitable advisor for a student. The Q-MAXSAT instance generated consists of 10 queries, 0.3 million variables, and 7.9 million clauses.

The second application is Entity Resolution (*ER*), which identifies duplicate entities in a database. The dataset is generated from the Cora bibliographic dataset [95]. The queries in this application specify whether two entities in the dataset are the same. The Q-MAXSAT instance generated consists of 25 queries, 3 thousand variables, and 0.1 million clauses.

The third application is Information Extraction (*IE*), which extracts information from text or semi-structured sources. The dataset is also generated from the Cora dataset. The queries in this application specify extractions of the author, title, and venue of a publication record. The Q-MAXSAT instance generated consists of 6 queries, 47 thousand variables, and 0.9 million clauses.

Table 4.6: Performance of our PILOT and the baseline approach (**BASELINE**). In all experiments, we used a memory limit of 3 GB and a time limit of one hour for each invocation of the MAXSAT solver in both approaches. Experiments that timed out exceeded either of these limits.

application	benchmark	running time (in seconds)		peak memory (in MB)		problem size (in thousands)		final solver time (in seconds)	iterations
		PILOT	**BASELINE**	PILOT	**BASELINE**	final	max		
datarace analysis	`ftp`	53	5	387	589	892	1,400	3	7
	`hedc`	45	4	260	387	586	925	2	6
	`weblech`	2	1	199	340	561	937	1	1
	`avrora`	55	5	416	576	991	1,521	2	6
	`luindex`	72	4	278	441	657	1,120	2	6
	`lusearch`	45	3	223	388	575	994	2	8
	`xalan`	145	21	1,328	1,781	3,649	4,937	15	5
pointer analysis	`ftp`	16	11	16	1,262	29	2,982	0.1	9
	`hedc`	23	21	181	1,918	400	4,821	3	7
	`weblech`	4	***timeout***	363	***timeout***	922	8,353	4	1
	`antlr`	190	***timeout***	1,405	***timeout***	3,304	12,993	14	9
	`avrora`	178	***timeout***	1,095	***timeout***	2,649	16,344	13	8
	`chart`	253	***timeout***	721	***timeout***	1,770	22,325	8	6
	`luindex`	169	***timeout***	944	***timeout***	2,175	11,882	12	8
	`lusearch`	115	***timeout***	659	***timeout***	1,545	10,917	8	9
	`xalan`	646	***timeout***	1,312	***timeout***	3,350	18,713	19	8
AR	-	4	***timeout***	4	***timeout***	2	7,968	0.3	7
ER	-	13	2	6	44	9	104	0.2	19
IE	-	2	2,760	13	335	27	944	0.05	7

4.2.5.2 Evaluation Result

To evaluate the benefits of being query-guided, we measured the running time and memory consumption of PILOT. We used MiFuMaX as the baseline by running it on MAXSAT instances generated from our Q-MAXSAT instances by removing queries. To better understand the benefits of being query-guided, we also study the size of clauses posed to the underlying MAXSAT solver in the last iteration of PILOT, and the corresponding solver running time.

Further, to understand the cost of resolving each query, we pick one Q-MAXSAT instance from both domains and evaluate the performance of PILOT by resolving each query separately.

Finally, we study the sensitivity of PILOT's performance to the underlying MAXSAT solver by running PILOT using three different solvers besides MiFuMaX.

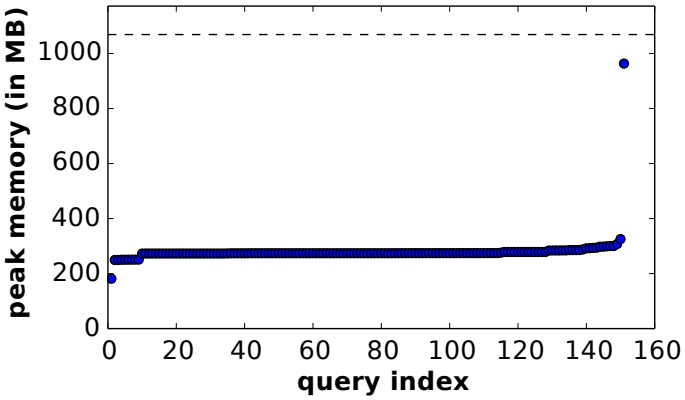
Performance of our approach *vs.* baseline approach. Table 4.6 summarizes our evaluation results on Q-MAXSAT instances generated from both domains.

Our approach successfully terminated on all instances and significantly outperformed the baseline approach in memory consumption on all instances, while the baseline only finished on twelve of the twenty instances in total. On the eight largest instances, the baseline approach either ran out of memory (exceeded 3 GB), or timed out (exceeded one hour).

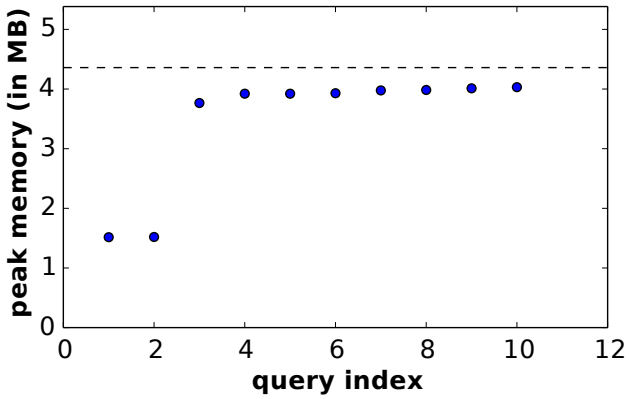
Column 'peak memory' shows the peak memory consumption of our approach and the baseline approach on all instances. For the instances on which both approaches terminated, our approach consumed 55.7% less memory compared to the baseline. Moreover, for large instances containing more than two million clauses, this number further improves to 71.6%.

We next compare the running time of both approaches under the 'running time' column. For most instances on which both approaches terminated, our approach does not outperform the baseline in running time. One exception is *IE* where our approach terminated in 2 seconds while the baseline approach spent 2,760 seconds, yielding a 1,380X speedup. Due to the iterative nature of PILOT, on simple instances where the baseline approach runs efficiently, the overall running time of PILOT can exceed that of the baseline approach. As column 'iteration' shows, PILOT takes multiple iterations on most instances. However, for challenging instances like *IE* and other instances where the baseline approach failed to finish, PILOT yields significant improvement in running time.

To better understand the gains of being query-guided, we study the number of clauses PILOT explored under the 'problem size' column. Column 'final' shows the number of clauses PILOT posed to the underlying solver in the last iteration, while column 'max' shows the total number of clauses in the Q-MAXSAT instances. On average, PILOT only explored 35.2% clauses in each instance. Moreover, for large instances containing over two million clauses, this number improves to 19.4%. Being query-guided allows our approach to lazily explore the problem and only solve the clauses that are relevant to the queries. As



(a) pointer analysis.



(b) *AR*.

Figure 4.6: The memory consumption of PILOT when it resolves each query separately on instances generated from (a) pointer analysis and (b) *AR*. The dotted line represents the memory consumption of PILOT when it resolves all queries together.

a result, our approach significantly outperforms the baseline approach.

The benefit of being query-guided is also reflected by the running time of the underlying MAXSAT solver in our approach. Column 'final solver time' shows the running time of the underlying solver in the last iteration of PILOT. Despite the fact that the baseline approach outperforms PILOT in overall running time for some instances, the running time of the underlying solver is consistently lower than that of the baseline approach. On average, this time is only 42.1% of the time for solving the whole instance.

Effect of resolving queries separately. We studied the cost of resolving each query by evaluating the memory consumption of PILOT when applying it to each query separately.

Table 4.7: Performance of our approach and the baseline approach with different underlying MAXSAT solvers.

instance	solver	running time **(in sec.)**		peak memory **(in MB)**	
		PILOT	**BASELINE**	PILOT	**BASELINE**
pointer analysis	CCLS2akms	***timeout***	***timeout***	***timeout***	***timeout***
	Eva500	2,267	***timeout***	1,379	***timeout***
	MaxHS	555	***timeout***	1,296	***timeout***
	WPM-2014.co	609	***timeout***	1,127	***timeout***
	MiFuMaX	178	***timeout***	1,095	***timeout***
AR	CCLS2akms	148	***timeout***	13	***timeout***
	Eva500	21	***timeout***	2	***timeout***
	MaxHS	4	***timeout***	9	***timeout***
	WPM-2014.co	6	***timeout***	2	***timeout***
	MiFuMaX	4	***timeout***	4	***timeout***

Figure 4.6 shows the result on one instance generated from the pointer analysis and the instance generated from *AR*. The program used in the pointer analysis is `avrora`. For comparison, the dotted line in the figure shows the memory consumed by PILOT when resolving all queries together. The other instances yield similar trends and we omit showing them for brevity.

For instances generated from the pointer analysis, when each query is resolved separately, except for one outlier, PILOT uses less than 30% of the memory it needs when all queries are resolved together. The result shows that each query is decided by different clauses in the program analysis instances. By resolving them separately, we further improve the performance of PILOT. This is in line with locality in program analysis, which is exploited by various query-driven approaches in program analysis.

For the instance generated from *AR*, we observed different results for each query. For eight queries, PILOT uses over 85% of the memory it needs when all queries are resolved together. For the other two queries, however, it uses less than 37% of that memory. After further inspection, we found that PILOT uses a similar set of clauses to produce the solution to the former eight queries. This indicates that for queries correlated with each other, batching them together in PILOT can improve the performance compared to the cumulative performance of resolving them separately.

Effect of different underlying solvers. To study the effect of the underlying MAXSAT solver, we evaluated the performance of PILOT using CCLS2akms, Eva500, MaxHS, and wpm-2014.co as the underlying solver. CCLS2akms and Eva500 were the winners in the MaxSAT'14 competition for random instances and industrial instances, respectively, while MaxHS ranked third for crafted instances (neither of the solvers performing better than MaxHS is publicly available). We used each solver itself as the baseline for comparison. In the evaluation, we used two instances from different domains, one generated from running the pointer analysis on `avrora`, and the other generated from *AR*.

Table 4.7 shows the running time and memory consumption of PILOT and the baseline approach under each setting. For convenience, we also include the result with MiFuMaX as the underlying solver in the table. As the table shows, except for the run with CCLS2akms on the pointer analysis instance, PILOT terminated under all the other settings, while none of the baseline approaches finished on any instance. This shows that our approach consistently gives improved performance regardless of the underlying MAXSAT solver it invokes.

4.2.6 Related Work

The MAXSAT problem is one of the optimization extensions of the SAT problem. The original form of the MAXSAT problem does not allow any hard clauses, and requires each soft clause to have a unit weight. A model of this problem is a complete assignment to the variables that maximizes the number of satisfied clauses. Two important variations of this problem are the *weighted* MAXSAT problem and the *partial* MAXSAT problem. The weighted MAXSAT problem allows each soft clause to have an arbitrary positive weight instead of limiting it to a unit weight; a model of this problem is a complete assignment that maximizes the sum of the weights of satisfied soft clauses. The partial MAXSAT problem allows hard clauses mixed with soft clauses; a model of this problem is a complete assignment that satisfies all hard clauses and maximizes the number of satisfied soft clauses. The combination of both these variations yields the *weighted partial* MAXSAT problem, which

is commonly referred to simply as the MAXSAT problem, and is the problem addressed in our work.

MAXSAT solvers are broadly classified into approximate and exact. Approximate solvers are efficient but do not guarantee optimality (i.e., may not maximize the objective value) [128] or even soundness (i.e., may falsify a hard clause) [129], although it is common to provide error bounds on optimality [130]. Our solver, on the other hand, is exact as it guarantees optimality and soundness.

There are a number of different approaches for exact MAXSAT solving, including branch-and-bound based, satisfiability-based, unsatisfiability-based, and their combinations [115, 116, 117, 118, 119, 120, 121, 122, 123]. The most successful of these on real-world instances, as witnessed in annual Max-SAT evaluations [131], perform iterative solving using a SAT solver as an oracle in each iteration [117, 118]. Such solvers differ primarily in how they estimate the optimal cost (e.g., linear or binary search), and the kind of information that they use to estimate the cost (e.g. cores, the structure of cores, or satisfying assignments). Many algorithms have been proposed that perform search on either upper bound or lower bound of the optimal cost [118, 117, 121, 116], Some algorithms efficiently perform a combined search over two bounds [123, 122]. A drawback of the most sophisticated combined search algorithms is that they modify the formula using expensive Pseudo Boolean (PB) constraints that increase the size of the formula and, potentially, slow down the solver. A recent promising approach [115] avoids this problem by using succinct formula transformations that do not use PB constraints and can be applied incrementally.

Our approach is similar in spirit to the above exact approaches in aspects such as iterative solving and optimal cost estimation. But we solve a new and fundamentally different optimization problem Q-MAXSAT, which enables our approach to be demand-driven, unlike existing approaches. In particular, it enables our approach to entirely avoid exploring vast parts of a given, very large MAXSAT formula that are irrelevant to deciding the values of a (small set of) queries in some model of the original MAXSAT formula. For this pur-

pose, our approach invokes an off-the-shelf exact MAXSAT solver on small sub-formulae of a much larger MAXSAT formula. Our approach is thus also capable of leveraging advances in solvers for the MAXSAT problem. Conversely, it would be interesting to explore how to integrate our query-guided approach into search algorithms of existing MAXSAT solvers.

The MAXSAT problem has also been addressed in the context of probabilistic logics for information retrieval [132], such as PSLs (Probabilistic Soft Logics) [133] and MLNs (Markov Logic Networks) [105]. These logics seek to reason efficiently with very large knowledge bases (KBs) of imperfect information. Examples of such KBs are the *AR*, *ER*, and *IE* applications in our empirical evaluation (Chapter 4.2.5). In particular, a fully grounded formula in these logics is a MaxSAT formula, and finding a model of this formula corresponds to solving the Maximum-a-Posteriori (MAP) inference problem in those logics [80, 81, 82, 83, 84]. A query-guided approach has been proposed for this problem [134] with the same motivation as ours, namely, to reason about very large MAXSAT formulae. However, this approach as well as all other (non-query-guided) approaches in the literature on these probabilistic logics sacrifice optimality as well as soundness.

In contrast, query-guided approaches have witnessed tremendous success in the domain of program reasoning. For instances, program slicing [135] is a popular technique to scale program analyses by pruning away the program statements which do not affect an assertion (query) of interest. Likewise, counter-example guided abstraction refinement (CEGAR) based model checkers [4, 1, 136] and refinement-based pointer analyses [137, 138, 75, 34] compute a cheap abstraction which is precise enough to answer a given query. However, the vast majority of these approaches target constraint *satisfaction* problems (i.e., problems with only hard constraints) as opposed to constraint *optimization* problems (i.e., problems with mixed hard and soft constraints). To our knowledge, our approach is the first to realize the benefits of query-guided reasoning for constraint optimization problems—problems that are becoming increasingly common in domains such as program reasoning [139, 140,

8, 141, 142].

4.2.7 Conclusion

We introduced a new optimization problem Q-MAXSAT that extends the well-known MAXSAT problem with queries. The Q-MAXSAT problem is motivated by the fact that many real-world MAXSAT problems pose scalability challenges to MAXSAT solvers, and the observation that many such problems are naturally equipped with queries. We proposed efficient exact algorithms for solving Q-MAXSAT instances. The algorithms lazily construct small MAXSAT sub-problems that are relevant to answering the given queries in a much larger MAXSAT problem. We implemented our Q-MAXSAT solver in a tool PILOT that uses off-the-shelf MAXSAT solvers to efficiently solve such sub-problems. We demonstrated the effectiveness of PILOT in practice on a diverse set of 19 real-world Q-MAXSAT instances ranging in size from 100 thousand to 22 million clauses. On these instances, PILOT used only 285 MB of memory on average and terminated in 107 seconds on average, whereas conventional MAXSAT solvers timed out for eight of the instances.

CHAPTER 5
FUTURE DIRECTIONS

So far, we have demonstrated three important applications that leverage combined logical and probabilistic reasoning and new algorithms to enable these applications. In this section, we identify a few fundamental challenges in the language, theory, and algorithms that are crucial for this new paradigm to be adopted more broadly.

Languages. Besides Markov Logic Networks, there are many competing languages that combine logic and probability. We chose Markov Logic Networks as their logical fragment is the closest to logical languages such as Datalog that are applied widely in conventional program analyses. However, it possess a few 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 a Markov Logic Network, given a soft constraint (c_h, w), all instances in the grounding of the logical formula c_h are assigned the same weight w. 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, Markov Logic Networks lack built-in support for the least fixed point operator, which is used prevalently in abstract-interpretation-based program analyses. In the static bug detection application which needs such support, we mitigate the issue by adding additional soft constraints to enforce the least fixed point semantics. However, a more fundamental and elegant solution may be conceivable by refining the language design.

Guarantees. The correctness of conventional logical analyses is enforced by rigorous safety guarantees, most notably soundness. We can continue to enforce such guarantees

in our combined analysis thanks to the logical part. This is an advantage of our approach compared to a pure probabilistic approach, as enforcing safety in the latter is much more challenging and has only been studied recently [143, 144, 145]. However, such logical guarantees alone are often not expressive enough to characterize the quality of results produced by the combined approach. One possible solution to address this challenge is to incorporate statistical guarantees. 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. To enforce such guarantees, we plan to investigate how to leverage the literature of probably approximately correct learning (PAC learning) [146].

Explainability. One benefit of the conventional logical approaches is that their results come with explanations such as provenance information. In our combined approach, while one can inspect the ground constraints related to result tuples to get intuitive explanations, there are no systematic approaches to extract explanatory information such as provenance from these constraints. To address this challenge, we plan to investigate provenance of weighted constraints.

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 and the probabilistic parts of the analysis specification could be learned 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 [147] and program synthesis [148].

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. More concretely, we plan to investigate the following techniques to further improve the effectiveness of the inference engine:

Magic Sets Transformation. Locality is almost universal to program analysis and is the key to the effectiveness of query-driven and demand-driven program analysis. NICHROME has currently only exploited locality in the solving phase. Ideally, we want to exploit it in the overall ground-solve framework. One promising idea is to leverage Magic Set transformation [149] from the Datalog evaluation literature. The idea is to apply the current framework but rewrite the constraint formulation so that both grounding and solving are driven by the demand of queries. In this way, we are able to only consider the constraints that are related to the queries while leveraging existing efficient algorithms that are agnostic to queries.

Lifted Inference. While our current ground-solve framework effectively leverages advances in MaxSAT solvers, it loses the high-level information while translating problems in our constraint language into low-level propositional formulae. Lifted inference [150, 151, 152, 153, 154] is a technique that aims to solve the constraint problem symbolically without grounding. While lifted inference can effectively avoid grounding large propositional formulae for certain problems, it fails to leverage existing efficient propositional solvers. One promising direction is to combine lifted inference with our ground-solve approach in a systematic way.

Compositional Solving. By exploiting modularity of programs, we envision compositional solving as an effective approach to improve the solver efficiency. The idea is to break a constraint problem into more tractable subproblems and solve them independently.

It is motivated by the success of compositional and summary-based analysis techniques in scaling to large programs.

Approximate Solving. Despite all the domain insights we exploit, MAP inference is a combinatorial optimization problem, which is known for its intractability. As a result, there will be pathological cases where none of the aforementioned techniques are effective. One idea to address this challenge is to investigate approximate solving, which trades precision for efficiency. Moreover, to trade precision for efficiency is a controlled manner, it is desirable to design an algorithm with tunable precision.

CHAPTER 6
CONCLUSION

This thesis presents a new paradigm to program analysis that combines logical and probabilistic reasoning. While preserving the benefits of conventional logic-based program analyses, the proposed paradigm provides analyses additional abilities to handle uncertainties, learn, and adapt. To support this paradigm, we described an end-to-end constraint-based system to automatically incorporate probabilities in an existing logical program analysis and demonstrated its effectiveness on three important program analysis applications.

The frontend of our system, PETABLOX (Chapter 3), takes a conventional analysis specified in Datalog as input and incorporate probabilities in it by converting it into a novel analysis specified in Markov Logic Networks, a language from the Artificial Intelligence community for unifying logic and probability. We showed that such treatment allows us to address fundamental challenges in prominent applications, including abstraction selection in automated program verification (Chapter 3.1), user effort reduction in interactive verification (Chapter 3.2), and alarm classification in static bug detection (Chapter 3.3).

To support the above applications, the backend of our system, NICHROME (Chapter 4) serves as a sound, optimal, and scalable inference and learning engine for Markov Logic Networks. To support effective inference and learning, it leverages domain insights to improve two key procedures: grounding and solving. By exploiting the observation that most solutions are sparse and the constraints came from logical analysis, it applies an iterative lazy-eager grounding algorithm (Chapter 4.1); by exploiting the observation that we are often only interested in a few tuples in the solver outputs that are related to analysis results, it applies a query-guided solving algorithm (Chapter 4.2). We demonstrated the effectiveness of NICHROME not only on constraint problems generated from aforementioned program analysis applications, but also problems generated from information retrieval applications.

We believe such a combined paradigm will help address long-standing problems in program analysis as well as enable new applications. This thesis aims to serve as a starting point of this exciting new body of research by showcasing important applications as well as describing a general receipt for enabling this paradigm. We envision wide adoption of the proposed paradigm and further research development in applications, languages, theory, and algorithms.

Appendices

APPENDIX A

PROOFS

A.1 Proofs for Results of Chapter 2

Lemma 33. *Let L be a complete lattice and $f, g \in L \to L$ two monotone functions. Then* lfp *preserves order, in the sense that*

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

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

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

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

$$\begin{aligned} & F_{C_1}(T) \subseteq F_{C_2}(T) \\ T_1 \subseteq T_2 \implies & F_{C_k}(T_1) \subseteq F_{C_k}(T_2) \qquad \text{for } k \in \{1, 2\} \end{aligned}$$

These properties can be readily verified from the definitions given in Figure 2.2, and from the assumption $C_1 \subseteq C_2$. By Lemma 33, $\mathsf{lfp} F_{C_1} \subseteq \mathsf{lfp} F_{C_2}$, which is the desired result. □

A.2 Proofs of Results of Chapter 3.1

A.2.1 Proofs of Theorems 4 and 5

Recall that for each Datalog constraint c, we have the function $[\![c]\!] : \mathcal{P}(\mathbb{T}) \to \mathcal{P}(\mathbb{T})$ in Figure 2.2, which determines the meaning of the constraint. We first prove a lemma that connects this function with a Markov Logic Network $\{c\}$. Note while the domain of constants of Markov Logic Network constraints varies, the domain of constants of Datalog constraints is always the set of all constants $\mathbb{N}$.

Lemma 34. *For every constraint c and all $T, T' \subseteq \mathbb{T}$ such that $[\![c]\!](T) \subseteq T$ and $T' \subseteq T$,*

$$[\![\{c\}]\!]_P(T') > 0 \quad \iff \quad [\![c]\!](T') \subseteq T',$$

where the domain of constants of Markov Logic Network $\{c\}$ is $\mathsf{constants}(T)$.

Proof. We prove the theorem by proving it separately under two cases: (1) when c has an empty boy, and (2) otherwise.

When c has an empty body, we have $\forall T'' \subseteq T. [\![c]\!](T'') = [\![c]\!](\emptyset)$. Its grounding is a set of tuples. Moreover, it is $[\![c]\!](\emptyset)$. To see why it is the case, first notice that $\mathsf{grounding}(c) = [\![c]\!](\emptyset)$ when the domain of constants of c in $\mathsf{grounding}$ is $\mathbb{N}$. Secondly, $[\![c]\!](\emptyset) = [\![c]\!](T) \subseteq T$, which in turn indicates that T contains all the constants in $[\![c]\!](\emptyset)$. Thus, we conclude $\mathsf{grounding}(c) = [\![c]\!](\emptyset)$, when the domain of constants of c in $\mathsf{grounding}$ is $\mathsf{constants}(\{c\} \cup T)$. Hence $[\![\{c\}]\!]_P(T') > 0$ is equivalent to $[\![c]\!](\emptyset) \subseteq T'$. Moreover, we can rewrite $[\![c]\!](T') \subseteq T'$ as $[\![c]\!](\emptyset) \subseteq T'$. Hence, the theorem holds when c has an empty body.

Next, we prove the theorem under the case where c has a nonempty body.

Consider c, T', T such that $[\![c]\!](T) \subseteq T$ and $T' \subseteq T$.

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

$$[\![\{c\}]\!]_P(T') > 0.$$

Assuming $[\![c]\!](T') \neq \emptyset$, pick $t \in [\![c]\!](T')$. Otherwise, the formula on the right-hand side trivially holds. By the definitions of $[\![c]\!]$ and $\mathsf{grounding}(c)$, there exist $t_1, \ldots, t_n$ such that

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

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

is a conjunct in $\mathsf{grounding}(c)$ where the domain of constants is $\mathsf{constants}(T' \cup [\![c]\!](T'))$.

Since $T' \subseteq T$, we have $[\![c]\!](T') \subseteq [\![c]\!](T)$. This, together with $[\![c]\!](T) \subseteq T$, we have $T' \cup [\![c]\!](T') \subseteq T$. Since $\mathsf{grounding}(c)$ monotonically grows as the domain of constants grows, the formula ϕ is also a conjunct in $\mathsf{grounding}(c)$ where the domain of constants is $\mathsf{constants}(T)$. By assumption, $[\![\{c\}]\!]_P(T') > 0$, so $T' \models \phi$. But all of $t_1, \ldots, t_n$ are in T'. Thus, this satisfaction relationship implies that $t \in T'$.

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

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

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

$$t \in [\![c]\!](\{t_1, \ldots, t_n\}) \subseteq [\![c]\!](T') \subseteq T',$$

where we use the monotonicity of $[\![c]\!]$ with respect to $\subseteq$. From $t \in T'$ that we have just proved follows the desired $T' \models \phi$. $\square$

Next, we prove a similar result for Datalog programs C which is broken into two lemmas. We introduce an auxiliary function $F_c : \mathcal{P}(\mathbb{T}) \rightarrow \mathcal{P}(\mathbb{T})$ such that $F_c(T) = \bigcup_{c \in C} [\![c]\!](T)$.

We prove that this function is closely connected with the Markov Logic Network C.

Lemma 35. *For all T, T' such that $F_C(T) = T$,*

$$[\![C]\!]_P(T') > 0 \quad \Longrightarrow \quad [\![C \cup (T' \cap T)]\!] \subseteq T',$$

where the domain of constants for Markov Logic Network C is $\mathsf{constants}(T)$.

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

Suppose that

$$[\![C]\!]_P(T') > 0,$$

where the domain of constants is $\mathsf{constants}(T)$. Let

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

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

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

We prove this sufficient condition by induction on n.

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

2. **Inductive case** $n > 0$**.** In this case,

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

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

as desired. Otherwise, there exist $c \in C$ and tuples

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

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

$$[\![c]\!](T'') \subseteq T''.$$

We show this subset relationship using Lemma 34. Specifically, we show the following three properties:

$$T'' \subseteq T, \quad [\![c]\!](T) \subseteq T, \quad \text{and} \quad [\![c]\!]_P(T'') > 0,$$

where the domain of constants is $\mathsf{constants}(T)$. The first property holds because $T'' = T' \cap T$, and the second is true because T is a fixed point of F_C and c is a constraint in C. We now show the third property. By assumption, $[\![C]\!]_P(T') > 0$, where the domain of constant is $\mathsf{constants}(T)$. Hence,

$$[\![c]\!]_P(T') > 0,$$

where the domain of constants is $\mathsf{constants}(T)$. We further have $[\![c]\!]_P(T') > 0$ when the domain of constants reduces to $\mathsf{constants}(T)$. Since $[\![c]\!](T) \subseteq T$, for every constraint instance $t_1 \wedge \ldots \wedge t_n \implies t$ in $\mathsf{grounding}(c)$, we have either $t \in T$ or $\exists t_i . t_i \notin T$. We can derive the same results for T' since $[\![c]\!]_P(T') > 0$. Based on these two results, we can also derive the same result for $T' \cap T$. The reasoning is as follows: for a given constraint instance $t_1 \wedge \ldots \wedge t_n \implies t$ in $\mathsf{grounding}(c)$, it is either the case that there exists t_i such that $t_i \notin T \vee t_i \notin T'$, or the case that $t \in T \wedge t \in T'$.

Such result implies

$$[\![c]\!]_P(T' \cap T) > 0.$$

That is, $[\![c]\!]_P(T'') > 0$, the very property that we want to prove.

□

Lemma 36. *For all T, T' such that $F_C(T) = T$, $T' \subseteq T$*

$$[\![C \cup (T' \cap T)]\!] \subseteq T' \quad \Longrightarrow \quad [\![C]\!]_P(T') > 0,$$

where the domain of constants for Markov Logic Network C is $\mathsf{constants}(T)$.

Proof. Suppose $T'' = T' \cap T$. Following the assumption, we have

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

We should show that $[\![C]\!]_P(T') > 0$. Suppose not. Then, there exist $c \in C$ and tuples $t_1, \ldots, t_n \in T$ and t such that

1. $t_1, \ldots, t_n \in T'$ but $t \notin T'$; and
2. when $t_1, \ldots, t_n, t$ are used as variables corresponding to these tuples,

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

is in $\mathsf{grounding}(c)$ where the domain of constants is $\mathsf{constants}(T)$.

Then, $t \in [\![c]\!](\{t_1, \ldots, t_n\})$. Since $T'' = T' \cap T$,

$$t_1, \ldots, t_n \in T''.$$

Hence,

$$t \in [\![\{c\} \cup \{t_1, \ldots, t_n\}]\!] \subseteq [\![C \cup T'']\!].$$

But $[\![C \cup T'']\!] \subseteq T'$. Thus, $t \in T'$. But this contradicts the fact that $t \notin T'$. □

Theorem 4. *Given a set of Datalog constraints C and a set of tuples A, let A' be a subset of A. Then the Datalog run result $[\![C \cup A']\!]$ is fully determined by the Markov Logic Network C where the domain of constants is* $\mathsf{constants}([\![C \cup A]\!])$*, as follows:*

$$t \in [\![C \cup A']\!] \quad \iff \quad \forall T.(T = \mathrm{MAP}(C \cup A')) \implies t \in T,$$

where $\mathsf{constants}([\![C \cup A]\!])$ *is the domain of constants of Markov Logic Network $C \cup A'$.*

Proof. Consider tuple sets T, A' such that $A' \subseteq A$ and $T = [\![C \cup A]\!]$. Throughout the proof, we assume the domains of constants for all Markov Logic Networks are $\mathbb{N}([\![C \cup A]\!])$.

First, we prove the only-if direction. Pick $t \in [\![C \cup A']\!]$. Consider T'' such that $[\![C \cup A']\!]_P(T'') > 0$. In other words, T'' is a solution to the MAP problem $C \cup A'$. This means that

$$A' \subseteq T'' \quad \text{and} \quad [\![C]\!]_P(T'') > 0. \tag{A.1}$$

Using these facts, we will prove that

$$t \in T''. \tag{A.2}$$

The key step of our proof is to show that

$$[\![C \cup (T'' \cap T)]\!] \subseteq T''. \tag{A.3}$$

This gives the set membership in (A.2), as shown in the following reasoning:

$$t \in [\![C \cup A']\!] \subseteq [\![C \cup (T'' \cap T)]\!] \subseteq T''.$$

The first subset relationship here holds because A' is a subset of both T (by the choice of A' and T) and T'' (by the first conjunct in (A.1)), and $[\![-]\!]$ is monotone. Now it remains

to discharge the subset relationship in (A.3). This is easy, because it is an immediate consequence of Lemma 35 and the fact that $[\![C]\!]_P(T'') > 0$, the second conjunct in (A.1).

Second, we show the if direction. Consider $t \in \mathbb{T}$ such that

$$\forall T'' \subseteq \mathbb{T}.(T'' = \mathrm{MAP}(C \cup A')) \implies t \in T''. \tag{A.4}$$

Let $T'' = [\![C \cup A']\!]$. Then,

$$T'' \models \bigwedge_{t' \in A'} t' \quad \text{and} \quad T'' \subseteq [\![C \cup A]\!] = T \quad \text{and} \quad [\![C \cup (T'' \cap T)]\!] \subseteq T''. \tag{A.5}$$

The first conjunct holds because $A' \subseteq T''$. The second conjunct holds because the monotonicity of Datalog. The third conjunct holds because

$$[\![C \cup (T'' \cap T)]\!] \subseteq [\![C \cup T'']\!] = T''.$$

Here the subset relationship follows from the monotonicity of $[\![-]\!]$ with respect to $\subseteq$, and the equality holds because T'' includes A', it is a fixed point of $F_{C \cup A'}$, and every fixed point T_0 of $F_{C \cup A'}$ equals $[\![C \cup A' \cup T_0]\!]$. By Lemma 36, the second conjunct in (A.5) implies

$$[\![C]\!]_P(T'') > 0.$$

This, together with $T'' \models \bigwedge_{t' \in A'} t'$, we have

$$[\![C \cup A']\!]_P(T'') > 0.$$

In other words, $T'' = \mathrm{MAP}(C \cup A')$. What we have proved so far and our choice of t in (A.4) imply that $T'' \models t$. That is, $t \in T''$. If we unroll the definition of T'', we get the desired set membership:

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

□

Theorem 5. *Let $A_1, \ldots, A_n$ be sets of tuples. For all $A' \subseteq \bigcup_{i\in[1,n]} A_i$,*

$$\forall T.(T = \mathrm{MAP}(C \cup A')) \implies t \in T \quad \implies \quad t \in [\![C \cup A']\!],$$

where $\bigcup_{i\in[1,n]} \mathsf{constants}([\![C \cup A_i]\!])$ is the domain of constants of Markov Logic Network $C \cup A'$.

Proof. Consider t and T' such that

$$t \in T', T' = \{t' \mid t' \in [\![C \cup A']\!] \wedge \mathsf{constants}(\{t'\}) \subseteq \bigcup_{i\in[1,n]} \mathsf{constants}([\![C \cup A_i]\!])\}. \quad \text{(A.6)}$$

It suffices to show

$$T' = \mathrm{MAP}(C \cup A').$$

Suppose it does not hold, then there exists $t_1 \wedge \ldots \wedge t_n \rightarrow t \quad \in \quad \mathsf{grounding}(C \cup A')$ where the domain of constants is $\bigcup_{i\in[1,n]} \mathsf{constants}([\![C \cup A_i]\!])\}$, such that

$$\{t_1, \ldots, t_n\} \subseteq T' \quad \text{and} \quad t \notin T'.$$

We can show that $t \in [\![C \cup A']\!]$ given $\{t_1, \ldots, t_n\} \subseteq T' \subseteq [\![C \cup A']\!]$. Based on the construction of T', we further conclude $t \in T'$, which contradicts the assumption.

□

A.2.2 Proof of Theorem 6

Theorem 6. *If $\mathsf{choose}(T, C, Q')$ evaluates to an element of the set $\mathbf{A} \setminus \gamma(T, C, Q')$ whenever such an element exists, and to* impossible *otherwise, then Algorithm 1 is partially correct: it returns (R, I) such that $R = \mathcal{R}(\mathbf{A}, Q)$ and $I = Q \setminus R$. In addition, if $\mathbf{A}$ is finite, then Algorithm 1 terminates.*

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

1. $R \subseteq \mathcal{R}(\mathbf{A}, Q)$;
2. $T = \emptyset$ or $T = T_1 \cup \ldots \cup T_n$ where $T_1, \ldots, T_n$ are some fixed points of F_C; and
3. $\mathcal{R}(\mathbf{A} \setminus \gamma(T, C, I), I) = \mathcal{R}(\mathbf{A}, I)$.

Let us start by proving that the invariant holds initially. When the loop of Algorithm 6 is entered for the first time,

$$R = \emptyset \wedge T = \emptyset \wedge I = Q.$$

Hence, the first and second conditions of our invariant hold in this case. For the third condition, we notice that

$$\begin{aligned}\gamma(T, C, I) = \gamma(\emptyset, C, I) &= \big\{\, A \in \mathbf{A} \;\big|\; \forall T'.(T' = \mathrm{MAP}(C \cup (A \cap \emptyset))) \implies I \subseteq T', \\ &\qquad\qquad \text{where the domain of constants is } \emptyset \,\big\} \\ &= \big\{\, A \in \mathbf{A} \;\big|\; \forall T'.T' \subseteq \mathbb{T} \implies I \subseteq T' \big\} \\ &= \big\{\, A \in \mathbf{A} \;\big|\; false \big\} = \emptyset.\end{aligned}$$

The second step holds as $\mathsf{grounding}(C \cup (A \cap \emptyset)) = \emptyset$ with $\emptyset$ as the domain of constants and any set of tuples satisfies an empty set of constraints. Hence, the third condition holds.

Next, we prove that our invariant is preserved by the loop of Algorithm 1. Assume that the invariant holds for R, T, and I. Also, assume that the result A of $\mathsf{choose}(T, C, I)$ is not $\mathsf{impossible}$. Let

$$R' = R \cup (Q \setminus [\![C \cup A]\!]), \quad T' = T \cup [\![C \cup A]\!], \quad I' = Q \setminus R'.$$

We should show that the three conditions of our invariant hold for R', T', and I'. The first

condition is

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

which holds because $R \subseteq \mathcal{R}(\mathbf{A}, Q)$ and $(Q \setminus [\![C \cup A]\!]) \subseteq \mathcal{R}(\mathbf{A}, Q)$. The second condition also holds because $[\![C \cup A]\!]$ is a fixed point of F_C. It remains to prove the third condition:

$$\mathcal{R}(\mathbf{A} \setminus \gamma(T', C, I'), I') = \mathcal{R}(\mathbf{A}, I').$$

For this, we will show the following sufficient condition:

$$\forall A' \in \gamma(T', C, I').\ I' \setminus [\![C \cup A']\!] = \emptyset.$$

Pick $A' \in \gamma(T', C, I')$. Then,

$$\forall T''.(T'' = \mathrm{MAP}(C \cup (A \cap T'))) \implies I' \subseteq T'',$$

where the domain of constants is $\mathsf{constants}(T')$. Since T' is the union of some fixed points of F_C, by Theorem 5, the above formula implies that

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

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

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

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

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

loop invariant,

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

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

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

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

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

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

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

$$R' = R \cup (Q \setminus [\![C \cup A]\!]), \quad T' = T \cup [\![C \cup A]\!], \quad I' = Q \setminus R'.$$

Since T is a subset of T', for any $A \in \mathbf{A}$, $T'' \subseteq \mathbb{T}$ we have

$$T'' = \mathrm{MAP}(C \cup (A \cap T')) \text{ where the domain of constants is } \mathsf{constants}(T') \implies$$
$$T'' = \mathrm{MAP}(C \cup (A \cap T)) \text{ where the domain of constants is } \mathsf{constants}(T).$$

Since $I \subset I'$, for any T'', we have $T'' \subseteq I$ implies $T'' \subseteq I''$. These two together imply

$$\begin{gathered}\{\, A \in \mathbf{A} \,|\forall T''.(T'' = \mathrm{MAP}(C \cup (A \cap T))) \implies I \subseteq T'', \\ \text{where the domain of constants is } T \,\} \subseteq \\ \{\, A \in \mathbf{A} \,|\forall T''.(T'' = \mathrm{MAP}(C \cup (A \cap T'))) \implies I \subseteq T'', \\ \text{where the domain of constants is } T' \,\}.\end{gathered}$$

Hence,

$$\gamma(C, T, I) \subseteq \gamma(C, T', I').$$

It remains to show that this subset relationship is strict. This is the case because $A \notin \gamma(T, C, I)$ by the assumption on choose, but it is in $\gamma(T', C, I')$. To see why $A \in \gamma(T', C, I')$, notice $I' \subseteq [\![C \cup A]\!]$ and $A \subseteq [\![C \cup A]\!]$. Hence, by Theorem 4,

$$\forall T''.(T'' = \mathrm{MAP}(C \cup A)) \implies I' \subseteq T'',$$

where the domain of constants is $\mathsf{constants}(C \cup A)$. This together with $A \subseteq T'$ and $[\![C \cup A]\!] \subseteq T'$ implies

$$\forall T''.(T'' = \mathrm{MAP}(C \cup (A \cap T'))) \implies I' \subseteq T'',$$

where the domain of constants is $\mathsf{constants}(T')$. This is equivalent to $A \in \gamma(\phi', I')$. □

A.3 Proofs of Results of Chapter 3.2

Lemma 10. *A sound analysis C can derive the ground truth; that is,* $\mathsf{True} = [\![C]\!]_{\mathsf{False}}$.

Proof. By (3.2), we have $[\![C]\!]_{\mathsf{False}} \supseteq \mathsf{True}$. By the augmented semantics (Figure 3.11(b)), we also know that $[\![C]\!]_{\mathsf{False}}$ and False are disjoint. The result follows. □

Lemma 11. *In Algorithm 2, suppose that* Heuristic *returns all tuples* $\mathbb{T}$*. Also, assume* (3.1)*,* (3.2)*, and* (3.3)*. Then, Algorithm 2 returns the true alarms* $\mathbf{A} \cap \mathsf{True}$*.*

Proof. The key invariant is that

$$\mathbf{F} \subseteq \mathsf{False} \subseteq \mathbf{Q} \tag{3.4}$$

The invariant is established by setting $\mathbf{F} := \emptyset$ and $\mathbf{Q} := \mathbf{T}$. By (3.1), we know that $Y \subseteq \mathsf{True}$ and $N \subseteq \mathsf{False}$, on line 5. It follows that the invariant is preserved by removing Y from $\mathbf{Q}$, on line 6. It also follows that $\mathbf{F} \cup N \subseteq \mathsf{False}$ and, by (3.2), that $[\![C]\!]_{\mathbf{F} \cup N} \supseteq \mathsf{True}$. So, the invariant is also maintained by line 7. We conclude that (3.4) is indeed an invariant.

For termination, let us start by showing that $|\mathbf{Q} \setminus \mathbf{F}|$ is nonincreasing. According to lines 6 and 7, the values of $\mathbf{Q}$ and $\mathbf{F}$ in the next iteration will be $\mathbf{Q}' := \mathbf{Q} \setminus Y$ and $\mathbf{F}' := (\mathbf{Q} \setminus Y) \setminus [\![C]\!]_{\mathbf{F} \cup N}$. We now show that $\mathbf{F} \subseteq \mathbf{F}'$ and $\mathbf{Q}' \subseteq \mathbf{Q}$. Consider an arbitrary $f \in \mathbf{F}$. By (3.4), $f \in \mathbf{Q}$. Using $Y \subseteq \mathsf{True}$ and (3.4), we conclude that Y and $\mathbf{F}$ are disjoint; using the augmented semantics in Figure 3.11(b), we conclude that $[\![C]\!]_{\mathbf{F} \cup N}$ and $\mathbf{F}$ are disjoint. Thus, $f \in \mathbf{F}'$, and, since f was arbitrary, we conclude $\mathbf{F} \subseteq \mathbf{F}'$. For $\mathbf{Q}' \setminus \mathbf{Q}$, it suffices to notice that $Y \subseteq \mathsf{True}$.

Now let us show that $|\mathbf{Q} \setminus \mathbf{F}|$ is not only nonincreasing but in fact decreasing. By (3.3), we know that $\mathbf{R}$ is non-empty, and thus at least one of Y or N is non-empty. If Y is non-empty, then $\mathbf{Q}' \subset \mathbf{Q}$. If N is non-empty, then $\mathbf{F} \subset \mathbf{F}'$. We can now conclude that 2 terminates.

When the main loop terminates, we have $\mathbf{F} = \mathbf{Q}$. Together with the invariant (3.4), we obtain that $\mathbf{F} = \mathsf{False}$. By Lemma 10, it follows that Algorithm 2 returns $\mathbf{A} \cap \mathsf{True}$. □

A.4 Proofs of Results of Chapter 4.1

Theorem 15 (Optimal initial grounding for Horn constraints). *If a Markov Logic Network comprises a set of hard constraints* C_h*, each of which is a Horn constraint* $\bigwedge_{i=1}^{n} l_i \implies$

l_0*, whose least solution is desired:*

$$T = \mathsf{lfp}\ \lambda T'.\ T' \cup \{\ [\![l_0]\!](\sigma) \mid (\textstyle\bigwedge_{i=1}^{n} l_i \implies l_0) \in \mathbf{C}_h$$
$$\wedge\ \forall i \in [1,n].[\![l_i]\!](\sigma) \in T' \wedge \sigma \in \Sigma\ \},$$

then for such a system, (a) $\texttt{Lazy}(C_h, \emptyset)$ *grounds at least* $|T|$ *constraints, and (b)* $\texttt{CEGAR}(C_h, \emptyset)$ *with the initial grounding* ϕ *does not ground any more constraints where*

$$\phi = \bigcup\{\ \bigvee_{i=1}^{n} \neg[\![l_i]\!](\sigma) \vee [\![l_0]\!](\sigma) \mid (\bigwedge_{i=1}^{n} l_i \implies l_0) \in C_h \wedge \forall i \in [0,n].[\![l_i]\!](\sigma) \in T \wedge \sigma \in \Sigma\ \}.$$

Proof. To prove (a), we will show that for each $t \in T$, $\texttt{Lazy}(H, \emptyset)$ must ground some constraint with t on the r.h.s. Let the sequence of sets of constraints grounded in the iterations of this procedure be $C_1, ..., C_n$. Then, we have:
Proposition (1): each ground constraint $\bigwedge_{i=1}^{m} t_i \implies t'$ in any C_j was added because the previous solution set all t_i to true and t' to false. This follows from the assumption that all rules in H are Horn rules. Let $x \in [1..n]$ be the earliest iteration in whose solution t was set to true. Then, we claim that t must be on the r.h.s. of some ground constraint ρ in C_x. Suppose for the sake of contradiction that no clause in C_x has t on the r.h.s. Then, it must be the case that there is some ground constraint ρ' in C_x where t is on the l.h.s. (the MAXSAT procedure will not set variables to true that do not even appear in any clause in C_x). Suppose ground constraint ρ' was added in some iteration $y < x$. Applying proposition (1) above to ρ' and $j = x$, it must be that t was true in the solution to iteration y, contradicting the assumption above that x was the earliest iteration in whose solution t was set to true.

To prove (b), suppose $\texttt{CEGAR}(H, \emptyset)$ grounds an additional constraint, that is, there exists a $(\bigwedge_{i=1}^{n} l_i \implies l_0) \in H$ and a σ such that $T \not\models \bigvee_{i=1}^{n} \neg[\![l_i]\!](\sigma) \vee [\![l_0]\!](\sigma)$. The only way by which this can hold is if $\forall i \in [1..n] : [\![l_i]\!](\sigma) \in T$ and $[\![l_0]\!](\sigma) \notin T$, but this contradicts the definition of T. □ □

Theorem 16 (**Soundness and Optimality of IPR**). *For any Markov Logic Network $C_h \cup C_s$ where hard constraints C_h is satisfiable,* $\texttt{CEGAR}(C)$ *produces a sound and optimal solution.*

Proof. We extend the function WEIGHT (Chapter 4.1.2) to hard clauses, yielding $-\infty$ if any such clause is violated:

$$\mathtt{W} = \lambda(T, \phi \cup \psi).\ \mathit{if}\ (\exists \rho \in \phi : T \not\models \rho)\ \mathit{then}\ -\infty \\ \mathit{else}\ \textsc{Weight}(T, \psi).$$

It suffices to show that the solution produced by $\texttt{CEGAR}(H, S)$ has the same weight as the solution produced by the eager approach. The eager approach (denoted by `Eager`) generates the solution by posing clauses generated from full grounding to a WPMS solver.

First, observe that $\texttt{CEGAR}(H, S)$ terminates: in each iteration of the loop in line 6 of Algorithm 7, it must be the case that at least one new ground hard constraints is added to ϕ or at least one new ground soft constraint is added to ψ, because otherwise the condition on line 12 will hold and the loop will be exited.

Now, suppose that in last iteration of the loop in line 6 for computing $\texttt{CEGAR}(H, S)$, we have:

(1) $gc_1 = hgc_1 \cup sgc_1$ is the set of ground hard and soft constraints accumulated in (ϕ, ψ) so far (line 9);

(2) $T_1 = \emptyset$ and $w_1 = 0$ (line 5), or $T_1 = \textsc{MaxSat}(hgc_1, sgc_1)$ and its weight is w_1 (lines 10 and 11);

(3) $gc_2 = hgc_2 \cup sgc_2$ is the set of all ground hard and soft constraints that are violated by T_1 (lines 7 and 8);

(4) $T_2 = \textsc{MaxSat}(hgc_1 \cup hgc_2, sgc_1 \cup sgc_2)$ and its weight is w_2 (lines 10 and 11);

and the condition on line 12 holds as this is the last iteration:

(5) $w_1 = w_2$ and $hgc_2 = \emptyset$.

Then, the result of $\texttt{CEGAR}(H, S)$ is T_1. On the other hand, the result of $\texttt{Eager}(H, S)$ is:

(6) $T_f = \textsc{MaxSat}(hgc_f, sgc_f)$ where:

(7) $gc_f = hgc_f \cup sgc_f$ is the set of fully grounded hard and soft constraints (Figure 2.4). Thus, it suffices to show that T_1 and T_f are equivalent.

Define $gc_m = gc_2 \setminus gc_1$.

(8) For any T, we have:

$$\begin{aligned}
\mathtt{W}(T, gc_1 \cup gc_2) &= \mathtt{W}(T, gc_1) + \mathtt{W}(T, gc_m) \\
&= \mathtt{W}(T, gc_1) + \mathtt{W}(T, hgc_2 \setminus hgc_1) + \\
&\quad \mathtt{W}(T, sgc_2 \setminus sgc_1) \\
&= \mathtt{W}(T, gc_1) + \mathtt{W}(T, sgc_2 \setminus sgc_1) \quad [a] \\
&\geq \mathtt{W}(T, gc_1) \quad [b]
\end{aligned}$$

where [a] follows from (5), and [b] from $\mathtt{W}(T, sgc_2) >= 0$ (i.e., soft constraints do not have negative weights). Instantiating (8) with T_1, we have: (9): $\mathtt{W}(T_1, gc_1 \cup gc_2) \geq \mathtt{W}(T_1, gc_1)$. Combining (2), (4), and (5), we have: (10): $\mathtt{W}(T_1, gc_1) = \mathtt{W}(T_2, gc_1 \cup gc_2)$. Combining (9) and (10), we have: (11)$\mathtt{W}(T_1, gc_1 \cup gc_2) \geq \mathtt{W}(T_2, gc_1 \cup gc_2)$. This means T_1 is a better solution than T_2 on $gc_1 \cup gc_2$. But from (4), we have that T_2 is an optimum solution to $gc_1 \cup gc_2$, so we have: (12): T_1 is also an optimum solution to $gc_1 \cup gc_2$.

It remains to show that T_1 is also an optimum solution to the set of fully grounded hard and soft constraints gc_f, from which it will follow that T_1 and T_f are equivalent. Define $gc_r = gc_f \setminus (gc_1 \cup gc_2)$. For any T, we have:

$$\begin{aligned}
\mathtt{W}(T, gc_f) &= \mathtt{W}(T, gc_1 \cup gc_2 \cup gc_r) \\
&= \mathtt{W}(T, gc_1 \cup gc_2) + \mathtt{W}(T, gc_r) \\
&\leq \mathtt{W}(T_1, gc_1 \cup gc_2) + \mathtt{W}(T, gc_r) \quad [c] \\
&\leq \mathtt{W}(T_1, gc_1 \cup gc_2) + \mathtt{W}(T_1, gc_r) \quad [d] \\
&= \mathtt{W}(T_1, gc_1 \cup gc_2 \cup gc_r) \\
&= \mathtt{W}(T_1, gc_f)
\end{aligned}$$

i.e. $\forall T, \mathtt{W}(T, gc_f) \leq \mathtt{W}(T_1, gc_f)$, proving that T_1 is an optimum solution to gc_f. Inequality

[c] follows from (11), that is, T_1 is an optimum solution to $gc_1 \cup gc_2$. Inequality [d] holds because from (3), all clauses that T_1 possibly violates are in gc_2, whence T_1 satisfies all constraints in gc_r, whence $\mathtt{W}(T, gc_r) \leq \mathtt{W}(T_1, gc_r)$.

□

APPENDIX B

ALTERNATE USE CASE OF URSA: COMBINING TWO STATIC ANALYSES

Another use case of our approach to interactive verification is to combine a fast yet imprecise analysis and a slow yet precise analysis, in order to balance the overall precision and scalability. We can enable this use case by instantiating Decide with a precise yet expensive analysis instead of a human user. Such iterative combination of the two analyses are beneficial as it may take a significant amount of time to apply the precise analysis to resolve all potential causes in the imprecise analysis. On the other hand, URSA allows the precise analysis to resolve only the causes that are relevant to the alarms, and focus on causes with high payoffs. We next demonstrate this use case by combining two a pointer analyses as an example. We first describe the base analysis, its notions of alarms and causes, and our implementation of the procedure Heuristic. Then we describe the oracle analysis. Finally, we empirically evaluate our approach.

Base Analysis. Our base pointer analysis is a flow/context-insensitive, field-sensitive, Anderson-style analysis with on-the-fly call-graph construction [73]. It uses allocation sites as the heap abstraction. It comprises 46 rules, 29 input relations, and 18 output relations.

We treat points-to facts (denoted by relation `pointsTo`) and call-graph edges (denoted by `callEdge`) as the alarms as they are directly consumed by client analyses built atop the pointer analysis. Given the large number of tuples in these two relations, we further limit the alarms to tuples in the application code. We treat the same set of tuples as the universe of potential causes since the tuples in these two relations are used to derive each other in a recursive manner.

We are not aware of effective static heuristics for pointer analysis alarms that are client-agnostic. Therefore, we only provide a Heuristic instantiation that leverages a dynamic

Table B.1: Numbers of alarms (denoted by $|\mathbf{A}|$) and tuples in the universe of potential causes (denoted by $|\mathbf{Q}_U|$) of the pointer analysis, where k stands for thousands.

	$\|\mathbf{A}\|$			$\|\mathbf{Q}_U\|$
	false	total	false%	
raytracer	56	950	5.9%	950
montecarlo	5	867	0.6%	867
sor	0	159	0	159
elevator	3	369	0.8%	369
jspider	925	5.1k	18.1%	5.1k
hedc	1.2k	3.9k	29.8%	3.9k
ftp	5.7k	12.5k	45.5%	12.5k
weblech	440	3.9k	11.2%	3.9k

analysis:

$$\texttt{dynamic}() = \{\texttt{pointsTo}(v, o) \mid \text{ variable } v \text{ is accessed in the runs and never points to object } o\} \cup$$
$$\{\texttt{callEdge}(p, m) \mid \text{invocation site } p \text{ is reached and never invokes method } m \text{ in the runs } \}$$

Oracle Analysis. The oracle analysis is a query-driven k-object-sensitive pointer analysis [8]. This analysis improves upon the precision of the pointer analysis by being simultaneously context- and object-sensitive, but it achieves this higher precision by targeting queries of interest, which in our setting are individual alarms and potential causes.

Empirical Evaluation. We use a setting that is similar to the one described in Chapter 3.2.6.1. In particular, to evaluate the effectiveness of our approach in reducing false alarm rates, we obtained answers to all the alarms and potential causes offline using the oracle analysis. Table B.1 shows the statistics of alarms and tuples in the universal of potential causes. We next describe the generalization results and prioritization results of our approach.

Figure B.1 shows the generalization results of URSA with `dynamic`, the only available Heuristic instantiation for the pointer analysis. To evaluate the effectiveness of `dynamic`, we also show the results of the ideal case where the oracle answers are used to implement

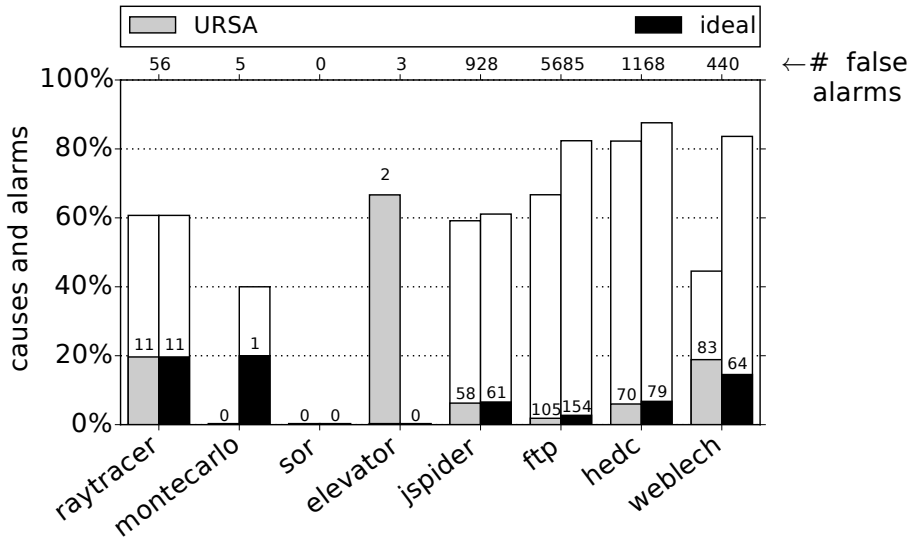


Figure B.1: Number of questions asked over total number of false alarms (denoted by the lower dark part of each bar) and percentage of false alarms resolved (denoted by the upper light part of each bar) by URSA for the pointer analysis.

Heuristic.

URSA is able to eliminate 44.8% of the false alarms with an average payoff of $8\times$ per question. Excluding the three small benchmarks with under five false alarms each, the gain rises to 63.2% and the average payoff increases to $11\times$. These results show that, most of the false alarms can indeed be eliminated by inspecting only a few common root causes. And by applying the expensive oracle analysis only to these few root causes, URSA can effectively improve the precision of the base analysis without significantly increasing the overall runtime.

In the ideal case, URSA eliminates an additional 15.5% of false alarms. While the improvement is modest for most benchmarks, an additional 39% false alarms are eliminated on `weblech` in the ideal case. The reason for this anomaly is that the input set used in `dynamic` does not yield sufficient code coverage to produce accurate predictions for the desired root causes. We thus conclude that overall, the `dynamic` instantiation is effective in identifying common root causes of false alarms.

Figure B.2 shows the prioritization results of URSA. Every point in the plots represents the number of false alarms eliminated (y-axis) and the number of questions (x-axis) asked

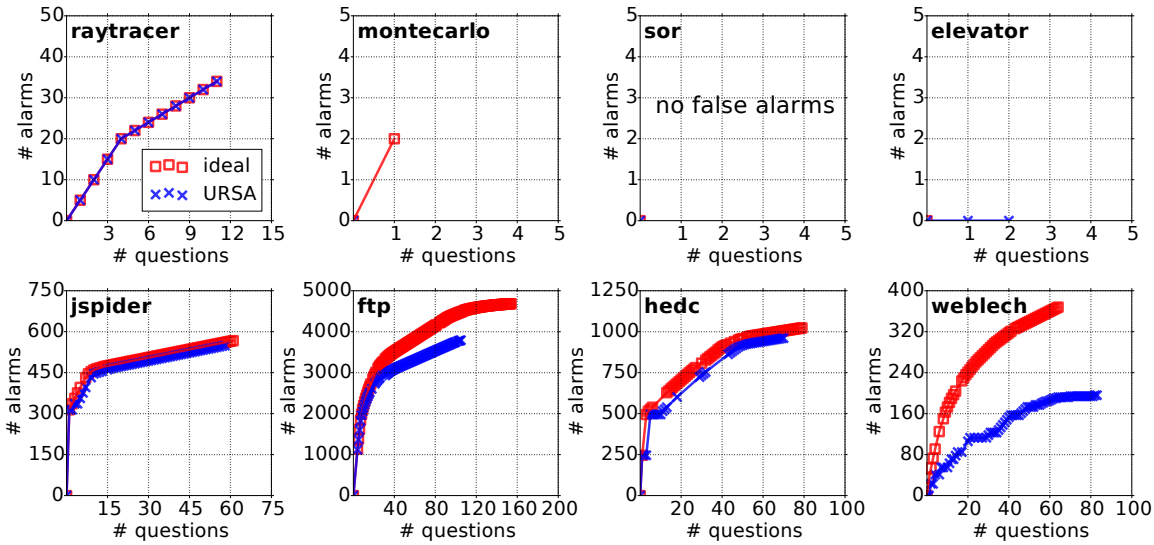


Figure B.2: Number of questions asked and number of false alarms resolved by URSA in each iteration for the pointer analysis (k = thousands).

up to the current iteration. As before, we compare the results of URSA to the ideal case.

We observe that a few causes often yield most of the benefits. For instance, four causes can resolve 1,133 out of 5,685 false alarms on `ftp`, yielding a payoff of 283×. URSA successfully identifies these causes with high payoffs and poses them to the oracle analysis in the earliest few iterations. In fact, for the first three iterations on most benchmarks, URSA asks exactly the same set of questions as the ideal setting. The results of these two settings only differ in later iterations, where the payoff becomes relatively low. We also notice that there can be multiple causes in the set that gives the highest benefit (for instance, the aforementioned `ftp` results). The reason is that there can be multiple derivations to each alarm. If we naively search the potential causes by fixing the number of questions in advance, we can miss such causes. URSA, on the other hand, successfully finds them by solving the optimal root set problem iteratively.

The fact that URSA is able to prioritize causes with high payoffs allows us to stop the interaction of the two analyses after a few iterations and still get most of the benefits. URSA terminates either when the expected payoff drops to one or when there are no more questions to ask. But in practice, we might choose to stop the interaction even earlier. For

instance, we might be satisfied with the current result, or we may find limited payoffs in running the oracle analysis, which can take a long time comparing to inspecting the rest alarms manually.

We study these causes with high payoffs more closely. the main causes are the points-to facts and call-graph edges that lead to one or multiple spurious call-graph edges, which in turn lead to many false tuples. Such spurious tuples are produced due to context insensitivity of the analysis.

REFERENCES

[1] T. Ball, V. Levin, and S. K. Rajamani, "A decade of software model checking with SLAM," *Commun. ACM*, vol. 54, no. 7, pp. 68–76, 2011.

[2] P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival, "The ASTREÉ analyzer," in *Programming Languages and Systems, 14th European Symposium on Programming,ESOP 2005, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2005, Edinburgh, UK, April 4-8, 2005, Proceedings*, 2005, pp. 21–30.

[3] *Coverity*, `http://www.coverity.com`, Accessed: 2017-06-02.

[4] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith, "Counterexample-guided abstraction refinement for symbolic model checking," *J. ACM*, vol. 50, no. 5, pp. 752–794, 2003.

[5] A. Aiken, "Introduction to set constraint-based program analysis," *Sci. Comput. Program.*, vol. 35, no. 2, pp. 79–111, 1999.

[6] S. Abiteboul, R. Hull, and V. Vianu, *Datalog and Recursion*. Addison-Wesley, 1995, ch. 12, pp. 271–310.

[7] Y. Smaragdakis, G. Kastrinis, and G. Balatsouras, "Introspective analysis: Context-sensitivity, across the board," in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2014, Edinburgh, United Kingdom, June 09-11, 2014*, 2014, pp. 485–495.

[8] X. Zhang, R. Mangal, R. Grigore, M. Naik, and H. Yang, "On abstraction refinement for program analyses in datalog," in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2014, Edinburgh, United Kingdom, June 09-11, 2014*, 2014, pp. 239–248.

[9] R. Mangal, X. Zhang, A. V. Nori, and M. Naik, "A user-guided approach to program analysis," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, Bergamo, Italy, August 30-September 4, 2015*, 2015, pp. 462–473.

[10] M. Madsen, M. Yee, and O. Lhoták, "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, Santa Barbara, CA, USA, June 13-17, 2016*, 2016, pp. 194–208.

[11] H. Jordan, B. Scholz, and P. Subotic, "Soufflé: On synthesis of program analyzers," in *Computer Aided Verification - 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part II*, 2016, pp. 422–430.

[12] M. Richardson and P. M. Domingos, "Markov logic networks," *Machine Learning*, vol. 62, no. 1-2, pp. 107–136, 2006.

[13] H. G. Rice, "Classes of recursively enumerable sets and their decision problems," *Transactions of the American Mathematical Society*, vol. 74, no. 2, pp. 358–366, 1953.

[14] X. Zhang, R. Grigore, X. Si, and M. Naik, "Effective interactive resolution of static analysis alarms," in *Proceedings of the 32th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2017, October 25-27, 2017, Vancouver, British Columbia, Canada*, 2017.

[15] J. Whaley and M. S. Lam, "Cloning-based context-sensitive pointer alias analysis using binary decision diagrams," in *Proceedings of the 25th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2004, Washington, DC, USA, June 9-11, 2004*, 2004, pp. 131–144.

[16] M. Bravenboer and Y. Smaragdakis, "Strictly declarative specification of sophisticated points-to analyses," in *Proceedings of the 24th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2009, October 25-29, 2009, Orlando, Florida, USA*, 2009, pp. 243–262.

[17] G. Kastrinis and Y. Smaragdakis, "Hybrid context-sensitivity for points-to analysis," in *Proceedings of the 34th Annual ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2013, Seattle, WA, USA, June 16-19, 2013*, 2013, pp. 423–434.

[18] Y. Smaragdakis, M. Bravenboer, and O. Lhoták, "Pick your contexts well: Understanding object-sensitivity," in *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011*, 2011, pp. 17–30.

[19] J. Whaley, D. Avots, M. Carbin, and M. S. Lam, "Using datalog with binary decision diagrams for program analysis," in *Programming Languages and Systems, Third Asian Symposium, APLAS 2005, Tsukuba, Japan, November 2-5, 2005, Proceedings*, 2005, pp. 97–118.

[20] Y. Smaragdakis and M. Bravenboer, "Using datalog for fast and easy program analysis," in *Datalog Reloaded - First International Workshop, Datalog 2010, Oxford, UK, March 16-19, 2010. Revised Selected Papers*, 2010, pp. 245–251.

[21] O. Lhoták and L. J. Hendren, "Jedd: A bdd-based relational extension of java," in *Proceedings of the 25th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2004, Washington, DC, USA, June 9-11, 2004*, 2004, pp. 158–169.

[22] K. Hoder, N. Bjørner, and L. M. de Moura, "μZ- an efficient engine for fixed points with constraints," in *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*, 2011, pp. 457–462.

[23] J. Queille and J. Sifakis, "Specification and verification of concurrent systems in CESAR," in *International Symposium on Programming, 5th Colloquium, Torino, Italy, April 6-8, 1982, Proceedings*, 1982, pp. 337–351.

[24] T. A. Henzinger, R. Jhala, R. Majumdar, and K. L. McMillan, "Abstractions from proofs," in *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2004, Venice, Italy, January 14-16, 2004*, 2004, pp. 232–244.

[25] S. Chaki, E. M. Clarke, A. Groce, S. Jha, and H. Veith, "Modular verification of software components in C," in *Proceedings of the 25th International Conference on Software Engineering, May 3-10, 2003, Portland, Oregon, USA*, 2003, pp. 385–395.

[26] R. Grigore and H. Yang, "Abstraction refinement guided by a learnt probabilistic model," in *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, 2016, pp. 485–498.

[27] A. Milanova, A. Rountev, and B. G. Ryder, "Parameterized object sensitivity for points-to and side-effect analyses for java," in *Proceedings of the International Symposium on Software Testing and Analysis, ISSTA 2002, Roma, Italy, July 22-24, 2002*, 2002, pp. 1–11.

[28] S. J. Fink, E. Yahav, N. Dor, G. Ramalingam, and E. Geay, "Effective typestate verification in the presence of aliasing," *ACM Trans. Softw. Eng. Methodol.*, vol. 17, no. 2, 9:1–9:34, 2008.

[29] T. W. Reps, S. Horwitz, and S. Sagiv, "Precise interprocedural dataflow analysis via graph reachability," in *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 1995, San Francisco, California, USA, January 23-25, 1995*, 1995, pp. 49–61.

[30] X. Zhang, M. Naik, and H. Yang, "Finding optimum abstractions in parametric dataflow analysis," in *Proceedings of the 34th ACM SIGPLAN Conference on Pro-*

gramming Language Design and Implementation, PLDI 2013, Seattle, WA, USA, June 16-19, 2013, 2013, pp. 365–376.

[31] S. Grebenshchikov, A. Gupta, N. P. Lopes, C. Popeea, and A. Rybalchenko, "HSF(C): A software verifier based on horn clauses - (competition contribution)," in *Tools and Algorithms for the Construction and Analysis of Systems - 18th International Conference, TACAS 2012, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2012, Tallinn, Estonia, March 24 - April 1, 2012. Proceedings*, 2012, pp. 549–551.

[32] N. Bjørner, K. L. McMillan, and A. Rybalchenko, "On solving universally quantified horn clauses," in *Static Analysis - 20th International Symposium, SAS 2013, Seattle, WA, USA, June 20-22, 2013. Proceedings*, 2013, pp. 105–125.

[33] T. A. Beyene, C. Popeea, and A. Rybalchenko, "Solving existentially quantified horn clauses," in *Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings*, 2013, pp. 869–882.

[34] P. Liang and M. Naik, "Scaling abstraction refinement via pruning," in *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*, 2011, pp. 590–601.

[35] W. Lee, W. Lee, and K. Yi, "Sound non-statistical clustering of static analysis alarms," in *Verification, Model Checking, and Abstract Interpretation - 13th International Conference, VMCAI 2012, Philadelphia, PA, USA, January 22-24, 2012. Proceedings*, 2012, pp. 299–314.

[36] W. Le and M. L. Soffa, "Path-based fault correlations," in *Proceedings of the 18th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2010, Santa Fe, NM, USA, November 7-11, 2010*, 2010, pp. 307–316.

[37] *Apache FTP Server*, `http://mina.apache.org/ftpserver-project/`.

[38] M. Naik, *Chord: A program analysis platform for Java*, `http://jchord.googlecode.com/`, 2006.

[39] I. Abío and P. J. Stuckey, "Encoding linear constraints into SAT," in *Principles and Practice of Constraint Programming - 20th International Conference, CP 2014, Lyon, France, September 8-12, 2014. Proceedings*, 2014, pp. 75–91.

[40] B. Livshits, M. Sridharan, Y. Smaragdakis, O. Lhoták, J. N. Amaral, B. E. Chang, S. Z. Guyer, U. P. Khedker, A. Møller, and D. Vardoulakis, "In defense of soundiness: A manifesto," *Commun. ACM*, vol. 58, no. 2, pp. 44–46, 2015.

[41] N. Ayewah, D. Hovemeyer, J. D. Morgenthaler, J. Penix, and W. Pugh, "Using static analysis to find bugs," *IEEE Software*, vol. 25, no. 5, pp. 22–29, 2008.

[42] T. Copeland, *Pmd applied*, 2005.

[43] M. Naik, A. Aiken, and J. Whaley, "Effective static race detection for java," in *Proceedings of the ACM SIGPLAN 2006 Conference on Programming Language Design and Implementation, PLDI 2006, Ottawa, Ontario, Canada, June 11-14, 2006*, 2006, pp. 308–319.

[44] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin, "Dynamically discovering likely program invariants to support program evolution," *IEEE Trans. Software Eng.*, vol. 27, no. 2, pp. 99–123, 2001.

[45] J. R. Quinlan, *C4.5: Programs for Machine Learning*. Morgan Kaufmann, 1993, ISBN: 1-55860-238-0.

[46] *Upwork*, `http://www.upwork.com`, Accessed: 2015-11-19, 2015.

[47] I. Dillig, T. Dillig, and A. Aiken, "Automated error diagnosis using abductive inference," in *Proceedings of the 33th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2012, Beijing, China - June 11 - 16, 2012*, 2012, pp. 181–192.

[48] T. Kremenek, K. Ashcraft, J. Yang, and D. Engler, "Correlation exploitation in error ranking," in *Proceedings of the 12th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2004, Newport Beach, CA, USA, October 31 - November 6, 2004*, 2004, pp. 83–93.

[49] H. Zhu, T. Dillig, and I. Dillig, "Automated inference of library specifications for source-sink property verification," in *Programming Languages and Systems - 11th Asian Symposium, APLAS 2013, Melbourne, VIC, Australia, December 9-11, 2013. Proceedings*, 2013, pp. 290–306.

[50] O. Bastani, S. Anand, and A. Aiken, "Specification inference using context-free language reachability," in *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*, 2015, pp. 553–566.

[51] L. N. Q. Do, K. Ali, B. Livshits, E. Bodden, J. Smith, and E. R. Murphy-Hill, "Just-in-time static analysis," in *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2017, Santa Barbara, CA, USA, July 10 - 14, 2017*, 2017, pp. 307–317.

[52] O. Padon, K. L. McMillan, A. Panda, M. Sagiv, and S. Shoham, "Ivy: Safety verification by interactive generalization," in *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, Santa Barbara, CA, USA, June 13-17, 2016*, 2016, pp. 614–630.

[53] 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 *Static Analysis, 12th International Symposium, SAS 2005, London, UK, September 7-9, 2005, Proceedings*, 2005, pp. 203–217.

[54] T. Kremenek and D. Engler, "Z-ranking: Using statistical analysis to counter the impact of static analysis approximations," in *Static Analysis, 10th International Symposium, SAS 2003, San Diego, CA, USA, June 11-13, 2003, Proceedings*, 2003, pp. 295–315.

[55] S. Blackshear and S. Lahiri, "Almost-correct specifications: A modular semantic framework for assigning confidence to warnings," in *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2013, Seattle, WA, USA, June 16-19, 2013*, 2013, pp. 365–376.

[56] S. Hallem, B. Chelf, Y. Xie, and D. R. Engler, "A system and language for building system-specific, static analyses," in *Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2002, Berlin, Germany, June 17-19, 2002*, 2002, pp. 69–82.

[57] M. Renieris and S. P. Reiss, "Fault localization with nearest neighbor queries," in *18th IEEE International Conference on Automated Software Engineering (ASE 2003), 6-10 October 2003, Montreal, Canada*, 2003, pp. 30–39.

[58] T. Ball, M. Naik, and S. K. Rajamani, "From symptom to cause: Localizing errors in counterexample traces," in *Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2003, New Orleans, Louisisana, USA, January 15-17, 2003*, 2003, pp. 97–105.

[59] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan, "Scalable statistical bug isolation," in *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation, PLDI 2005, Chicago, IL, USA, June 12-15, 2005*, 2005, pp. 15–26.

[60] J. A. Jones and M. J. Harrold, "Empirical evaluation of the tarantula automatic fault-localization technique," in *20th IEEE/ACM International Conference on Automated Software Engineering (ASE 2005), November 7-11, 2005, Long Beach, CA, USA*, 2005, pp. 273–282.

[61] J. A. Jones, M. J. Harrold, and J. T. Stasko, "Visualization of test information to assist fault localization," in *Proceedings of the 24th International Conference on Software Engineering, ICSE 2002, 19-25 May 2002, Orlando, Florida, USA*, 2002, pp. 467–477.

[62] D. von Dincklage and A. Diwan, "Optimizing programs with intended semantics," in *Proceedings of the 24th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2009, October 25-29, 2009, Orlando, Florida, USA*, 2009, pp. 409–424.

[63] ——, "Integrating program analyses with programmer productivity tools," *Softw., Pract. Exper.*, vol. 41, no. 7, pp. 817–840, 2011.

[64] G. Nelson and D. C. Oppen, "Simplification by cooperating decision procedures," *ACM Trans. Program. Lang. Syst.*, vol. 1, no. 2, pp. 245–257, 1979.

[65] M. Naik, H. Yang, G. Castelnuovo, and M. Sagiv, "Abstractions from tests," in *Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2012, Philadelphia, Pennsylvania, USA, January 22-28, 2012*, 2012, pp. 373–386.

[66] H. Oh, W. Lee, K. Heo, H. Yang, and K. Yi, "Selective x-sensitive analysis guided by impact pre-analysis," *ACM Trans. Program. Lang. Syst.*, vol. 38, no. 2, 6:1–6:45, 2016.

[67] S. Wei, O. Tripp, B. G. Ryder, and J. Dolby, "Revamping javascript static analysis via localization and remediation of root causes of imprecision," in *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016, Seattle, WA, USA, November 13-18, 2016*, 2016, pp. 487–498.

[68] S. Narayanasamy, Z. Wang, J. Tigani, A. Edwards, and B. Calder, "Automatically classifying benign and harmful data racesallusing replay analysis," in *Proceedings of the 28th ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation, PLDI 2007, San Diego, California, USA, June 10-13, 2007*, 2007, pp. 22–31.

[69] M. Naik, C. Park, K. Sen, and D. Gay, "Effective static deadlock detection," in *Proceedings of the 31st International Conference on Software Engineering, ICSE 2009, May 16-24, 2009, Vancouver, Canada, Proceedings*, 2009, pp. 386–396.

[70] M. C. Martin, V. B. Livshits, and M. S. Lam, "Finding application errors and security flaws using PQL: a program query language," in *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems,*

Languages, and Applications, OOPSLA 2005, October 16-20, 2005, San Diego, CA, USA, 2005, pp. 365–383.

[71] S. Guarnieri and V. B. Livshits, "GATEKEEPER: mostly static enforcement of security and reliability policies for javascript code," in *18th USENIX Security Symposium, Montreal, Canada, August 10-14, 2009, Proceedings*, 2009, pp. 151–168.

[72] V. B. Livshits, J. Whaley, and M. S. Lam, "Reflection analysis for java," in *Programming Languages and Systems, Third Asian Symposium, APLAS 2005, Tsukuba, Japan, November 2-5, 2005, Proceedings*, 2005, pp. 139–160.

[73] O. Lhoták, "Spark: A flexible points-to analysis framework for Java," Master's thesis, McGill University, 2002.

[74] O. Lhoták and L. J. Hendren, "Context-sensitive points-to analysis: Is it worth it?" In *Compiler Construction, 15th International Conference, CC 2006, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2006, Vienna, Austria, March 30-31, 2006, Proceedings*, 2006, pp. 47–64.

[75] M. Sridharan and R. Bodík, "Refinement-based context-sensitive points-to analysis for java," in *Proceedings of the 26th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2006, Ottawa, Ontario, Canada, June 11-14, 2006*, 2006, pp. 387–400.

[76] V. B. Livshits and M. S. Lam, "Finding security vulnerabilities in java applications with static analysis," in *Proceedings of the 14th USENIX Security Symposium, Baltimore, MD, USA, July 31 - August 5, 2005*, 2005.

[77] S. M. Blackburn, R. Garner, C. Hoffmann, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. L. Hosking, M. Jump, H. B. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanovic, T. VanDrunen, D. von Dincklage, and B. Wiedermann, "The dacapo benchmarks: Java benchmarking development and analysis," in *Proceedings of the 21th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2006, October 22-26, 2006, Portland, Oregon, USA*, 2006, pp. 169–190.

[78] *Securibench Micro*, `http://suif.stanford.edu/~livshits/work/securibench-micro/index.html`.

[79] *Pjbench*, `https://code.google.com/p/pjbench/`.

[80] F. Niu, C. Ré, A. Doan, and J. W. Shavlik, "Tuffy: Scaling up statistical inference in markov logic networks using an RDBMS," *PVLDB*, vol. 4, no. 6, pp. 373–384, 2011.

[81] M. Richardson and P. M. Domingos, "Markov logic networks," *Machine Learning*, vol. 62, no. 1-2, pp. 107–136, 2006.

[82] A. T. Chaganty, A. Lal, A. V. Nori, and S. K. Rajamani, "Combining relational learning with SMT solvers using CEGAR," in *Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings*, 2013, pp. 447–462.

[83] J. Noessner, M. Niepert, and H. Stuckenschmidt, "Rockit: Exploiting parallelism and symmetry for MAP inference in statistical relational models," in *Proceedings of the Twenty-Seventh AAAI Conference on Artificial Intelligence, July 14-18, 2013, Bellevue, Washington, USA*, 2013.

[84] S. Riedel, "Improving the accuracy and efficiency of MAP inference for markov logic," in *UAI 2008, Proceedings of the 24th Conference in Uncertainty in Artificial Intelligence, Helsinki, Finland, July 9-12, 2008*, 2008, pp. 468–475.

[85] R. Mangal, X. Zhang, A. Kamath, A. V. Nori, and M. Naik, "Scaling relational inference using proofs and refutations," in *Proceedings of the 30th AAAI Conference on Artificial Intelligence, February 12-17, 2016, Phoenix, Arizona, USA.*, 2016, pp. 3278–3286.

[86] E. I. Psallida, "Relational representation of the LLVM intermediate language," B.S. Thesis, University of Athens, Jan. 2014.

[87] K. Hoder, N. Bjørner, and L. M. de Moura, "μZ- an efficient engine for fixed points with constraints," in *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*, 2011, pp. 457–462.

[88] T. Kremenek, P. Twohey, G. Back, A. Ng, and D. Engler, "From uncertainty to belief: Inferring the specification within," in *7th Symposium on Operating Systems Design and Implementation (OSDI 2006), November 6-8, Seattle, WA, USA*, 2006, pp. 161–176.

[89] V. B. Livshits, A. V. Nori, S. K. Rajamani, and A. Banerjee, "Merlin: Specification inference for explicit information flow problems," in *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2009, Dublin, Ireland, June 15-21, 2009*, 2009, pp. 75–86.

[90] N. E. Beckman and A. V. Nori, "Probabilistic, modular and scalable inference of typestate specifications," in *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*, 2011, pp. 211–221.

[91] X. Zhang, R. Mangal, A. V. Nori, and M. Naik, "Query-guided maximum satisfiability," in *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, 2016, pp. 109–122.

[92] P. Singla and P. M. Domingos, "Discriminative training of markov logic networks," in *Proceedings, The Twentieth National Conference on Artificial Intelligence and the Seventeenth Innovative Applications of Artificial Intelligence Conference, July 9-13, 2005, Pittsburgh, Pennsylvania, USA*, 2005, pp. 868–873.

[93] ——, "Memory-efficient inference in relational domains," in *Proceedings, The Twenty-First National Conference on Artificial Intelligence and the Eighteenth Innovative Applications of Artificial Intelligence Conference, July 16-20, 2006, Boston, Massachusetts, USA*, 2006, pp. 488–493.

[94] C. Mencía, A. Previti, and J. Marques-Silva, "Literal-based MCS extraction," in *Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence, IJCAI 2015, Buenos Aires, Argentina, July 25-31, 2015*, 2015, pp. 1973–1979.

[95] M. Bilenko and R. J. Mooney, "Adaptive duplicate detection using learnable string similarity measures," in *Proceedings of the Ninth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, Washington, DC, USA, August 24 - 27, 2003*, 2003, pp. 39–48.

[96] H. Poon, P. M. Domingos, and M. Sumner, "A general method for reducing the complexity of relational inference and its application to MCMC," in *Proceedings of the Twenty-Third AAAI Conference on Artificial Intelligence, AAAI 2008, Chicago, Illinois, USA, July 13-17, 2008*, 2008, pp. 1075–1080.

[97] R. de Salvo Braz, E. Amir, and D. Roth, "Lifted first-order probabilistic inference," in *IJCAI-05, Proceedings of the Nineteenth International Joint Conference on Artificial Intelligence, Edinburgh, Scotland, UK, July 30 - August 5, 2005*, 2005, pp. 1319–1325.

[98] B. Milch, L. S. Zettlemoyer, K. Kersting, M. Haimes, and L. P. Kaelbling, "Lifted probabilistic inference with counting formulas," in *Proceedings of the Twenty-Third AAAI Conference on Artificial Intelligence, AAAI 2008, Chicago, Illinois, USA, July 13-17, 2008*, 2008, pp. 1062–1068.

[99] D. Poole, "First-order probabilistic inference," in *IJCAI-03, Proceedings of the Eighteenth International Joint Conference on Artificial Intelligence, Acapulco, Mexico, August 9-15, 2003*, 2003, pp. 985–991.

[100] C. H. Papadimitriou, *Computational complexity*. Addison-Wesley, 1994, ISBN: 978-0-201-53082-7.

[101] S. A. Cook, "The complexity of theorem-proving procedures," in *Proceedings of the 3rd Annual ACM Symposium on Theory of Computing, May 3-5, 1971, Shaker Heights, Ohio, USA*, 1971, pp. 151–158.

[102] J. Gu, P. W. Purdom, J. Franco, and B. W. Wah, "Algorithms for the satisfiability (SAT) problem: A survey," in *Satisfiability Problem: Theory and Applications, Proceedings of a DIMACS Workshop, Piscataway, New Jersey, USA, March 11-13, 1996*, 1996, pp. 19–152.

[103] M. Jose and R. Majumdar, "Cause clue clauses: Error localization using maximum satisfiability," in *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*, 2011, pp. 437–446.

[104] ——, "Bug-assist: Assisting fault localization in ANSI-C programs," in *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*, 2011, pp. 504–509.

[105] M. Richardson and P. M. Domingos, "Markov logic networks," *Machine Learning*, vol. 62, no. 1-2, pp. 107–136, 2006.

[106] P. M. Domingos, S. Kok, D. Lowd, H. Poon, M. Richardson, and P. Singla, "Markov logic," in *Probabilistic Inductive Logic Programming - Theory and Applications*, 2008, pp. 92–117.

[107] S. Miyazaki, K. Iwama, and Y. Kambayashi, "Database queries as combinatorial optimization problems," in *CODAS*, 1996, pp. 477–483.

[108] M. Aref, B. Kimelfeld, E. Pasalic, and N. Vasiloglou, "Extending datalog with analytics in logicblox," in *Proceedings of the 9th Alberto Mendelzon International Workshop on Foundations of Data Management, Lima, Peru, May 6 - 8, 2015.*, 2015.

[109] H. Xu, R. A. Rutenbar, and K. A. Sakallah, "Sub-sat: A formulation for relaxed boolean satisfiability with applications in routing," in *Proceedings of 2002 International Symposium on Physical Design, ISPD 2002, Del Mar, CA, USA, April 7-10, 2002*, 2002, pp. 182–187.

[110] A. Gracca, I. Lynce, J. Marques-Silva, and A. L. Oliveira, "Efficient and accurate haplotype inference by combining parsimony and pedigree information," in *Algebraic and Numeric Biology - 4th International Conference, ANB 2010, Hagenberg, Austria, July 31- August 2, 2010, Revised Selected Papers*, 2010, pp. 38–56.

[111] D. M. Strickland, E. R. Barnes, and J. S. Sokol, "Optimal protein structure alignment using maximum cliques," *Operations Research*, vol. 53, no. 3, pp. 389–402, 2005.

[112] M. Vasquez and J. Hao, "A "logic-constrained" knapsack formulation and a tabu algorithm for the daily photograph scheduling of an earth observation satellite," *Comp. Opt. and Appl.*, vol. 20, no. 2, pp. 137–157, 2001.

[113] Q. Yang, K. Wu, and Y. Jiang, "Learning action models from plan examples using weighted MAX-SAT," *Artificial Intelligence*, vol. 171, no. 2-3, pp. 107–143, Feb. 2007.

[114] F. Juma, E. I. Hsu, and S. A. McIlraith, "Preference-based planning via maxsat," in *Advances in Artificial Intelligence - 25th Canadian Conference on Artificial Intelligence, Canadian AI 2012, Toronto, ON, Canada, May 28-30, 2012. Proceedings*, 2012, pp. 109–120.

[115] N. Bjørner and N. Narodytska, "Maximum satisfiability using cores and correction sets," in *Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence, IJCAI 2015, Buenos Aires, Argentina, July 25-31, 2015*, 2015, pp. 246–252.

[116] A. Morgado, C. Dodaro, and J. Marques-Silva, "Core-guided maxsat with soft cardinality constraints," in *Principles and Practice of Constraint Programming - 20th International Conference, CP 2014, Lyon, France, September 8-12, 2014. Proceedings*, 2014, pp. 564–573.

[117] A. Morgado, F. Heras, M. H. Liffiton, J. Planes, and J. Marques-Silva, "Iterative and core-guided maxsat solving: A survey and assessment," *Constraints*, vol. 18, no. 4, pp. 478–534, 2013.

[118] C. Ansótegui, M. L. Bonet, and J. Levy, "SAT-based MaxSAT algorithms," *Artificial Intelligence*, vol. 196, pp. 77–105, 2013.

[119] J. Marques-Silva and J. Planes, "Algorithms for maximum satisfiability using unsatisfiable cores," in *Design, Automation and Test in Europe, DATE 2008, Munich, Germany, March 10-14, 2008*, 2008, pp. 408–413.

[120] R. Martins, S. Joshi, V. M. Manquinho, and I. Lynce, "Incremental cardinality constraints for maxsat," in *Principles and Practice of Constraint Programming - 20th International Conference, CP 2014, Lyon, France, September 8-12, 2014. Proceedings*, 2014, pp. 531–548.

[121] N. Narodytska and F. Bacchus, "Maximum satisfiability using core-guided maxsat resolution," in *Proceedings of the Twenty-Eighth AAAI Conference on Artificial*

Intelligence, July 27 -31, 2014, Québec City, Québec, Canada., 2014, pp. 2717–2723.

[122] A. Ignatiev, A. Morgado, V. M. Manquinho, I. Lynce, and J. Marques-Silva, "Progression in maximum satisfiability," in *ECAI 2014 - 21st European Conference on Artificial Intelligence, 18-22 August 2014, Prague, Czech Republic - Including Prestigious Applications of Intelligent Systems (PAIS 2014)*, 2014, pp. 453–458.

[123] F. Heras, A. Morgado, and J. Marques-Silva, "Core-guided binary search algorithms for maximum satisfiability," in *Proceedings of the Twenty-Fifth AAAI Conference on Artificial Intelligence, AAAI 2011, San Francisco, California, USA, August 7-11, 2011*, 2011.

[124] R. Mangal, X. Zhang, A. V. Nori, and M. Naik, "Volt: A lazy grounding framework for solving very large maxsat instances," in *Theory and Applications of Satisfiability Testing - SAT 2015 - 18th International Conference, Austin, TX, USA, September 24-27, 2015, Proceedings*, 2015, pp. 299–306.

[125] M. Janota, *MiFuMax — a literate MaxSAT solver*, 2013.

[126] *AI Genealogy Project*, http://aigp.eecs.umich.edu.

[127] *Dblp: Computer science bibliography*, http://http://dblp.uni-trier.de.

[128] J. Marques-Silva, F. Heras, M. Janota, A. Previti, and A. Belov, "On computing minimal correction subsets," in *Proceedings of the 23rd International Joint Conference on Artificial Intelligence, IJCAI 2013, Beijing, China, August 3-9, 2013*, 2013, pp. 615–622.

[129] H. A. Kautz, B. Selman, and Y. Jiang, "A general stochastic approach to solving problems with hard and soft constraints," in *Satisfiability Problem: Theory and Applications, Proceedings of a DIMACS Workshop, Piscataway, New Jersey, USA, March 11-13, 1996*, 1996, pp. 573–586.

[130] V. V. Vazirani, *Approximation algorithms*. Springer Science & Business Media, 2013.

[131] *Max-SAT Evaluation*, http://www.maxsat.udl.cat/.

[132] L. Getoor and B. Taskar, *Introduction to Statistical Relational Learning (Adaptive Computation and Machine Learning)*. The MIT Press, 2007.

[133] A. Kimmig, S. H. Bach, M. Broecheler, B. Huang, and L. Getoor, "A short introduction to probabilistic soft logic," in *NIPS Workshop on Probabilistic Programming: Foundations and Applications*, 2012.

[134] W. Y. Wang, K. Mazaitis, N. Lao, and W. W. Cohen, "Efficient inference and learning in a large knowledge base - reasoning with extracted information using a locally groundable first-order probabilistic logic," *Machine Learning*, vol. 100, no. 1, pp. 101–126, 2015.

[135] S. Horwitz, T. W. Reps, and D. Binkley, "Interprocedural slicing using dependence graphs," in *Proceedings of the 26th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), PLDI 1988, Atlanta, Georgia, USA, June 22-24, 1988*, 1988, pp. 35–46.

[136] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre, "Software verification with BLAST," in *Model Checking Software, 10th International SPIN Workshop. Portland, OR, USA, May 9-10, 2003, Proceedings*, 2003, pp. 235–239.

[137] S. Z. Guyer and C. Lin, "Client-driven pointer analysis," in *Static Analysis, 10th International Symposium, SAS 2003, San Diego, CA, USA, June 11-13, 2003, Proceedings*, 2003, pp. 214–236.

[138] M. Sridharan, D. Gopan, L. Shan, and R. Bodík, "Demand-driven points-to analysis for java," in *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2005, October 16-20, 2005, San Diego, CA, USA*, 2005, pp. 59–76.

[139] N. Bjørner and A. Phan, "νz - maximal satisfaction with Z3," in *6th International Symposium on Symbolic Computation in Software Science, SCSS 2014, Gammarth, La Marsa, Tunisia, December 7-8, 2014*, 2014, pp. 1–9.

[140] Y. Li, A. Albarghouthi, Z. Kincaid, A. Gurfinkel, and M. Chechik, "Symbolic optimization with SMT solvers," in *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2014, San Diego, CA, USA, January 20-21, 2014*, 2014, pp. 607–618.

[141] D. Larraz, A. Oliveras, E. Rodríguez-Carbonell, and A. Rubio, "Proving termination of imperative programs using max-smt," in *Formal Methods in Computer-Aided Design, FMCAD 2013, Portland, OR, USA, October 20-23, 2013*, 2013, pp. 218–225.

[142] D. Larraz, K. Nimkar, A. Oliveras, E. Rodríguez-Carbonell, and A. Rubio, "Proving non-termination using max-smt," in *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings*, 2014, pp. 779–796.

[143] G. Katz, C. W. Barrett, D. L. Dill, K. Julian, and M. J. Kochenderfer, "Reluplex: An efficient SMT solver for verifying deep neural networks," in *Computer Aided*

Verification - 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part I, 2017, pp. 97–117.

[144] X. Huang, M. Kwiatkowska, S. Wang, and M. Wu, "Safety verification of deep neural networks," in *Computer Aided Verification - 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part I*, 2017, pp. 3–29.

[145] O. Bastani, Y. Ioannou, L. Lampropoulos, D. Vytiniotis, A. V. Nori, and A. Criminisi, "Measuring neural net robustness with constraints," in *Advances in Neural Information Processing Systems 29: Annual Conference on Neural Information Processing Systems 2016, NIPS 2016, December 5-10, 2016, Barcelona, Spain*, 2016, pp. 2613–2621.

[146] L. G. Valiant, "A theory of the learnable," *Commun. ACM*, vol. 27, no. 11, pp. 1134–1142, 1984.

[147] S. Muggleton and L. D. Raedt, "Inductive logic programming: Theory and methods," *J. Log. Program.*, vol. 19/20, pp. 629–679, 1994.

[148] R. Alur, R. Bodík, G. Juniwal, M. M. K. Martin, M. Raghothaman, S. A. Seshia, R. Singh, A. Solar-Lezama, E. Torlak, and A. Udupa, "Syntax-guided synthesis," in *Formal Methods in Computer-Aided Design, FMCAD 2013, Portland, OR, USA, October 20-23, 2013*, 2013, pp. 1–8.

[149] K. A. Ross, "Modular stratification and magic sets for datalog programs with negation," *J. ACM*, vol. 41, no. 6, pp. 1216–1266, 1994.

[150] D. Poole, "First-order probabilistic inference," in *IJCAI-03, Proceedings of the Eighteenth International Joint Conference on Artificial Intelligence, Acapulco, Mexico, August 9-15, 2003*, 2003, pp. 985–991.

[151] R. de Salvo Braz, E. Amir, and D. Roth, "Lifted first-order probabilistic inference," in *IJCAI-05, Proceedings of the Nineteenth International Joint Conference on Artificial Intelligence, Edinburgh, Scotland, UK, July 30 - August 5, 2005*, 2005, pp. 1319–1325.

[152] P. Singla and P. M. Domingos, "Lifted first-order belief propagation," in *Proceedings of the Twenty-Third AAAI Conference on Artificial Intelligence, AAAI 2008, Chicago, Illinois, USA, July 13-17, 2008*, 2008, pp. 1094–1099.

[153] B. Milch, L. S. Zettlemoyer, K. Kersting, M. Haimes, and L. P. Kaelbling, "Lifted probabilistic inference with counting formulas," in *Proceedings of the Twenty-Third AAAI Conference on Artificial Intelligence, AAAI 2008, Chicago, Illinois, USA, July 13-17, 2008*, 2008, pp. 1062–1068.

[154] G. V. den Broeck, N. Taghipour, W. Meert, J. Davis, and L. D. Raedt, "Lifted probabilistic inference by first-order knowledge compilation," in *IJCAI 2011, Proceedings of the 22nd International Joint Conference on Artificial Intelligence, Barcelona, Catalonia, Spain, July 16-22, 2011*, 2011, pp. 2178–2185.