Abstraction Refinement Guided by a Learnt Probabilistic Model

Radu Grigore Hongseok Yang
Oxford University

Abstract

The core challenge in designing an effective static program analysis is to find a good program abstraction – one that retains only details relevant to a given query. In this paper, we present a new approach for automatically finding such an abstraction. Our approach uses a pessimistic strategy, which can optionally use guidance from a probabilistic model. Our approach applies to parametric static analyses implemented in Datalog, and is based on counterexample-guided abstraction refinement. For each untried abstraction, our probabilistic model provides a probability of success, while the size of the abstraction provides an estimate of its cost in terms of analysis time. Combining these two metrics, probability and cost, our refinement algorithm picks an optimal abstraction. Our probabilistic model is a variant of the Erdős–Rényi random graph model, and it is tunable by what we call hyperparameters. We present a method to learn good values for these hyperparameters, by observing past runs of the analysis on an existing codebase. We evaluate our approach on an object sensitive pointer analysis for Java programs, with two client analyses (PolySite and Downcast).

Categories and Subject Descriptors D.2.4 [*Software Engineering*]: Software/Program Verification

Keywords Datalog, Horn, hypergraph, probability

1. Introduction

We wish that static program analyses would become better as they see more code. Starting from this motivation, we designed an abstraction refinement algorithm that incorporates knowledge learnt from observing its own previous runs, on an existing codebase. For a given query about a program, this knowledge guides the algorithm towards a good abstraction that retains only the details of the program relevant to the query. Similar guidance also features in existing abstraction refinement algorithms [4, 8, 20], but is based on nontrivial heuristics that are developed manually by analysis designers. These heuristics are often suboptimal and difficult to transfer from one analysis to another. Our algorithm has the potential to improve itself by learning from past runs, and it applies to almost any analysis implemented in Datalog.

Prior work on abstraction refinement for Datalog [54] implicitly uses an optimistic strategy: the search is geared towards finding an abstraction that would show the current counterexample to be spurious. We take the complimentary approach: our search is geared towards finding an abstraction that would show the current counterexample to be unavoidable. Furthermore, we bias the search by using a probabilistic model, which is tuned using information from previous runs of the analysis.

In other approaches to program analysis that are based on learning [42, 53], the analysis designer must choose appropriate features. A feature is a measurable property of the program, usually a numeric one. Choosing features that are effective for program analysis is nontrivial, and involves knowledge of both the analysis and the probabilistic model. In our approach, the analysis designer does not need to choose appropriate features.

Instead of observing features, our models observe directly the internal representations of analysis runs. Parametric static analyses implemented in Datalog consist of universally quantified Horn clauses, and work by instantiating the universal quantification of these clauses, while respecting the constraints on instantiation imposed by a given parameter setting. These instantiated Horn clauses are typically implications of the form

$$h \leftarrow t_1, t_2, \ldots, t_n$$

and can be understood as a directed (hyper) arc from the source vertices $t_1, \ldots, t_n$ to the target vertex h. Thus, the instantiated Horn clauses taken altogether form a hypergraph. This hypergraph changes when we try the analysis again with a different parameter setting. Given a hypergraph obtained under one parameter setting, we build a probabilistic model that predicts how the hypergraph would change if a new and more precise parameter setting were used. In particular, the probabilistic model estimates how likely it is that the new parameter setting will end the refinement process, which happens when the new hypergraph includes evidence that the analysis will never prove a query. Technically, our probabilistic model is a variant of the Erdős–Rényi random graph model [11]: given a template hypergraph G, each of its subhypergraphs H is assigned a probability, which depends on the values of the hyperparameters. Intuitively, this probability quantifies the chance that H correctly describes the changes in G when the analysis is run with the new and more precise parameter settings. The hyperparameters quantify how much approximation occurs in each of the quantified Horn clauses of the analysis. We provide an efficient method for learning hyperparameters from prior analysis runs. Our method uses certain analytic bounds in order to avoid the combinatorial explosion of a naive learning method based on maximum likelihood; the explosion is caused by H being a latent variable, which can be observed only indirectly.

The next parameter setting to try is chosen by our refinement algorithm based on predictions of the probabilistic model but also

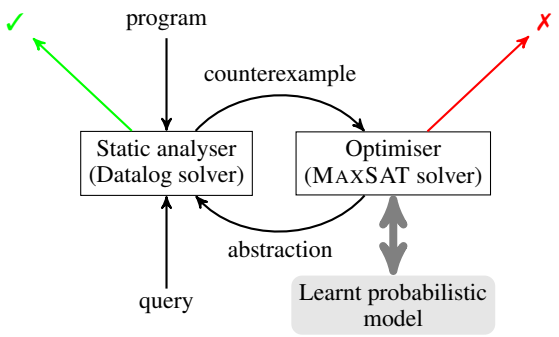

Figure 1. Architecture

```
    object x, y, z, v
    assume x.dirty
    x.value := 10
0:  smudge2(x, y)
0': y.value := y.value + 2 * x.value
1:  smudge3(y, z)
    if z.dirty && y.value > 5
      v.value := x.value + y.value
2:  smudge3(z, v)
    ...
3:  smudge5(x, y)
    ...
4:  smudge7(y, v)
    assert !v.dirty
```

Figure 2. Example program to analyse

based on an estimate of the runtime cost. For each parameter setting, the probability of successfully handling the query is evaluated by our model, and the runtime is estimated to increase with the precision of the parameter setting. We prove that our method of integrating these two metrics is optimal, under reasonable assumptions.

The paper starts with an informal overview of our approach (Section 2) and a review of notations from probability theory (Section 3), and is followed by a description of our probabilistic model (Section 4) and its learning algorithm (Section 5). The probabilistic model is then used to implement a refinement loop that optimally chooses the next parameter setting (Section 6). The experimental evaluation (Section 7) shows the value of the pessimistic strategy, but suggests we need better optimisers in order to take full advantage of the probabilistic model. Section 8 positions our work in the various attempts to combine probabilistic reasoning and static analyses, and Section 9 concludes the paper. Most proofs are in the full version.

2. Overview

Figure 1 gives a high level overview of our abstraction refinement algorithm, and in particular it shows the role of our probabilistic model. The refinement loop is standard, with analysis on one side and refinement on the other. Our contribution lies in the refinement part, which receives guidance from a learnt probabilistic model and chooses the next abstraction by balancing the model's prediction and the estimated cost of running the analysis under each abstraction.

We assume that the analysis is given and obeys two constraints. The first is that the analysis is implemented in Datalog – it is specified in terms of universally quantified Horn clauses, such as

$$\texttt{pointsto}(\alpha, \ell) \leftarrow \texttt{precise}(\alpha), \texttt{pointsto}(\beta, \ell), \texttt{assignTo}(\beta, \alpha) \quad (1)$$

in which all the free variables α, β, ℓ are implicitly universally quantified. We call these clauses ***Datalog rules***. The analysis works by instantiating the quantification of these rules, and thus deriving new facts. A query is a particular fact such as $\texttt{pointsto}(x, h)$, which is an instantiation of the left side of the rule (1), with $\alpha := x$ and $\ell := h$. The query represents an undesirable situation in the program being analysed. The analysis could derive the query because the undesirable situation really occurs at runtime. But, the analysis could also derive the query because it approximates the runtime semantics. Our task is to decide whether it is possible to avoid deriving the query by approximating less. If the query is derived, then the set of all instances of Datalog rules constitute a counterexample, which is then used for refinement.

The second constraint is that the analysis is parametric. For instance, it might have a parameter for each program variable, which specifies whether the variable should be tracked precisely or not. The analysis would encode a setting of these parameters in Datalog by using relations `cheap` and `precise`. In fact, the Datalog rule (1) assumes such parametrisation and fires only when the parameter setting dictates the precise tracking of the variable α. For a parametric analysis, an abstraction can be specified by a parameter setting, and so we use these two terms interchangeably.

The refinement part analyses a counterexample, and suggests a new promising parameter setting. If the counterexample derives the query without relying on approximations, then the refinement part reports impossibility and stops [50, 54, 55]. If the counterexample derives the query by relying on approximations, then the refinement part sets itself the goal to find a similar counterexample that does not rely on approximations. This is a pessimistic goal. To find such a similar counterexample, the analysis must be run with a different parameter setting. Which one? On the one hand, the parameter setting should be likely to uncover a similar counterexample. On the other hand, the parameter should be as cheap as possible. The refinement part uses a MaxSAT solver to balance these desiderata.

Consider now the example program in Figure 2. The language is idiosyncratic, and so will be the analysis. The language and the analysis are chosen to allow a concise rendering of the main ideas. In this toy language, each object has two fields, the boolean *dirty* and the integer *value*. Initially, all *value* fields are 0. Object x is dirty at the beginning, and we are interested in whether object v is dirty at the end. Dirtiness is propagated from one object to another only by the primitive commands `smudgeK`. The effect of the command $\texttt{smudgeK}(x, y)$ is equivalent to the following pseudocode:

$$\text{if } (x.value + y.value) \bmod K = 0$$
$$y.dirty := x.dirty \vee y.dirty$$

That is, if the sum of the values of objects x and y is a multiple of K, then dirt propagates from x to y.

To decide whether object v is dirty at the end, an analysis may need to track the values of multiple objects. The values can be changed by guarded assignments. The guard of an assignment can be any boolean expression; the right hand side of an assignment can be any integer expression. In short, tracking values and relations between values could be expensive.

However, tracking all values may also be unnecessary. In the first iteration, the analysis treats all non-smudge commands as `skip`. As a result, the analysis knows nothing about the *value* fields. To remain sound, it assumes that smudge commands always propagate dirtiness; that is, it treats the command $\texttt{smudgeK}(x, y)$ as equivalent to the following pseudocode, dropping the guard:

$$y.dirty := x.dirty \vee y.dirty$$

If, using these approximate semantics, the analysis concluded that v is clean at the end, then it would stop. But, in our example, v could

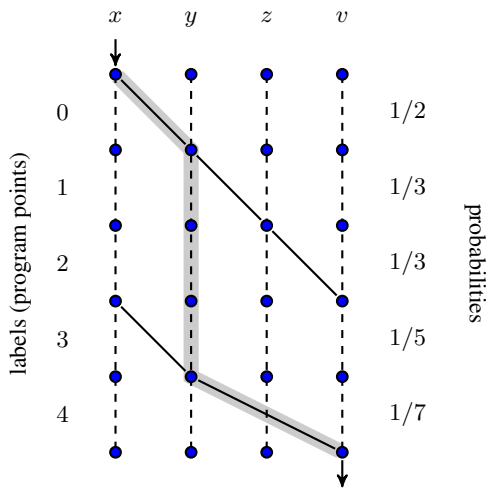

Figure 3. Abstract view of the program in Figure 2. Each label on the left identifies a smudge command. The dashed, vertical lines signify that once an object is dirty it remains dirty. The solid, oblique lines signify that smudge commands might propagate dirtiness. Depending on the values of the objects, a `smudgeK` command propagates dirtiness with probability $1/K$. The highlighted path illustrates one way in which dirtiness could propagate from object x to object v, thus violating the assertion.

be dirty at the end, for example because of the smudge commands on lines 0 and 4: the smudge on line 0 propagates dirtiness from x to y, and the smudge on line 4 propagates dirtiness from y to v. This scenario corresponds to the highlighted path in Figure 3.

Before seeing what happens in the next iteration, let us first describe the analysis in more detail. The approximate semantics of the command `smudge2` are modelled by the following Datalog rule:

$$\texttt{dirty}(\ell', \beta) \leftarrow \texttt{cheap}(\ell), \texttt{dirty}(\ell, \alpha), \texttt{flow}(\ell, \ell'), \texttt{smudge2}(\ell, \alpha, \beta) \tag{2}$$

The rule makes use of the following relations:

$\texttt{flow}(\ell, \ell')$	the control flow goes from ℓ to ℓ'
$\texttt{smudge2}(\ell, \alpha, \beta)$	the command at ℓ is $\texttt{smudge2}(\alpha, \beta)$
$\texttt{cheap}(\ell)$	the command at ℓ should be approximated
$\texttt{dirty}(\ell, \alpha)$	$\alpha.\mathit{dirty}$ is **true** before the command at ℓ

The relations `flow` and `smudge2` encode the program that is being analysed. The relation `cheap` parametrises the analysis, by allowing it or disallowing it to approximate the semantics of particular commands. Finally, the relation `dirty` expresses facts about executions of the program that is being analysed. From the point of view of the analysis, `flow`, `smudge2`, and `cheap` are part of the input, while `dirty` is part of the output. The relations `flow` and `smudge2` are simply a transliteration of the program text. The relation `cheap` is computed by a refinement algorithm, which we will see later.

The precise semantics of `smudge2` can also be encoded with a Datalog rule, albeit a more complicated one.

$$\begin{aligned}\texttt{dirty}(\ell', \beta) \leftarrow\ & \texttt{precise}(\ell), \texttt{dirty}(\ell, \alpha), \texttt{flow}(\ell, \ell'), \\ & \texttt{smudge2}(\ell, \alpha, \beta), \texttt{value}(\ell, \alpha, a), \\ & \texttt{value}(\ell, \beta, b), (a + b) \bmod 2 = 0\end{aligned} \tag{3}$$

This rule makes use of two further relations:

$\texttt{precise}(\ell)$	the command at ℓ should not be approximated
$\texttt{value}(\ell, \alpha, a)$	$\alpha.\mathit{value} = a$ holds before the command at ℓ

Like `cheap`, the relation `precise` is part of the input. If the input relation `precise` activates rules like the one above, then the analysis takes longer not only because the rule is more complicated, but also because it needs to compute more facts about the relation `value`.

The refinement algorithm ensures that for each program point ℓ exactly one of $\texttt{cheap}(\ell)$ and $\texttt{precise}(\ell)$ holds. In the first iteration, $\texttt{cheap}(\ell)$ holds for all ℓ, and `precise` holds for no ℓ. In each of the next iterations, the refinement algorithm switches some program points from cheap to precise semantics.

Let us see what happens when one program point is switched from cheap to precise. In the first iteration, $\texttt{cheap}(0)$ is part of the input, and the following rule instance derives $\texttt{dirty}(0', \texttt{y})$:

$$\texttt{dirty}(0', \texttt{y}) \leftarrow \texttt{cheap}(0), \texttt{dirty}(0, \texttt{x}), \texttt{flow}(0, 0'), \texttt{smudge2}(0, \texttt{x}, \texttt{y})$$

Let us now look at the scenario in which for the second iteration the fact $\texttt{cheap}(0)$ is replaced by the fact $\texttt{precise}(0)$. In this case, $\texttt{dirty}(0', \texttt{y})$ is still derived, this time by the following rule instance:

$$\begin{aligned}\texttt{dirty}(0', \texttt{y}) \leftarrow\ & \texttt{precise}(0), \texttt{dirty}(0, \texttt{x}), \texttt{flow}(0, 0'), \\ & \texttt{smudge2}(0, \texttt{x}, \texttt{y}), \texttt{value}(0, \texttt{x}, 10), \\ & \texttt{value}(0, \texttt{y}, 0), (10 + 0) \bmod 2 = 0\end{aligned}$$

To be able to apply this rule, the analysis had to work harder, to derive the intermediate results $\texttt{value}(0, \texttt{x}, 10)$ and $\texttt{value}(0, \texttt{y}, 0)$. Using $\texttt{precise}(0)$ influences other Datalog rules as well, and forces the analysis to derive these intermediate results, so that $\texttt{dirty}(0', \texttt{y})$ is still derived. This is not always the case. For example, the `smudge3` command at program point 1 will not propagate dirtiness if the precise semantics is used.

Let us now step back and see which parts of the example generalise.

Model. If we replace $\texttt{cheap}(\ell)$ by $\texttt{precise}(\ell)$, then the set of Datalog rule instances could change unpredictably. Yet, we observe empirically that the change is confined to one of two cases:

(a) $\texttt{precise}(\ell)$ eventually derives facts similar to those facts that $\texttt{cheap}(\ell)$ derives, but with more work; or

(b) $\texttt{precise}(\ell)$ no longer derives the facts that $\texttt{cheap}(\ell)$ derived.

This dichotomy is by no means necessary. Intuitively, it holds because the Datalog rules are not arbitrary: they are implementing a program analysis. In our example, case (a) occurs when $\texttt{cheap}(0)$ is replaced by $\texttt{precise}(0)$, and case (b) occurs when $\texttt{cheap}(1)$ is replaced by $\texttt{precise}(1)$. In general, we formalise this dichotomy by requiring that a certain predictability condition holds. The condition is flexible, in that it allows one to choose the meaning of 'similar' in case (a) by defining a so called projection function. In our example, no projection is necessary. In context sensitive analyses, projection corresponds to truncating contexts. In general, by adjusting the definition of the projection function we can exploit more knowledge about the analysis, if we so wish. If we do not, then it is always possible to choose a trivial projection for which the meaning of 'similar' is 'exactly the same'.

Provided that the predictability condition holds, which is a formal way of saying that the dichotomy between cases (a) and (b) holds, it is natural to define the probabilistic model as a variant of the Erdős–Rényi random graph model. Our sets of Datalog rule instances are seen as sets of arcs of a hypergraph. Each arc of the hypergraph is either selected or not, with a certain probability. Being selected corresponds to case (a) – having a counterpart in the precise

hypergraph; being unselected corresponds to case (b) – not having a counterpart in the precise hypergraph.

For the predictability condition and for the projection function, we drew inspiration from abstract interpretation [10]. Intuitively, our projection functions correspond to concretisation maps, and our predictability condition corresponds to correctness of approximation. However, we did not formalise this intuitive correspondence.

Learning. The model predicts that each rule instance is selected (that is, has a precise counterpart) with some probability. How to pick this probability? Figure 3 gives an intuitive representation of a set of instances. In particular, each dashed arc and each solid arc represents some rule instance. We assume that instances represented by dashed arcs are selected with probability 1. These are instances of some rule which says that a dirty object remains dirty. We also assume that instances represented by solid arcs are selected with probability $1/K$. These are instances of rules of the form (2), which describe the semantics of `smudgeK` commands. These probabilities make intuitive sense. In particular, it is reasonable to expect that a number is a multiple of K with probability $1/K$.

But, how can we design an algorithm to find these probabilities, without appealing to intuition and knowledge about arithmetic? The answer is that we run the analysis on many programs, and observe whether rule instances have precise counterparts or not. In our example, if the training sample is large enough, we would observe that instances of the form (2) do indeed have counterparts of the form (3) in about $1/K$ of cases. In general, it is not possible to observe directly which rules have precise counterparts. It is difficult to decide which rule is a counterpart of which rule. Instead, we make indirect observations based on which similar facts are derived.

Refinement. In terms of Figure 3, refinement can be understood intuitively as follows. We are interested in whether there is a path from the input on the top left to the output on the bottom right. We know the dashed arcs are really present: they have a precise counterpart with probability 1. We do not know if the solid arcs are really present: we see them only because we used a cheap parameter setting, and they have a precise counterpart only with probability $1/K$. We can find out whether the solid arcs are really present or just an illusion, by running the analysis with a more precise parameter setting. But, we have to pay a price, because more precise parameter settings are also more expensive.

The question is then which of the solid arcs should we enquire about, such that we decide quickly whether there is a path from input to output. There are several possible strategies, in particular there is an optimistic strategy and a pessimistic strategy. The optimistic strategy hopes that there is no path, so object v is clean at the end. Accordingly, the optimistic strategy considers asking about those sets of solid arcs that could disconnect the input from the output, if the arcs were not really there. The pessimistic strategy hopes that there is a path, so object v is dirty at the end. Accordingly, the pessimistic strategy considers asking about those sets of solid arcs that could connect the input to the output, if the arcs were really there. The highlighted path in Figure 3 corresponds to replacing `cheap`(0) by `precise`(0), and also `cheap`(4) by `precise`(4). Thus, let us denote its set of arcs as 04. There are two other paths that the pessimistic strategy will consider, whose sets of arcs are 012 and 34. The path 04 gets a probability $1/2 \times 1/7$ of surviving; the path 012 gets a probability $1/2 \times 1/3 \times 1/3$ of surviving; the path 34 gets a probability $1/5 \times 1/7$ of surviving. According to probabilities, the path 04 has the highest chance of showing that v is dirty at the end.

We designed an algorithm which generalises the pessimistic strategy described above by taking into account unions of paths and also the runtime cost of trying a parameter setting. Our refinement algorithm has to work in a more general setting than suggested by Figure 3. In particular, it must handle hypergraphs, not just graphs.

3. Preliminaries and Notations

In this section we recall several basic notions from probability theory. At the same time, we introduce the notation used throughout the paper.

A ***finite probability space*** is a finite set Ω together with a function $\Pr : \Omega \to \mathbb{R}$ such that $\Pr(\omega) \geq 0$ for all $\omega \in \Omega$, and $\sum_{\omega \in \Omega} \Pr(\omega) = 1$. An ***event*** is a subset of Ω. The ***probability of an event*** A is

$$\Pr(A) := \sum_{\omega \in A} \Pr(\omega) = \sum_{\omega \in \Omega} \Pr(\omega)[\omega \in A]$$

The notation $[\Psi]$ is the Iverson bracket: if Ψ is true it evaluates to 1, if Ψ is false it evaluates to 0. A ***random variable*** is a function $\mathbf{X} : \Omega \to \mathcal{X}$. For each value $x \in \mathcal{X}$, the set $\mathbf{X}^{-1}(x)$ is an event, traditionally denoted by $(\mathbf{X} = x)$. In particular, we write $\Pr(\mathbf{X} = x)$ for its probability; occasionally, we may write $\Pr(x = \mathbf{X})$ for the same probability. A ***boolean random variable*** is a function $\mathbf{X} : \Omega \to \{0, 1\}$. For a random variable $\mathbf{X}$ with $\mathcal{X} \subseteq \mathbb{R}$, we define its ***expectation*** $\mathrm{E}\,\mathbf{X}$ by

$$\mathrm{E}\,\mathbf{X} := \sum_{x \in \mathcal{X}} x \Pr(\mathbf{X} = x) = \sum_{\omega \in \Omega} \Pr(\omega)\mathbf{X}(\omega)$$

In particular, if $\mathbf{X}$ is a boolean random variable, then

$$\mathrm{E}\,\mathbf{X} = \Pr(\mathbf{X} = 1)$$

Events $A_1, \ldots, A_n$ are said to be ***independent*** when

$$\Pr(A_1 \cap \ldots \cap A_n) = \prod_{i=1}^{n} \Pr(A_i)$$

Note that n events could be pairwise independent, but still dependent when taken altogether. Random variables $\mathbf{X}_1, \ldots, \mathbf{X}_n$ are said to be independent when the events $(\mathbf{X}_1 = x_1), \ldots, (\mathbf{X}_n = x_n)$ are independent for all $x_1, \ldots, x_n$ in their respective domains. In particular, if $\mathbf{X}_1, \ldots, \mathbf{X}_n$ are independent boolean random variables, then $\mathbf{X_1} \wedge \ldots \wedge \mathbf{X}_n$ is also a boolean random variable, and

$$\mathrm{E}(\mathbf{X}_1 \wedge \ldots \wedge \mathbf{X}_n) = \prod_{i=1}^{n} \mathrm{E}\,\mathbf{X}_i$$

Events A and B are said to be ***incompatible*** when they are disjoint. In that case, $\Pr(A \cup B) = \Pr(A) + \Pr(B)$. In particular, if $\mathbf{X}_1, \ldots, \mathbf{X}_n$ are boolean random variables such that the events $(\mathbf{X}_1 = 1), \ldots, (\mathbf{X}_n = 1)$ are pairwise incompatible, then

$$\mathrm{E}(\mathbf{X}_1 \vee \ldots \vee \mathbf{X}_n) = \sum_{i=1}^{n} \mathrm{E}\,\mathbf{X}_i$$

4. Probabilistic Model

The probabilistic model predicts what analyses would do if they were run with precise parameter settings. To make such predictions, the model relies on several assumptions: the analysis must be implemented in Datalog (Section 4.1) and its precision must be configurable by parameters (Section 4.2); furthermore, increasing precision should correspond to invalidating some derivation steps (Section 4.3). Given probabilities that individual derivation steps survive the increase in precision, we compute probabilities that sets of derivation steps survive the increase in precision (Section 4.4). Given which set of derivation steps survives the increase in precision, we can tell whether a given query, which signifies a bug, is still reachable (Section 4.5).

4.1 Datalog Programs and Hypergraphs

We shall use a simplified model of Datalog programs, which is essentially a directed hypergraph. The semantics will then be given

by reachability in this hypergraph. For readers already familiar with Datalog, it may help to think of vertices as elements of Datalog relations, and to think of arcs as instances of Datalog rules with non-relational constraints removed. For readers not familiar with Datalog, simply thinking in terms of the hypergraph introduced below will be sufficient to understand the rest of the paper.

We assume a finite universe of ***facts***. An ***arc*** is a pair (h, B) of a head h and a body B; the ***head*** is a fact; the ***body*** is a set of facts. A ***hypergraph*** is a set of arcs. The ***vertices*** of a hypergraph are those facts that appear in its arcs. If a hypergraph G contains an arc (h, B), then we say that h is reachable from B in G. In general, given a hypergraph G and a set T of facts, the set $\mathcal{R}_G T$ of facts reachable from T in G is defined as the least fixed-point of the following recursive equation:

$$\{\, h \mid (h, B) \in G \text{ and } B \subseteq \mathcal{R}_G T \,\} \cup T \;\subseteq\; \mathcal{R}_G T$$

The following monotonicity properties are easy to check.

Proposition 1. *Let G, G_1 and G_2 be hypergraphs; let T, T_1 and T_2 be sets of facts.*

(a) If $T_1 \subseteq T_2$, then $\mathcal{R}_G T_1 \subseteq \mathcal{R}_G T_2$.
(b) If $G_1 \subseteq G_2$, then $\mathcal{R}_{G_1} T \subseteq \mathcal{R}_{G_2} T$.

Given a hypergraph G and a set T of facts, the ***induced sub-hypergraph*** $G[T]$ retains those arcs that mention facts from T:

$$G[T] := \{\, (h, B) \in G \mid h \in T \text{ and } B \subseteq T \,\}$$

4.2 Analyses

We use Datalog programs to implement static analyses that are parametric and monotone. Thus, the Datalog programs we consider have additional properties:

1. Because the Datalog program implements a static analysis, a subset of facts encode queries, corresponding to assertions in the program being analysed.
2. Because the static analysis is parametric, a subset of facts encode parameter settings.
3. Because the static analysis is monotone, parameter settings that are more expensive are also more precise.

For example, in Section 2, queries are facts from the relation `dirty`; parameter settings are encoded by relations `cheap` and `precise`; and switching a parameter from `cheap` to `precise` makes the analysis more expensive but cannot grow the relation `dirty`.

If we only assume that the analysis is parametric, monotone, and implemented in Datalog, then we can already make good predictions in some cases, such as the case of the analysis in Section 2. In other cases, we require more information about the relationship between what the analysis does when run in a precise mode and what the analysis does when run in an imprecise mode. We assume that this information comes in the form of a partial function that projects facts. The technical requirements on the projection function are mild, so the analysis designer has considerable leeway in choosing an appropriate projection. In some cases, the choice is straightforward. For example, if the analysis is k-object sensitive, meaning that it tracks calling contexts using sequences of allocation sites, then a good choice of projection corresponds to truncating these sequences.

An ***analysis*** $\mathcal{A}$ is a tuple (G, Q, P, p_0, p_1, π), where G is a hypergraph called the ***global provenance***, Q is a set of facts called ***queries***, P is a finite set of ***parameters***, the ***encoding functions*** p_0 and p_1 map parameters to facts, and π is a partial function from facts to facts called ***projection***. A ***parameter setting*** a of an analysis $\mathcal{A}$ is an assignment of booleans to the parameters P. We sometimes refer to parameter settings as ***abstractions***, for brevity. We encode the abstraction a as two sets of facts, $P_0(a)$ and $P_1(a)$, defined by

$$P_k(a) := \{\, p_k(x) \mid x \in P \text{ and } a(x) = k \,\} \quad \text{for } k \in \{0, 1\}$$

The set $\mathcal{A}(a)$ of facts ***derived*** by the analysis $\mathcal{A}$ under abstraction a is defined to be $\mathcal{R}_G\big(P_0(a) \cup P_1(a)\big)$. Abstractions form a complete lattice with respect to the pointwise order: $a \leq a'$ iff $a(x) \leq a'(x)$ for all $x \in P$. We write $\bot$ for the ***cheapest abstraction*** that assigns 0 to all parameters, and $\top$ for the ***most precise abstraction*** that assigns 1 to all parameters.

For an analysis $\mathcal{A}$, we sometimes consider the restriction of its hypergraph to those facts derived under a given abstraction a: $G^a := G[\mathcal{A}(a)]$. In particular, $G^\bot$ is called the ***cheap provenance***, and $G^\top$ is called the ***precise provenance***.

An analysis is ***well formed*** when it obeys further restrictions: (i) facts derived under the cheapest abstraction are fixed-points of the projection, $\pi(x) = x$ for $x \in \mathcal{A}(\bot)$, (ii) the image of the projection π is included in $\mathcal{A}(\bot)$, (iii) only fixed-points project on queries, $\pi^{-1}(q) \subseteq \{q\}$ for $q \in Q$, (iv) the encoding functions p_0 and p_1 are injective and have disjoint images, and (v) projection is compatible with parameter encoding, $\pi \circ p_1 = p_0$. From (i) and (ii) it follows that π is idempotent. These conditions are technical: they ease the treatment that follows, but do not restrict which analyses can be modelled.

An analysis $\mathcal{A}$ is said to be ***monotone*** when the set of derived queries decreases as a function of the abstraction: $a \leq a'$ implies $\big(Q \cap \mathcal{A}(a)\big) \supseteq \big(Q \cap \mathcal{A}(a')\big)$.

We can now formally define the main problem.

Problem 2. Given are a well formed, monotone analysis $\mathcal{A}$, and a query q for $\mathcal{A}$. Does there exist an abstraction a such that $q \notin \mathcal{A}(a)$?

Because the analysis is monotone, $q \in \mathcal{A}(a)$ for all a if and only if $q \in \mathcal{A}(\top)$. Thus, one way to solve the problem is to check if q is derived by $\mathcal{A}$ under the most precise abstraction $\top$. However, this is typically too expensive. Instead, we consider a class of solutions called ***monotone refinement algorithms***. A monotone refinement algorithm evaluates the analysis for a sequence $a_1 \leq \cdots \leq a_n$ of abstractions. Refinement algorithms terminate when one of two conditions holds: (i) $q \notin \mathcal{A}(a_n)$ or (ii) $q \in \mathcal{R}_{G^{a_n}}\big(P_1(a_n)\big)$. It is easy to see why $q \notin \mathcal{A}(a_n)$ implies that Problem 2 has answer 'yes'. It is less easy to see why $q \in \mathcal{R}_{G^{a_n}}\big(P_1(a_n)\big)$ implies that Problem 2 has answer 'no'. Intuitively, this second termination condition says that the query q is reachable even if we rely only on precise semantics. In other words, our abstract counterexample does not actually have any abstract step. Formally, we rely on the following lemma:

Lemma 3. *Let q be a query for a well formed, monotone analysis $\mathcal{A}$. If $q \in \mathcal{R}_{G^a}\big(P_1(a)\big)$ for some abstraction a, then $q \in \mathcal{A}(a')$ for all abstractions a'.*

Proof. By Proposition 1(a), $q \in \mathcal{R}_{G^a}\big(P_1(a)\big) = \mathcal{R}_G\big(P_1(a)\big) \subseteq \mathcal{R}_G\big(P_1(\top)\big) = \mathcal{A}(\top)$. We conclude by noting that the analysis is monotone. □

4.3 Predictability

The precise provenance $G^\top$ contains all the information necessary to answer Problem 2. Unfortunately, the precise provenance $G^\top$ is typically very large and hard to compute. In contrast, the cheap provenance $G^\bot$ is typically smaller and easier to compute. In fact, most refinement algorithms start with the cheapest abstraction, $a_1 = \bot$. Fortunately, we observed empirically that $G^\top$ and $G^\bot$ are compatible, in a way made precise next.

We begin by lifting the projection π to sets T of facts as follows:

$$\pi(T) := \{\, t' \mid t' = \pi(t) \text{ and } t \in T \,\}$$

In particular, if the partial function π is not defined for any $t \in T$, then $\pi(T) = \emptyset$. Our empirical observation is that

$$\pi \circ \mathcal{R}_{G^\top} \circ P_1 = \mathcal{R}_H \circ \pi \circ P_1 \qquad \text{for some } H \subseteq G^\perp \qquad (4)$$

An analysis $\mathcal{A}$ that obeys condition (4) is said to be ***predictable***. A hypergraph H that witnesses condition (4) is said to be a ***predictive provenance*** of analysis $\mathcal{A}$. For a predictable analysis, reachability and projection almost commute on the image of P_1, except that if projection is done first, then reachability must ignore some arcs.

The inspiration for condition (4) came from the notion of correct approximation, as used in abstract interpretation. But, it is not the same. We tested condition (4) on analyses that do not explicitly follow the abstract interpretation framework, and we were surprised that it holds. Then we designed the example analysis from Section 2 so that the reason why condition (4) holds is apparent: Datalog rules come in pairs, one encoding precise semantics, the other encoding approximate semantics. But, for real analyses, we could not discern any such simple reason. Thus, we consider our empirical finding as surprising and intriguing.

Recall that refinement algorithms use two termination conditions: $q \notin \mathcal{A}(a)$ and $q \in \mathcal{R}_{G^a}\big(P_1(a)\big)$. Predictive provenances help us evaluate the termination conditions of refinement algorithms.

Lemma 4. *Let $\mathcal{A}$ be a well formed, monotone analysis. Let a be an abstraction, and let H be a predictive provenance. Finally, let q be a query derived by $\mathcal{A}$ under the cheapest abstraction $\perp$.*

(a) If $q \notin \mathcal{A}(a)$, then $q \notin \mathcal{R}_{G^\perp}(P_0(a))$ and $q \notin \mathcal{R}_H(\pi(P_1(a)))$.
(b) Also, $q \in \mathcal{R}_{G^a}\big(P_1(a)\big)$ if and only if $q \in \mathcal{R}_H(\pi(P_1(a)))$.

Part (a) lets us approximate the termination condition $q \notin \mathcal{A}(a)$; part (b) lets us evaluate the termination condition $q \in \mathcal{R}_{G^a}\big(P_1(a)\big)$. In both cases, only small parts of the global provenance G are used, namely $G^\perp$ and H. The assumption $q \in \mathcal{A}(\perp)$ is reasonable: otherwise the refinement algorithm terminates after the first iteration.

Proof. Assume that $q \in \mathcal{R}_H(\pi(P_1(a)))$. We have

$$\mathcal{R}_H(\pi(P_1(a))) = \pi\big(\mathcal{R}_{G^\top}(P_1(a))\big) \qquad \text{by (4)}$$
$$q \in \pi\big(\mathcal{R}_{G^\top}(P_1(a))\big) \Rightarrow q \in \mathcal{R}_{G^\top}(P_1(a)) \qquad \text{by } \pi^{-1}(q) \subseteq \{q\}$$
$$\mathcal{R}_{G^\top}(P_1(a)) = \mathcal{R}_{G^a}(P_1(a)) \subseteq \mathcal{A}(a) \qquad \text{by Prop. } 1(a)$$

Putting these together, we conclude that $q \in \mathcal{A}(a)$. Using a very similar argument we can show that $q \in \mathcal{R}_{G^\perp}(P_0(a))$ implies $q \in \mathcal{A}(a)$. This concludes the proof of part (a).

The proof of part (b) is similar. □

Lemma 4 tells us that we could evaluate termination conditions more efficiently if we knew a predictive provenance. Alas, we do not know a predictive provenance.

4.4 Probabilities of Predictive Provenances

If we do not know a predictive provenance, then a naive way forward is as follows: enumerate each possible predictive provenance, see what it predicts, and take an average of the predictions. Our model is only marginally more complicated: it considers some possible predictive provenances as more likely than others. On the face of it, enumerating all possible predictive provenances takes us back to an inefficient algorithm. We will see later how to deal with this problem (Section 6). Now, let us define the probabilistic model formally.

The blueprint of the probabilistic model is given by a cheap provenance $G^\perp$. To each arc $e \in G^\perp$, we associate a boolean random variable $\mathbf{S}_e$, and call it the ***selection variable*** of e. Selection variables are independent but may have different expectations. We partition $G^\perp$ into ***types*** $G^\perp_1, \ldots, G^\perp_t$, and we do not require selection variables to have the same expectation unless they have the same type. Each type $G^\perp_k$ has an associated ***hyperparameter*** θ_k: if $e \in G^\perp_k$, then we say that e has type k, and we require that $\mathrm{E}\,\mathbf{S}_e = \theta_k$. Recall that $\mathrm{E}\,\mathbf{S}_e = \Pr(\mathbf{S}_e = 1)$. We define, in terms of the selection variables, a random variable $\mathbf{H}$ whose values are predictive provenances, by requiring that $\mathbf{S}_e = [e \in \mathbf{H}]$. Thus, the probability of a predictive provenance H is

$$\Pr(\mathbf{H} = H) = \prod_{k=1}^{t} \theta_k^{|G^\perp_k \cap H|}(1-\theta_k)^{|G^\perp_k \setminus H|} \qquad (5)$$

For example, if all arcs have the same type, then the model has only one hyperparameter θ, and $\Pr(\mathbf{H} = H)$ is $\theta^{|H|}(1-\theta)^{|G^\perp \setminus H|}$. At the other extreme, if all arcs have their own type, then the model has one hyperparameter θ_e for each arc $e \in G^\perp$, and $\Pr(\mathbf{H} = H)$ is $\prod_{e \in G^\perp} \theta_e^{[e \in H]}(1-\theta_e)^{[e \notin H]}$.

How many types should there be? Few types could lead to underfitting, many types could lead to overfitting. In the implementation, we have one type per Datalog rule. Intuitively, this means that we trust the judgement of whoever implemented the analysis.

4.5 Use of the Model

Before using the probabilistic model in a refinement algorithm, we must choose appropriate values for hyperparameters. This is done offline, in a learning phase (Section 5). After learning, each Datalog rule has an associated probability – its hyperparameter.

After the first invocation of the analysis we know the cheap provenance $G^\perp$, which we use as a blueprint for the probabilistic model. Then, our model predicts whether $q \in \mathcal{R}_{G^a}(P_1(a))$, where a is some abstraction not yet tried. Recall that $q \in \mathcal{R}_{G^a}(P_1(a))$ is one of the termination conditions. The hypergraph G^a is unknown, and thus we model it by a random variable $\mathbf{G}^a$. However, we do know from Lemma 4(b) that $q \in \mathcal{R}_{G^a}(P_1(a))$ if and only if $q \in \mathcal{R}_H(\pi(P_1(a)))$. Thus,

$$\begin{aligned}\Pr\big(q \in \mathcal{R}_{\mathbf{G}^a}(P_1(a))\big) &= \Pr\big(q \in \mathcal{R}_{\mathbf{H}}(\pi(P_1(a)))\big) \\ &= \sum_{\substack{R \\ q \in R}} \Pr\big(\mathcal{R}_{\mathbf{H}}(\pi(P_1(a))) = R\big)\end{aligned}$$

where R ranges over subsets of vertices of $G^\perp$. It remains to compute a probability of the form $\Pr\big(\mathcal{R}_{\mathbf{H}}T = R\big)$. Explicit expressions for such probabilities are also needed during learning, so they are discussed later (Section 5).

Intuitively, one could think that the refinement algorithm runs a simulation in which the static analyser is approximated by the probabilistic model. However, it would be inefficient to actually run a simulation, and we will have to use heuristics that have a similar effect (Section 6), namely to minimise the expected total runtime.

5. Learning

The probabilistic model (Section 4) lets us compute the probability that a given abstraction will provide a definite answer, and thus terminate the refinement. These probabilities are computed as a function of hyperparameters. The values of the hyperparameters, however, remain to be determined. To find good hyperparameters, we shall use a standard method from machine learning, namely MLE (maximum likelihood estimation).

MLE works as follows. First, we set up an experiment. The result of the experiment is that we observe an event O. Next, we compute the ***likelihood*** $\Pr(O)$ according to the model, which is a function of the hyperparameters. Finally, we pick for hyperparameters values that maximise the likelihood.

The standard challenge in deploying the MLE method is in the last phase: the likelihood is typically a complicated function of the hyperparameters. Often, to maximise the likelihood, analytic methods do not exist, and numeric methods could be unstable

or inefficient. This is indeed the case for our model: analytic methods do not apply, and many numeric methods are inefficient. But, we did find one numeric method that is both stable and efficient (Section 7.2). In addition to the standard challenge, our setting presents an additional difficulty. The expression of $\Pr(O)$ is exponentially large if the cheap provenance has cycles. We will handle this difficulty by finding bounds that approximate $\Pr(O)$.

5.1 Training Experiment

For the training experiment, we collect a set of programs. For the formal development, it is convenient to consider the set of programs as one larger program. We run the analysis on this large training program several times, each time under a different abstraction. The abstractions $a_1, \ldots, a_n$ are chosen randomly, with bias. In particular, they have to be cheap enough so that the analysis terminates in reasonable time. As a result of running the analysis, we observe the provenances $G^{a_1}, \ldots, G^{a_n}$. To connect these observed provenances to a probabilistic event, we shall use the predictability condition (4) together with the following simple fact.

Proposition 5. *Let G be a hypergraph, and let T_1 and T_2 be sets of facts. If $T_1 \subseteq T_2$, then $\mathcal{R}_G T_1 = \mathcal{R}_{G'} T_1$, where $G' = G[\mathcal{R}_G T_2]$.*

Corollary 6. *Let a be an abstraction for analysis $\mathcal{A}$. We have $\mathcal{R}_{G^\top}(P_1(a)) = \mathcal{R}_{G^a}(P_1(a))$.*

Given an efficient way to compute the projection π, we can compute the sets of facts $R_k := \pi\big(\mathcal{R}_{G^{a_k}}(P_1(a_k))\big)$, for each $k \in \{1, \ldots, n\}$. Using Corollary 6 and condition (4), we have that $R_k = \mathcal{R}_H(\pi(P_1(a_k)))$, for $k \in \{1, \ldots, n\}$. We define the following events:

$$\begin{aligned} O_k &:= \big(\mathcal{R}_{\mathbf{H}}(\pi(P_1(a_k))) = R_k\big) \qquad \text{for } k \in \{1, \ldots, n\} \\ O &:= \big(O_1 \cap \ldots \cap O_n\big) \end{aligned}$$

The event O is what we observe. It is completely described by the pairs (a_k, R_k). The abstraction a_k is sampled at random. The set R_k of facts is easily computed from G^{a_k}. The provenance G^{a_k} is obtained from the set of instantiated Datalog rules during the analysis under abstraction a_k, and it records all the reasoning steps of the analysis.

5.2 Bounds on Likelihood

There appears to be no formula that computes the likelihood $\Pr(O)$ and that is not exponentially large. However, there exist reasonably small formulas that provide lower and upper bounds. We shall use the lower bound for learning, and we shall use both bounds to evaluate the quality of the model.

One could define different bounds on likelihood. Our choice relies on the concept of forward arc, which leads to several desirable properties we will see later. Given a hypergraph G, we define the ***distance*** $d_T^{(G)}(h)$ from vertices T to vertex h by requiring $d_T^{(G)}$ to be the unique fixed-point of the following equations:

$$\begin{aligned} d_T^{(G)}(h) &= 0 && \text{if } h \in T \\ d_T^{(G)}(h) &= \infty && \text{if } h \notin \mathcal{R}_G T \\ d_T^{(G)}(h) &= \min_{e=(h,B)\in G} \max_{b\in B} (d_T^{(G)}(b) + 1) && \text{otherwise} \end{aligned}$$

We omit the superscript when the hypergraph is clear from context. A ***forward arc*** with respect to T is an arc $e = (h, B) \in G$ such that $d_T(h) > d_T(b)$ for every $b \in B$.

Theorem 7. *Consider the probabilistic model associated with the cheap provenance $G^\perp$ of some analysis $\mathcal{A}$. Let $T_1, \ldots, T_n$ and $R_1, \ldots, R_n$ be subsets of vertices of $G^\perp$. If $h \notin B$ for all arcs (h, B) in $G^\perp$ and $R_k \subseteq \mathcal{R}_{G^\perp} T_k$ for all k, then we have the following lower and upper bounds on $\Pr\big(\bigcap_{k=1}^n (\mathcal{R}_{\mathbf{H}} T_k = R_k)\big)$:*

$$\begin{aligned} &\prod_{e\in N} \mathrm{E}\,\bar{\mathbf{S}}_e \prod_{\substack{h \\ C_h \neq \emptyset}} \sum_{\substack{E_1 \\ E_1 \subseteq A_h \\ \forall k \in C_h,\ E_1 \cap F_k \neq \emptyset}} \prod_{e\in E_1} \mathrm{E}\,\mathbf{S}_e \prod_{e \in A_h \setminus E_1} \mathrm{E}\,\bar{\mathbf{S}}_e \\ \leq\ &\Pr\Big(\bigcap_{k=1}^n (\mathcal{R}_{\mathbf{H}} T_k = R_k)\Big) \\ \leq\ &\prod_{e\in N} \mathrm{E}\,\bar{\mathbf{S}}_e \prod_{\substack{h \\ C_h \neq \emptyset}} \sum_{\substack{E_1 \\ E_1 \subseteq A_h \\ \forall k \in C_h,\ E_1 \cap D_k \neq \emptyset}} \prod_{e\in E_1} \mathrm{E}\,\mathbf{S}_e \prod_{e \in A_h \setminus E_1} \mathrm{E}\,\bar{\mathbf{S}}_e \end{aligned}$$

where

$$\begin{aligned} N &:= \{\, (h', B') \in G^\perp \mid B' \subseteq R_{k'} \text{ and } h' \notin R_{k'} \text{ for some } k' \,\} \\ C_h &:= \{\, k' \mid h \in R_{k'} \setminus T_{k'} \,\} \qquad A_h := \{\, (h, B') \in G^\perp \,\} \setminus N \\ D_k &:= \{\, (h', B') \in G^\perp \mid B' \subseteq R_k \,\} \\ F_k &:= \{\, e = (h', B') \in D_k \mid e \text{ is a forward arc w.r.t. } T_k \,\} \end{aligned}$$

Intuitively, the arcs in N are those arcs that were observed to be not selected; thus, the factor $\prod_{e\in N} \mathrm{E}\,\bar{\mathbf{S}}_e$. For each reachable vertex, there is a factor that requires a justification, in terms of other reachable vertices and in terms of selected arcs. Let us consider a simple example, in which the lower and upper bounds coincide: there are four arcs $e_k = (h, \{b_k\})$ for $k \in \{1, 2, 3, 4\}$, and we observed $R_1 = \{b_1\}$, $R_2 = \{b_1, b_2, b_4, h\}$, and $R_3 = \{b_3, b_4, h\}$. In R_1, vertex h is not reachable but b_1 is, so $\mathbf{S}_{e_1}$ must not hold. In R_2, vertex h is reachable and could be justified by one of e_1, e_2, e_4, so $\mathbf{S}_{e_1} \vee \mathbf{S}_{e_2} \vee \mathbf{S}_{e_4}$ must hold. In R_3, vertex h is reachable and could be justified by one of e_3, e_4, so $\mathbf{S}_{e_3} \vee \mathbf{S}_{e_4}$ must hold. In all,

$$\begin{aligned} &\bar{\mathbf{S}}_{e_1} \wedge (\mathbf{S}_{e_1} \vee \mathbf{S}_{e_2} \vee \mathbf{S}_{e_4}) \wedge (\mathbf{S}_{e_3} \vee \mathbf{S}_{e_4}) \\ &\quad = \bar{\mathbf{S}}_{e_1} \wedge (\mathbf{S}_{e_2} \vee \mathbf{S}_{e_4}) \wedge (\mathbf{S}_{e_3} \vee \mathbf{S}_{e_4}) \end{aligned} \tag{6}$$

must hold. The expectation of this quantity is written in Theorem 7 as $\mathrm{E}\,\bar{\mathbf{S}}_{e_1}(\mathrm{E}\,\bar{\mathbf{S}}_{e_2}\,\mathrm{E}\,\bar{\mathbf{S}}_{e_3}\,\mathrm{E}\,\mathbf{S}_{e_4} + \cdots + \mathrm{E}\,\mathbf{S}_{e_2}\,\mathrm{E}\,\mathbf{S}_{e_3}\,\mathrm{E}\,\mathbf{S}_{e_4})$, where the inner sum enumerates the models of $(\mathbf{S}_{e_2} \wedge \mathbf{S}_{e_3}) \vee \mathbf{S}_{e_4}$.

The situation becomes more complicated when the hypergraph has cycles. In the presence of cycles, the recipe from the previous example does not compute the likelihood, but it does compute an upper bound. The reason is that it counts all cyclic justifications as if they were valid. Indeed, this is the upper bound given in Theorem 7. For the lower bound, we first eliminate cycles by dropping some arcs, thus lowering the likelihood; then, we apply the same recipe. Theorem 7 indicates that the arcs which should be dropped are the nonforward arcs. Why is this a good choice? One might think that we should drop a minimum number of arcs if we want a good lower bound. However, (1) it is NP-hard to find the minimum number of arcs [26, Feedback Arc Set], and (2) the set of such arcs is not uniquely determined. In contrast, we can find the set of nonforward arcs in polynomial time, and the solution is unique.

Another nice property of the set of forward arcs is that, if for each reachable vertex h we retain at least one forward arc whose head is h, then all reachable vertices remain reachable. This property is desirable for detecting impossibility (see Lemma 15). In terms of the lower bound, this property means that we never lower bound a positive probability by 0.

In the implementation, we sometimes heuristically drop forward arcs, in order to keep the size of the formula small. But, we only choose to drop a forward arc with head h if there are more than 8 forward arcs with head h. For example, if we drop arc e_2 in our running example, the effect is that we lower bound (6) by

$$\bar{\mathbf{S}}_{e_1} \wedge \mathbf{S}_{e_4} \wedge (\mathbf{S}_{e_3} \vee \mathbf{S}_{e_4})$$

We simply drop the corresponding variable $\mathbf{S}_{e_2}$ from the formula, thus making the formula smaller. Similarly, we can reduce the size of the formula for the upper bound, at the cost of weakening the bond. This time, we drop clauses rather than variables. For example, we can upper bound (6) by

$$\bar{\mathbf{S}}_{e_1} \wedge (\mathbf{S}_{e_2} \vee \mathbf{S}_{e_4})$$

For each vertex, our implementation drops all clauses except for the longest one.

Although the probabilistic model is simple, computing the likelihood of an event of the form '$\mathcal{R}_{\mathbf{H}}T_1 = R_1$ and ... and $\mathcal{R}_{\mathbf{H}}T_n = R_n$' is not computationally easy. The full version of this paper gives an exact formula that has size exponential in the number of vertices of the cheap provenance, but also points to evidence that a significantly smaller formula is unlikely to exist [31]. The size explosion is caused mainly by the cycles of the cheap provenance. Theorem 7 gives likelihood lower and upper bounds that are exponential only in the maximum in-degree of the cheap provenance. These formulas are still too large to be used in practice. However, there are simple heuristics that can be applied to reduce the size of the formulas, at the cost of weakening the bounds.

We use the lower bound to learn hyperparameters (Section 7.2). We use the upper bound to measure the quality of the learnt hyperparameters (Section 7.3).

5.3 Results

We learnt hyperparameters for a flow insensitive but object sensitive aliasing analysis. The aliasing analysis is implemented in 59 Datalog rules. All but 5 rules get a hyperparameter of 1. A rule with a hyperparameter of 1 is a rule that was not observed to be involved in any approximation, in the training set. For two of the remaining five rules, the learnt hyperparameters were essentially random, because the likelihood lower bound did not depend on them. The reason is that the training set did not contain enough data, or that the lower bound was too weak.

For the remaining three rules the hyperparameters were 0.997, 0.985, and 0.969. These values were robust, in the sense that they varied little when the training subset changed. For example, the rule with a hyperparameter of 0.969 is

$$\texttt{CVC}(c, u, o) \leftarrow \texttt{DVDV}(c, u, d, v), \texttt{CVC}(d, v, o), \texttt{VCfilter}(u, o)$$

Looking briefly at the aliasing analysis implementation we see that (a) $\texttt{CVC}(c, u, o)$ means 'in context c, variable u may point to object o', and (b) the relation DVDV is responsible for copying method arguments and returned values. We interpret this as evidence that the approximations done by the aliasing analysis are closely related to approximations of the call graph.

We are not the authors of the aliasing analysis; it is taken from Chord. Our learning algorithm automatically identified the three rules that are most interesting, from the point of view of approximation.

6. Refinement

The probabilistic model is interesting from a theoretical point of view (Section 4). The learning algorithm is already useful, because it lets us find which rules of a static analysis approximate the concrete semantics, and by how much (Section 5). In this section we explore another potential use of the learnt probabilistic model: to speed up the refinement of abstractions.

We consider a refinement algorithm that is applicable to analyses implemented in Datalog (Section 6.1). The key step of refinement is choosing the next abstraction to try. Abstractions that make good candidates share several desirable properties. In particular, they are likely to answer the posed query (Section 6.2), and they are likely to be cheap to try (Section 6.3). These two desiderata need to be balanced (also Section 6.3). Once we formalise how desirable an abstraction is, the next task is to search for the most desirable one (Section 6.4).

```
Given: A well formed, monotone analysis 𝒜, and a query q.
SOLVE
1  a := ⊥        // ⊥ as initial abstraction
2  repeat
3      G^a := G[𝒜(a)]        // invokes analysis
4      if q ∉ 𝒜(a) then return "yes"
5      if q ∈ ℛ_{G^a}(P_1(a)) then return "no"
6      a := CHOOSENEXTABSTRACTION(G^a, q, a)
```

Figure 4. The refinement algorithm used to solve Problem 2.

6.1 Refinement Algorithm

The refinement algorithm is straightforward (Figure 4). It repeatedly obtains the provenance G^a by running the analysis under abstraction a (line 3), checks if one of the two termination conditions holds (lines 4 and 5), and invokes CHOOSENEXTABSTRACTION to update the current abstraction (line 6). The correctness of this algorithm follows from the discussion in Section 4.2, and in particular Lemma 3.

Let a' be the result of CHOOSENEXTABSTRACTION(G^a, q, a). For termination, we require that a' is strictly more precise than a. This is sufficient because the lattice of abstractions is finite. The next abstraction to try should satisfy two further requirements:

1. The termination conditions are likely to hold for a'.
2. The estimated runtime of $\mathcal{A}$ under a' is small.

Next, we discuss these two requirements in turn. To some degree, we will make each of them more precise. But, we caution that from now on the discussion leaves the realm of hard theoretical guarantees, and enters the land of heuristic reasoning, where discussions about static program analysis are typically found.

6.2 Making Termination Likely

The key step of the refinement algorithm (Figure 4) is the procedure CHOOSENEXTABSTRACTION. The simplest implementation that would ensure correctness is the following: return a random element from the set of feasible abstractions $\{\, a' \mid a' > a \,\}$. Note that if a were the most precise abstraction then the procedure CHOOSENEXTABSTRACTION would not be called, so the feasible set from above is indeed guaranteed to be nonempty.

One idea to speed up refinement is to restrict the set of feasible solutions to those abstractions that are likely to provide a definite answer. Let A_{y} and A_{n} be the sets of abstractions that will lead the refinement algorithm to terminate on the next iteration with the answer 'yes' or, respectively, 'no':

$$A_{\mathrm{y}} := \{\, a' \mid a' > a \text{ and } q \notin \mathcal{A}(a') \,\}$$
$$A_{\mathrm{n}} := \{\, a' \mid a' > a \text{ and } q \in \mathcal{R}_{G^{a'}}(P_1(a')) \,\}$$

Of course, exactly one of the two sets A_{y} and A_{n} is nonempty, but we do not know which. More generally, we cannot evaluate these sets exactly without running the analysis. But, we can approximate them, because CHOOSENEXTABSTRACTION has access to G^a. For A_{y} we can compute an upper bound $A_{\mathrm{y}}^{\supseteq}$; for A_{n} we use a heuristic approximation $A_{\mathrm{n}}^{\approx}$.

$$A_{\mathrm{y}}^{\supseteq} := \{\, a' \mid a' > a \text{ and } q \notin \mathcal{R}_{G^a}(P_0(a')) \,\}$$
$$A_{\mathrm{n}}^{\approx} := \{\, a' \mid a' > a \text{ and } q \in \mathcal{R}_{H}(T(a, a')) \,\}$$

for some $H \subseteq G^a$, where

$$T(a, a') \;:=\; P_1(a) \cup \pi(P_1(a') \setminus P_1(a))$$

It is easy to see why $A_{\mathrm{y}}^{\supseteq} \supseteq A_{\mathrm{y}}$; it is less easy to see why $A_{\mathrm{n}}^{\approx} \approx A_{\mathrm{n}}$. Let us start with the easy part.

Lemma 8. *Let $A_{\mathrm{y}}^{\supseteq}$ and A_{y} be defined as above. Then $A_{\mathrm{y}}^{\supseteq} \supseteq A_{\mathrm{y}}$.*

Proof. Assume that $a' > a$, as in the definitions of $A_{\mathrm{y}}^{\supseteq}$ and A_{y}. Then $P_0(a') \subseteq P_0(a)$. By Proposition 5 and Proposition 1,

$$\mathcal{R}_{G^a}(P_0(a')) = \mathcal{R}_G(P_0(a')) = \mathcal{R}_{G^{a'}}(P_0(a')) \subseteq \mathcal{A}(a')$$

The claimed inclusion now follows. □

Let us now discuss the less obvious claim that $A_{\mathrm{n}}^{\approx} \approx A_{\mathrm{n}}$. One could wonder why we did not define $A_{\mathrm{n}}^{\approx}$ by

$$\{\, a' \mid a' > a \text{ and } q \in \mathcal{R}_H(\pi(P_1(a'))) \,\}$$

for some $H \subseteq G^{\perp}$. This definition is simpler and is also guaranteed to be equivalent to A_{n}, by the predictability condition (4). In the implementation, we use the more complicated definition of $A_{\mathrm{n}}^{\approx}$ for two reasons. First, we note that (4) implies $A_{\mathrm{n}}^{\approx} = A_{\mathrm{n}}$ if $a = \perp$. Thus, the claim that $A_{\mathrm{n}}^{\approx} = A_{\mathrm{n}}$ can be seen as a generalisation of (4). We did not use this generalisation of (4) in the more theoretical parts (Section 4 and Section 5) because it would complicate the presentation considerably. For example, instead of one projection π, we would have a family of projections that compose. In principle, however, it would be possible to take $A_{\mathrm{n}}^{\approx} = A_{\mathrm{n}}$ as an axiom, from the point of view of the theoretical development. Second, the more complicated definition of $A_{\mathrm{n}}^{\approx}$ exploits all the information available in G^a. The simpler version can also incorporate information from G^a by conditioning H to be compatible with G^a, via (4). However, this conditioning would only use the projected set of vertices of G^a, rather than its full structure.

Furthermore, the definition of $A_{\mathrm{n}}^{\approx}$ used in the implementation has the following intuitive explanation. The condition $A_{\mathrm{n}}^{\approx} \approx A_{\mathrm{n}}$ tells us that in order to predict $\mathcal{R}_{G^{a'}}(P_1(a'))$ by using G^a we should do the following: (i) split $P_1(a')$ into $P_1(a)$ and $P_1(a') \setminus P_1(a)$; (ii) use the facts $P_1(a)$ as they are, because they already appear in G^a; (iii) approximate the facts in $P_1(a') \setminus P_1(a)$ by their projections, because they do not appear in G^a; and (iv) define the predictive provenance H with respect to G^a, because it is the most precise provenance available so far.

We defined two possible restrictions of the feasible set, namely $A_{\mathrm{y}}^{\supseteq}$ and $A_{\mathrm{n}}^{\approx}$. The remaining question is now which one should we use, or whether we should use some combination of them such as $A_{\mathrm{y}}^{\supseteq} \cap A_{\mathrm{n}}^{\approx}$. The restriction to $A_{\mathrm{y}}^{\supseteq}$ could be called the optimistic strategy, because it hopes the answer will be 'yes'; the restriction to $A_{\mathrm{n}}^{\approx}$ could be called the pessimistic strategy, because it hopes the answer will be 'no'. The optimistic strategy has been used in previous work [54]. The pessimistic strategy is used in our implementation. We found that it leads to smaller runtime (Section 7.4). It would be interesting to explore combinations of the two strategies, as future work.

In the optimistic strategy, one needs to check whether $A_{\mathrm{y}}^{\supseteq} = \emptyset$. In this case, it must be that $A_{\mathrm{y}} = \emptyset$ and thus the answer is 'no'. In other words, the main loop of the refinement algorithm needs to be slightly modified to ensure correctness. In the pessimistic strategy, it is never the case that $A_{\mathrm{n}}^{\approx} = \emptyset$, and so the main loop of the refinement algorithm is correct as given in Figure 4. The pessimistic restriction $A_{\mathrm{n}}^{\approx}$ is nonempty because it always contains $\top$, by choosing $H = G^a$ (see Lemma 15).

The set $A_{\mathrm{n}}^{\approx}$ is defined in terms of an unknown predictive provenance H. Thus, we work in fact with the random variable

$$\mathbf{A}_{\mathrm{n}}^{\approx} := \{\, a' \mid a' > a \text{ and } q \in \mathcal{R}_{\mathbf{H}}(T(a, a')) \,\}$$

Cases	`all-one`	`fine`	`coarse`
95.0%	0	$(-0.22, -0.20)$	$(-0.73, -0.72)$
3.8%	$-\infty$	$(-15, -14)$	$(-33, -32)$
1.2%	$-\infty$	$-\infty$	$(-12, -11)$

Table 1. Bounds on the average log-likelihood, in base e.

defined in a probabilistic model with respect to G^a, instead of $G^{\perp}$. We wish to choose an abstraction a' that is likely in $\mathbf{A}_{\mathrm{n}}^{\approx}$. In other words, we want to maximise $\Pr(a' \in \mathbf{A}_{\mathrm{n}}^{\approx})$. There is no simple expression to compute this probability. For optimisation, we will use the following lower bound.

Lemma 9. *Let $\mathbf{A}_{\mathrm{n}}^{\approx}$ be defined as above, with respect to an analysis $\mathcal{A}$, an abstraction a, and a query q. Let a' be some abstraction such that $a' > a$. Let H be some subgraph of G^a such that $q \in \mathcal{R}_H(T(a, a'))$. Then*

$$\Pr(a' \in \mathbf{A}_{\mathrm{n}}^{\approx}) \;\geq\; \prod_{e \in H} \mathrm{E}\, \mathbf{S}_e$$

where $\mathbf{S}_e$ is the selection variable of arc e.

Before describing the search procedure (Section 6.4), we must see how to balance maximising the probability of termination with minimising the running cost.

6.3 Balancing Probabilities and Costs

We are looking for an abstraction that is likely to answer the query but, at the same time, is not too expensive. Most of the time, these two desiderata point in opposite directions: expensive abstractions are more likely to provide an answer. This raises the question of how to balance the two desiderata. We model the problem as follows.

Definition 10 (Action Scheduling Problem)**.** Suppose that we have a list of $m \geq 1$ actions, which can succeed or fail. The success probabilities of these actions are $p_1, \ldots, p_m \in (0, 1]$, and the costs for executing these actions are $c_1, \ldots, c_m > 0$. Find a permutation σ on $\{1, \ldots, m\}$ that minimises the cost $C(\sigma)$:

$$C(\sigma) = \sum_{k=1}^{m} q_k(\sigma) c_{\sigma(k)}, \qquad q_k(\sigma) = \prod_{j=1}^{k-1} \big(1 - p_{\sigma(j)}\big).$$

Intuitively, $C(\sigma)$ represents the average cost of running actions according to σ until we hit success.

In the setting of our algorithm, the m actions correspond to all the possible next abstractions $a'_1, \ldots, a'_m$. The p_i is $\Pr(a'_i \in \mathbf{A}_{\mathrm{n}}^{\approx})$, and c_i is the cost of running the analysis under abstraction a'_i. Hence, a solution to this action scheduling problem tells us how we should combine probability and cost, and select the next abstraction a'.

Lemma 11. *Consider an instance of the action scheduling problem (Definition 10). Assume the success probabilities of the actions are independent. A permutation σ has minimum cost $C(\sigma)$ if and only if $p_{\sigma(1)}/c_{\sigma(1)} \geq \cdots \geq p_{\sigma(m)}/c_{\sigma(m)}$.*

Corollary 12. *Under the conditions of Lemma 11, if the cost of permutation σ is minimum, then $\sigma(1) \in \arg\max_i p_i/c_i$.*

6.4 MaxSAT encoding

We saw a refinement algorithm (Section 6.1) whose key step chooses an abstraction to try next. Then we saw how to estimate whether an abstraction a' is a good choice (Section 6.2 and Section 6.3): it should have a high ratio between success probability and runtime cost. But, since the number of abstractions is exponential in the number of parameters, it is infeasible to enumerate all in the search for the best one. Instead of performing a naive exhaustive search, we encode the search problem as a MaxSAT problem.

Configuration		Solved queries		
Strategy	Optimiser	Ruled out	Impossible	Limit
optimistic	exact	6	48	365
optimistic	approximating	6	0	413
pessimistic	exact	20	82	317
pessimistic	approximating	20	82	317
probabilistic	exact	20	70	329
probabilistic	approximating	16	81	322

Table 2. Outcomes. All queries are assertions that seem to be violated when the cheapest abstraction is used. A ***ruled out*** query is an assertion that is shown not to be violated. An ***impossible*** query is an assertion that seems violated even if the most precise abstraction is used. The exact optimiser is MiFuMax [22]. The approximating optimiser is based on MCSIs [35].

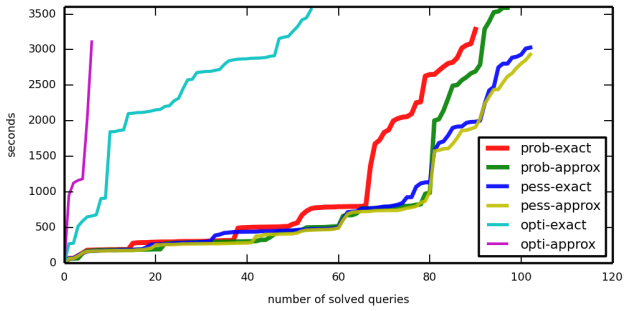

Figure 5. Runtime comparison.

Let us summarise the search problem. Given are a query q, an abstraction a and its local provenance G^a. We want to find an abstraction $a' > a$ that maximises the ratio $\Pr(a' \in \mathbf{A}_{\mathrm{n}}^{\approx})/c(a')$, where $c(a')$ is an estimate of the runtime of the analysis under abstraction a' (see Corollary 12). We will approximate $\Pr(a' \in \mathbf{A}_{\mathrm{n}}^{\approx})$ by a lower bound (see Lemma 9). Based on empirical observations, we estimate the runtime of the analysis to increase exponentially with the number $\sum_{x \in P} a(x)$ of precise parameters. In short, we want to evaluate the following expression:

$$\arg\max_{\substack{a' \\ a' > a}} \left(\left(\max_{\substack{H \\ H \subseteq G^a \\ q \in \mathcal{R}_H(T(a,a'))}} \prod_{e \in H} \mathrm{E}\,\mathbf{S}_e \right) \Big/ \exp\left(\alpha \sum_{x \in P} a'(x) \right) \right)$$

Or, after absorbing $\max$ in $\arg\max$, taking the log of the resulting objective value, and simplifying the outcome:

$$\arg\max_{\substack{a', H \\ a' > a,\ H \subseteq G^a \\ q \in \mathcal{R}_H(T(a,a'))}} \left(\sum_{e \in H} \log(\mathrm{E}\,\mathbf{S}_e) - \sum_{\substack{x \in P \\ a'(x) = 1}} \alpha \right) \tag{7}$$

We shall evaluate this expression by using a MaxSAT solver. The idea is to encode the range of $\arg\max$ as hard constraints, and the objective value as soft constraints.

There exist several distinct versions of the MaxSAT problem. We define here a version that is most convenient to our development. We consider arbitrary boolean formulas, not necessarily in some normal form. We view assignments as sets of variables; in particular,

$$\begin{array}{lll} M \models x & \text{iff} & x \in M \\ M \models \bar{x} & \text{iff} & x \notin M \\ M \models \phi_1 \wedge \phi_2 & \text{iff} & M \models \phi_1 \text{ and } M \models \phi_2 \end{array}$$

The evaluation rules for other boolean connectives are as expected. If $M \models \phi$ holds, we say that the assignment M is a ***model*** of formula ϕ.

Problem 13 (MaxSAT)**.** Given are a boolean formula Φ and a weight $w(x)$ for each variable x that occurs in Φ. Find a model M of Φ that maximises $\sum_{x \in M} w(x)$.

We refer to Φ as the ***hard constraint***.

Remark 14. Technically, Problem 13 is none of the standard variations of MaxSAT. It is easy to see, although we do not prove it here, that Problem 13 is polynomial-time equivalent to partial weighted MaxSAT [3, 37]: the reduction in one direction uses the Tseytin transformation, while the reduction in the other direction introduces relaxation variables.

The idea of the encoding is to define the hard constraint Φ such that (i) the models of Φ are in one-to-one correspondence with the possible choices of H and T such that $H \subseteq G^a$ and $P_0(a) \subseteq T \subseteq P_0(a) \cup P_1(a)$, and moreover (ii) each model also encodes the reachable set $\mathcal{R}_H T$. To construct a hard constraint Φ with these properties, we use the same technique as we used for computing the likelihood (Section 5.2). As was the case for likelihood, cycles lead to an exponential explosion. We again deal with cycles by retaining only forward arcs:

$$G^a_{\rightarrow} \;:=\; \{\, e \in G^a \mid e \text{ is a forward arc w.r.t. } P_0(a) \cup P_1(a) \,\}$$

The hard constraint is a formula whose variables correspond to vertices and arcs of $G^a_{\rightarrow}$. More precisely, its set of variables is $X_V(G^a_{\rightarrow}) \cup X_E(G^a_{\rightarrow})$, where

$$X_V(G) := \{x_u \mid u \text{ vertex of } G\} \quad X_E(G) := \{x_e \mid e \text{ arc of } G\}$$

We construct the hard constraint Φ as follows:

$$\begin{aligned} \Phi \;&:=\; \exists_{e \in G^a_{\rightarrow}} y_e \left(\Phi_1 \wedge \Phi_2 \wedge \Phi_3 \right) \\ \Phi_1 \;&:=\; \bigwedge_{e=(h,B) \in G^a_{\rightarrow}} \left(\left(y_e \leftrightarrow \left(x_e \wedge \bigwedge_{b \in B} x_b \right) \right) \wedge \left(y_e \rightarrow x_h \right) \right) \\ \Phi_2 \;&:=\; \bigwedge_{\substack{h \\ \text{vertex of } G^a_{\rightarrow} \\ h \notin P_0(a) \cup P_1(a)}} \left(x_h \rightarrow \left(\bigvee_{e=(h,B) \in G^a_{\rightarrow}} y_e \right) \right) \\ \Phi_3 \;&:=\; x_q \wedge \left(\bigwedge_{u \in P_0(a)} x_u \right) \wedge \left(\bigvee_{u \in P_1(a)} x_u \right) \end{aligned} \tag{8}$$

The notation $\exists_{e \in G^a_{\rightarrow}} y_e$ stands for several existential quantifiers, one for each variable in the set $\{\, y_e \mid e \in G^a_{\rightarrow} \,\}$. Intuitively, the constraints Φ_1 and Φ_2 ensure that the models correspond to reachable sets, and the constraint Φ_3 ensures that the query is reachable and that $a' > a$.

The formula Φ defined above has several desirable properties: its size is linear in the size of the local provenance G^a, it is satisfiable, and each of its models represents a pair (a', H) that satisfies the range conditions of (7). The satisfiability of Φ is important for the correctness of the refinement algorithm, and it follows from how we remove cycles, by retaining forward arcs. To state these properties more precisely, let us denote the range of (7) by $F(G^a)$ where

$$F(G) := \{\, (a', H) \mid a' > a \text{ and } H \subseteq G \text{ and } q \in \mathcal{R}_H(T(a, a')) \,\} \tag{9}$$

Lemma 15. *Let a be an abstraction, and let q be a query, for some analysis $\mathcal{A}$. Let $F(G)$ and $G^a_{\rightarrow}$ be defined as above. If $a < \top$ and $q \in \mathcal{A}(a)$, then $(\top, G^a_{\rightarrow}) \in F(G^a_{\rightarrow}) \subseteq F(G^a)$.*

The conditions $a < \top$ and $q \in \mathcal{A}(a)$ are guaranteed to hold when ChooseNextAbstraction is called on line 6 of Figure 4.

Lemma 16. *Let a be an abstraction, and let q be a query, for some analysis $\mathcal{A}$. Let the hard constraint Φ be defined as in* (8)*: let the feasible set $F(G^a_{\to})$ be defined as in* (9)*. There is a bijection between the models M of Φ and the elements (a', H) of $F(G^a_{\to})$. According to this bijection,*

$$M \cap X_E(G^a_{\to}) = X_E(H)$$
$$M \cap X_V(G^a_{\to}) = X_V\big(\mathcal{R}_H(T(a, a'))\big)$$

The proof of this lemma, given in the submitted supplement, relies on techniques very similar to those used to prove Theorem 7.

At this point, we know how to define the hard constraint Φ, so that its models form a subrange of the range of (7). It remains to encode the value $\sum_{e \in H} \log(\mathrm{E}\,\mathbf{S}_e) - \alpha \sum_{x \in P} a'(x)$ by assigning weights to variables. This is very easy. Each arc variable x_e is assigned the weight $w(x_e) = \log(\mathrm{E}\,\mathbf{S}_e)$. Each vertex variable x_u corresponding to $u \in P_0(a) \cup P_1(a)$ is assigned the weight $w(x_u) = -\alpha$. All other variables are assigned the weight 0.

7. Empirical Evaluation

In the empirical evaluation[1] we aim to answer three questions: (a) Which optimisation algorithm should be used for learning (Section 7.2)? (b) How well does the probabilistic model predict what the analysis does (Section 7.3)? (c) What is the effect of the new refinement algorithm on the total runtime (Section 7.4)?

7.1 Experimental Design

For experiments, our goal was to improve upon the refinement algorithm of Zhang et al. [54]. Accordingly, we use the same test suite and the same aliasing analysis. The test suite consists of 8 Java programs, which amount to 0.45 MiB of application bytecode plus 1 MiB of library bytecode.

We try three refinement strategies: optimistic, pessimistic, and probabilistic. The optimistic strategy uses the baseline refinement algorithm. The pessimistic strategy uses our refinement algorithm with all hyperparameters set to 1. The probabilistic strategy uses our refinement algorithm with hyperparameters learnt. We use a time limit of 60 minutes per query, and a memory limit of 25 GiB.

For learning, we observe what the analysis does on a small set of queries and abstractions. Each observation is essentially an event of the form '$\mathcal{R}_{\mathbf{H}}T_1 = R_1$ and ... and $\mathcal{R}_{\mathbf{H}}T_n = R_n$' (Section 5.1). From these observations we learn hyperparameters, by optimising a lower bound on the likelihood (Section 5.2). The hyperparameters we use to solve a query are learnt only from observations made on the other programs.

7.2 Numeric Optimisation of Likelihood

First, from the 8 programs, we chose a random sample of 26 queries. Then, for each query, we chose a random sample of 10 abstractions (Section 5.1). In total, the training set has 260 samples.

We first tried three numerical optimisers from the SciPy toolkit [23]: `tnc`, `slsqp`, and `basinhopping`. They all fail. Then we implemented a couple of numeric optimisers ourselves. We found that the cyclic coordinate ascent method works well on our problem. In the implementation, we use `basinhopping` and `slsqp` as subroutines, for line search.

Intuitively, cyclic coordinate ascent behaves well because the likelihood tends to be concave along a coordinate, and tends to not be concave along an arbitrary direction. Concave functions are much easier to optimise than non-concave functions, and so the line search algorithm has an easier task when applied along coordinates.

[1] `http://rgrig.appspot.com/static/papers/popl2016experiments.html`

7.3 Predictive Power of the Probabilistic Model

In addition to the 260 samples used for training, we obtain, using the same method, another set of 260 samples used for evaluation. Given a model, which is determined by an assignment of values to hyperparameters, we can evaluate likelihood bounds for each of the 260 evaluation samples. In absolute terms, these numbers are hard to interpret: are they good or bad? To make the numbers more meaningful, we consider three models, and we see how good they are relative to each other.

The three models are: `fine`, `coarse`, and `all-one`. The `fine` model is learnt as described above. The `coarse` model is also learnt as described above, but under the constraint that all hyperparameters have the same value. The `all-one` model simply assigns value 1 to all hyperparameters, and thus corresponds to the pessimistic refinement strategy.

Table 1 presents the results of the three models on the evaluation set. For the aliasing analysis we consider, it turns out that an abstraction chosen at random does no better than the cheapest abstraction in 95% of cases. The `all-one` model predicts that all abstractions do no better than the cheapest one, so it is exactly right in these 95% of cases; conversely, it thinks the other 5% of cases cannot happen. More interestingly, the `fine` model thinks that 1.2% samples from the evaluation set cannot happen. This means that some hyperparameter is 1 but should be < 1. We expect that the number of such situations would decrease as the training set grows. Assuming this is true, we can conclude that the `fine` model is better than the `coarse` model.

It is not possible to conclude which of `all-one` and `fine` is better. One difficulty is that the 95% is a property of the analysis. It might very well be that for another analysis this percent (of cases in which precision helps) is higher or lower. A lower percentage would favour the `fine` model; a high percentage favours the `all-one` model.

7.4 Total Analysis Runtime

In the 8 programs there are in total 1450 queries. We report results for a random sample of 419 queries. The first thing to notice in Table 2 is that most queries are not solved. This is in stark contrast with Zhang et al. [54] where all queries are reported as solved. The difference is explained by several differences between their setup and ours. (1) In addition to their PolySite queries, we also include Downcast queries. The latter are more difficult. (2) We used less space and time: they used a machine with 128 GiB of memory, whereas we only had 25 GiB available; they did not have an explicit time limit, whereas we used 1 hour as our time limit. (3) One of our modifications to the code (unfortunate, with hindsight), was that we loaded in memory the results of the Datalog analysis, which further increased our memory use. (4) They solve multiple queries at once, whereas we solve one at a time. By solving one query at a time, we can make a more fine grained comparison.

These differences notwithstanding, we stress that the results reported here are for running different algorithms under conditions that are as similar as possible. For example, as much as possible of the implementation is shared.

From the number of solved queries (Table 2), we see that the refinement strategies, from best to worst, are: pessimistic, probabilistic, optimistic. The pessimistic strategy solves the same set of 102 queries regardless of the optimiser it uses. The probabilistic strategy solves 101 queries in total, if we take the union over the two optimisers. There is exactly one query solved by the pessimistic strategy but not by the probabilistic one. The pessimistic strategy solves this query in four iterations, whereas the probabilistic strategy dies in the second iteration. The exact optimiser times out. The approximate optimiser increases the precision more than necessary after the first iteration, the Datalog solver does cope with the

increased precision, but an out of memory error happens while Datalog's answer is loaded in memory.

Figure 5 compares the six configurations from the point of view of runtime. We see that both the pessimistic and the probabilistic strategies are better than the optimistic strategy.

7.5 Discussion

According to Table 2 and Figure 5, setting all hyperparameters to 1 works better than using learnt hyperparameters. Given this, is there any point in learning hyperparameters? We believe the answer is yes. Initially we tried only an exact MAXSAT solver[2]. When the pessimistic strategy succeeds but the probabilistic strategy fails, the cause is always that the MAXSAT solver times out. Our encoding in MAXSAT is already an approximation, so an approximate answer would do. We conjectured that replacing the exact solver with an approximate one would improve performance. We are not aware of an off-the-shelf approximate MAXSAT solver, so we implemented one. Comparing `prob-exact` with `prob-approx`, we see that using an approximate solver does improve the results, but not enough. However, our approximate solver is so dumb that we feel it ought to be possible to do much better.

Another reason to learn hyperparameters is independent of their use for refinement: learnt hyperparameters identify interesting parts of an analysis implemented in Datalog (Section 5.3). This is especially useful when one wants to understand an analysis implemented by a third party.

Finally, we note that our empirical evaluation of refinement strategies shows promise but is not comprehensive. In future work, we intend to try better approximate MAXSAT solvers, and we intend to evaluate refinement algorithms on more analyses implemented in Datalog. But, first, we need better approximate MAXSAT solvers, and we need more analyses implemented in Datalog.

8. Related and Future Work

The potential of using machine learning techniques or probabilistic reasoning for addressing challenges in static analysis [4, 10] has been explored by several researchers in the past ten years. Three dominant directions so far are: to infer program specifications automatically using probabilistic models or other inductive learning techniques [5, 27, 32, 36, 42, 43, 45], to guess candidate program invariants from test data or program traces using generalisation techniques from machine learning [33, 40, 47], and to predict properties of potential or real program errors, such as true positiveness and cause, probabilistically [29, 30, 53, 56]. Our work brings a new dimension to this line of research by suggesting the use of a probabilistic model for predicting the effectiveness of program abstractions: a probabilistic model can be designed for predicting how well a parametric static analysis would perform for a given verification task when it is given a particular abstraction, and this model can help the analysis to select a good program abstraction for the task in the context of abstraction refinement. Another important message of our work is that the derivations computed during each analysis run include a large amount of useful information, and exploiting this information could lead to more beneficial interaction between probabilistic reasoning and static analysis.

Machine learning techniques have been used before to speed up abstraction refinement [9, 18], but in the setting of bounded model checking of hardware.

Several probabilistic models for program source code have been proposed in the past [1, 2, 21, 25, 34, 42, 43], and used for extracting natural coding conventions [1], helping the correct use of library functions [43], translating programs between different languages [25], and cleaning program source code and inferring likely properties [42]. These models are different from ours in that they are not designed to predict the behaviours of program analyses under different program abstractions, the main task of our probabilistic models.

Our probabilistic models are examples of first-order probabilistic logic programs studied in the work on statistical relational learning [12, 13, 19, 46]. In our case, models are large, and training data provides only partial information about the random variable $\mathbf{H}$ used in the models. To overcome this difficulty, we designed an algorithm tailored to our needs, which is based on the idea of variational inference [24, 51]. More precisely, we optimised a lower bound on the likelihood.

Our work builds on a large amount of research for automatically finding good program abstraction, such as CEGAR [4, 7–9, 20, 44], parametric static analysis with parameter search algorithms [28, 39, 54, 55], and static analysis based on Datalog or Horn solvers [6, 16, 17, 48, 52]. The novelty of our work lies in the use of adding a bias in this abstraction search using a probabilistic model, which predicts the behaviour of the static analysis under different abstractions.

One future direction would be to find new applications for our probabilistic techniques. For example, one could try to use our techniques in order to improve other, non-probabilistic approaches to estimating the impact of abstractions [41, 49]. Another future direction would be to better characterise the theoretical properties of our refinement algorithm. For example, if applied in the setting of abstract interpretation, how does it interact with the notion of completeness [14, 15]?

9. Conclusion

We have presented a new approach to abstraction refinement, one that receives guidance from a learnt probabilistic model. The model is designed to predict how well would the static analysis perform for a given verification task under different parameter settings. The model is fully derived from the specification of the analysis, and does not require manually crafted features. Instead, our model's prediction is based on all the reasoning steps performed by the analysis in a failed run. To make these predictions, the model needs to know how much approximation is involved in each Datalog rule that implements the static analysis. We have shown how to quantify the approximation, by using a learning algorithm that observes the analysis running on a large codebase. Finally, we have shown how to combine the predictions of the model with a cost measure in order to choose an optimal next abstraction to try during refinement. Our empirical evaluation with an object-sensitive pointer analysis shows that our approach is promising.

Acknowledgments

We thank Mikoláš Janota and Xin Zhang for suggesting that we invert Datalog implications. We thank Mayur Naik for giving us access to the private parts of Chord [38]. We thank Yongsu Park for giving us access to a server on which we ran preliminary experiments. We thank reviewers for their suggestions on how to improve the paper. This work was supported by EPSRC Programme Grant Resource Reasoning (EP/H008373/2). Yang was also supported by EPSRC and Institute for Information & communications Technology Promotion (IITP) grant funded by the Korea government (MSIP) (No. R0190-15-2011, Development of Vulnerability Discovery Technologies for IoT Software Security).

References

[1] Miltiadis Allamanis, Earl T. Barr, Christian Bird, and Charles A. Sutton. Learning natural coding conventions. In *FSE*, 2014.

[2] also, at submission time, we had not tried setting all hyperparameters to 1

[2] Miltiadis Allamanis, Daniel Tarlow, Andrew D. Gordon, and Yi Wei. Bimodal modelling of source code and natural language. In *ICML*, 2015.

[3] Carlos Ansótegui, Maria Luisa Bonet, and Jordi Levy. SAT-based MaxSAT algorithms. *Artif. Intell.*, 196:77–105, 2013.

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

[5] Nels E. Beckman and Aditya V. Nori. Probabilistic, modular and scalable inference of typestate specifications. In *PLDI*, 2011.

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

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

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

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

[10] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, 1977.

[11] Paul Erdős and Alfréd Rényi. On the evolution of random graphs. *Publ. Math. Inst. Hung. Acad. Sci*, 1960.

[12] Daan Fierens, Guy Van den Broeck, Joris Renkens, Dimitar Sht. Shterionov, Bernd Gutmann, Ingo Thon, Gerda Janssens, and Luc De Raedt. Inference and learning in probabilistic logic programs using weighted boolean formulas. *To appear in Theory and Practice of Logic Programming*, 2013.

[13] Lise Getoor and Ben Taskar, editors. *Introduction to Statistical Relational Learning*. The MIT Press, 2007.

[14] Roberto Giacobazzi, Francesco Logozzo, and Francesco Ranzato. Analyzing program analyses. In *POPL*, 2015.

[15] Roberto Giacobazzi, Francesco Ranzato, and Francesca Scozzari. Making abstract interpretations complete. *J. ACM*, 2000.

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

[17] Sergey Grebenshchikov, Nuno P. Lopes, Corneliu Popeea, and Andrey Rybalchenko. Synthesizing software verifiers from proof rules. In *PLDI*, 2012.

[18] Anubhav Gupta. *Learning Abstractions for Model Checking*. PhD thesis, Carnegie Mellon, 2006.

[19] Bernd Gutmann, Ingo Thon, and Luc De Raedt. Learning the parameters of probabilistic logic programs from interpretations. In *ECML PKDD*, 2011.

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

[21] Abram Hindle, Earl T. Barr, Zhendong Su, Mark Gabel, and Premkumar T. Devanbu. On the naturalness of software. In *ICSE*, 2012.

[22] Mikoláš Janota. MiFuMax — a literate MaxSat solver. `http://sat.inesc-id.pt/~mikolas/sw/mifumax/`, 2013.

[23] Eric Jones, Travis Oliphant, Pearu Peterson, et al. SciPy: Open source scientific tools for Python, 2001–. [Online; accessed 2015-07-06].

[24] Michael I. Jordan, Zoubin Ghahramani, Tommi S. Jaakkola, and Lawrence K. Saul. An introduction to variational methods for graphical models. *Machine Learning*, 1999.

[25] Svetoslav Karaivanov, Veselin Raychev, and Martin T. Vechev. Phrase-based statistical translation of programming languages. In *Onward!*, 2014.

[26] Richard M. Karp. Reducibility among combinatorial problems. In *Complexity of Computer Computations*, 1972.

[27] Ted Kremenek, Andrew Y. Ng, and Dawson R. Engler. A factor graph model for software bug finding. In *IJCAI*, 2007.

[28] Percy Liang and Mayur Naik. Scaling abstraction refinement via pruning. In *PLDI*, 2011.

[29] Ben Liblit, Alexander Aiken, Alice X. Zheng, and Michael I. Jordan. Bug isolation via remote program sampling. In *PLDI*, 2003.

[30] Ben Liblit, Mayur Naik, Alice X. Zheng, Alexander Aiken, and Michael I. Jordan. Scalable statistical bug isolation. In *PLDI*, 2005.

[31] Vladimir Lifschitz and Alexander A. Razborov. Why are there so many loop formulas? *ACM Trans. Comput. Log.*, 7(2):261–268, 2006.

[32] V. Benjamin Livshits, Aditya V. Nori, Sriram K. Rajamani, and Anindya Banerjee. Merlin: specification inference for explicit information flow problems. In *PLDI*, 2009.

[33] Alexey Loginov, Thomas W. Reps, and Shmuel Sagiv. Abstraction refinement via inductive learning. In *CAV*, 2005.

[34] Chris J. Maddison and Daniel Tarlow. Structured generative models of natural source code. In *ICML*, 2014.

[35] João Marques-Silva, Federico Heras, Mikolás Janota, Alessandro Previti, and Anton Belov. On computing minimal correction subsets. In *IJCAI*, 2013.

[36] Alon Mishne, Sharon Shoham, and Eran Yahav. Typestate-based semantic code search over partial programs. In *OOPSLA*, 2012.

[37] António Morgado, Federico Heras, Mark H. Liffiton, Jordi Planes, and João Marques-Silva. Iterative and core-guided MaxSAT solving: A survey and assessment. *Constraints*, 2013.

[38] Mayur Naik. jChord. `https://bitbucket.org/pag-lab/jchord`.

[39] Mayur Naik, Hongseok Yang, Ghila Castelnuovo, and Mooly Sagiv. Abstractions from tests. In *POPL*, 2012.

[40] Aditya V. Nori and Rahul Sharma. Termination proofs from tests. In *FSE*, 2013.

[41] Hakjoo Oh, Wonchan Lee, Kihong Heo, Hongseok Yang, and Kwangkeun Yi. Selective context-sensitivity guided by impact pre-analysis. In *PLDI*, 2014.

[42] Veselin Raychev, Martin T. Vechev, and Andreas Krause. Predicting program properties from "Big Code". In *POPL*, 2015.

[43] Veselin Raychev, Martin T. Vechev, and Eran Yahav. Code completion with statistical language models. In *PLDI*, 2014.

[44] Andrey Rybalchenko and Viorica Sofronie-Stokkermans. Constraint solving for interpolation. In *VMCAI*, 2007.

[45] Sriram Sankaranarayanan, Franjo Ivancic, and Aarti Gupta. Mining library specifications using inductive logic programming. In *ICSE*, 2008.

[46] Taisuke Sato. A statistical learning method for logic programs with distribution semantics. In *ICLP*, 1995.

[47] Rahul Sharma, Aditya V. Nori, and Alex Aiken. Interpolants as classifiers. In *CAV*, 2012.

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

[49] Yannis Smaragdakis, George Kastrinis, and George Balatsouras. Introspective analysis: context-sensitivity, across the board. In *PLDI*, 2014.

[50] Tachio Terauchi. Explaining the effectiveness of small refinement heuristics in program verification with CEGAR. In *SAS*, 2015.

[51] Martin J. Wainwright and Michael I. Jordan. Graphical models, exponential families, and variational inference. *Foundations and Trends in Machine Learning*, 2008.

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

[53] Kwangkeun Yi, Hosik Choi, Jaehwang Kim, and Yongdai Kim. An empirical study on classification methods for alarms from a bug-finding static C analyzer. *Inf. Process. Lett.*, 2007.

[54] X. Zhang, R. Mangal, R. Grigore, M. Naik, and H. Yang. On abstraction refinement for program analyses in Datalog. In *PLDI*, 2014.

[55] Xin Zhang, Mayur Naik, and Hongseok Yang. Finding optimum abstractions in parametric dataflow analysis. In *PLDI*, 2013.

[56] Alice X. Zheng, Michael I. Jordan, Ben Liblit, Mayur Naik, and Alex Aiken. Statistical debugging: simultaneous identification of multiple bugs. In *ICML*, 2006.