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, by using guidance from a probabilistic model, which itself is tuned by observing prior runs of the analysis. 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 implemented our approach on an object-sensitive pointer analysis for Java programs with two client analyses (PolySite and Downcast). Experiments show the benefits of our approach on reducing the runtime of the analysis.

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, 10, 18], 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 attempts to avoid these shortcomings by automatically learning an effective heuristic for finding a good abstraction, given a static analysis and a codebase with typical programs.

In this paper we present our abstraction refinement algorithm and its companion probabilistic model. Our algorithm applies to any parametric static analysis implemented in Datalog, provided that its precision increases when the values of the parameters increase. Such analyses are typically run in a loop that iteratively refines the parameter setting. Our idea is to equip such an analysis with a probabilistic model that can predict, for every untried parameter setting, what would happen if the analysis were run with the setting. The model makes this prediction using information found by the analysis in the failed iterative process so far, and guides the analysis when it chooses a next parameter setting. The probabilistic model itself is parametrised. To distinguish the parameters of the analysis from those of the probabilistic model, we call the latter hyperparameters. Good values for the hyperparameters are learnt by observing runs of the analysis on an existing codebase, and are later used when new unseen programs are analysed.

In other approaches to program analysis that are based on learning [38, 50], 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, features are not required: the model is derived fully from the specification of the corresponding analysis.

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 [13]: 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 so called 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 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

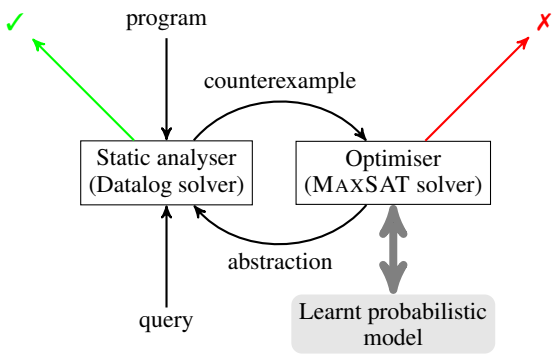

Figure 1. Architecture

```
    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

the parameter setting. We prove that our method of integrating these two metrics is optimal, under reasonable assumptions. We have implemented our algorithm and probabilistic model on an object-sensitive pointer analysis for Java programs. Experiments show that our algorithm substantially reduces the runtime of the analysis.

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). Section 7 positions our work in the various attempts to combine probabilistic reasoning and static analyses, and Section 8 concludes the paper.

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). 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. When it receives a counterexample from the analysis (that is, a collection of instantiated Datalog rules), it first checks whether using a more precise parameter settings could remove this counterexample. If not, the refinement part reports impossibility (or irrefutability) and stops [46, 51, 52]. Otherwise, it moves on to its main task of choosing a next parameter setting. The refinement part consults our probabilistic model, which uses the counterexample and predicts, for each untried parameter setting, the probability that the analysis under that setting ends the refinement loop. Based on the outcome of this consultation, the refinement part formulates an optimisation goal that balances these probabilities and the estimated costs of running the analysis under untried parameter settings. The resulting optimisation problem is then solved by a weighted MAXSAT solver, and its solution becomes the next parameter setting that the analysis tries.

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 very 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

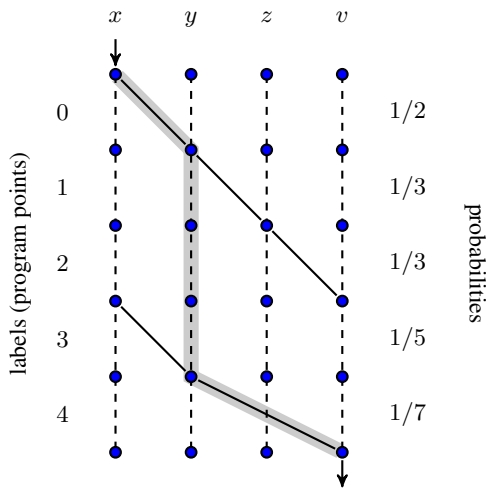

Figure 3. Abstract view of the program in Figure 2. Each label on the left identifies a smudge command. The dashed, vertical lines signify that once an object is dirty it remains dirty. The solid, oblique lines signify that smudge commands might propagate dirtiness. Depending on the values of the objects, 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.

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, we do expect that a number is a multiple of K with probability of about $1/K$.

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. This complicates the algorithm that learns probabilities, but we have found an efficient solution.

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 uses the pessimistic strategy described above but also takes 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), the analysis must be parametric (for instance, it may have parameters for controlling the degree of context sensitivity) (Section 4.2), and the results obtained with precise parameter settings are compatible with the results obtained with cheap parameter settings (Section 4.3). Given these assumptions, the probabilistic model assigns a probability to a subset of the arcs of a directed hypergraph (Section 4.4).

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. In particular, increasing the value of a parameter will not cause an assertion to change from pass to fail.

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, however, 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 very 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 a k-object sensitive aliasing analysis and 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) for each query q, only q itself can be projected on q (i.e., $\pi^{-1}(\{q\}) \subseteq \{q\}$), (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 $\big(Q \cap \mathcal{A}(a)\big) \supseteq \big(Q \cap \mathcal{A}(a')\big)$.

In practice, all analyses are well formed and many are monotone. In what follows, all analyses are assumed to be both well formed and monotone.

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. Based on empirical observations, we estimate the total running time of such an algorithm to be $c(a_1) + \cdots + c(a_n)$, where $c(a) = \exp(\alpha(\sum_{x \in P} a(x)))$ for some $\alpha > 0$. This means that the cost of running an analysis under an abstraction a is exponential in the number of parameters set to 1 in a. Refinement algorithms terminate when one of two conditions holds: (i) $q \notin \mathcal{A}(a_n)$ and the answer is 'yes', or (ii) $q \in \mathcal{R}_{G^{a_n}}\big(P_1(a_n)\big)$ and the answer is 'no'. The first termination condition is trivial, and corresponds to the successful verification of the query q by the analysis. The second condition is more subtle. It says that the query q is derived by the analysis under the abstraction a_n but that this derivation does not depend on the parameters set to 0 in a_n. When this happens, the analysis concludes that q cannot be proved by any abstraction, including $\top$. This reasoning is based 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 Empirical Observation

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

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

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.

All the analyses we tried turned out to be predictable, for a simple choice of projection π. We do not know a simple theoretical explanation of why it should be so. Intuitively, though, condition (4) makes sense. When run with the cheap abstraction, the analysis approximates the semantics of the programming language being analysed. Some of these approximations turn out to be correct, and some turn out to be wrong. Correspondingly, some arcs of the cheap provenance are retained in the predictive provenance, and some arcs of the cheap provenance are not retained in the predictive provenance.

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 $\bot$.*

(a) If $q \notin \mathcal{A}(a)$, then $q \notin \mathcal{R}_{G^\bot}(P_0(a))$ 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 $G^\bot$ and H. The assumption $q \in \mathcal{A}(\bot)$ is reasonable: otherwise the refinement algorithm would have terminated after the first iteration.

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

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

Putting these together, we conclude that $q \in \mathcal{A}(a)$. Using a very similar argument we can show that $q \in \mathcal{R}_{G^\bot}(P_0(a))$ 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

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 $G^\bot$. To each arc $e \in G^\bot$, we associate a boolean random variable S_e, and call it the ***selection variable*** of e. Recall that an expectation of S_e is just the probability of $S_e = 1$. Selection variables are independent but may have different expectations. We partition $G^\bot$ into ***types*** $G^\bot_1, \ldots, G^\bot_t$, and we do not require selection variables to have the same expectation unless they have the same type. Each type $G^\bot_k$ has an associated ***hyperparameter*** θ_k: if $e \in G^\bot_k$, then we say that e has type k, and we require that $\mathrm{E}\, S_e = \theta_k$. We define, in terms of the selection variables, a random variable $\mathbf{H}$ whose values are predictive provenances:

$$(\mathbf{H} = H) \text{ if and only if } (e \in H \Leftrightarrow S_e = 1 \text{ for all } e \in G^\bot)$$

In other words, $\mathbf{H}$ is defined such that $S_e = [e \in \mathbf{H}]$. Thus, the probability of a predictive provenance H is

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

For example, if all arcs have the same type, then the model has only one hyperparameter θ, and $\Pr(\mathbf{H} = H)$ is $\theta^{|H|}(1-\theta)^{|G^\bot \setminus H|}$. If $\theta = 1/2$, then all predictive provenances are assigned the probability $2^{-|G^\bot|}$. At the other extreme, if all arcs have their own type, then the model has one hyperparameter θ_e for each arc $e \in G^\bot$, and $\Pr(\mathbf{H} = H)$ is $\prod_{e \in G^\bot} \theta_e^{[e \in H]} (1 - \theta_e)^{[e \notin H]}$.

This concludes the formal presentation of the probabilistic model. But, one question presents itself: How should we group arcs into types? To see why the answer is important, consider two extreme situations: if all arcs have the same type, then the model is very inflexible, and it will likely underfit empirical data; if each arc has its own type, then the model is very flexible, and it will likely overfit empirical data. There is a natural choice for how to define types. Recall that arcs are instances of Datalog rules. The natural choice is to define types to be sets of instances of the same Datalog rule. This natural choice is the one used in experiments (Section 5.4 and Section 6.5), and the results are good. Intuitively, defining types in terms of Datalog rules amounts to using the same granularity as was deemed appropriate by whoever implemented the analysis in Datalog. With such a definition of types, one can affect the flexibility of the probabilistic model by refactoring the Datalog implementation of the given analysis. We did not need to do so.

Finally, recall 'all models are wrong, but some are useful' [7].

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. To use the probabilistic model, it is also necessary to know the cheap provenance $G^\bot$.

After the first iteration, the model can predict what the analysis would do for abstractions not yet tried. In particular, it can predict whether $q \in \mathcal{R}_{G^a}(P_1(a))$, which is one of the two conditions under which refinement algorithms terminate. 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,

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

where R ranges over subsets of vertices of $G^\bot$. It remains to compute a probability of the form $\Pr\big(R = \mathcal{R}_{\mathbf{H}} T\big)$. 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, and it cannot be maximised using analytic methods. Even numeric methods can be unstable or inefficient. This is indeed the case for our model: analytic methods do not apply, some numeric methods are unstable, and some numeric methods are inefficient. But, we did find one numeric method that is both stable and efficient (Section 5.3).

In addition to the standard challenge, our setting presents two additional difficulties, both of them related to the computation of the likelihood. Usually, O has the form $O_1 \cap \ldots \cap O_n$, where the events $O_1, \ldots, O_n$ are independent. In our setting, the event O does indeed have the form $O_1 \cap \ldots \cap O_n$, but the events $O_1, \ldots, O_n$ are not independent. We will handle this difficulty by finding a way to compute $\Pr(O)$ other than the factorisation $\prod_{i=1}^{n} \Pr(O_i)$. Even so, a second difficulty will arise: the expression of $\Pr(O)$ does not fit in the memory of a typical computer if the cheap provenance has cycles. We will handle this difficulty by finding bounds that approximate $\Pr(O)$. In short, the difficulties we need to overcome are: (1) there are dependencies among $O_1, \ldots, O_n$, which we need to account for, and (2) the cycles of the cheap provenance make it infeasible to compute the likelihood exactly.

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, but 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:

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

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.

In practice, we do not analyse all training programs at once, but rather one at a time. This optimisation is straightforward, and does not warrant additional explanation.

5.2 Likelihood

There appears to be no simple and general formula that computes the likelihood $\Pr(O)$. However, there exist reasonably simple 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.

To state the main result on likelihood computation, we need to define forward arcs. 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:

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

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 for $\Pr(\bigcap_{k=1}^{n}(R_k = \mathcal{R}_{\mathbf{H}} T_k))$:*

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

where

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

The proof is fairly technical, and so it is given in Appendix A. Also in Appendix A, one can find an exact formula for computing the likelihood. However, the exact formula is exponential in the size of $G^\perp$, not only in the worst case but also in all of our experiments. The bounds given in Theorem 7 are much smaller than the exact formula, but still too big to be used in practice. Further reduction is needed.

The following result yields a lower bound formula that is small enough to be practical.

Proposition 8. *Let A_h be a set of arcs; let S_e be a selection variable, for each $e \in A_h$. Let C_h be a set of indices; let F_k be a set of arcs, for each $k \in C_h$. Then*

$$\sum_{\substack{E_1 \\ E_1 \subseteq A_h \\ \forall k \in C_h,\ E_1 \cap F_k \neq \emptyset}} \prod_{e \in E_1} \mathrm{E}\, S_e \prod_{e \in A_h \setminus E_1} \mathrm{E}\, \bar{S}_e = \sum_{\substack{E_1 \\ E_1 \subseteq F^h \\ \forall k \in C_h,\ E_1 \cap F_k \neq \emptyset}} \prod_{e \in E_1} \mathrm{E}\, S_e \prod_{e \in F^h \setminus E_1} \mathrm{E}\, \bar{S}_e$$

where $F^h := A_h \cap \left(\bigcup_{k \in C_h} F_k\right)$.

Proof. Let us write $S(X)$ for

$$\sum_{\substack{E_1 \\ E_1 \subseteq X \\ \forall k \in C_h,\ E_1 \cap F_k \neq \emptyset}} \prod_{e \in E_1} \mathrm{E}\, S_e \prod_{e \in X \setminus E_1} \mathrm{E}\, \bar{S}_e$$

We want to show that $S(A_h) = S(F^h)$. We will show that $S(X_1) = S(X_2)$ whenever X_1 and X_2 agree inside $\bigcup_{k \in C_h} F_k$. For this, it is sufficient to consider the case in which $X_1 \cup \{e\} = X_2$ and $e \notin X_1 \cup \bigcup_{k \in C_h} F_k$. Consider some subset E_1 of X_1 such that $E_1 \cap F_k \neq \emptyset$ for all $k \in C_h$. Then E_1 and $E_1 \cup \{e\}$ are subsets of X_2 with the same property. Thus, $S(X_1) = (\mathrm{E}\, S_e) \cdot S(X_1) + (\mathrm{E}\, \bar{S}_e) \cdot S(X_1) = S(X_2)$. □

Proposition 8 reduces the size of the formula from $O\big(2^{|A_h|}\big)$ to $O\big(2^{|F^h|}\big)$. In our experiments, it is often the case that $|A_h| > 20$ and it is sometimes the case that $|A_h| > 900$. On the other hand, it is often the case that $|F^h| < 5$ and it is always the case that $|F^h| < 20$. Of course, Proposition 8 can also be used to reduce the size of the upper bound from Theorem 7, by substituting D_k for F_k. Unfortunately, the sets D_k tend to be significantly larger than the sets F_k. However, we will only need an upper bound for the situation in which all hyperparameters have the value $1/2$. In this case, the following result suffices.

Proposition 9. *Given are sets A_h and C_h, as well as the sets D_k indexed by $k \in C_h$. We have*

$$\sum_{\substack{E_1 \\ E_1 \subseteq A_h \\ \forall k \in C_h,\ E_1 \cap D_k \neq \emptyset}} \frac{1}{2^{|A_h|}} \leq 1 - \frac{1}{2^{\min_{k \in C_h} |A_h \cap D_k|}}$$

Proof. Let n be the number of subsets E_1 of A_h that do *not* satisfy the condition that $E_1 \cap D_k \neq \emptyset$ for all $k \in C_h$. We want to prove that $(2^{|A_h|} - n)/2^{|A_h|} \leq 1 - 1/2^{|A_h \cap D_{k'}|}$ for all $k' \in C_h$. This is equivalent to $n \geq 2^{|A_h \setminus D_{k'}|}$. But this is obvious, because none of the subsets E_1 of $A_h \setminus D_{k'}$ satisfies the condition that $E_1 \cap D_k \neq \emptyset$ for all $k \in C_h$. □

Proposition 9 can be used in an obvious way to weaken the upper bound from Theorem 7. The benefit is that the resulting formula has polynomial size relative to $|G^{\perp}|$, and therefore the corresponding upper bound is easy to compute.

Although the probabilistic model is simple, computing the likelihood of an event of the form '$R_1 = \mathcal{R}_{\mathbf{H}} T_1$ and ... and $R_n = \mathcal{R}_{\mathbf{H}} T_n$' is not easy. Appendix A 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. The size explosion is caused mainly by the cycles of the cheap provenance. Theorem 7 gives lower and upper bounds for the likelihood, which are exponential only in the maximum in-degree of the cheap provenance. The lower bound is small enough to be practical, after reducing its size further using Proposition 8. But, the upper bound needs to be weakened, by using Proposition 9. The resulting upper bound formula has polynomial size, and therefore its corresponding algorithm computes an upper bound in polynomial time. We use the reduced lower bound for learning hyperparameters, and the upper bound in Proposition 9 for measuring the quality of the learnt hyperparameters.

Algorithm	Log Lower Bound	Time [min]
`tnc`	fail	1
`slsqp`	−335.878	1
`basinhopping`	−2164.416	48
`hill`	−341.852	3
`coord`	−335.844	1

Table 1. The logarithms of maximum lower bounds obtained by different optimisers.

5.3 Numeric Optimisation

We ran two analyses (Downcast and PolySite) on 6 programs using 22000 abstractions. We recorded the sets N, C_h, A_h, D_k, and F_k as defined in Theorem 7; this amounts to 2.3 GiB of data. Using this training data we optimised the lower bound by using several numeric optimisers. Table 1 shows the logarithms of maximum lower bounds found by five optimisers, which are interesting because they represent different approaches: `tnc`, `slsqp`, `basinhopping`, `hill`, and `coord`.

The optimisers `tnc`, `slsqp` and `basinhopping` are off-the-shelf optimisers, which are part of the SciPy toolkit [21]. The optimiser `tnc` implements Newton's method, but also has support for bounds, so that the resulting hyperparameters are always in the interval $[0, 1]$. The optimiser `slsqp` uses sequential least squares programming. The optimiser `basinhopping` uses the Metropolis algorithm, but also improves proposals using a local search algorithm; we used `slsqp` as the local search algorithm.

The optimisers `hill` and `coord` are implemented by us. The optimiser `hill` implements gradient ascent with an exponentially decreasing step, and with support for the bounds $[0, 1]$. The optimiser `coord` implements cyclic coordinate ascent, and uses `basinhopping` with `slsqp` for line search.

Out of all five optimisers, `basinhopping` is the only one specifically designed to look for a global maximum, rather than a local maximum.

In addition to the dataset for which results are given in Table 1, we also tried several other smaller training datasets. The results were consistent across multiple datasets.

We found that `tnc` usually failed to give any reasonable answer. The optimiser `slsqp` often finds a very good solution quickly, as seen in the table. However, it also has non-convergent behaviour: if left to run for more iterations, the results *worsen* significantly. The optimiser `basinhopping` is significantly slower than the others. The table reports the result for ≈ 50 min, but it is true that better results would be found if more time were available. Our optimiser `hill` typically converges to a local optimum.

As can be seen in Table 1, `coord` performed as good as `slsqp`. Unlike in the case of `slsqp`, we did not notice any convergence issue. Intuitively, `coord` 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.

In short, we found empirically that for this particular problem, coordinate ascent is the numerical optimisation method of choice. Next come sequential least squares programming (`slsqp`) and gradient ascent (`hill`). We found that `slsqp` sometimes does not

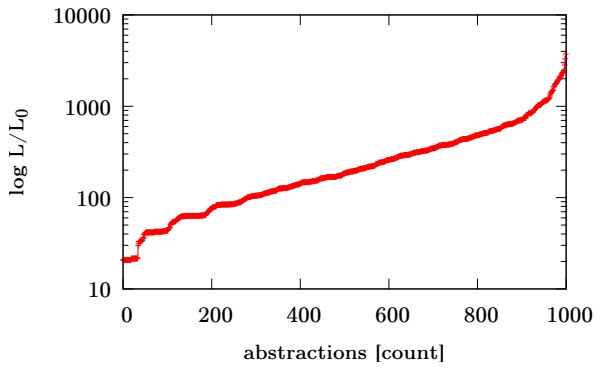

Figure 4. Model quality. Note that the y-axis gives $\log L/L_0$ on a logarithmic scale; thus, it has double-logarithmic scale with respect to L/L_0.

converge, at least with the implementation in SciPy, and `hill` almost always finds a local optimum but not a global optimum. Next comes a method based on Markov chains (`basinhopping`), which is extremely slow. Finally, Newton's method always fails, at least with the implementation in SciPy.

5.4 Evaluation

One way to evaluate a probabilistic model is the following. We begin by observing a sequence of events of the kind we want to predict. For each of these events we compute L and L_0: the probability of the event according to our model, and the probability of the event according to random guessing. In our case, random guessing corresponds to setting all hyperparameters to the value $1/2$. The model is good when L/L_0 is large.

In our case, we do not have an efficient algorithm for computing L/L_0, nor L or L_0 individually. But, to ensure that L/L_0 is large, we only need to compute a lower bound. If $L^{\text{lb}} \leq L$ and $L_0^{\text{up}} \geq L_0$, then $L/L_0 \geq L^{\text{lb}}/L_0^{\text{ub}}$. We compute L^{lb} using the lower bound formula in Theorem 7 improved with the simplification in Proposition 8, as before. We compute L_0^{ub} using the upper bound formula in Theorem 7 weakened and simplified by using Proposition 9.

We find that for the training event we have $(L/L_0) > e^{13109} \approx 1.5 \times 10^{5693}$. A large gain is to be expected for the training event. The question is whether we observe large gains for other events we are interested in predicting.

We chose 1000 abstractions at random. For each abstraction a, we computed $T := \pi(P_1(a))$ and $R := \pi\big(\mathcal{R}_{G^\top}(P_1(a))\big)$. For each such pair (R, T), we checked what is the likelihood gain for the event $R = \mathcal{R}_{\mathbf{H}}T$. Figure 4 shows lower bounds for the likelihood gain, where the lower bound is computed as described above. For more than 70% of events, the gain L/L_0 is greater than $e^{100} \approx 2.6 \times 10^{43}$.

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 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 5. The refinement algorithm used to solve Problem 2.

6.1 Refinement Algorithm

The refinement algorithm is straightforward (Figure 5). 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 5) 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':

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

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 $A_{\text{y}}^{\supseteq}$; for A_{n} we use a heuristic approximation $A_{\text{n}}^{\approx}$.

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

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 10. *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 [51]. The pessimistic strategy is used in our implementation. We found that it leads to fewer iterations and smaller runtime (Section 6.5). 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 5. The pessimistic restriction $A_{\mathrm{n}}^{\approx}$ is nonempty because it always contains $\top$, by choosing $H = G^a$ (see Lemma 17).

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

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

defined in a probabilistic model with respect to G^a, *not* $G^{\perp}$. 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 11. *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*

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

where S_e is the selection variable of arc e.

Proof. The proof is a straightforward calculation.

$$\begin{aligned}
\Pr(a' \in \mathbf{A}_{\mathrm{n}}^{\approx}) \;&=\; \sum_{\substack{H' \\ H' \subseteq G^a}} [q \in \mathcal{R}_{H'}(T(a, a'))] \Pr(H') \\
&\geq \sum_{\substack{H' \\ H \subseteq H' \subseteq G^a}} [q \in \mathcal{R}_{H'}(T(a, a'))] \Pr(H') \\
&= \sum_{\substack{H' \\ H \subseteq H' \subseteq G^a}} \Pr(H') \;=\; \prod_{e \in H} \mathrm{E}\, S_e
\end{aligned}$$

The second equality uses two facts: (i) $q \in \mathcal{R}_H(T(a, a'))$, and (ii) $\mathcal{R}_H(T(a, a')) \subseteq \mathcal{R}_{H'}(T(a, a'))$ for all $H' \supseteq H$. □

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

The next abstraction a' should be more precise than the last abstraction a. It is desirable that a' is likely to lead to termination: $\Pr(a' \in \mathbf{A}_{\mathrm{n}}^{\approx})$ should be big. At the same time, it is desirable that a' is cheap: $c(a') = \exp\bigl(\alpha \sum_{x \in P} a'(x)\bigr)$ should be small. This raises the question of how to integrate these two metrics.

Definition 12 (Action Scheduling Problem)**.** Suppose that we have a list of m actions for $m \geq 1$, 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 $C(\sigma)$:

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

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 future 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'.

We prove that under some independence assumption, we can solve the action scheduling problem:

Lemma 13. *Consider an instance of the action scheduling problem (Definition 12). Assume the success probabilities of the actions are independent. A permutation σ has the minimal cost $C(\sigma)$ if and only if for all $1 \leq i, j \leq m$,*

$$i \leq j \quad \Rightarrow \quad \frac{c_{\sigma(i)}}{p_{\sigma(i)}} \leq \frac{c_{\sigma(j)}}{p_{\sigma(j)}}. \tag{6}$$

For the proof, see Appendix B.

Corollary 14. *Under the conditions of Lemma 13, for every permutation σ, if σ has the minimal cost 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.

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 $\Pr(a' \in \mathbf{A}_{\mathrm{n}}^{\approx})/c(a')$ (see Corollary 14). In doing so, we will approximate $\Pr(a' \in \mathbf{A}_{\mathrm{n}}^{\approx})$ by a lower bound (see Lemma 11). In short, we want to evaluate the following expression:

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

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

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

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,

$M \models x$	iff	$x \in M$
$M \models \bar{x}$	iff	$x \notin M$
$M \models \phi_1 \wedge \phi_2$	iff	$M \models \phi_1$ and $M \models \phi_2$

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 15 (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 16. Technically, Problem 15 is none of the standard variations of MaxSAT. It is easy to see, although we do not prove it here, that Problem 15 is polynomial-time equivalent to partial weighted MaxSAT [3, 34]: 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 and Appendix A). As was the case for likelihood, cycles lead to an exponential explosion. We 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:

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

The notation $\exists_{e \in G^a_{\rightarrow}} y_e$ stands for several existential quantifiers, one for each variable in the set $\{y_e\}_{e \in G^a_{\rightarrow}}$. 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). 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 17. *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 5.

Proof. The inclusion $F(G^a_{\rightarrow}) \subseteq F(G^a)$ follows from $G^a_{\rightarrow} \subseteq G^a$. We have $(\top, G^a_{\rightarrow}) \in F(G^a_{\rightarrow})$ because (a) $\top > a$ by assumption, (b) $G^a_{\rightarrow} \subseteq G^a_{\rightarrow}$ trivially, and (c) $q \in \mathcal{R}_{G^a_{\rightarrow}}(T(a, \top))$. To see why (c) holds, notice that removing nonforward arcs with respect to $T(a, \top) = P_0(a) \cup P_1(a)$ preserves distances and reachability from $T(a, \top)$, and so $\mathcal{R}_{G^a_{\rightarrow}}(T(a, \top)) = \mathcal{R}_{G^a}(T(a, \top))$. □

Lemma 18. *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 $F(G^a_{\rightarrow})$ be defined as in (9). There is a bijection between the models M of Φ and the elements (a', H) of $F(G^a_{\rightarrow})$. According to this bijection,*

$$\begin{aligned}
M \cap X_E(G^a_{\rightarrow}) &= X_E(H) \\
M \cap X_V(G^a_{\rightarrow}) &= X_V\big(\mathcal{R}_H(T(a, a'))\big)
\end{aligned}$$

The proof of this lemma, given in Appendix B, 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 $\sum_{e \in H} \log(\mathrm{E}\, S_e) - \sum_{\substack{x \in P \\ a'(x) = 1}} \alpha$ by assigning weights to variables. This is very easy. Each arc variable x_e is assigned the weight $w(x_e) = \log(\mathrm{E}\, S_e)$. 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.

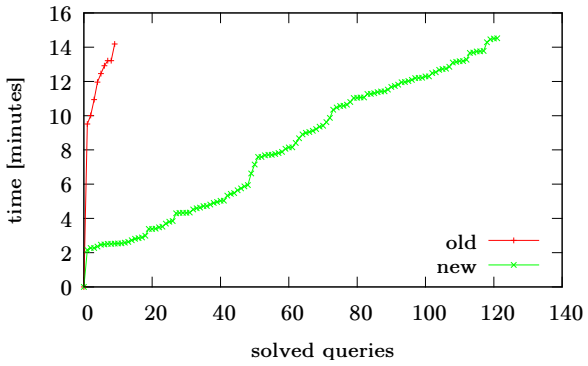

Figure 6. Comparison between an existing algorithm (old) and ours (new). The old algorithm, of Zhang et al. [51], uses an optimistic refinement strategy. Our new algorithm uses a pessimistic refinement strategy and also a probabilistic model.

6.5 Evaluation

To evaluate the refinement algorithm, we use it to perform pointer analysis on 6 programs taken from the benchmark suites Ashes [47] and DaCapo [6]: antlr (a parser generator), hedc (a web crawler), javasrc-p (pretty printer for Java code), schroeder-m (audio editing tool), toba-s (translates Java bytecode into C code), and weblech (website download tool).

Our implementation is based on that of [51]. In particular, we use the same Datalog implementation of the static analysis, and the same open-source Datalog and MAXSAT solvers (see [20, 49]). But, we have re-implemented their refinement algorithm and, alongside, we implemented our own refinement algorithm. This way, as much code as possible is shared.

The pointer analysis is flow insensitive but object sensitive. It determines for each expression in the program a set of possible dynamic types. The more precise the analysis, the more restricted the set of possible dynamic types. Based on these sets, the analysis answers two types of queries. A PolySite query asks whether the receiver of a method invocation has at most one possible dynamic type; a Downcast query asks whether all possible types of an expression being cast to T are subtypes of T.

Figure 6 shows the total runtime, for the solved queries. For each testcase, the following limits were enforced: 100 GiB of space and 15 minutes of time. Within these limits, our algorithm solves many more queries than the baseline algorithm.

7. Related work

The potential of using machine learning techniques or probabilistic reasoning for addressing challenges in static analysis [4, 12] 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, 24, 30, 33, 38, 39, 41], to guess candidate program invariants from test data or program traces using generalisation techniques from machine learning [31, 36, 43], and to predict properties of potential or real program errors, such as true positiveness and cause, probabilistically [27, 28, 50, 53]. 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.

A typical bottleneck in combining techniques from probabilistic reasoning with techniques from static analysis is that the former are inherently numeric while the latter are not. To bridge the gap, one needs to design so called features, which essentially translate between the non-numeric world of static analysis to the numeric world of probabilities and machine learning. But, designing such features is no easy task. Our work shows that it is possible to obtain good results without designing any feature at all, provided only that the analysis is implemented in Datalog.

Several probabilistic models for program source code have been proposed in the past [1, 2, 19, 23, 32, 38, 39], and used for extracting natural coding conventions [1], helping the correct use of library functions [39], translating programs between different languages [23], and cleaning program source code and inferring likely properties [38]. 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 [14, 15, 42]. In this line of work, using one hyperparameter θ for all arcs of the same type is a commonly used technique for fighting against overfitting to training data and for learning good hyperparameters. In our case, models are large, and training data provide only partial information about the random variable $\mathbf{H}$ used in the models. Learning hyperparameters in such cases is generally intractable, and we have overcome this intractability by analytically deriving the lower bound of probabilities in Theorem 7 and optimising this lower bound. Using such a proxy during learning is common in machine learning, in particular, in the work on variational inference [22, 48].

Our work builds on a large amount of research for automatically finding good program abstraction, such as CEGAR [4, 9–11, 18, 40], parametric static analysis with parameter search algorithms [26, 35, 51, 52], and static analysis based on Datalog or Horn solvers [8, 16, 17, 44, 49]. 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.

Recently, non-probabilistic approaches for estimating the impacts of different program abstractions on a given analysis or verification task have been proposed [37, 45]. One interesting future direction is to revisit these approaches from the perspective of probabilistic modelling explained in this paper, with the goal of obtaining probabilistic variants of their approaches that replace the current hard-coded heuristics for prediction by adaptable ones.

8. 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 the promise of our approach.

Acknowledgements. Yang was 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] 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] Stephen M. Blackburn, Robin Garner, Chris Hoffmann, Asjad M. Khan, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony L. Hosking, Maria Jump, Han Bok Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanovic, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. In *OOPSLA*, 2006.

[7] George E. P. Box. Science and statistics. *Journal of the American Statistical Association*, 1976.

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

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

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

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

[12] 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.

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

[14] 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.

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

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

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

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

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

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

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

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

[25] Joohyung Lee. A model-theoretic counterpart of loop formulas. In *IJCAI*, 2005.

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

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

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

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

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

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

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

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

[34] 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.

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

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

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

[38] Veselin Raychev, Martin T. Vechev, and Andreas Krause. Predicting program properties from "big code". In *POPL*, 2015.

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

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

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

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

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

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

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

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

[47] R. Vallée-Rai. Ashes suite collection. `http://www.sable.mcgill.ca/ashes/`.

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

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

[50] 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.

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

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

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

A. Proof of Theorem 7

We begin by restating in our notation a standard result from logic programming. A ***dependency graph*** of a hypergraph G is a directed graph that includes an arc (h, b) whenever $(h, B) \in G$ and $b \in B$ for some B. A ***loop*** L of a hypergraph G is a nonempty subset of its vertices that are strongly connected in the corresponding dependency graph. Note that loops are not required to be maximal. In particular, sets that contain single vertices are loops, called ***trivial loops***. The set $J_G(L)$ of ***justifications*** for loop L in G is defined as follows:

$$J_G(L) := \{\, (h, B) \in G \mid h \in L \text{ and } B \cap L = \emptyset \,\}$$

For a hypergraph G we define its ***forward formula*** $\phi_\rightarrow(G)$ and its ***backward formula*** $\phi_\leftarrow(G)$ as follows:

$$\phi_\rightarrow(G) \;:=\; \bigwedge_{e=(h,B)\in G} \Big(\big((\bigwedge_{b\in B} x_b) \leftrightarrow x_e \big) \wedge (x_e \rightarrow x_h) \Big)$$

$$\phi_\leftarrow(G) \;:=\; \bigwedge_{\substack{L \\ \text{loop of } G}} \Big((\bigwedge_{u\in L} x_u) \rightarrow (\bigvee_{e\in J_G(L)} x_e) \Big)$$

Both formulas are defined over the following set of variables:

$$\{\, x_u \mid u \text{ vertex of } G \,\} \cup \{\, x_e \mid e \text{ arc of } G \,\}$$

We define the ***formula*** $\phi(G)$ of a hypergraph G by

$$\phi(G) \;:=\; \exists_{e\in G}\, x_e \; \big(\phi_\rightarrow(G) \wedge \phi_\leftarrow(G) \big)$$

The notation $\exists_{e\in G}\, x_e$ stands for several existential quantifiers, one for each variable in the set $\{x_e\}_{e\in G}$ indexed by G. In the definition of $\phi(G)$ from above, the existential quantification is not strictly necessary, but convenient: Because the remaining free variables correspond to vertices, sets of variables are isomorphic to sets of vertices.

We view models M of a formula φ as sets of variables; that is,

$$\begin{array}{lll} M \models x & \text{iff} & x \in M \\ M \models \bar{x} & \text{iff} & x \notin M \\ M \models \varphi_1 \rightarrow \varphi_2 & \text{iff} & M \models \varphi_1 \text{ implies } M \models \varphi_2 \\ M \models \exists x\, \varphi & \text{iff} & M \models \varphi[x := 0] \text{ or } M \models \varphi[x := 1] \end{array}$$

and so on, in the standard way. There is an obvious one-to-one correspondence between sets of vertices and models; if S is a set of vertices, we write XS for the corresponding model, which is a set of variables:

$$XS \;:=\; \{\, x_s \mid s \in S \,\}$$

The following result is stated in [25, Section 3], in a slightly more general form and with slightly different notations:

Lemma 19. *Let G be a hypergraph, and let $\phi(G)$ be its formula, defined as above. Then $X\big(\mathcal{R}_G\emptyset\big)$ is the unique model of $\phi(G)$.*

For the proof, we refer to [25].

Remark 20. We note that $\phi_\rightarrow(G)$ is linear in the size of G, while $\phi_\leftarrow(G)$ is exponential in the size of G in the worst case. One could wonder whether it is possible to define $\phi(G)$ in a way that does not lead to exponentially large formulas but Lemma 19 still holds. It turns out there are reasons to suspect that such an alternative definition does not exist [29].

Here, we shall need a more flexible form of Lemma 19. Let S be a distinguished subset of vertices, none of which occurs in the head of an arc. Define

$$\phi^S_\leftarrow(G) \;:=\; \bigwedge_{\substack{L \\ \text{loop of } G \\ L\cap S=\emptyset}} \Big((\bigwedge_{u\in L} x_u) \rightarrow (\bigvee_{e\in J_G(L)} x_e) \Big)$$

and

$$\phi^S(G) \;:=\; \underset{e\in G}{\exists}\, x_e \left(\phi_{\rightarrow}(G) \wedge \phi^S_{\leftarrow}(G)\right) \tag{10}$$

Corollary 21. *Let G be a hypergraph, let S be a subset of vertices such that none of them occurs in the head of an arc, and let $\phi^S(G)$ be defined as above. For each subset T of S, there exists a unique model M of $\phi^S(G)$ such that $X^{-1}(M) \cap S = T$, namely $M = X(\mathcal{R}_G T)$.*

Proof. For a fixed but arbitrary $T \subseteq S$, construct the graph

$$G_T \;:=\; G \cup \{\, (t, \emptyset) \mid t \in T \,\}$$

It is easy to check that $\mathcal{R}_G T = \mathcal{R}_{G_T}\emptyset$. From Lemma 19, we know that $X\big(\mathcal{R}_{G_T}\emptyset\big)$ is the unique model of $\phi(G_T)$. Since the vertices of S do not occur in the heads of arcs, they appear only in trivial loops. Thus, we have

$$\begin{aligned}
\phi_{\rightarrow}(G_T) \;&=\; \phi_{\rightarrow}(G) \wedge \Big(\bigwedge_{t\in T} x_t\Big) \\
\phi_{\leftarrow}(G_T) \;&=\; \phi^S_{\leftarrow}(G) \wedge \Big(\bigwedge_{s\in S\setminus T} \bar{x_s}\Big)
\end{aligned}$$

(The formulas above eliminate via existential quantification the variables corresponding to the dummy arcs $(t, \emptyset)$ of G_T, but this is of little consequence.) And finally

$$\begin{aligned}
\phi_{\rightarrow}(G_T) \wedge \phi_{\leftarrow}(G_T) \;=\; &\phi_{\rightarrow}(G) \wedge \phi^S_{\leftarrow}(G) \\
&\wedge \Big(\bigwedge_{s\in S\setminus T} \bar{x_s}\Big) \wedge \Big(\bigwedge_{t\in T} x_t\Big)
\end{aligned}$$

This concludes the proof. □

We now take a special case of Corollary 21.

Corollary 22. *Let G be a hypergraph. Let (S, V) be a partition of its vertices such that no vertex in S occurs as the head of an arc. Let $\phi^S(G)$ be defined as above. Let R be a subset of V. Define*

$$\phi^{S,R}(G) \;:=\; \underset{u\in V}{\exists}\, x_u \left(\phi^S(G) \wedge \Big(\bigwedge_{u\in R} x_u\Big) \wedge \Big(\bigwedge_{u\in V\setminus R} \bar{x_u}\Big)\right)$$

For all $T \subseteq S$, we have that XT is a model of $\phi^{S,R}(G)$ if and only if $\mathcal{R}_G T = T \cup R$.

Proof. Let T be a subset of S. Then, XT is a model of $\phi^{S,R}(G)$ if and only if $X(T \cup R)$ is a model of $\phi^S(G)$. But by Corollary 21, this is equivalent to $\mathcal{R}_G T = T \cup R$. □

The key idea of our proof is to use Corollary 22 in such a way that subsets of S correspond to predictive provenances H. To this end, we define the ***extended cheap provenance*** $G^{\perp}_T$ with respect to the set T of vertices by

$$G^{\perp}_T := \{\, (h, B \cup \{s_e\}) \mid e = (h, B) \in G^{\perp} \,\} \cup \{\, (t, \emptyset) \mid t \in T \,\}$$

Recall our notation $G^{\perp}$ for the cheap provenance. For a predictive provenance $H \subseteq G^{\perp}$, let us write SH for $\{\, s_e \mid e \in H \,\}$. All the vertices of $SG^{\perp}$ are fresh: they appear in $G^{\perp}_T$ but not in $G^{\perp}$. The extended cheap provenance has the property that

$$\mathcal{R}_{G^{\perp}_T}(SH) = (SH) \cup \mathcal{R}_H T \tag{11}$$

for all predictive provenances $H \subseteq G^{\perp}$ and all sets of vertices T.

Suppose the cheap provenance $G^{\perp}$ and two subsets T and R of its vertices are given. The following lemma shows how to construct a boolean formula whose models are in one-to-one correspondence with the cheap provenances $H \subseteq G^{\perp}$ for which $R = \mathcal{R}_H T$.

Lemma 23. *Let $G^{\perp}$ be a cheap provenance, and let R and T be two subsets of its vertices. Define the extended cheap provenance $G^{\perp}_T$ with respect to T as above. We have that $R = \mathcal{R}_H T$ if and only if $X(SH)$ is a model of $\phi^{SG^{\perp},R}(G^{\perp}_T)$.*

Proof. In Corollary 22, set $S := SG^{\perp}$ and $T := SH$ and $G := G^{\perp}_T$. We obtain that

$$X(SH) \models \phi^{SG^{\perp},R}(G^{\perp}_T) \quad \text{iff} \quad \mathcal{R}_{G^{\perp}_T}(SH) = (SH) \cup R$$

Combining this with (11) we obtain

$$X(SH) \models \phi^{SG^{\perp},R}(G^{\perp}_T) \quad \text{iff} \quad (SH) \cup \mathcal{R}_H T = (SH) \cup R$$

Finally, since all the vertices in SH are fresh, we are done. □

What remains to be done is to make explicit the formula $\phi^{SG^{\perp},R}(G^{\perp}_T)$ mentioned in Lemma 23. This is only a matter of calculation. We begin by unfolding the definition of $\phi^{SG^{\perp},R}(G^{\perp}_T)$, and then that of $\phi^{SG^{\perp}}(G^{\perp}_T)$. Below, the notation $\varphi[x_R := v]$ means that in φ we substitute the variable x_u with value v for all indices $u \in R$. Also, we write V for the vertex set of $G^{\perp}$.

$$\begin{aligned}
&\phi^{SG^{\perp},R}(G^{\perp}_T) \\
&\quad= \underset{u\in V}{\exists}\, x_u \left(\phi^{SG^{\perp}}(G^{\perp}_T) \wedge \Big(\bigwedge_{u\in R} x_u\Big) \wedge \Big(\bigwedge_{u\in V\setminus R} \bar{x_u}\Big)\right) \\
&\quad= \phi^{SG^{\perp}}(G^{\perp}_T)[x_R := 1][x_{V\setminus R} := 0] \\
&\quad= \underset{e\in G^{\perp}_T}{\exists}\, x_e \left(\phi_{\rightarrow}(G^{\perp}_T) \wedge \phi^{SG^{\perp}}_{\leftarrow}(G^{\perp}_T)\right)[x_R := 1][x_{V\setminus R} := 0] \\
&\quad= \underset{e\in G^{\perp}_T}{\exists}\, x_e \left(\Psi_{\rightarrow} \wedge \Psi_{\leftarrow}\right)
\end{aligned}$$

where

$$\begin{aligned}
\Psi_{\rightarrow} \;&:=\; \phi_{\rightarrow}(G^{\perp}_T)[x_R := 1][x_{V\setminus R} := 0] \\
\Psi_{\leftarrow} \;&:=\; \phi^{SG^{\perp}}_{\leftarrow}(G^{\perp}_T)[x_R := 1][x_{V\setminus R} := 0]
\end{aligned}$$

Now we calculate $\Psi_{\rightarrow}$ and $\Psi_{\leftarrow}$, in turn. We begin with $\Psi_{\rightarrow}$. First we unfold the definition of $\phi_{\rightarrow}(G^{\perp}_T)$, then we unfold the definition of $G^{\perp}_T$, and finally we apply the substitutions. During the calculation, we identify x_{s_e} with S_e. This is partly notational convenience (to avoid double subscripts), but it will also allow us to weigh models according to the probabilistic model.

$$\begin{aligned}
\Psi_{\rightarrow} &= \phi_{\rightarrow}(G^{\perp}_T)[x_R := 1][x_{V\setminus R} := 0] \\
&= \bigwedge_{\substack{e\in G^{\perp}_T \\ e=(h,B)}} \left(\Big(\big(\bigwedge_{b\in B} x_b\big) \leftrightarrow x_e\Big) \wedge (x_e \rightarrow x_h)\right) \begin{bmatrix} x_R := 1 \\ x_{V\setminus R} := 0 \end{bmatrix} \\
&= \Bigg(\bigwedge_{\substack{e'\in G^{\perp} \\ e'=(h,B) \\ e=(h,B\cup\{s_{e'}\})}} \left(\Big(\big(\bigwedge_{b\in B\cup\{s_{e'}\}} x_b\big) \leftrightarrow x_e\Big) \wedge (x_e \rightarrow x_h)\right) \\
&\qquad \wedge \bigwedge_{\substack{t\in T \\ e=(t,\emptyset)}} (x_e \wedge x_t)\Bigg) \begin{bmatrix} x_R := 1 \\ x_{V\setminus R} := 0 \end{bmatrix} \\
&= \bigwedge_{\substack{e'\in G^{\perp} \\ e'=(h,B) \\ e=(h,B\cup\{s_{e'}\})}} \left(\Big(\big(S_{e'} \wedge [B \subseteq R]\big) \leftrightarrow x_e\Big) \wedge (x_e \rightarrow [h \in R])\right) \\
&\qquad \wedge \bigwedge_{\substack{t\in T \\ e=(t,\emptyset)}} (x_e \wedge [t \in R])
\end{aligned}$$

If $T \not\subseteq R$, then $\Psi_{\rightarrow} = 0$; otherwise,

$$\Psi_{\rightarrow} = \Big(\bigwedge_{\substack{e'=(h,B)\in G^{\perp} \\ e=(h,B\cup\{s_{e'}\}) \\ B \subseteq R \text{ and } h \in R}} (S_{e'} \leftrightarrow x_e) \Big) \wedge \Big(\bigwedge_{\substack{e'=(h,B)\in G^{\perp} \\ B \subseteq R \text{ and } h \notin R}} \bar{S_{e'}} \Big) \wedge \Big(\bigwedge_{\substack{e'=(h,B)\in G^{\perp} \\ e=(h,B\cup\{s_{e'}\}) \\ B \not\subseteq R}} \bar{x_e} \Big) \wedge \Big(\bigwedge_{\substack{t\in T \\ e=(t,\emptyset)}} x_e \Big) \tag{12}$$

Next, we calculate $\Psi_{\leftarrow}$.

$$\begin{aligned}
\Psi_{\leftarrow} &= \phi^{SG^{\perp}}_{\leftarrow}(G^{\perp}_T)[x_R := 1][x_{V\setminus R} := 0] \\
&= \bigwedge_{\substack{L \\ \text{loop of } G^{\perp}_T \\ L\cap SG^{\perp}=\emptyset}} \Big((\bigwedge_{u\in L} x_u) \rightarrow (\bigvee_{e\in J_{G^{\perp}_T}(L)} x_e) \Big) \begin{bmatrix} x_R := 1 \\ x_{V\setminus R} := 0 \end{bmatrix} \\
&= \bigwedge_{\substack{L \\ \text{loop of } G^{\perp}}} \Big((\bigwedge_{u\in L} x_u) \rightarrow (\bigvee_{e\in J_{G^{\perp}_T}(L)} x_e) \Big) \begin{bmatrix} x_R := 1 \\ x_{V\setminus R} := 0 \end{bmatrix} \\
&= \bigwedge_{\substack{L \\ \text{loop of } G^{\perp}}} \Big([L \subseteq R] \rightarrow (\bigvee_{e\in J_{G^{\perp}_T}(L)} x_e) \Big) \\
&= \bigwedge_{\substack{L \\ \text{loop of } G^{\perp} \\ L\subseteq R}} \Big(\big(\bigvee_{\substack{e' \\ e'=(h,B)\in J_{G^{\perp}}(L) \\ e=(h,B\cup\{s_{e'}\})}} x_e \big) \vee \big(\bigvee_{\substack{t\in T\cap L \\ e=(t,\emptyset)}} x_e \big) \Big)
\end{aligned}$$

When we calculate $\Psi_{\rightarrow} \wedge \Psi_{\leftarrow}$ we see that $\Psi_{\rightarrow}$ fixes the values of all the variables x_e corresponding to arcs.

$$\begin{aligned}
\phi^{SG^{\perp},R}(G^{\perp}_T) &= \exists_{e\in G^{\perp}_T} x_e \, (\Psi_{\rightarrow} \wedge \Psi_{\leftarrow}) \\
&= [T \subseteq R] \wedge \Big(\bigwedge_{\substack{e'=(h,B)\in G^{\perp} \\ B \subseteq R \text{ and } h \notin R}} \bar{S_{e'}} \Big) \wedge \Psi_{\leftarrow} \begin{bmatrix} x_e := S_{e'} \text{ for } (e,e') \in S \\ x_e := 0 \text{ for } e \in O \\ x_e := 1 \text{ for } e \in I \end{bmatrix}
\end{aligned}$$

where S, O, and I stand for corresponding ranges in (12). More precisely, letting $e' = (h, B)$ range over $G^{\perp}$ and letting e be its corresponding arc $(h, B \cup \{s_{e'}\})$ in $G^{\perp}_T$, we have $S := \{ (e, e') \mid B \subseteq R \text{ and } h \in R \}$ and $O := \{ e \mid B \not\subseteq R \}$. Also, I contains all the dummy arcs of the form $(t, \emptyset)$, for all $t \in T$. Now we apply these three substitutions to $\Psi_{\leftarrow}$, one by one. The first line just introduces a shorthand notation for each of the three kinds of substitutions.

$$\begin{aligned}
\Psi_{\leftarrow} \begin{bmatrix} x_e := S_{e'} \text{ for } (e,e') \in S \\ x_e := 0 \text{ for } e \in O \\ x_e := 1 \text{ for } e \in I \end{bmatrix} &= \Psi_{\leftarrow} \begin{bmatrix} \mathcal{S} \\ \mathcal{O} \\ \mathcal{I} \end{bmatrix} \\
&= \bigwedge_{\substack{L \\ \text{loop of } G^{\perp} \\ L\subseteq R}} \Big(\big(\bigvee_{\substack{e' \\ e'=(h,B)\in J_{G^{\perp}}(L) \\ e=(h,B\cup\{s_{e'}\})}} x_e \big) \vee \big(\bigvee_{\substack{t\in T\cap L \\ e=(t,\emptyset)}} x_e \big) \Big) \begin{bmatrix} \mathcal{S} \\ \mathcal{O} \\ \mathcal{I} \end{bmatrix} \\
&= \bigwedge_{\substack{L \\ \text{loop of } G^{\perp} \\ L\subseteq R\setminus T}} \bigvee_{\substack{e' \\ e'=(h,B)\in J_{G^{\perp}}(L) \\ e=(h,B\cup\{s_{e'}\})}} x_e \begin{bmatrix} \mathcal{S} \\ \mathcal{O} \end{bmatrix} \\
&= \bigwedge_{\substack{L \\ \text{loop of } G^{\perp} \\ L\subseteq R\setminus T}} \bigvee_{\substack{e' \\ e'=(h,B)\in J_{G^{\perp}}(L) \\ e=(h,B\cup\{s_{e'}\}) \\ B\subseteq R}} x_e[\mathcal{S}] \\
&= \bigwedge_{\substack{L \\ \text{loop of } G^{\perp} \\ L\subseteq R\setminus T}} \bigvee_{\substack{e' \\ e'=(h,B)\in J_{G^{\perp}}(L) \\ B\subseteq R}} S_{e'}
\end{aligned}$$

Finally, we conclude that

$$\phi^{SG^{\perp},R}(G^{\perp}_T) = [T \subseteq R] \wedge \Big(\bigwedge_{\substack{e=(h,B)\in G^{\perp} \\ B\subseteq R,\ h\notin R}} \bar{S_e} \Big) \wedge \Big(\bigwedge_{\substack{L \\ \text{loop in } G^{\perp} \\ L\subseteq R\setminus T}} \bigvee_{\substack{e=(h,B) \\ e\in J_{G^{\perp}}(L) \\ B\subseteq R}} S_e \Big) \tag{13}$$

Now observe that

$$\Pr\Big(\bigcap_{k=1}^{n} \big(R_k = \mathcal{R}_{\mathbf{H}}(T_k) \big) \Big) = \mathrm{E}\Big(\bigwedge_{k=1}^{n} \phi^{SG^{\perp},R_k}(G^{\perp}_{T_k}) \Big) \tag{14}$$

Putting together (13) and (14), we obtain the following lemma.

Lemma 24. *Consider the probabilistic model associated with the cheap provenance $G^{\perp}$ of an analysis $\mathcal{A}$. Let $T_1, \ldots, T_n$ and $R_1, \ldots, R_n$ be subsets of the vertices of $G^{\perp}$. If $T_k \subseteq R_k$ for all k, then*

$$\Pr\Big(\bigcap_{k=1}^{n} (R_k = \mathcal{R}_{\mathbf{H}} T_k) \Big) = \prod_{e\in N} \mathrm{E}\, \bar{S}_e \cdot \mathrm{E}\Big(\bigwedge_{\substack{L \\ \text{loop of } G^{\perp}}} \bigwedge_{\substack{k \\ L\subseteq R_k\setminus T_k}} \bigvee_{\substack{e=(h,B) \\ e\in J_{G^{\perp}}(L)\setminus N \\ B\subseteq R_k}} S_e \Big)$$

where

$$N := \{ (h, B) \in G^{\perp} \mid B \subseteq R_k \text{ and } h \notin R_k \text{ for some } k \}$$

Proof. We assume that $T_k \subseteq R_k$. Using (13) and (14), we transform $\bigwedge_{k=1}^{n} \phi^{SG^{\perp},R_k}(G^{\perp}_{T_k})$ as follows:

$$\begin{aligned}
&\bigwedge_{k=1}^{n} \phi^{SG^{\perp},R_k}(G^{\perp}_{T_k}) \\
&= \bigwedge_{k=1}^{n} \Big(\Big(\bigwedge_{\substack{e=(h,B)\in G^{\perp} \\ B\subseteq R_k,\ h\notin R_k}} \bar{S_e} \Big) \wedge \Big(\bigwedge_{\substack{L \\ \text{loop in } G^{\perp} \\ L\subseteq R_k\setminus T_k}} \bigvee_{\substack{e=(h,B) \\ e\in J_{G^{\perp}}(L) \\ B\subseteq R_k}} S_e \Big) \Big) \\
&= \Big(\bigwedge_{e\in N} \bar{S_e} \Big) \wedge \Big(\bigwedge_{k=1}^{n} \bigwedge_{\substack{L \\ \text{loop in } G^{\perp} \\ L\subseteq R_k\setminus T_k}} \bigvee_{\substack{e=(h,B) \\ e\in J_{G^{\perp}}(L) \\ B\subseteq R_k}} S_e \Big) \\
&= \Big(\bigwedge_{e\in N} \bar{S_e} \Big) \wedge \Big(\bigwedge_{k=1}^{n} \bigwedge_{\substack{L \\ \text{loop in } G^{\perp} \\ L\subseteq R_k\setminus T_k}} \bigvee_{\substack{e=(h,B) \\ e\in J_{G^{\perp}}(L)\setminus N \\ B\subseteq R_k}} S_e \Big) \\
&= \Big(\bigwedge_{e\in N} \bar{S_e} \Big) \wedge \Big(\bigwedge_{\substack{L \\ \text{loop in } G^{\perp}}} \bigwedge_{\substack{k\in\{1,\ldots,n\} \\ L\subseteq R_k\setminus T_k}} \bigvee_{\substack{e=(h,B) \\ e\in J_{G^{\perp}}(L)\setminus N \\ B\subseteq R_k}} S_e \Big)
\end{aligned}$$

The conclusion of the lemma now follows from the result of this calculation and the fact that S_e and $S_{e'}$ are independent whenever $e \neq e'$. □

We can finally prove Theorem 7. Recall its statement:

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 for $\Pr(\bigcap_{k=1}^n (R_k = \mathcal{R}_{\mathbf{H}} T_k))$*:*

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

where

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

If $T_k \not\subseteq R_k$ for some k, then the probability and both of its bounds are all 0. In what follows, we shall invoke Lemma 24, thus silently assuming that $T_k \subseteq R_k$ for all k. We first prove the claim about an upper bound, and then show the claim about a lower bound.

Proof of the Upper Bound in Theorem 7. We start with a short calculation which shows what happens if we consider only trivial loops. Recall the assumption that $h \notin B$ for all arcs (h, B).

$$\mathrm{E}\Big(\bigwedge_{\substack{L \\ \text{loop of } G^\perp}} \bigwedge_{\substack{k \\ L \subseteq R_k \setminus T_k}} \bigvee_{\substack{e=(h,B) \\ e \in J_{G^\perp}(L) \setminus N \\ B \subseteq R_k}} S_e\Big)$$
$$\leq \mathrm{E}\Big(\bigwedge_{\substack{h \\ \text{vertex of } G^\perp}} \bigwedge_{\substack{k \\ h \in R_k \setminus T_k}} \bigvee_{\substack{e=(h,B) \\ e \notin N, B \subseteq R_k}} S_e\Big)$$
$$= \mathrm{E}\Big(\bigwedge_{\substack{h \\ C_h \neq \emptyset}} \bigwedge_{k \in C_h} \bigvee_{\substack{e=(h,B) \\ e \notin N, B \subseteq R_k}} S_e\Big) \tag{15}$$
$$= \prod_{\substack{h \\ C_h \neq \emptyset}} \mathrm{E}\Big(\bigwedge_{k \in C_h} \bigvee_{\substack{e=(h,B) \\ e \notin N, B \subseteq R_k}} S_e\Big)$$
$$= \prod_{\substack{h \\ C_h \neq \emptyset}} \mathrm{E}\Big(\bigwedge_{k \in C_h} \bigvee_{\substack{e=(h,B) \in A_h \\ B \subseteq R_k}} S_e\Big)$$

The expression above has the form $\prod_h \mathrm{E}\, \Psi_h$. We rewrite Ψ_h, by essentially enumerating all of its models and checking if they satisfy Ψ_h. The result is the following equivalent form:

$$\bigvee_{\substack{E_1 \\ E_1 \subseteq A_h \\ \forall k \in C_h,\ E_1 \cap D_k \neq \emptyset}} \Big(\Big(\bigwedge_{e \in E_1} S_e\Big) \wedge \Big(\bigwedge_{e \in A_h \setminus E_1} \bar{S}_e\Big)\Big)$$

and so

$$\mathrm{E}\, \Psi_h = \sum_{\substack{E_1 \\ E_1 \subseteq A_h \\ \forall k \in C_h,\ E_1 \cap D_k \neq \emptyset}} \prod_{e \in E_1} \mathrm{E}\, S_e \prod_{e \in A_h \setminus E_1} \mathrm{E}\, \bar{S}_e \tag{16}$$

Finally, we multiply the inequality (15) on both sides by $\prod_{e \in N} \mathrm{E}\, \bar{S}_e$, plug in (16), and use Lemma 24. □

Note that the upper bound is tight if $G^\perp$ has no cycles and therefore all loops are trivial.

Proof of the Lower Bound in Theorem 7. By Lemma 24,

$$\Pr\Big(\bigcap_{k=1}^n (R_k = \mathcal{R}_{\mathbf{H}} T_k)\Big) =$$
$$\prod_{e \in N} \mathrm{E}\, \bar{S}_e \cdot \mathrm{E}\Big(\bigwedge_{\substack{L \\ \text{loop of } G^\perp}} \bigwedge_{\substack{k \\ L \subseteq R_k \setminus T_k}} \bigvee_{\substack{e=(h,B) \\ e \in J_{G^\perp}(L) \setminus N \\ B \subseteq R_k}} S_e\Big)$$

Thus, the main part of the lemma follows if we show that

$$\bigwedge_{\substack{h \\ C_h \neq \emptyset}} \bigvee_{\substack{E_1 \\ E_1 \subseteq A_h \\ \forall k \in C_h,\ E_1 \cap F_k \neq \emptyset}} \Big(\Big(\bigwedge_{e \in E_1} S_e\Big) \wedge \Big(\bigwedge_{e \in A_h \setminus E_1} \bar{S}_e\Big)\Big) \tag{17}$$

implies

$$\bigwedge_{\substack{L \\ \text{loop of } G^\perp}} \bigwedge_{\substack{k \\ L \subseteq R_k \setminus T_k}} \bigvee_{\substack{e=(h,B) \\ e \in J_{G^\perp}(L) \setminus N \\ B \subseteq R_k}} S_e \tag{18}$$

To show this implication, we will show that a fixed but arbitrary conjunct of (18) holds, assuming that (17) holds. A conjunct of (18) is determined by a loop L_0 and an index k_0. The idea is to show that loop L_0 is justified via its vertex that is closest to T_{k_0}.

Since L_0 and k_0 determine a conjunct of (18), we know that $L_0 \subseteq R_{k_0} \setminus T_{k_0}$. We need to find an arc $e = (h, B)$ such that

$$e \in J_{G^\perp}(L_0) \setminus N, \quad B \subseteq R_{k_0}, \quad \text{and} \quad S_e = 1. \tag{19}$$

Since L_0 is not empty and $L_0 \subseteq R_{k_0} \subseteq \mathcal{R}_{G^\perp} T_{k_0}$, we can choose $h \in L_0$ such that $d_{T_{k_0}}(h)$ is minimum. Since $h \in L_0 \subseteq R_{k_0} \setminus T_{k_0}$, we have that

$$k_0 \in C_h.$$

This lets us instantiate (17) with h, and derive that for some subset E_1 of A_h,

$$E_1 \cap F_k \neq \emptyset \text{ for all } k \in C_h \quad \text{and} \quad S_e = 1 \text{ for all } e \in E_1 \tag{20}$$

Since $k_0 \in C_h$, the first conjunct implies that $E_1 \cap F_{k_0} \neq \emptyset$. Thus, there exists an arc $e_0 = (h_0, B_0)$ in $E_1 \cap F_{k_0}$, and it satisfies the following conditions:

1. the head h_0 of e_0 is h;
2. e_0 is not in N;
3. $B_0 \subseteq R_{k_0}$; and
4. e_0 is a forward arc with respect to T_{k_0}.

Since e_0 is a forward arc w.r.t. T_{k_0} and h has the minimal distance from T_{k_0} among all the vertices in L_0,

$$e_0 \in J_{G^\perp}(L_0)$$

Also, by the second conjunct in (20),

$$S_{e_0} = 1$$

From what we have just shown follows that e_0 is the desired arc; it satisfies the requirements in (19). □

Note that the lower bound and the upper bound coincide if $D_k \cap A_h = F_k \cap A_h$ for all k and h. In this case, both bounds are tight.

B. Proofs for Results in Section 6

Lemma 13. *Consider an instance of the action scheduling problem (Definition 12). Assume the success probabilities of the actions are independent. A permutation σ has the minimal cost $C(\sigma)$ if and only if for all $1 \leq i, j \leq m$,*

$$i \leq j \quad \Rightarrow \quad \frac{c_{\sigma(i)}}{p_{\sigma(i)}} \leq \frac{c_{\sigma(j)}}{p_{\sigma(j)}}. \tag{6}$$

Proof. Pick an arbitrary permutation σ. We will study the effect of one transposition $(i \leftrightarrow i+1)$ on the cost. Let $\sigma' = \sigma \circ (i \leftrightarrow i+1)$; in other words

$$\sigma'(j) = \begin{cases} \sigma(i+1) & \text{if } j = i \\ \sigma(i) & \text{if } j = i+1 \\ \sigma(j) & \text{otherwise} \end{cases}$$

Observe that $q_k(\sigma)$ and $q_k(\sigma')$ differ for only *one* value of k:

$$q_k(\sigma') = \begin{cases} q_i(\sigma)(1 - p_{\sigma(i+1)}) & \text{if } k = i+1 \\ q_k(\sigma) & \text{otherwise} \end{cases}$$

Also notice that $q_k(\sigma) \neq 0$ and $q'_k(\sigma) \neq 0$ for all k. The difference in cost between σ' and σ is

$$\begin{aligned} C(\sigma') - C(\sigma) = {} & q_i(\sigma')c_{\sigma'(i)} + q_{i+1}(\sigma')c_{\sigma'(i+1)} \\ & - q_i(\sigma)c_{\sigma(i)} - q_{i+1}(\sigma)c_{\sigma(i+1)} \\ = {} & q_i(\sigma)c_{\sigma(i+1)} + q_i(\sigma)(1 - p_{\sigma(i+1)})c_{\sigma(i)} \\ & - q_i(\sigma)c_{\sigma(i)} - q_i(\sigma)(1 - p_{\sigma(i)})c_{\sigma(i+1)} \\ = {} & q_i(\sigma)(p_{\sigma(i)}c_{\sigma(i+1)} - p_{\sigma(i+1)}c_{\sigma(i)}). \end{aligned}$$

Thus,

$$\frac{C(\sigma') - C(\sigma)}{q_i(\sigma)} = p_i c_{i+1} - p_{i+1} c_i$$

where p_i denotes $p_{\sigma(i)}$, and c_i denotes $c_{\sigma(i)}$, for the fixed permutation σ.

All that remains is to interpret the result of these calculations. For the left-to-right direction, assume that σ has the minimal cost. Also, for the sake of contradiction, suppose that there exist i and j such that $i \leq j$ and $c_i/p_i > c_j/p_j$. Then, there must also exist an i such that $c_i/p_i > c_{i+1}/p_{i+1}$, which is equivalent to

$$p_i c_{i+1} - p_{i+1} c_i < 0.$$

Thus, the previous calculation shows that σ' would have a lower cost than σ. This contradicts the assumption that σ has the minimal cost.

For the right-to-left direction, pick σ and σ' that satisfy the RHS of (6). Then, we can convert σ to σ' by composing σ with a sequence of transpositions $i \leftrightarrow i+1$ for i such that

$$\frac{c_i}{p_i} = \frac{c_{i+1}}{p_{i+1}}.$$

Then the previous computation shows that such composition leaves the cost unchanged. Thus, σ and σ' have the same cost. But by what we have already shown, there should be at least one σ'' that satisfies the RHS of (6) and have the minimal cost. This implies that all of σ, σ' and σ'' are optimal. □

Lemma 18. *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 $F(G^a_{\rightarrow})$ be defined as in (9). There is a bijection between the models M of Φ and the elements (a', H) of $F(G^a_{\rightarrow})$. According to this bijection,*

$$\begin{aligned} M \cap X_E(G^a_{\rightarrow}) &= X_E(H) \\ M \cap X_V(G^a_{\rightarrow}) &= X_V\big(\mathcal{R}_H(T(a, a'))\big) \end{aligned}$$

Proof sketch. Let

$$\begin{aligned} G' &:= \{\, (h, e \cup B) \mid e = (h, B) \in G^a_{\rightarrow} \,\} \\ S' &:= \{\, e \mid e \in G^a_{\rightarrow} \,\} \end{aligned}$$

Because $G^a_{\rightarrow}$ has no cycles by construction, G' does not have cycles, either. We have that

$$\Big(\underset{e \in G^a_{\rightarrow}}{\exists} y_e \; (\Phi_1 \wedge \Phi_2)\Big) \quad \Leftrightarrow \quad \Big(\phi^{S' \cup P_0(a) \cup P_1(a)}(G')\Big)$$

where the latter uses the definition in (10). Thus, we can apply Corollary 21. Finally, note that Φ_3 ensures that $a' > a$ and $q \in \mathcal{R}_H(T(a, a'))$. □