Abstract Interpretation with Confidence

Quantifying the Precision of Dataflow Analysis with Probabilities

YUANFENG SHI, Peking University, China
ZIYUE JIN, Peking University, China
XIN ZHANG*, Peking University, China

Abstract interpretation has served as a foundational framework for static program analysis, enabling the over approximation of program semantics to be sound (i.e., no false negatives) but often at the cost of false alarms due to incompleteness. Although prior efforts to address false alarms have incorporated probabilistic techniques to compute confidence values for alarms, these methods are largely guided by empirical intuitions and lack a theoretical foundation. This paper bridges this gap by proposing a principled framework to quantify the confidence in results produced by a dataflow analysis based on abstract interpretation. Specifically, we define the problem as calculating the probability of the abstract interpreter being locally complete for a sampled program from the distribution of programs consistent with such abstract interpretation. By proposing a compositional denotational semantics $\langle\!\langle \cdot \rangle\!\rangle$, we derive the distribution of program outputs to compute those confidence probabilities. Moreover, to ensure tractability, we propose another denotational semantics $\langle\!\langle \cdot \rangle\!\rangle^{lc}$ that under-approximates $\langle\!\langle \cdot \rangle\!\rangle$. The paper proves both the correctness of the two semantics, and therefore establishes a theoretical foundation for quantifying the precision of static program analysis with probabilities.

CCS Concepts: • **Theory of computation** → **Semantics and reasoning**; **Program analysis**; **Abstraction**.

Additional Key Words and Phrases: Abstract Interpretation, Program Analysis, Local Completeness.

ACM Reference Format:
Yuanfeng Shi, Ziyue Jin, and Xin Zhang. 2026. Abstract Interpretation with Confidence: Quantifying the Precision of Dataflow Analysis with Probabilities. *Proc. ACM Program. Lang.* 10, PLDI, Article 273 (June 2026), 25 pages. https://doi.org/10.1145/3808351

1 Introduction

Abstract interpretation [11, 13, 14], a theoretical framework for approximating program semantics, has been the cornerstone of static program analyses in the past decades. As any non-trivial program analysis problem is theoretically undecidable in the worst case [40], the key idea of abstract interpretation is to over-approximate the program semantics so that the computed states are a super-set of the reachable program states. Such a property ensures that a program analysis based on abstract interpretation does not miss any program defect of the kind it targets (i.e., no false negatives), which is referred to as *soundness*. While static analyses based on abstract interpretation have made a significant impact in both academia and industry [18, 36, 41, 42], a major problem is that a sound analysis can produce false alarms (i.e., is *incomplete*). In other words, whether a

*Corresponding author.

Authors' Contact Information: Yuanfeng Shi, Key Laboratory of High Confidence Software Technologies (Peking University), Ministry of Education and School of Computer Science, Peking University, China, friedrich22@stu.pku.edu.cn; Ziyue Jin, School of Computer Science, Peking University, China; Xin Zhang, Key Laboratory of High Confidence Software Technologies (Peking University), Ministry of Education and School of Computer Science, Peking University, China, xin@pku.edu.cn.

ACM 2475-1421/2026/6-ART273
https://doi.org/10.1145/3808351

reported alarm represents an actual bug or an undesirable program property is uncertain, which greatly restricts the application of static program analysis tools.

While it is impossible to design an analysis that is both sound and complete in general, an emerging trend to address the issue from a practical angle is to statistically determine how likely a reported bug is true [1, 2, 22, 24, 26, 27, 29, 32, 34, 38, 39, 45]. The main idea behind these approaches is to integrate probabilities to quantify the uncertainty in the analysis results, and hence for each alarm they can compute a confidence value which measures how likely that the alarm represents a real bug. A direct typical application is to set a threshold for these confidence values, and hence they can filter out those low-probability alarms. As a result, those works can significantly reduce the number of false alarms presented to the user. These confidence values can also be used for alarm clustering [28, 30], ranking [26, 27], and classification [22, 34, 47].

Despite the fact that these works have produced impressive empirical results, a major issue is that the integration with probabilities lacks a theoretical foundation, leaving many important questions unanswered. Most importantly, what are the sources of randomness so that one can assign meanings to the alarm probabilities by defining their distribution? Prior works either do not provide such a definition [26, 27, 34, 39] or attribute randomness to drawing program inputs from a given distribution [38]. The latter essentially solves a different problem: How likely a reported alarm can be triggered by a randomly sampled input? However, the uncertainty that causes analysis imprecision comes from abstracting program semantics rather than from the random inputs.

Secondly, even the distribution of alarms is defined based on the source of randomness, how can one describe a procedure to compute the probabilities of alarms in a principled way? Ideally, if the procedure cannot precisely compute the alarm probabilities, it should approximate them in a principled way (e.g., providing upper bounds or lower bounds). Moreover, the computation should be compositional so that it can be done in an efficient way.

To address the challenges, we propose a novel framework to quantify the confidence of results produced by an important class of program analysis based on abstract interpretation–dataflow analysis. The key idea is that, given the execution of an abstract interpreter, we consider the distribution of all programs whose abstract interpretation matches the execution and use the probability of the abstract interpreter being complete on the distribution as the confidence of the execution. Intuitively, an abstract interpreter approximates any program as an abstract program and an abstract program corresponds to multiple programs. As an analogy, given a program analysis, an expert has run the analysis on many programs and inspected whether the alarms are true. Now, they run the analysis on a new program and by considering the past runs which have the same abstract interpretation, they can gauge how likely the produced alarms are true.

Following the setup by Bruni et al. [4], we assume the alarm specification is expressible in the abstract domain. For example, in an interval analysis, the property of $v > 0$ can be expressed as $v \in [0, +\infty)$. Under this assumption, an alarm being true can be implied by that the abstract interpretation processing deriving the alarm is complete. Formally, we judge whether the abstract interpreter is locally complete for a program [4], that is whether the abstract interpreter is complete on the program for a given input set (e.g., the set of all inputs).[1]

With the above setups, we formalize our core problem as given the abstract interpretation of a program, an input set and a proper distribution of programs matching the abstract interpretation, calculating the probability of the abstract interpreter being locally complete on a program that is drawn from the distribution for the input set.

[1]A related concept is global completeness which means that analysis is complete for arbitrary input set, which is shown to be very rare to be satisfied in practice [19, 20].

To compute the above property, we propose a general compositional denotational semantics $\langle\!\langle \cdot \rangle\!\rangle$ to derive the distribution of the outputs for the programs matching the given abstract interpretation process, which in turn can derive the probability of the abstract interpreter being local complete on the input. Given the distribution of each atomic statement, our semantics computes the distribution of program outputs for the given abstract interpretation through rules of sequential composition, conditional branching, and loops. We have proven the correctness of our denotational semantics.

Moreover, to make the computation tractable, we provide another denotational semantics $\langle\!\langle \cdot \rangle\!\rangle^{lc}$ which under-approximates $\langle\!\langle \cdot \rangle\!\rangle$. The key difference is that $\langle\!\langle \cdot \rangle\!\rangle^{lc}$ conservatively tracks the concrete program states that are complete and discards states that the abstract interpreter is potentially incomplete on. As a result, the probability calculated by $\langle\!\langle \cdot \rangle\!\rangle^{lc}$ is a lower bound of the real value. We have proven the soundness of $\langle\!\langle \cdot \rangle\!\rangle^{lc}$ with respect to $\langle\!\langle \cdot \rangle\!\rangle$.

In summary, the paper makes the following contributions:

- We are the first to provide a principled framework to measure the confidence of a dataflow analysis by formalizing the problem as calculating the probability of the abstract interpreter being locally complete on a program that is sampled from the distribution of programs matching the abstract interpretation process.
- We have defined a denotational semantics to compositionally calculate the distribution of outputs for programs matching an abstract interpretation process, which in turn can derive the probability of the abstract interpreter being complete, and have proven its correctness.
- For tractability, we have defined another denotational semantics through under-approximation to calculate lower bounds of probabilities of abstract interpretation being locally complete and proven its soundness.

2 Overview

This section gives an informal overview of our approach with a standard sign analysis [13] that checks whether the result of a program can be zero.

2.1 The Programs and the Sign Analysis

The programs we consider involve one integer variable x, which serves as both the input and output. The programs are sequences of statements that assign arithmetic expressions formed with x and arbitrary integers. For simplicity, we only consider operations $(+, \times)$ in the arithmetic expressions as $-$ can be simulated with $+$ and negative numbers, and $/$ is not closed for integers. An example program s is defined as follows: $s \triangleq x := x + 1;\ x := x \times 2$. The program state at each program point is a set of integers, which are different values of x each of which corresponds to its different initial values. Thus, the concrete domain is $\mathcal{P}(\mathbb{Z})$, which denotes the powerset of integers. The goal is to check whether x at the end of these programs can be zero, which simulates the practical usage of sign analyses to check whether a program has division-by-zero errors.

The widely used sign analysis [3, 4, 13, 14, 20] is flow-sensitive and tries to compute possible values of x at each program point by over-approximating the program states. Concretely, we define an abstract domain $A = \{\emptyset, \mathbb{Z}_{<0}, \mathbb{Z}_{=0}, \mathbb{Z}_{>0}, \mathbb{Z}_{\leq 0}, \mathbb{Z}_{\neq 0}, \mathbb{Z}_{\geq 0}, \mathbb{Z}\}$. The eight values represent sets of integers which satisfy the condition in their subscripts respectively. For example, $\mathbb{Z}_{>0}$ represents the set of all the positive integers. They can be defined via the concretization function $\gamma : A \to \mathcal{P}(\mathbb{Z})$, which maps an element in A to its corresponding set of integers. To model program executions in the abstract domain A, we begin by defining the abstraction function $\alpha : \mathcal{P}(\mathbb{Z}) \to A$, which maps the program states to their corresponding abstractions in A. Here, α maps a set of integers to the most precise element in A that over-approximates its concrete values. For example, both $\mathbb{Z}$ and $\mathbb{Z}_{\neq 0}$ can over-approximate the set $\{-1, 1\}$, but $\mathbb{Z}_{\neq 0}$ is more precise as 0 is not in $\{-1, 1\}$, and

hence $\alpha(\{-1, 1\}) = \mathbb{Z}_{\neq 0}$. Next we describe the abstract operators $(+_A, \times_A)$ over the abstract domain A for arithmetic operations $(+, \times)$, which in turn can be used to define abstract transfer functions for program statements. For all $a_1, a_2 \in A$, we define the results of $a_1 +_A a_2$ and $a_1 \times_A a_2$ as the most precise element in A which can over-approximates the concrete executions $\gamma(a_1)+\gamma(a_2)$ and $\gamma(a_1)\dot{\times}\gamma(a_2)$ respectively. Here $\dot{+}$ and $\dot{\times}$ are extended to $\mathcal{P}(\mathbb{Z})$ by performing operations pairwise.

Then we can simulate the program execution in A. For example, given an input set $I = \{0, 1\} \in \mathcal{P}(\mathbb{Z})$, the execution of s can be approximated as $O_A = (\alpha(I) +_A \mathbb{Z}_{>0}) \times_A \mathbb{Z}_{>0} = \mathbb{Z}_{>0}$, where O_A is the abstract output. As the concrete result on the input set $I = \{0, 1\}$ is $\{2, 4\}$, in this scenario the sign analysis precisely captures all possible signs of the program outputs. The analysis raises an alarm when x can be 0 by checking whether $O_A \in \{\mathbb{Z}_{=0}, \mathbb{Z}_{\leq 0}, \mathbb{Z}_{\geq 0}, \mathbb{Z}\}$ holds.

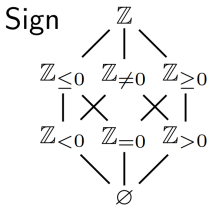

Fig. 1. Sign abstract domain

The sign analysis guarantees soundness, which ensures that the final abstract result will not miss any possible sign of the concrete outputs. As a result, the analysis will not miss any error and has no false negative. This property is reflected by the partial order $>_A$ in A depicted by Figure 1: If a_2 is higher than a_1 and there is a path between them, then $a_1 <_A a_2$, while $a_1 <_A a_2 \vee a_1 = a_2$ is simplified as $a_1 \leq_A a_2$. This implies, for the concrete output set $O \in \mathcal{P}(\mathbb{Z})$ and the abstract output O_A, $\alpha(O) \leq_A O_A$, which ensures the soundness.

However, O_A may introduce spurious signs that do not exist in concrete executions, leading to false positives. For example, when the input set I is $\{-3, -2, 1\}$, $O_A = (\mathbb{Z}_{\neq 0} +_A \mathbb{Z}_{>0}) \times_A \mathbb{Z}_{>0} = \mathbb{Z} \times_A \mathbb{Z}_{>0} = \mathbb{Z}$. But the concrete output set O is $\{-4, -2, 2\}$, and $\alpha(O) = \mathbb{Z}_{\neq 0}$. Here, the abstract result is imprecise, since 0 is not a possible output, leading to a false alarm.

The existence of false positives is universal for program analyses based on abstract interpretation, and greatly hinders the usefulness of analyses in practice. In the rest of the section, we describe how our approach addresses this problem for the sign analysis by assigning confidence to its alarms.

2.2 Assigning Probabilities to Quantify the Confidence of the Analysis Results

We aim to use probabilities to quantify the confidence of alarms produced by the sign analysis in Section 2.1. Our key idea is to formulate the problem as calculating the probability of the analysis being complete when it runs on a program that is sampled from the distribution of programs matching the current abstract interpretation.

The first part of our idea is to define the source of randomness as sampling from a distribution of programs matching the current abstract interpretation. To begin with, since the analysis does not abstract the control flow, the execution of the underlying abstract interpreter on the example program s can be modeled as running an abstract program S by replacing each statement with its abstract transfer function: $x := x +_A \mathbb{Z}_{>0}$; $x := x \times_A \mathbb{Z}_{>0}$. This abstract program first abstracts the program input set $I \in \mathcal{P}(\mathbb{Z})$ to $\alpha(I) \in A$, then sequentially executes those two abstract operators $+_A$ and $\times_A$. As a result, the abstract output O_A is computed as $(\alpha(I) +_A \mathbb{Z}_{>0}) \times_A \mathbb{Z}_{>0}$. We denote *Prog* as the set of all programs whose abstract interpretation matches such an abstract program: $Prog = \{s(i, j) : x := x + i;\ x := x \times j \mid i, j \in \mathbb{Z}, i > 0, j > 0\}$. Here, $s(i, j)$ is a symbol which denotes the program $x := x + i;\ x := x \times j;$. The key idea to introduce randomness is to consider the distribution of all such programs. In practice, based on historical analysis application, the distribution can be designed to approximate empirical frequencies for these programs. For simplicity, in this overview, we just assign the same probability to all possible programs and construct a uniform distribution $Dist : Prog \to [0, 1]$ where $Dist(s(i, j)) = \frac{1}{N^2}$ $(\forall 1 \leq i, j \leq N)$. Note that *Dist* is a function mapping a program in Prog to a real number between 0 and 1 representing

its probability, and N is the maximum positive integer we considered. We set this threshold N in order to restrict the sample space of programs to a finite set.

The second part of our idea is to use the probability of the abstract program S being complete on the distribution as the confidence of the reported alarm. We inherit the notion of *local completeness* [4]: for a given input set I, a concrete output set O, and an abstract output O_A, we say that S is locally complete on I iff $\alpha(O) = O_A$. For program $s(i, j)$, its concrete output set is computed as: $O = (I \dot{+} i) \dot{\times} j$. Then given an input set I, we denote the probability of S being locally complete on I as $LC(S, I, Dist)$ and compute it as: $LC(S, I, Dist) = \sum_{i=1}^{N} \sum_{j=1}^{N} Dist(s(i, j)) \cdot \delta(\alpha(O) = O_A)$, where $\delta(\Psi)$ evaluates to 1 if Ψ is true and evaluates to 0 if Ψ is false. Since all programs in *Prog* have the same probability, $LC(S, I, Dist)$ can be regarded as the proportion of programs which satisfy the local completeness condition.

Note that $LC(S, I, Dist)$ is actually a lower bound of the probability of the alarm of $x = 0$ being true, which is denoted as $AT(S, I, Dist)$. The reason is that an incomplete abstract interpretation can still derive true alarms (i.e., the spurious states are not the ones introducing the alarms). For example, consider the set of programs in *Prog* assuming the input $I = \{-3, -2, 1\}$. Then the abstract output O_A is $\mathbb{Z}$ and the concrete output $O = (I \dot{+} i) \dot{\times} j = \{(-3 + i) * j, (-2 + i) * j, (1 + i) * j\}$. Here, $\alpha(O) = \mathbb{Z}$ iff $(-2 + i) * j = 0$ where $i > 0, j > 0$. Then we can derive that $LC(S, I, Dist) = \frac{1}{N}$ as the analysis is only complete when $i = 2$. However, an alarm will be raised when $0 \in O$, and it occurs iff $(-2 + i) * j = 0$ or $(-3 + i) * j = 0$ which indicates $i = 2$ or $i = 3$. We can derive $AT(S, I, Dist) = \frac{2}{N}$. The reason for such under-estimation is that the program with $i = 3$ is not counted in computing $LC(S, I, Dist)$, as its abstract output $\mathbb{Z}$ is incomplete for $O = \{0, j, 4j\}$ where $\alpha(O) = \mathbb{Z}_{\geq 0}$.

To conclude, given the abstract program S, an input set I, we calculate the probability $LC(S, I, Dist)$ denoting that S is locally complete in a program drawn from the distribution of programs *Dist* that matches the abstract program. Such a probability $LC(S, I, Dist)$ can be used as a guide for the user of the sign analysis, as it measures how likely that the alarm represents a real bug. In our example, $LC(S, I, Dist) = \frac{1}{N}$ is too low for the user to admit that there is a real alarm.

2.3 A Denotational Semantics for Computing Output Distributions Compositionally

Despite that we provide a principled approach to measure analysis confidence in Section 2.2, it exhibits computational inefficiency in sampling all possible programs, particularly for complex abstract programs. For example, for the abstract program S,there are N^2 concrete programs in the sampling distribution *Dist*. This is because for each $x := x +_A Z_{>0}$ (or $x := x \times_A Z_{>0}$), it corresponds to N possible concrete executions: $x := x + i$ (or $x := x \times i$) where $i \in \{1, 2, \ldots, N\}$. This inefficiency is exponentially exacerbated with increasing the abstract program's length, as the sampling number becomes N^k when the length becomes k. We address the inefficiency problem through compositionality and under-approximation. Before introducing our full method, in this subsection, we first introduce a denotational semantics that computes the concrete output distribution corresponding to a program distribution in a compositional way. By comparing the abstract output with the concrete output distribution, we can directly derive the probability of the analysis being complete. This semantics serves as both basis for compositionally computing the probabilities and a correctness ground truth for our semantics through under-approximation introduced in the next subsection.

To achieve compositionality, we decompose each abstract program into atomic abstract programs (i.e., abstract transfer functions for each statement). In our example, we decompose the abstract program S into sequentially executed atomic abstract programs $\hat{S}_1 : x := x +_A \mathbb{Z}_{>0}$ and $\hat{S}_2 : x := x \times_A \mathbb{Z}_{>0}$, and they are equipped with uniforms distributions $\{\{\hat{S}_1\}\}$ and $\{\{\hat{S}_2\}\}$ on $Prog_1$ and $Prog_2$ respectively ($Prog_j = \{x := x \odot_j i \mid i = 1, 2, \ldots, N\}, \odot_1 = +, \odot_2 = \times$). The distribution *Dist* in Section 2.2 can be regarded as the joint distribution of $\{\{\hat{S}_1\}\}$ and $\{\{\hat{S}_2\}\}$: $Dist(s(i, j)) = \{\{\hat{S}_1\}\}$ ($x :=$

$x + i) * \{\{\hat{S}_2\}\}$ $(x := x \times j)$. Here for simplicity, we suppose that $\{\{\hat{S}_1\}\}$ and $\{\{\hat{S}_2\}\}$ are independent, and we will discuss the dependence between the distributions of atomic abstract programs in Section 4.1 and handle it in Section 5.

Next, we propose a denotational semantics $\langle\langle\cdot\rangle\rangle$, which is a map that assigns a mathematical object $\langle\langle S\rangle\rangle : \mathcal{P}(\mathbb{Z}) \to D(\mathcal{P}(\mathbb{Z}))$ called a denotation to each abstract program S. Here, $D(\mathcal{P}(\mathbb{Z}))$ denotes the probability function $\mathcal{P}(\mathbb{Z}) \to [0, 1]$. For each atomic abstract program $\hat{S}$, the denotation $\langle\langle \hat{S} \rangle\rangle$ maps an input set in $\mathcal{P}(\mathbb{Z})$ to the distribution of all possible outputs by sampling concrete programs from the corresponding $\{\{\hat{S}\}\}$. Semantics $\langle\langle \hat{S}_1 \rangle\rangle$ and $\langle\langle \hat{S}_2 \rangle\rangle$ are defined as follows:

$$\langle\langle \hat{S}_1 \rangle\rangle (X)(Y) \triangleq \begin{cases} \frac{1}{N} & \text{if } \exists\, 1 \le i \le N, Y = X \dotplus i \\ 0 & \text{otherwise} \end{cases} \quad \langle\langle \hat{S}_2 \rangle\rangle (X)(Y) \triangleq \begin{cases} \frac{1}{N} & \text{if } \exists\, 1 \le j \le N, Y = X \dot{\times} j \\ 0 & \text{otherwise} \end{cases}$$

For example, given the input set $I = \{-3, -2, 1\}$. Only for $O = \{-3 + k, -2 + k, 1 + k\}$ where $k \in \mathbb{Z}$ and $1 \le k \le N$, $\langle\langle \hat{S}_1 \rangle\rangle (I)(O) = \frac{1}{N}$.

On the other hand, for the non-atomic abstract program, its semantics can be **compositionally** expressed in terms of the semantics of all its comprising atomic abstract programs. Using S as an example, semantics $\langle\langle S\rangle\rangle$ is computed as follows:

$$\langle\langle S\rangle\rangle (X_1)(X_2) \triangleq \sum_{X_3 \in \mathcal{P}(\mathbb{Z})} \langle\langle \hat{S}_1 \rangle\rangle (X_1)(X_3) \cdot \langle\langle \hat{S}_2 \rangle\rangle (X_3)(X_2) \qquad \forall\, X_1, X_2 \in \mathcal{P}(\mathbb{Z}) \tag{1}$$

Since the sample spaces for $\{\{\hat{S}_1\}\}$ and $\{\{\hat{S}_2\}\}$ are both countable, in Equation (1) the number of non-zero terms is also countable, and hence the sum is well-defined.

As a result, $\langle\langle S\rangle\rangle (I)$ records the distribution of all possible output integer sets using sampled programs on the input set $I = \{-3, -2, 1\}$:

$$\langle\langle S\rangle\rangle (I)(O) = \begin{cases} \frac{1}{N^2} & \text{if } \exists i, j \in \{1, 2, \ldots, N\} \text{ s.t. } O = \{(-3 + i) * j, (-2 + i) * j, (1 + i) * j\} \\ 0 & \text{otherwise} \end{cases}$$

It is equal to the result by directly sampling from *Dist*, and we can calculate $LC(S, I, Dist)$ (the probability of S being local complete on I) using $\langle\langle S\rangle\rangle (I)$ by: $LC(S, I, Dist) = \sum_{O \in \mathcal{P}(\mathbb{Z})} \langle\langle S\rangle\rangle (I)(O) \cdot \delta(\alpha(O) = O_A)$. By pre-computing the semantics $\langle\langle \hat{S} \rangle\rangle$ for each atomic abstract program $\hat{S}$, we can compute $LC(S, I, Dist)$ without sampling. Moreover, the computation of $\langle\langle\cdot\rangle\rangle$ is modular, as the result of $\langle\langle \hat{S} \rangle\rangle$ for each atomic abstract program can be reused.

In Section 5, we introduce our full denotational semantics including treatment to support composition in more complex program structures such as branching statements and while loops.

2.4 Under-Approximation for Efficiently Computing Completeness Probabilities

In order to make the computation of $LC(S, I, Dist)$ more tractable, we provide another denotational semantics $\langle\langle\cdot\rangle\rangle^{lc}$ that under-approximates $\langle\langle\cdot\rangle\rangle$ and directly computes the probability of local completeness. The key difference is that $\langle\langle\cdot\rangle\rangle^{lc}$ conservatively tracks program samples on which the abstract interpreter is complete on and discard program samples on which the abstract interpreter is potentially incomplete at each step. As a result, the probability calculated by $\langle\langle\cdot\rangle\rangle^{lc}$ is a lower bound of the real value. Note that this lower-bound approximation is consistent with the fact that the probability of local completeness is itself a lower bound on the probability that the alarm is true, ensuring that the final probability estimate remains a lower bound on the alarm's actual probability.

For each atomic abstract program $\hat{S}$, with a slight abuse of notation, we denote its abstract result of the input $\alpha(I)$ as $\hat{S}(\alpha(I))$ and its semantics $\langle\langle \hat{S} \rangle\rangle^{lc}$ can be defined as:

$$\langle\langle \hat{S} \rangle\rangle^{lc} (X)(Y) \triangleq \langle\langle \hat{S} \rangle\rangle (X)(Y) \cdot \delta(\alpha(Y) = \hat{S}(\alpha(X))) \tag{2}$$

Here, $\left\langle\!\left\langle \hat{S} \right\rangle\!\right\rangle^{lc}$ returns the probability for a concrete input-output set pair on which the analysis is complete for $\hat{S}$. For example, for $I = \{-3, -2, 1\}$, $\left\langle\!\left\langle \hat{S}_1 \right\rangle\!\right\rangle (I) = \langle\!\langle x := x +_A \mathbb{Z}_{>0} \rangle\!\rangle (I)$ is computed as:

$$\left\langle\!\left\langle \hat{S}_1 \right\rangle\!\right\rangle (I)(O) = \begin{cases} \frac{1}{N} & \text{if } O = \{-1, 0, 3\} \\ 0 & \text{otherwise} \end{cases} \tag{3}$$

This is because $\hat{S}_1(\alpha(I)) = \hat{S}_1(\mathbb{Z}) = \mathbb{Z} +_A \mathbb{Z}_{>0} = \mathbb{Z}$, and during all possible outputs $\{-3+i, -2+i, 1+i\}$ where $i \in \{1, 2, \ldots, N\}$, only $O = \{-1, 0, 3\}$ satisfies $\alpha(O) = \mathbb{Z}$. While for the non-atomic abstract program, its semantics is still compositionally computable. Using S as an example:

$$\langle\!\langle S \rangle\!\rangle^{lc} (X_1)(X_2) \triangleq \sum_{X_3 \in \mathcal{P}(\mathbb{Z})} \left\langle\!\left\langle \hat{S}_1 \right\rangle\!\right\rangle^{lc} (X_1)(X_3) \cdot \left\langle\!\left\langle \hat{S}_2 \right\rangle\!\right\rangle^{lc} (X_3)(X_2) \qquad \forall\, X_1, X_2 \in \mathcal{P}(\mathbb{Z}) \tag{4}$$

Here, $\langle\!\langle \cdot \rangle\!\rangle^{lc}$ ensures that the results of the two subprograms $\hat{S}_1$ and $\hat{S}_2$ are locally complete on the corresponding inputs, and then the results in $\langle\!\langle S \rangle\!\rangle^{lc}$ are also local complete. Therefore, a lower bound of $LC(S, I, Dist)$ can be calculated as $\sum_{O \in \mathcal{P}(\mathbb{Z})} \langle\!\langle S \rangle\!\rangle^{lc} (I)(O)$. Note this is a lower bound instead of an exact computation because the composition of an incomplete statement with another statement can still be complete. As an example, consider an abstract program $x := x +_A \mathbb{Z}_{\geq 0};\ x := x \times_A \mathbb{Z}_{=0}$ where the analysis is always complete by returning $\mathbb{Z}_{=0}$ and can be incomplete for the first statement.

With this under-approximation, this removal of intermediate outputs in compositional computation enables semantics $\langle\!\langle \cdot \rangle\!\rangle^{lc}$ to lower computational costs, as shown in computing $\langle\!\langle S \rangle\!\rangle^{lc} (\{-3, -2, 1\})$:

$$\langle\!\langle S \rangle\!\rangle^{lc} (\{-3, -2, 1\})(O) = \begin{cases} \frac{1}{N^2} & \text{if } \exists j \in \{1, 2, \ldots, N\} \text{ s.t. } O = \{-1 * j, 0, 3 * j\} \\ 0 & \text{otherwise} \end{cases}$$

Since only $O = \{-1, 0, 3\}$ can satisfy that $\left\langle\!\left\langle \hat{S}_1 \right\rangle\!\right\rangle^{lc} (\{-3, -2, 1\})(O) \neq 0$, the summation in Equation (4) reduces to a single-term evaluation $\left\langle\!\left\langle \hat{S}_1 \right\rangle\!\right\rangle^{lc} (\{-3, -2, 1\})(\{-1, 0, 3\}) * \left\langle\!\left\langle \hat{S}_2 \right\rangle\!\right\rangle^{lc} (\{-1, 0, 3\})(X_2)$. In contrast, to compute $\langle\!\langle S \rangle\!\rangle (\{-3, -2, 1\})$, it requires iterating N possible $X_3 = \{-3+i, -2+i, 1+i\}$ ($i \in \{1, 2, \ldots, N\}$). Moreover, in this case, $\langle\!\langle S \rangle\!\rangle^{lc} (\{-3, -2, 1\})(O)$ returns the exact result instead of a lower bound while improving the efficiency.

3 Preliminaries

This section establishes the required background: basic mathematical notations, probability notations, the syntax and semantics of the programs studied in this paper, the theory of abstract interpretation, and the formalization of the class of analysis we focus on—namely, dataflow analysis.

3.1 Basic Notations

Lattices. A complete lattice $\langle C, \leq_C, \vee_C, \wedge_C, \top_C, \bot_C \rangle$ is denoted by C, with partial order $\leq_C$, lub (least upper bound) $\vee_C$, glb (greatest lower bound) $\wedge_C$, greatest element $\top_C$ and least element $\bot_C$. We use $\mathcal{P}(X)$ to denote the powerset complete lattice over a set X ordered by inclusion, in which case the standard symbols $\subseteq$, $\cup$, etc., denote its order-theoretic structure.

Notations Related to Functions. If there exists $f : X \to Y$ then f is overloaded to denote its lifting $f : \mathcal{P}(X) \to \mathcal{P}(Y)$ (often called collecting) to powersets: $f(S) \triangleq \{f(x) \mid x \in S\}$. Given two functions $f : X \to Y$ and $g : Y \to Z$ we denote $g \circ f$ as their composition. For the function $f : X \to X$ and $\forall n \in \mathbb{N}$ we let $f^n : X \to X$ be defined inductively as: $f^0 \triangleq id_X$ and $f^{n+1} \triangleq f^n \circ f$. Here id_X is the identity function on a set X and the subscript is omitted when given by the context. Given complete lattices C, C_1 and C_2, a function $f : C_1 \to C_2$ is monotone if for all $x, y \in C_1$ such

that when $x \leq_{C_1} y$, f preserves the order, i.e. $f(x) \leq_{C_2} f(y)$. A function $f : C_1 \to C_2$ is additive if for all $x, y \in C_1$, $f(x \vee_{C_1} y) = f(x) \vee_{C_2} f(y)$. For $f, g \in C \to C$ the partial order $f \leq_C g$ is defined as that for all $c \in C$, $f(c) \leq_C g(c)$, and we denote $f \vee_C g$ as $\lambda c. f(c) \vee_C g(c)$. The least fixpoint of a function $f : C \to C$ is denoted, when it exists, by $\mathsf{lfp}\,(f)$.

3.2 Probability Notations

In this subsection, we recall several basic notions from probability theory and introduce notations used throughout the paper. For simplicity, we consider elementary probabilities over *countable* spaces without losing generality. A probability space is a countable set X equipped with a *probability mass function* (pmf) $m : X \to [0, 1]$ such that $m(x) \geq 0$ for all $x \in X$, and $\sum_{x \in X} m(x) = 1$. We denote by $D(X)$ the set of all pmfs over X, and we also refer to a pmf as a distribution and write $x \sim m$ to mean that x is drawn from X according to m. The *support* of m is the set of elements that can occur with nonzero probability: $supp(m) = \{x \in X \mid m(x) > 0\}$.

In this paper, we consider the probability space as the set (denoted as *Prog*) of all syntactically valid programs, which will be defined later in Section 3.3. Semantics $\{\{\cdot\}\}$ assigns a pmf for each abstract execution. Therefore, in Section 2, $\{\{S_1\}\}$ is actually defined as: $\{\{S_1\}\}(s) = \frac{1}{N}$ if $s \in Prog_1$ and $\{\{S_1\}\}(s) = 0$ otherwise. Moreover, it is easy to infer that $supp(\{\{S_1\}\}) = Prog_1$

Given an event (predicate) $\Psi : X \to \{\mathsf{True}, \mathsf{False}\}$, its probability under m is $\Pr_{x \sim m}(\Psi) = \sum_{x \in X} m(x) \cdot \delta(\Psi(x))$, where $\delta(\Psi(x))$ is the indicator (1 if $\Psi(x)$ holds and 0 otherwise). In our setting, "confidence" values are exactly probabilities of certain events (e.g., local completeness) computed over a sample space induced by an abstract execution.

3.3 Programs

In our paper, we consider a language of standard imperative programs whose syntax is defined as:

$$\textsc{Program} \qquad Prog \ni s ::= e \in BaseCmd \mid s; s \mid s \oplus s \mid s^*$$

The set *BaseCmd* consists of basic commands, such as assignments and Boolean guards. This language also has the standard compound constructs: sequential composition $(s; s)$, non-deterministic choice $(s \oplus s)$, and Kleene iteration (s^*), which are similar to those in previous works [4, 7, 37, 46, 48]. It is general enough to cover other programming paradigms that include, e.g., nondeterministic and probabilistic computations, and equational systems such as Kleene algebras with tests [25].

For simplicity, the basic commands we currently consider are standard basic commands used in deterministic while-programs, which include no-op instructions, assignments, Boolean guards, integer expressions, and are no runtime errors. The syntax of the basic commands is shown below.

$$\begin{array}{ll}
\textsc{Arithmetic Expression} & AExp \ni a ::= v \in Var \mid n \in \mathbb{Z} \mid a + a \mid a - a \mid a \times a \\
\textsc{Boolean Expression} & BExp \ni b ::= \mathsf{True} \mid \mathsf{False} \mid a = a \mid a < a \mid a \leq a \mid \neg b \mid b \wedge b \\
\textsc{Basic Command} & BaseCmd \ni e ::= \mathtt{skip} \mid v := a \mid b?
\end{array}$$

Note that the if-then-else branching and while-loop commands presented in the standard deterministic imperative language [46] can be defined as syntactic sugar:

$$\mathtt{if}\ b\ \mathtt{then}\ s_1\ \mathtt{else}\ s_2 \triangleq (b?; s_1) \oplus (\neg\, b?; s_2) \qquad \mathtt{while}\ b\ \mathtt{do}\ s \triangleq (b?; s)^*; \neg\, b?$$

3.4 Concrete Semantics of Programs

Let $Var = \{x_1, \ldots, x_n\}$ denote a finite set of all variables, and let a program state $\sigma : Var \to \mathbb{Z}$ be a total function mapping each variable to an integer. Note that when only one $x_i \in Var$ ($i \in \{1, 2, \cdots, n\}$) is in the program s, it is simplified to x. We denote by $\Sigma := Var \to \mathbb{Z}$ the set of program

$$\llbracket \texttt{skip} \rrbracket\, c \triangleq c \qquad \llbracket v := a \rrbracket\, c \triangleq \{\sigma\,[v \mapsto (\!|a|\!)\,\sigma] \mid \sigma \in c\} \qquad \llbracket b? \rrbracket\, c \triangleq \{\sigma \in c \mid (\!|b|\!)\,\sigma = \text{True}\}$$
$$\llbracket s_1; s_2 \rrbracket\, c \triangleq \llbracket s_2 \rrbracket\,(\llbracket s_1 \rrbracket\, c) \qquad \llbracket s_1 \oplus s_2 \rrbracket\, c \triangleq \llbracket s_1 \rrbracket\, c \cup \llbracket s_2 \rrbracket\, c \qquad \llbracket s^* \rrbracket\, c \triangleq \bigcup \{(\llbracket s \rrbracket)^n c \mid n \in \mathbb{N}\}$$

Fig. 2. Definitions of collecting denotational concrete semantics $\llbracket \cdot \rrbracket$

states, where *Var* is a finite set of variables (left implicit when clear from context). The state update operation $\sigma[x_i \mapsto n]$ is defined in the usual way: $\sigma[x_i \mapsto n](x_j) = n$ (if $j = i$) or $\sigma(x_j)$ (otherwise).

Figure 2 presents the (collecting) denotational concrete semantics of programs. Here the semantics of arithmetic and Boolean expressions are defined, respectively, by the functions $(\!|\cdot|\!) : \mathit{AExp} \to \Sigma \to \mathbb{Z}$ and $(\!|\cdot|\!) : \mathit{BExp} \to \Sigma \to \{\text{True}, \text{False}\}$ whose definitions are straightforward and therefore omitted. Now that each basic command has a semantics $\llbracket \cdot \rrbracket : \mathit{BaseCmd} \to \mathcal{P}(\Sigma) \to \mathcal{P}(\Sigma)$, the semantics $\llbracket \cdot \rrbracket : \mathit{Prog} \to \mathcal{P}(\Sigma) \to \mathcal{P}(\Sigma)$ of arbitrary program is inductively defined as shown in the right-hand side Figure 2, which is standard for designing static program analysis [13].

In the rest of the paper, some of the example programs only involve one variable, for which we will use $\mathcal{P}(\mathbb{Z})$ instead of $\mathcal{P}(\Sigma)$ as the program state for simplicity. For example, $\{-2, 1\}$ will represent $\{\sigma_1, \sigma_2\}$ where $\sigma_1(x) = -2$ and $\sigma_2(x) = 1$.

3.5 Abstract Interpretation

In this subsection, we recall the basic framework of abstract interpretation [12, 14, 15], and inherit the notations and definitions from a series of related previous works [4, 5, 7].

Abstract and Concrete Domains. In abstract interpretation, the concrete domain C and the abstract domain A are complete lattices related by a pair of monotone functions $\langle \alpha : C \to A, \gamma : A \to C \rangle$ called a Galois connection (GC for short). A GC satisfies that $\forall a \in A$ and $c \in C : \alpha(c) \leq_A a \Leftrightarrow c \leq_C \gamma(a)$, and α is additive. In the following text, unless otherwise specified, the concrete domain C will refer to $\mathcal{P}(\Sigma)$ ordered by inclusion $\subseteq$, and lub operation $\vee_C$ will be replaced by the set union $\cup$.

Correctness (Soundness) and Completeness. In abstract interpretation, each program is associated with a monotone concrete function $f : C \to C$ as the semantics, and is approximated as an abstract transfer function $f^A : A \to A$. Given an abstract domain A and its Galois connection $\langle \alpha, \gamma \rangle$ to the concrete domain C, as well as the abstract transfer function f^A, we can construct an abstract interpreter (A, α, γ, f^A) to analyze the concrete semantics f. In this setting, f^A should satisfy that $\alpha \circ f \leq_A f^A \circ \alpha$, and we say that it is *correct* (or *sound*). Then for any $I \in C$, the result by the abstract interpreter is an over-approximation: $f(I) \leq_C \gamma \circ f^A \circ \alpha(I)$. If f^A is a sound approximation of f and there exist both the least fixpoints of f and f^A, then $\alpha(\mathsf{lfp}\,(f)) \leq_A \mathsf{lfp}\,(f^A)$ holds. The *best correct approximation* (bca) of f on A is the abstract function $f^\alpha \triangleq \alpha \circ f \circ \gamma : A \to A$, as it turns out that any other abstract function f^A is a correct approximation of f iff $f^\alpha \leq_A f^A$.

Moreover, the abstract function f^A is a *complete* approximation of f if $\alpha \circ f = f^A \circ \alpha$ holds [14, 20]. The completeness of f^A intuitively encodes optimal precision when abstracting f into the abstract domain A, as the loss of precision is only due to α and not to the abstract function f^A itself.

Local Completeness. The completeness property above is to some extent a *global* property: It is universally quantified on all possible concrete elements, i.e., $\forall c \in C. \alpha \circ f(c) = f^A \circ \alpha(c)$. However, this global completeness can be hard/impossible to achieve, as argued by the previous work [4], it may well happen that completeness holds locally, i.e. just for some input properties. Therefore, we inherit and slightly alter their definition of *local completeness*: an abstract function $f^A : A \to A$ is locally complete for a concrete function $f : C \to C$ on a value $c \in C$ iff $\alpha \circ f(c) = f^A \circ \alpha(c)$. Note that in [4] they define local completeness only on the abstract domain A and let f^A be the bca f^α, but we consider any abstract function f^A instead. Also, local completeness is usually enough to judge whether an analysis is precise in practice, as the concrete element is usually a set of program states, and people often care whether the analysis is complete for the set of all legal inputs.

(a) Abstract Program $\quad AProg \ni \rho ::= \theta \in \Theta \mid \rho;\rho \mid \rho \oplus \rho \mid \rho^*$

(b) $\rho_1;\rho_2 = \rho_2 \circ \rho_1 \qquad \rho_1 \oplus \rho_2 = \rho_1 \vee_A \rho_2 \qquad \rho^* = \lambda a_0.\mathsf{lfp}\,(\lambda a_1.\ a_1 \triangledown_A (a_0 \vee_A \rho(a_1)))$

Fig. 3. (a) Syntax and (b) Semantics of the Abstract Program

3.6 Dataflow Analysis and the Abstract Semantics

Our work focuses on dataflow analysis, which attempts to compute program facts at each program locations by analyzing the flow of data through a program's control flow graph (CFG). A dataflow analysis typically does not abstract the control flow of the program. So once an abstract domain is given and assuming the join operation is realized by the lub operator $\vee_A$, for a program described in Section 3.3, a dataflow analysis can be fully specified by defining the abstract transfer function for each basic command. The design of abstract transfer functions varies by the specific analysis, and in our sign analysis example, they are designed as the best correct approximations $\{\alpha \circ [\![e]\!] \circ \gamma \mid e \in \mathit{BaseCmd}\}$. To speedup convergence and ensure termination, some dataflow analyses are also equipped with a widening operator. As a result, we define a dataflow analysis as a five-tuple $\mathrm{DA} := (A, \alpha, \gamma, \Theta, M_A, \triangledown_A)$ where A is an abstract domain, α and γ are the functions linking the abstract domain with the concrete domain, Θ is the domain of abstract transfer functions, $M_A : \mathit{BaseCmd} \to \Theta$ is a map from basic commands to abstract transfer functions, and $\triangledown_A : A \times A \to A$ is a widening operator.

Given a dataflow analysis $\mathrm{DA} := (A, \alpha, \gamma, \Theta, M_A, \triangledown_A)$, we can view its execution on a program $s \in$ Prog as the interpretation of an abstract program where each basic command e in s is replaced with its abstract transfer function $M_A(e) \in \Theta$. Intuitively, an abstract program is the abstraction of a program according to the dataflow analysis, and a program corresponds to only one abstract program while an abstract program can correspond to more than one program. We give the formal definition of an abstract program below.

Definition 3.1 (*Abstract Program*)**.** Given a dataflow analysis $\mathrm{DA} := (A, \alpha, \gamma, \Theta, M_A, \triangledown_A)$ and a program $s \in \mathit{Prog}$, an abstract program $\rho \in \mathit{AProg}$ can be derived by replacing each basic command e with its abstract transfer function $M_A(e) = \theta : A \to A \in \Theta$, and those atomic abstract programs are composed following the syntax in Figure 3, which is similar to the syntax of programs in *Prog*. The semantics is also defined by structural induction in Figure 3. With slight abuse of the notation, we define $\rho := M_A(s)$.

With the above definition, we define an execution of a dataflow analysis $\mathrm{DA} := (A, \alpha, \gamma, \Theta, M_A, \triangledown_A)$ on a program s as a pair (DA, ρ), where ρ is the abstract program for s. Similar to an abstract program, an execution can map to more than one program.

4 The Analysis Result Confidence Problem

We first define the source of randomness in our setting and then define our core problem–How to calculate the confidence of results produced by a dataflow analysis. Briefly, an execution of a dataflow analysis can match a set of programs, and the confidence of results produced by the execution is defined by the expectation of the analysis being locally complete on such programs.

4.1 Mapping an Abstract Program to a Distribution of Concrete Programs

Given an analysis execution (DA, ρ) where $\mathrm{DA} := (A, \alpha, \gamma, \Theta, M_A, \triangledown_A)$, the set of (concrete) programs for ρ is defined as $\mathit{reverse}(\rho) := \{s \in \mathit{Prog} \mid M_A(s) = \rho\}$. Here, $\mathit{reverse}(\rho)$ is the set of programs that match the given analysis execution, and is the sample space we will use to define the analysis

result confidence. The corresponding σ-algebra is simply $\mathcal{P}(reverse(\rho))$ and the measurable space is $(reverse(\rho), \mathcal{P}(reverse(\rho)))$.

With the measurable space defined, we need to define a probability measure in order to define the distribution of programs that match a given dataflow analysis execution. Since the analysis does not abstract the control flow, the probability of a program s in $reverse(\rho)$ is defined by the joint probability of its basic commands (denoted as $\bar{\phi}(s) \in \mathcal{P}(\mathbb{Z}^+ \times BaseCmd)$, a set of basic commands with their line numbers). Such a distribution can be computed by considering syntactical and semantic constraints or learnt from data (as seen in existing Bayesian program analysis literature [23]), and our focus does not include how to compute it but is how to quantify the confidence of analysis results once it is given. For example, in Section 2.2 , the set *Prog* corresponds to $reverse(S)$ and *Dist* is the distribution.

Let $\phi(\rho) \in \mathcal{P}(\mathbb{Z}^+ \times \Theta)$ be the set of all abstract transfer functions in ρ (a line number is used to distinguish the same abstract transfer function at different program locations). We divide $\phi(\rho)$ into two disjoint sets: $\phi(\rho) := \phi_{dep}(\rho) \cup \phi_{ind}(\rho)$, where all abstract transfer functions corresponding to dependent basic commands are included in $\phi_{dep}(\rho)$ and the ones that correspond to each pair in $\phi_{ind}(\rho)$ are distributed independently. Such dependencies can come from different syntactical and semantic constraints, for example: (1) Multiple abstract transfer functions must correspond to the same basic command as they abstract a reusable piece of code such as a function; (2) The basic commands corresponding to several abstract transfer functions must be distributed in a way to ensure the generated program is correct in terms of syntax and semantics.

Example. For a while-loop program `while` $x < 10$ `do` $x := x + 1$, it is written as $s = (b?; e)^*; \neg\, b?$ in our syntax, where b is $x < 10$ and e is $x := x + 1$. For its corresponding abstract program $\rho = M_A(s)$, $\phi(\rho) = \{(1, M_A(b?)), (2, M_A(e)), (3, M_A(\neg\, b?))\}$, where the line number is assigned in a left-to-right order. $\phi_{dep}(\rho) = \{(1, M_A(b?)), (3, M_A(\neg\, b?))\}$ due to the syntax relation of b and $\neg\, b$.

For illustration, we assume $\phi_{dep}(\rho) = \{(1, \theta_1), (2, \theta_2), \ldots, (k, \theta_k)\}$ and $\phi_{ind}(\rho) = \{(k{+}1, \theta_{k+1}), (k{+}2, \theta_{k+2}), \ldots, (k+m, \theta_{k+m})\}, k, m \in \mathbb{N}$. Let $\{\{(k+i, \theta_{k+i})\}\} : D(BaseCmd), i \in \{1, \ldots, m\}$ denote the distribution of independent basic commands in $\phi_{ind}(\rho)$, while the joint distribution for $(j, \theta_j) \in \Theta_{dep}, j \in \{1, \ldots, k\}$ is encoded as a function $\mathfrak{J}D : D(BaseCmd^k)$. Then for a sequence of basic commands $e_1, e_2, ..., e_k, e_{k+1}, \ldots, e_{k+m}$, its joint probability is defined as:

$$\{\{e_1, e_2, ..., e_k, e_{k+1}, ..., e_{k+m}\}\} := \mathfrak{J}D(e_1, \ldots, e_k) \times \prod_{i \in [1,m]} \{\{k+i, \theta_{k+i}\}\}\,(e_{k+i}).$$

We now formally define the distribution of concrete programs that match an analysis execution.

Definition 4.1 (*Distribution of Concrete Programs*)**.** Given a dataflow analysis execution (DA, ρ), where $\mathrm{DA} := (A, \alpha, \gamma, \Theta, M_A, \nabla_A)$, let $\phi(\rho) := \phi_{ind}(\rho) \cup \phi_{dep}(\rho)$ be the numbered abstract transfer functions in ρ whose joint probability distribution of corresponding basic commands is specified using a mass function $\{\{e_1, e_2, ..., e_n\}\} := \prod_{\theta_i = M_A(e_i) \wedge (i,\theta_i) \in \phi_{ind}(\rho)} \{\{i, \theta_i\}\}\,(e_i) \times \mathfrak{J}D(e_{d_1}, \ldots, e_{d_k})$, where $e_{d_j} \in M_A(\theta_{d_j})$ and $(d_j, \theta_{d_j}) \in \phi_{dep}, j \in \{1, \ldots, k\}$, then the probability distribution of all concrete programs that match the dataflow analysis execution can be specified using a mass function $\{\{\cdot\}\} : reverse(\rho) \to [0, 1]$:

$$\{\{s\}\} = \{\{e_1, \ldots, e_n\}\} \text{ where } (i, e_i) \in \bar{\phi}(s), i \in \{1, ..., n\} \text{ for any } s \in reverse(\rho).$$

We write the distribution as $Dist(\mathrm{DA}, \rho)$.

4.2 Local Completeness as an Under-Approximation of Alarm Holding

After defining the distribution of concrete programs matching a dataflow analysis execution, we can define the confidence of the corresponding analysis result as the probability of the analysis result holding on a sampled program from the distribution. Formally:

Definition 4.2 (*Confidence of Static Analysis Results*)**.** Given a dataflow analysis execution (DA, ρ), where DA := $(A, \alpha, \gamma, \Theta, M_A, \nabla_A)$, a set of all possible input states $I \in C$, a distribution $Dist(\text{DA}, \rho)$ of concrete programs matching the execution, and a correctness specification $Spec \in C$ used in the static analysis, assuming $\gamma(\rho(\alpha(I))) \not\subseteq Spec$, the probability $AT(\text{DA}, \rho, I, Dist(\text{DA}, \rho), Spec)$ which quantifies the confidence of the static analysis results is defined as:

$$AT(\text{DA}, \rho, I, Dist(\text{DA}, \rho), Spec) = \Pr_{s \sim Dist(\text{DA}, \rho)} \left([\![s]\!] (I) \not\subseteq Spec \right)$$

Since the form of *Spec* can vary widely, we attempt to propose a confidence value that does not depend on the specific form of *Spec*. Technically, we say that a *specification Spec* $\in C$ *is expressible in* A iff $Spec = \gamma \circ \alpha(Spec)$. Following the setup by Bruni et al. [4], we assume that **the alarm specification is expressible in the abstract domain**.

Example. For the sign analysis in Section 2.1, $Spec_1 = \{x | x \neq 0, x \in \mathbb{Z}\} \in \mathcal{P}(Z)$ is expressible in A, since $\gamma \circ \alpha(Spec_1) = \gamma(\mathbb{Z}_{\neq 0}) = Spec_1$. However, $Spec_2 = \{x | x \neq 1, x \in \mathbb{Z}\} \in \mathcal{P}(Z)$ is not expressible in A, since $\gamma \circ \alpha(Spec_2) = \gamma(\mathbb{Z}) \neq Spec_2$.

Note that when this assumption does not hold, we can refine the abstract domain to ensure that this assumption holds as the previous work did [4, 6]. For example, consider a coarser sign abstract domain $A' = \{\emptyset, \mathbb{Z}_{>0}, \mathbb{Z}_{=0}, \mathbb{Z}_{<0}, \mathbb{Z}\}$ where $Spec = \{x | x \neq 0, x \in \mathbb{Z}\}$ is not expressible. This issue can be overcome by refining it to the abstract domain A that we used in Section 2.

To precisely compute the analysis result confidence, one needs to compute the set of reachable concrete states for every concrete program that matches the analysis execution, which is more expensive than determining whether the analyzed program satisfies *Spec*. To enable compositional computation and a learning paradigm which is adopted by current Bayesian program analysis, we compute the probability of the analysis execution being locally complete, which underestimates the real probability. We define it formally below:

Definition 4.3 (*Probability of Local Completeness*)**.** Given $(\text{DA}, \rho, I, Dist(\text{DA}, \rho))$ which are the same as those in Definition 4.2, the probability $LC(\text{DA}, \rho, I, Dist(\text{DA}, \rho))$ of analysis execution (DA,ρ) being locally complete on a sampled program $s \sim Dist(\text{DA}, \rho)$ for the input I is defined as:

$$LC(\text{DA}, \rho, I, Dist(\text{DA}, \rho)) = \Pr_{s \sim Dist(\text{DA}, \rho)} \left(\rho(\alpha(I)) = \alpha([\![s]\!] (I)) \right).$$

Under the assumption that *Spec* is expressible in the abstract domain, we next show the probability of local completeness is an underestimation of the analysis result confidence. We first prove a theorem shows local completeness is an under-approximation of analysis results being valid.

Theorem 4.1 (Local Completeness as an Under-approximation). *Let Spec be expressible in A and for an abstract transfer* $f^A : A \to A$ *we have* $\gamma(f^A(\alpha(I))) \not\subseteq Spec$*, then*

$$\forall f : C \to C, f^A(\alpha(I)) = \alpha(f(I)) \implies f(I) \not\subseteq Spec$$

Its proof is given in the appendix. Following the above theorem, it is evident that the probability of local completeness underestimates the analysis result confidence.

Theorem 4.2. *Suppose Spec is expressible in A and the datalow analysis DA raises an alarm, then*

$$LC(DA, \rho, I, Dist(DA, \rho)) \leq AT(DA, \rho, I, Dist(DA, \rho), Spec).$$

In subsequent sections of this paper, we mainly focus on computing the probability $LC(\text{DA}, \rho, I, Dist(\text{DA}, \rho))$ (abbreviated as LC when its parameters can be clearly inferred from the context), as it is a lower bound for the confidence of arbitrary specification *Spec* which satisfies our assumption. We emphasize that LC is intended to measure *analysis-induced* imprecision, rather than limitations of the abstract domain's expressive power. If the assumption is not satisfied (*Spec* is not expressible), false alarms may persist even when LC holds, reflecting a domain/specification mismatch that must be addressed via domain refinement rather than by LC itself.

5 Denotational Semantics to Compute Probabilities of Completeness

We first introduce a compositional semantics to compute the probability of local completeness to avoid the cost of directly sampling programs from the joint distribution $Dist(\text{DA}, \rho)$. Then, we introduce an improved semantics with under-approximation to further improve the tractability.

Auxiliary Notations. We introduce auxiliary notations before describing our semantics. Since our semantics will need to compute the results about partial programs in order to support compositional reasoning, we use $SubStr(\rho)$ to denote the set of all sub-strings $\rho' \in AProg$ of $\rho \in AProg$. Further, for an abstract program ρ and each of its sub-string $\rho' \in SubStr(\rho)$, we assign a line number to each abstract transfer function in a left-to-right order to uniquely identify each at a program location. For example, $\rho = \theta_1; ((\theta_2)^* \oplus \theta_3)$ is numbered as $\rho = (1, \theta_1); ((2, \theta_2)^* \oplus (3, \theta_3))$. The functions *reverse* and ϕ in Section 4.1 are extended to take a $\rho' \in SubStr(\rho)$ as input. We introduce a new deterministic function $\varphi : \phi(\rho') \to BaseCmd$ that identifies a partial concrete program in $reverse(\rho')$ by mapping each abstract transfer function in $\phi(\rho')$ to a basic command. By slightly abusing the notation, we use $\phi(\rho')$ to denotes the program generated by replacing each (n, θ) in ρ' with $\varphi((n, \theta))$. Finally, similar to Section 4.1, given a dataflow analysis execution $(A, \alpha, \gamma, \Theta, M_A, \triangledown_A, \rho)$, we divide the set of abstract transfer functions into the independent set and the dependent set by supposing $\exists k, m \in \mathbb{N}$, such that $\phi(\rho) := \phi_{dep}(\rho) \cup \phi_{ind}(\rho)$ where $\phi_{dep}(\rho) = \{(n_i, \theta_i) \mid 1 \le i \le k\}$ and $\phi_{ind}(\rho) = \{(n_i, \theta_i) \mid k+1 \le i \le k+m\}$ $(n_i \in \mathbb{N}, \theta_i \in \Theta)$. We also write $\phi_{dep}(\rho)$ and $\phi_{ind}(\rho)$ as Θ_{dep} and Θ_{ind} respectively when ρ is obvious from the context.

Compositional Semantics for Probabilities of Local Completeness. Our semantics to compute the probabilities of local completeness takes a (partial) abstract program $\rho' \in SubStr(\rho)$ and a set of concrete states as input, and computes a joint distribution of pairs of a set of concrete states and the set of dependent basic commands that are used in the corresponding concrete partial program. Intuitively, tracking the distribution of sets of concrete states allows computing the probability of ρ' being locally complete by comparing to the abstract state computed by it, and finally leads to computing the probability of being locally complete for ρ. And since the distribution of subsequent partial programs in ρ may depend on the partial program sampled from ρ', the output also includes the basic commands in the sampled partial program that correspond to abstract transfer functions with dependence (those in $\phi_{dep}(\rho)$). As a result, the output of the semantics on ρ would include a distribution of basic commands corresponding to all abstract transfer functions in $\phi_{dep}(\rho)$, enabling evaluating the joint probability of these commands through mass function $\mathcal{J}D$. Note the independent basic commands do not need to be tracked as they do not affect the distribution of other basic commands. Finally, we introduce empty string ε to represent unsampled abstract transfer functions (which are not included in the input partial abstract program) to pad the semantics output so that the output sample always includes k basic commands for illustration. By putting these together, the type of our denotational semantics is defined as $\langle\!\langle \cdot \rangle\!\rangle : AProg \to (C \to D(C \times (BaseCmd \cup \{\varepsilon\})^k)$.

Figure 4 shows the rules of our semantics, which is defined inductively. The fact that $\langle\!\langle \cdot \rangle\!\rangle$ are inductively defined means that the semantics of compound constructs (e.g., sequences, non-determinism, loops) are systematically derived from the semantics of their sub-strings. This compositionality makes the framework modular and extensible. We use $(t)_i$ to denote the ith element of a tuple t. Intuitively, $\langle\!\langle \rho' \rangle\!\rangle\,(c)(t)$ computes the probability that a program s produces an output $(t)_1 \in C$ with input c, where s is constructed by replacing each θ_i at location n_i by $(t)_{i+1}$ and replacing each independent θ by a sampled basic command. We next explain the rules for each construct in detail.

The definition for $\langle\!\langle (n_i, \theta_i) \rangle\!\rangle$ handles the atomic case for basic commands. We use $t \in C \times (BaseCmd \cup \{\epsilon\})^k$ to denote a sample from its output distribution and $(t)_i$ is its ith element (with $(t)_1$ being the output concrete state set). For $(n_i, \theta_i) \in \phi_{ind}(\rho)$ $(k+1 \le i \le k+m)$, the output $(t)_1$

$$\langle\!\langle (n_i,\theta_i)\rangle\!\rangle\,(c)(t) = \begin{cases} \Pr_{e\sim\{\{(n_i,\theta_i)\}\}}\left[[\![e]\!]\,(c) = (t)_1\right] & \text{if } k+1 \le i \le k+m \text{ and } \forall j \in [2,k+1].(t)_j = \varepsilon \\ 1 & \text{if } 1 \le i \le k, (t)_{i+1} \in \mathit{reverse}((n_i,\theta_i)), \\ & [\![(t)_{i+1}]\!]\,(c) = (t)_1, \text{ and } \forall j \in [2,i] \cup [i+2,k].(t)_j = \varepsilon \\ 0 & \text{otherwise} \end{cases}$$

$$\langle\!\langle \rho_1;\rho_2\rangle\!\rangle\,(c)(t_3) = \sum_{c'=(t_1)_1 \wedge (t_2)_1=(t_3)_1 \wedge \mathbf{Valid}(t_1,t_2,t_3) \wedge \mathbf{non\text{-}zero\text{-}term}} \langle\!\langle \rho_1\rangle\!\rangle\,(c)(t_1)\cdot\langle\!\langle \rho_2\rangle\!\rangle\,(c')(t_2)$$

$$\langle\!\langle \rho_1 \oplus \rho_2\rangle\!\rangle\,(c)(t_3) = \sum_{(t_3)_1=(t_1)_1 \cup (t_2)_1 \wedge \mathbf{Valid}(t_1,t_2,t_3) \wedge \mathbf{non\text{-}zero\text{-}term}} \langle\!\langle \rho_1\rangle\!\rangle\,(c)(t_1)\cdot\langle\!\langle \rho_2\rangle\!\rangle\,(c)(t_2)$$

$$\left\langle\!\!\left\langle (\rho')^*\right\rangle\!\!\right\rangle\,(c)(t) = \begin{cases} \sum_{\varphi} \prod_{\zeta_j \in \phi_{ind}(\rho')} \{\!\{\zeta_j\}\!\}\,(\varphi(\zeta_j)) & \text{for all } \varphi : \phi(\rho) \to \mathit{BaseCmd} \text{ s.t. } [\![(\varphi(\rho'))^*]\!]\,(c) = (t)_1 \text{ and} \\ & \text{for } 1 \le i \le k, (t)_{i+1} = \begin{cases} \varphi((n_i,\theta_i)) & (n_i,\theta_i) \in \phi_{dep}(\rho') \\ \varepsilon & (n_i,\theta_i) \notin \phi_{dep}(\rho') \end{cases} \\ 0 & \text{otherwise} \end{cases}$$

Fig. 4. Semantics for computing the probabilities of local completeness.

is obtained by sampling a basic command from $\{\{n_i,\theta_i\}\}$ and evaluating it on the input $c \in C$, while $\langle\!\langle (n_i,\theta_i)\rangle\!\rangle\,(c)(t)$ records its probability. Note that $(t)_i$ $(2 \le i \le k+1)$ will be ε as no $(n_j,\theta_j) \in \Theta_{dep}$ appears. On the other hand, for $(n_i,\theta_i) \in \Theta_{dep}$ $(k+1 \le i \le k+m)$, $(t)_1$ is obtained by choosing a basic command e from $\mathit{reverse}((n_i,\theta_i))$, as the probability for a dependent basic command needs to be calculated with all the other dependent basic commands together. Next, $(t)_{i+1}$ will be set as e, and other elements will be set as ε as the corresponding abstract transfer has not appeared.

Example. Consider the sign analysis example in Section 2. The abstract program $\rho = x := x +_A \mathbb{Z}^+; x := x \times_A \mathbb{Z}^+$ and input set is $\{1,2\}$. Assuming the distribution corresponding to both abstract transfer functions are independent and uniform, then $\langle\!\langle (1, x := x +_A \mathbb{Z}^+)\rangle\!\rangle\,(\{1,2\})((\{1+i, 2+i\})) = \frac{1}{N}$ for any i between 1 and the maximum integer N, and $\langle\!\langle (1, x := x +_A \mathbb{Z}^+)\rangle\!\rangle\,(\{1,2\})((\{1+i, 2+j\})) = 0$ for any $i \neq j$ or $i \le 0$. If the two distributions are dependent, then $\langle\!\langle (1, x := x +_A \mathbb{Z}^+)\rangle\!\rangle\,(\{1,2\})((\{1+i, 2+i\}, x := x + i, \varepsilon)) = 1$ for $i \in [1, N]$ and the probability is 0 for other cases.

Next, the sequencing and non-deterministic rules for $\langle\!\langle \rho_1;\rho_2\rangle\!\rangle$ and $\langle\!\langle \rho_1 \oplus \rho_2\rangle\!\rangle$ compose the semantics of $\langle\!\langle \rho_1\rangle\!\rangle$ and $\langle\!\langle \rho_2\rangle\!\rangle$ respectively. Note that we require that terms should be non-zero in the summation of the compositional computation for $\langle\!\langle \rho_1;\rho_2\rangle\!\rangle$ and $\langle\!\langle \rho_1 \oplus \rho_2\rangle\!\rangle$, as denoted by a predicate **non-zero-term**. It ensures that our semantics is well-defined and computable, and we summarize it with the following theorems whose proof is included in the appendix:

Theorem 5.1. *$\mathit{reverse}((n_i,\theta_i))$ is finite $(1 \le i \le k+m)$ and all possibly sampled programs terminate, then $\forall \rho' \in \mathit{SubStr}(\rho), c \in C$, the non-zero terms $\langle\!\langle \rho\rangle\!\rangle\,(c)(t)$ are finite and hence $\langle\!\langle \rho\rangle\!\rangle$ is computable.*

Both rules marginalize the probability of intermediate concrete states computed by the corresponding intermediate values (c' for example). The key is that for the partial abstract programs ρ_1, ρ_2, and the composed abstract program $\rho_1;\rho_2$ or $\rho_1 \oplus \rho_2$, the sampled dependent basic commands should be consistent in order to form a valid concrete program sample. We capture the idea using a predicate **Valid**. It checks whether the basic commands stored in the same location for t_1, t_2 and t_3 are consistent, and it is also satisfied when one location of t_1 or t_2 stores the empty string ε and t_3 is equal to the other non-empty element. Formally:

$$\mathbf{Valid}(t_1,t_2,t_3) = \bigwedge_{i=2}^{k+1} ((t_1)_i = (t_2)_i = (t_3)_i) \vee ((t_3)_i = (t_1)_i \wedge (t_2)_i = \varepsilon) \vee ((t_3)_i = (t_2)_i \wedge (t_1)_i = \varepsilon)$$

The loop case is handled by the rule of $\langle\!\langle (\rho)^* \rangle\!\rangle$. If a Kleene iteration $(\varphi(\rho'))^*$ can produce output $(t)_1$ given input c, the tuple t will record the reversed program of each $(n_i, \theta_i) \in \phi_{dep}(\rho')$ (which means it appears in ρ') at corresponding $(t)_{i+1}$. Finally, $\langle\!\langle (\rho)^* \rangle\!\rangle\ (c)(t)$ will be set as the probability of independently sampling $\varphi(\zeta_j)$ from $\{\!\{\zeta_j\}\!\}$ for each independent $\zeta_j = (n_j, \theta_j) \in \phi_{ind}(\rho')$.

Note that our semantics $\langle\!\langle \rho' \rangle\!\rangle$ actually records each possible concrete execution $\varphi(\rho')$ that can be sampled from ρ' when handling $\langle\!\langle (\rho')^* \rangle\!\rangle$. Therefore, the definition for the loop case in Figure 4 is also compositional as it takes $\langle\!\langle \rho' \rangle\!\rangle$ as input. A minor issue is that the independently sampled basic commands in the loop body are not guaranteed to be consistent across iterations. However, it can be easily fixed by tracking them along with the dependent ones, as shown in the appendix.

For $a_1 \in A$ (Recall that $A = \mathsf{Sign}$), we define an auxiliary function $(a_1?) : A \to A$ as $\forall a_2 \in A, (a_1?)a_2 = a_1 \wedge_A a_2$. $a?$ can abstract the concrete program $b?$ if $\alpha(\{x \in \mathbb{Z} \mid (\!|b?|\!)\, x = \text{True}\}) = a$. For example, $(\mathbb{Z}_{>0}?)$ can abstract $(x > 1)?$. With this definition, we offer an example with loop:

Example. $\rho = ((\mathbb{Z}_{<0}?); x \coloneqq x +_A \mathbb{Z}_{>0})^*; (\mathbb{Z}_{\geq 0}?)$ encodes a sign analysis execution of a program with Kleene iteration. Suppose $\theta_1 = (\mathbb{Z}_{<0}?), \theta_2 = (\mathbb{Z}_{\geq 0}?), \Theta_{dep} = \{\theta_1, \theta_2\}$, and $\mathcal{JD}((x < 0)?, (x \geq 0)?) = 1$, while $\theta_3 = x \coloneqq x +_A \mathbb{Z}_{>0}$, $\Theta_{ind} = \{\theta_3\}$ and $\{\!\{\theta_3\}\!\}\ (x \coloneqq x + n) = \frac{1}{N}$ (For a $N > 0$ and $1 \leq n \leq N$). For the input $I = \{-1\}$, $\rho' = (\theta_1; \theta_3)^*$, non-zero terms of $\langle\!\langle \rho' \rangle\!\rangle\ (I)(t)$ are:

$$\langle\!\langle \rho' \rangle\!\rangle\ (I)(\{-1, n-1\}, (x < 0)?) = \frac{1}{N}\ (\forall 1 \leq n \leq N)$$

We will now show that our semantics indeed computes the desired output distributions, as captured by the following theorems, whose proofs are included in the appendix. Note that we use ζ_i to denote the pair (n_i, θ_i) for the sake of conciseness in expression.

Theorem 5.2 (Correctness of $\langle\!\langle \cdot \rangle\!\rangle$). *For all $\rho' \in \mathit{SubStr}(\rho), c \in C, t \in C \times (\mathit{BaseCmd} \cup \{\varepsilon\})^k$:*

$$\langle\!\langle \rho' \rangle\!\rangle\ (c)(t) = \sum_{\varphi} \prod_{\zeta \in \phi_{ind}(\rho')} \{\!\{\zeta\}\!\}\ (\varphi(\zeta)) \cdot \delta\left([\![\varphi(\rho')]\!]\ (c) = (t)_1 \wedge \bigwedge_{\zeta_i \in \phi_{dep}(\rho')} \varphi(\zeta_i) = (t)_{i+1} \right)$$

Theorem 5.2 shows that our denotational semantics accurately reflect the concrete behavior of programs sampled from the underlying distributions. As a direct consequence, we can use it to model the entire distribution $\mathit{Dist}(\mathrm{DA}, \rho)$ and compute the value LC:

Corollary 5.3.

$$Pr_{s \sim \mathit{Dist}(DA,\rho)}\ [[\![s]\!]\ (c) = d] = \sum_{(t)_1 = d} \langle\!\langle \rho \rangle\!\rangle\ (c)(t) \cdot \mathcal{JD}((t)_2, (t)_3, \ldots, (t)_{k+1})$$

Recall from Section 4, $LC(DA, \rho, I, \mathit{Dist}(\mathrm{DA}, \rho)) = \mathrm{Pr}_{s \sim \mathit{Dist}(\mathrm{DA},\rho)} \delta\,(\rho(\alpha(I)) = \alpha([\![s]\!]\ (I)))$, and it can be computed as $\sum_{\alpha(O) = \rho(\alpha(I))} \mathrm{Pr}_{s \sim \mathit{Dist}(\mathrm{DA},\rho)}\ ([\![s]\!]\ (I) = O)$.

Semantics with Under-approximation. While $\langle\!\langle \cdot \rangle\!\rangle$ provides a compositional way to compute the probability of an analysis execution being locally complete, it is done at the cost of recording all possible concrete executions for $\phi_{dep}(\rho)$, which may introduce significant computational overhead. A common scenario of redundancy is that the analysis has become incomplete for certain programs in the middle and does not recover. What is worse, the semantics is incomputable when a sampled program does not terminate, since it will perform the Kleene iteration $(\varphi(\rho'))^*$ in computing $\langle\!\langle (\rho')^* \rangle\!\rangle$. As an example, for an abstract program $\rho = ((\mathbb{Z}_{<0}?); x \coloneqq x \times \mathbb{Z}_{>0})^*; (\mathbb{Z}_{\geq 0}?)$, $\langle\!\langle (\rho)^* \rangle\!\rangle\ (\{-1, 1\})$ is incomputable for the input $\{-1, 1\}$.

To address these issues, we define an augmented denotational semantics $\langle\!\langle \cdot \rangle\!\rangle^{lc}$ that retains only executions guaranteed to be locally complete at each step. As a result, its computed results is an underestimation of the probability of the analysis being locally complete, and therefore still an underestimation of the analysis result confidence. Intuitively, $\langle\!\langle \rho \rangle\!\rangle^{lc}$ filters out the parts of the

```
1   select(a)  {
2       i = a % 9;
3       queue = Shuffle(12);
4       return queue[i];
5   }
```

```
6   x = select(x);
7   if (x <= 5)
8       x := x - 1;
9   else
10      x := x + 1;
```

Fig. 5. Example program with lost specification.

program execution that fail to satisfy the local completeness condition, keeping only samples for which the abstract programs agree with the concretization at each atomic step. Since the composition rules of $\langle\!\langle\cdot\rangle\!\rangle^{lc}$ for sequences, non-determinism are structurally identical to those of the standard semantics $\langle\!\langle\cdot\rangle\!\rangle$, we omit them and offer the inductive definitions of abstract transfer functions and the Kleene iteration:

$$\text{if } \theta \in \Theta, \langle\!\langle\theta\rangle\!\rangle^{lc}(c)(t) = \langle\!\langle\theta\rangle\!\rangle(c)(t) \cdot \delta(\alpha((t)_1) = \theta \circ \alpha(c))$$

$$\langle\!\langle(\rho)^*\rangle\!\rangle^{lc}(c)(t) = \left\langle\!\!\left\langle \bigoplus_{i=0}^{w} (\rho)^i \right\rangle\!\!\right\rangle^{lc}(c)(t) \cdot \delta(\alpha((t)_1) = (\rho)^* \circ \alpha(c))$$

Here $\bigoplus_{i=0}^{w}(\rho)^i = id_A \oplus \rho \oplus \rho^2 \oplus \cdots \oplus \rho^w$, where ρ^i is the sequence composed of k repetitions of the abstract program ρ.

In the $\langle\!\langle\cdot\rangle\!\rangle^{lc}$ version for abstract transfer functions, we only count outputs for which the result of the concrete execution on which the abstract transfer function is complete.

To ensure computability in the case of a non-terminating concrete execution, we handle the Kleene iteration $\langle\!\langle(\rho')^*\rangle\!\rangle^{lc}$ with special treatment. Intuitively, $\langle\!\langle\cdot\rangle\!\rangle^{lc}$ expands it w times (where w is a pre-set hyperparameter that controls the precision) and retaining only those corresponding output concrete states C' that are consistent with $(\rho)^*$ after abstraction (i.e., $\forall c' \in C', \alpha(c') = (\rho)^*(\alpha(c))$). In other words, $\langle\!\langle(\rho')^*\rangle\!\rangle^{lc}$ will only retain those concrete executions that have produced concrete states matching the fixed point of the analysis on the loop within w iterations of the loop. Note that this treatment will lead to the fact that for each$\langle\!\langle(\rho')^*\rangle\!\rangle^{lc}(c)(t) \neq 0$ the stored possible output $(t)_1 \in C$ is not a true intermediate result. However, since it is consistent with the execution of $(\rho')^*$ in the abstract domain A, we will continue to use it to compute the value LC.

Example. Re-consider the abstract program $\rho = ((\mathbb{Z}_{<0}?); x := x \times_A \mathbb{Z}_{>0})^*; (\mathbb{Z}_{\geq 0}?)$ now, and suppose abstract transfer functions are denoted as (i, θ_i) $(i = 1, 2, 3)$ from left to right. Suppose $\Theta_{dep} = \{\theta_1, \theta_2\}$, and $\mathcal{JD}((x < 0)?, (x \geq 0)?) = 1$, and $\{\{\theta_3\}\}$ $(x := x * n) = \frac{1}{N}$ (For a $N > 0$ and $1 \leq n \leq N$). We assume the unrolling parameter w to be 2, then for the input $I = \{-1, 1\}$, $\rho' = (\theta_1; \theta_3)^*$, non-zero terms of $\langle\!\langle\rho'\rangle\!\rangle(I)(t)$ are: $\langle\!\langle\rho'\rangle\!\rangle(I)(\{-1, -n, -n^2\}, (x < 0)?) = \frac{1}{N}$ $(\forall 1 \leq n \leq N)$.

We summarize the soundness of the under-approximation by $\langle\!\langle\cdot\rangle\!\rangle^{lc}$ with the following theorems whose proofs are included in the appendix:

Theorem 5.4 ($\langle\!\langle\cdot\rangle\!\rangle^{lc}$ computes an under-approximation of the value LC).

$$LC(DA, \rho, I, Dist(DA, \rho)) \geq \sum_{\alpha(O)=\rho(\alpha(I))} \sum_{(t)_1=O} \langle\!\langle\rho\rangle\!\rangle^{lc}(I)(t) \cdot \mathcal{JD}((t)_2, (t_3), \ldots, (t_{k+1}))$$

6 Extended Application: Interval Analysis with Lost Specification

This section instantiates our approach on the standard interval analysis (Detailed definition is given in the appendix) to demonstrate another usage scenario called *lost specification* (simplified as *lost spec*) other than quantifying the uncertainties incurred by abstracting program semantics. This scenario arises when library functions lack accessible or precise specifications, either because their source code is unavailable (e.g., Windows APIs), only partially available, or implemented in other

programming languages. For instance, for the example program in Figure 5, the function `Shuffle` is a pseudo random sequence generator which returns a permutation of $[1, 2, \ldots, 12]$. While the returned list may vary on different hardware platforms, it is the same on the same hardware platform. As a result, for the function `select` which returns the $(a\%9)$th-element of `queue` with given input a, we cannot capture its precise specification. We intend to quantify the confidence of an analysis that over-approximates all versions of the library function with *lost spec* together by considering the distribution of possible real specifications.

We extend the syntax in Section 3.3 to encode the *lost spec*:

$$\textsc{Extended Program} \qquad \mathit{EProg} \ni s ::= e \in \mathit{BaseCmd} \mid \mathit{lf} \in \mathit{Lib} \mid s; s \mid s \oplus s \mid s^*$$

where the set Lib consists of all these library functions with lost spec. When handling it using abstract interpretation, all possible versions must be considered, and we use a mapping $\Xi : \mathit{Lib} \to \mathcal{P}(\mathit{Prog})$ that maps a lf to its all possible versions. For example, let lf be `select`, $\Xi(\mathit{lf})$ consists of the 12! versions which correspond to different permutations of $[1, \ldots, 12]$ returned by different versions of `Shuffle`. Since `select` only selects from the first nine numbers returned from `Shuffle`, there are only C_{12}^9 versions if we consider semantic equivalence. Note that we distinguish *Prog* from *EProg* as the former does not have components with lost spec.

The interval analysis over-approximates `select` by considering all its possible versions together and therefore the corresponding abstract program essentially combines all theses versions. In other words, the analysis assumes `select` can return any number in $[1, \ldots, 12]$. But in reality, `select` returns an integer in $[l, u]$ where l and u are random integer variables with supports of $[1, 12]$ $(l < u)$, and the over-approximated specification is precise only when $l = 1$ and $u = 12$. We next introduce the interval analysis with lost spec and how our approach quantifies its confidence given the distribution of real specifications. For simplicity, we let A be the standard interval domain Int and omit all subscripts, then define the interval analysis with *lost spec*:

Definition 6.1 (*Interval Analysis of Lost Spec*)**.** The function M_A that maps each basic command to its abstract transfer function in Section 3.6 is extended to $\mathit{BaseCmd} \cup \mathit{Lib} \to \Theta$, where for a basic command in $\mathit{BaseCmd}$ it is mapped to its bcas for illustration and for each $\mathit{lf} \in \mathit{Lib}$, $M_A(\mathit{lf}) = \bigvee_{s \in \Xi(\mathit{lf})} M_A(s)$. The set Θ can be defined by applying M_A on $\mathit{BaseCmd} \cup \mathit{Lib}$. $\triangledown_A : A \times A \to A$ is the standard interval widening operator: $[l_1, u_1] \triangledown_A [l_2, u_2] \triangleq [l_3, u_3]$, where $l_3 = l_1$ if $l_1 \leq l_2$ and $l_3 = -\infty$ otherwise, on the other hand $u_3 = u_1$ if $u_1 \geq u_2$ and $u_3 = +\infty$ otherwise. Finally, the interval analysis is given by $(A, \alpha, \gamma, \Theta, M_A, \triangledown_A)$.

With the above definition, the execution of the interval analysis of the program in Figure 5 is encoded as the abstract program: $\rho = \theta_1; (\theta_2; \theta_3) \oplus (\theta_4; \theta_5)$, where $\theta_1 = M_A(lf), \theta_2 = M_A(x \leq 5?), \theta_3 = M_A(x := x - 1)), \theta_4 = M_A(x > 5?)$, and $\theta_5 = M_A(x := x + 1)$. Here lf represents $x :=$ `select`(x) and $\forall a \in A, \theta_1(a) = [1, 12]$. To quantify the precision of ρ, our framework assigns probabilities to all possible versions in $\Xi(\mathit{lf})$ and model it as a uniform distribution: $\{\{\theta_1\}\}\ (s) = \frac{1}{12!}\ (\forall s \in \Xi(\mathit{lf}))$. Since we already know the basic commands corresponding to θ_i $(2 \leq i \leq 5)$, we design each $\{\{\theta_i\}\}$ as the corresponding point-distributions. Here for simplicity, we omit the line numbers defined in Section 4.1 and suppose $\forall 1 \leq i \leq 5, \theta_i \in \Theta_{ind}$.

Given the set of all possible inputs $I_0 = \{1, \ldots, 9\}$, we calculate the value LC using $\langle\!\langle \cdot \rangle\!\rangle^{lc}$. For θ_1, while it has 12! specifications, there are only C_{12}^9 unique specifications considering semantic equivalence, each of which has a probability of $\frac{1}{C_{12}^9}$. The analysis is locally complete on θ_1 when $\{1, 12\}$ is a subset of the specification. As a result, the non-zero terms for $\langle\!\langle \theta_1 \rangle\!\rangle^{lc}(I_0)(\cdot)$ are $\langle\!\langle \theta_1 \rangle\!\rangle^{lc}(I_0)(I_1) = \frac{1}{C_{12}^9}\ (\{1, 12\} \subseteq I_1 \subseteq \{1 \ldots 12\}$ and $|I_1 \models 9)$ as $\alpha(I_1)$ should be $[1, 12]$. Next, from the fact that θ_2 is local complete for $[\![(x \leq 5?)]\!]$ on I_1 $(\{1, 12\} \subseteq I_1)$, we can derive $5 \in I_1$ to make the

$$\text{trace}(\theta) = \{\theta\} \qquad \text{trace}(\rho_1;\rho_2) = \{\tau_1;\tau_2 \mid \tau_1 \in \text{trace}(\rho_1) \wedge \tau_2 \in \text{trace}(\rho_2)\}$$
$$\text{trace}(\rho_1 \oplus \rho_2) = \text{trace}(\rho_1) \cup \text{trace}(\rho_2) \qquad \text{trace}(\rho^*) = \{\eta\} \bigcup_{k\in\mathbb{Z}^+} \text{trace}(\rho^k)$$

Fig. 6. Traces of an Abstract Program ρ. Symbol η denotes an empty trace.

term $\langle\!\langle \theta_2 \rangle\!\rangle^{lc}(I_1)(\cdot)$ non-zero. Similarly, we can derive $6 \in I_1$ for θ_3. Finally, for the entire execution ρ, the non-zero terms are $\langle\!\langle \rho \rangle\!\rangle^{lc}(I_0)(I_2) = \frac{1}{C_{12}^3}$ $(\{0,4,7,13\} \subseteq I_2 \subseteq \{0\ldots4\} \cup \{7\ldots13\}$ and $|I_2 \models 9)$. The value LC is computed as $LC = \sum_{\{0,4,7,13\}\subseteq I_2 \subseteq \{0\ldots13\}} \frac{1}{C_{12}^9} = \frac{C_8^5}{C_{12}^9} \approx 0.255$.

7 Discussion

7.1 A Principled Approach to Bayesian Program Analysis

A main motivation of our paper is to provide a theoretical foundation for new types of program analyses with probabilities. Bayesian program analysis [10, 21, 23, 34, 39, 43, 49, 50] is a representative which can adjust alarm probabilities by learning from posterior information such as user feedback on alarms. Next, we attempt to perform the Bayesian program analysis in our framework and provide a theoretical guarantee for it. In particular, we demonstrate how our approach updates the analysis result confidence when user feedback is provided on a subset of alarms.

We first extend the syntax of basic commands in Section 3.3 to encode runtime errors:

$$\textsc{Basic Command} \qquad \mathit{BaseCmd} \ni e ::= \texttt{skip} \mid v := a \mid b? \mid \texttt{assert}(b)$$

The concrete domain C is also extended to be $\mathcal{P}(\Sigma) \times \mathit{Alarm}$. For each input $(X, AL) \in C$, AL is initially set to be empty. Command $\texttt{assert}(b)$ adds an alarm $(l, \texttt{assert(b)})$ ($l \in \mathbb{N}$ denotes the program location) when a state in X does not satisfy the boolean guard $b?$, which is defined as:

$$[\![\texttt{assert(b)}]\!]^{ext}(X, AL) = \begin{cases} (X, AL \cup \{l, (\texttt{assert(b)})\}) & \text{if } [\![b?]\!](X) \neq X \\ (X, AL) & \text{if } [\![b?]\!](X) = X \end{cases}$$

where $[\![b?]\!]$ filters the states that do not satisfy the guard b. Moreover, for a non-assertion basic command e, $[\![e]\!]^{ext}(X, AL) = ([\![e]\!](X), AL)$.

Equipped with assertions, $s \in \mathit{Prog}$ can raise alarms during executions. Moreover, a dataflow analysis $DA = (A, \alpha, \gamma, \Theta, M_A, \triangledown_A)$ decides whether to raise an alarm based on whether $M_A(b?)(\alpha(X))$ is equal to X for the input (X, AL). Next, we define the set of execution traces $\text{trace}(\rho)$ of each abstract program $\rho \in \mathit{AProg}$ in Figure 6, which consists of all the execution sequences of atomic abstract transfers without non-determinism and Kleene iterations intuitively. Note that each $\tau \in \text{trace}(\rho)$ is also an abstract program that simulates the analysis execution along the trace. Then if an assertion $e_a = \texttt{assert}(b)$ raises an alarm, we can export the trace $\tau \in \text{trace}(\rho)$ that finally reaches e_a.

Let $\mathit{Spec} = \{(|b|)\ \sigma = \text{True}\}$ which consists of states that satisfy the boolean guard $b?$, then we can use the value LC to under-estimate AT (the probability of this alarm) when given $\mathit{Dist}(DA, \tau)$.

For example, in Figure 5, we replace the assignments $x := x-1$ and $x := x+1$ with two assertions, then we get the program: $s = \mathit{lf}; ((x \leq 5?); \texttt{assert}(x < 3)) \oplus ((x > 5?); \texttt{assert}(x > 7))$. Recall that lf represents the use of a library function: $x := \texttt{select}(x)$ and we still use the interval analysis. $M_A(s)$ generates two traces: $\tau_1 = \theta_1;\theta_2;M_A(\texttt{assert}(x < 3))$ and $\tau_2 = \theta_1;\theta_3;M_A(\texttt{assert}(x > 7))$, where $\theta_1 = M_A(\mathit{lf})$, $\theta_2 = M_A(x \leq 5?)$ and $\theta_3 = M_A(x > 5?)$. For input $I_0 = \{1\ldots9\}$, both alarms will be raised since $(\theta_1;\theta_2)(\alpha(I_0)) = [1,5]$ and $(\theta_1;\theta_3)(\alpha(I_0)) = [6,12]$.

Following a similar derivation as in Section 6, the non-zero terms for $\langle\!\langle \tau \rangle\!\rangle^{lc}(I_0)(I_1)$ are:

$$\langle\!\langle \tau \rangle\!\rangle^{lc}(I_0)(I_1) = \frac{1}{C_{12}^3}\ (\{0,5,12\} \subseteq I_1 \subseteq \{0\ldots12\} \text{ and } |I_1 \models 9)$$

With a slight abuse, the value $LC(\tau_1)$ is computed as $LC(\tau_1) = \sum_{\{0,5,12\} \subseteq I_1 \subseteq \{0 \ldots 12\}, |I_2|=9} \frac{1}{C_{12}^9} = \frac{C_9^6}{C_{12}^9} \approx 0.382$. Similarly, $LC(\tau_2) = \frac{C_9^6}{C_{12}^9} \approx 0.382$.

Those values quantify the precision of raised alarms in the prior distribution. Then a Bayesian-style analysis can compute the posterior probability when a negative feedback is given to the alarm ($\texttt{assert}(x < 3)$ in τ_1 for our example). From the perspective of reject sampling, all the sampled concrete executions that cause this alarm will be dropped. Since for the semantics $\langle\langle \cdot \rangle\rangle$, τ_1 is guaranteed to be local complete for all remained executions, all these executions will raise alarms according to *Theorem* 4.2. Therefore, the Bayesian analysis will drop all these executions.

In our example, all concrete versions by which the produced output set contains $\{0, 5, 12\}$ given input $\{1 \ldots 9\}$ will be discarded, then the number of unique specifications of θ will decrease from C_{12}^9 to $C_{12}^9 - C_9^6$. As a result, $LC(\tau_2)$ will be adjusted to $\frac{C_9^3 - C_8^3}{C_{12}^9 - C_9^6} \approx 0.206$. (Intuitively, this probability can be understood as selecting 9 different numbers which contain 0, 6 and 12 from a uniform distribution formed by $1 \ldots 12$, with the condition that 0, 5 and 12 cannot be selected simultaneously.)

This change of LC illustrates the process in Bayesian program analysis where the posterior probabilities of alarms sharing common roots decrease through learning and generalization from the negative feedback. In the general sense, each trace τ in our syntax is a derivation chain, and all these chains can compose a Bayesian inference graph. Given feedback on an alarm on one trace τ, all abstract transfer functions along the trace turn to be mutually dependent and form a posterior joint distribution conditioned on the feedback. Then the Bayesian program analysis can compute the posterior probability for other alarms.

7.2 Practical Implementation and Optimizations

While our focus is to lay a theoretical foundation, we informally discuss a possible future implemented pipeline and potential opportunities for optimization.

Implementation Guidance. Our framework can be integrated as a lightweight *post-processing layer* on top of a conventional over-approximating analyzer. A standard analyzer run already induces an abstract execution (an "abstract program") ρ for each alarm (or for an alarm slice). The confidence layer then decomposes ρ along the program structure into atomic abstract programs/transfers, and assigns each atom a distribution over its concrete realizations. In practice, these distributions can be instantiated as common *parameterized families* (e.g., geometric/Poisson or other discrete priors) and their parameters can be estimated from code corpus or historical analyzer runs via statistical learning. Confidence is then computed *compositionally* using our semantics, without global program-space sampling; atomic summaries can be cached and reused across repeated substrings of ρ and across alarms.

When multiple alarms are raised, this naturally supports a *Bayesian*-style learning workflow: using user feedback, dynamic evidence, or targeted tests, one can update the statement-level distributions to posteriors (as discussed in Section 7.1), cache them, and reuse the learned posteriors in subsequent runs to improve ranking/filtering and precision quantification.

Garbage Collection of Dependent Basic Commands. Our semantics tracks the samples of all dependent basic commands, while in practice, not all dependent commands correlate with each other, but it is more often the case that they form independent clusters. As a result, for either our semantics, whenever it finishes sampling basic commands in a cluster, it can stop tracking all of them. In other words, it can assume the corresponding elements in t introduced in Section 5 are empty strings ε, which will shrink the support of the output distribution and marginalize over the samples for these commands. This improves the time and memory efficiency of the implementation.

7.3 The Sensitivity of LC to the Choice of Priors

The previous subsection discussed how to implement our approach. We now turn to a complementary question that affects this practical deployment: *how sensitive are the resulting LC estimates to the choice of the prior distribution over concrete realizations?*

First, the sensitivity of the calculated probabilities is heavily affected by the program structure. In particular, in our setting, the impact of prior changes on *LC* (Probability of Local Completeness) is often *attenuated by program structure*: certain statements dominate or "wash out" earlier uncertainty. For example, in our simple language, once a command like $x := x \times 0$ occurs, the value of x becomes deterministically 0; consequently, priors over *all* earlier assignments to x cannot affect the subsequent concrete/abstract outputs relevant to *LC*. More generally, strong reset/overwrite operations and boolean guards that eliminate large portions of the state space can reduce sensitivity by making the downstream behavior insensitive to upstream priors.

Second, the required accuracy of the prior depends on the intended use. If our framework is deployed as an alarm-ranking system as seen in many existing probabilistic-style analysis systems [10, 31, 39], it is relatively insensitive to the prior as long as it provides a useful separation signal—i.e., it tends to assign lower confidence to "suspicious" alarms than to those that are typically true—since ranking/filtering is the primary objective.

Finally, in such a Bayesian-learning pipeline, the influence of the prior diminishes as more posterior evidence is accumulated: as we observe more user feedback (true/false alarms) or additional runtime/analysis evidence, the posterior concentrates and the resulting *LC* estimates become progressively less sensitive to the initial prior.

7.4 The Gap Between $\langle\!\langle \cdot \rangle\!\rangle$ and $\langle\!\langle \cdot \rangle\!\rangle^{lc}$

Recall that $\langle\!\langle \cdot \rangle\!\rangle$ is the exact (oracle) semantics, while $\langle\!\langle \cdot \rangle\!\rangle^{lc}$ computes a lower bound obtained by *dropping potentially incomplete samples early* for efficiency. In this part, we discuss when $\langle\!\langle \cdot \rangle\!\rangle^{lc}$ is tight, when the lower bound may collapse toward zero, and which factors influence the gap.

When $\langle\!\langle \cdot \rangle\!\rangle^{lc}$ Is Tight? The only way $\langle\!\langle \cdot \rangle\!\rangle^{lc}$ can be non-tight is when early dropping removes samples that could have become locally complete after further composition (i.e., *repair-by-composition*). A necessary condition for such non-tightness is the existence of an atomic abstract transfer $M_A(e) = f^A : A \to A$ in the abstract program ρ that is *noninjective*:

$$\exists a_1, a_2 \in A.\ a_1 \sqsubset_A a_2 \ \wedge\ f^A(a_1) = f^A(a_2).$$

Intuitively, this means f^A admits the possibility of mapping multiple previous outputs to the same abstract state, creating repair opportunities. To reason about (and potentially mitigate) this loss, we define the *repair set* for a transfer f^A with its abstract input a_{in}:

$$\mathit{repair}(f^A, a_{in}) \ = \ \{\, a \mid a \sqsubset_A a_{in} \ \wedge\ f^A(a) = f^A(a_{in}) \,\}.$$

If the abstract outputs produced by the samples dropped earlier never fall into $\mathit{repair}(f^A, a_{in})$, then f^A cannot repair earlier incompleteness, and $\langle\!\langle \cdot \rangle\!\rangle^{lc}$ remains tight at this point. For example, consider $\rho_1 = f_1^A; f_1^A$ with $f_1^A = x := x +_A \mathbb{Z}^+$ using the simplest sign domain $A = \{\bot, \mathbb{Z}^+, \mathbb{Z}^-, \mathbb{Z}^0, \top\}$ and the input set $I = \{-3, 5\}$. We have $f_1^A(\alpha(I)) = \top$ and $\mathit{repair}(f_1^A, \top) = \{\mathbb{Z}^-\}$. However, f_1^A cannot produce $\mathbb{Z}^-$ from any abstract input (adding a positive number cannot yield a negative sign), hence no dropped outputs can ever be "repaired" by the second f^A, and $\langle\!\langle \cdot \rangle\!\rangle^{lc}$ is tight.

When Can the Lower Bound Collapse Toward Zero? This happens when the vast majority of samples are dropped earlier (many incomplete intermediate outputs), while a later transfer has strong repair potential (a large *repair set*), so that $\langle\!\langle \cdot \rangle\!\rangle$ recovers many of those samples but $\langle\!\langle \cdot \rangle\!\rangle^{lc}$ cannot. For example, consider $\rho_2 = f_2^A; f_1^A$ with $I = \{-3, 5\}$, $f_2^A = x := x +_A \mathbb{Z}^-$ yields $f_1^A(\alpha(I)) = \top$.

Many concrete executions $x := x + (-k)$ with $k > 5$ produce a concrete output that abstracts to $\mathbb{Z}^-$ and are thus dropped by $\langle\langle\cdot\rangle\rangle^{lc}$, but the subsequent $f_1^A = x := x +_A \mathbb{Z}^+$ may "repair" them (Recall $\mathbb{Z}^- \in \mathit{repair}(f_1^A, \top)$). An even stronger repair example is $f_3^A = x := x * \mathbb{Z}^0$, for which $\mathit{repair}(f_3^A, \top) = \{\mathbb{Z}^+, \mathbb{Z}^-, \mathbb{Z}^0\}$: multiplying by zero collapses many distinct signs to the fixed output $\mathbb{Z}^0$, which repairs a wide range of previous imprecision, amplifying the gap between $\langle\langle\cdot\rangle\rangle$ and $\langle\langle\cdot\rangle\rangle^{lc}$.

Abstract Domain vs. Transfer Design. The gap between $\langle\langle\cdot\rangle\rangle$ and $\langle\langle\cdot\rangle\rangle^{lc}$ depends largely on the *design of abstract transfers* (and their repair potential) rather than purely on the granularity of the abstract domain. Even in coarse domains, many transfers are effectively non-repairing, and thus do not induce loss. For example, in the interval domain Int, affine assignments $f_4^A : x := k \times x + b$ do not create repair opportunities in the above sense ($\mathit{repair}(f_4^A, a_{in}) = \emptyset$ for any $a_{in} \in \mathsf{Int}$), so $\langle\langle\cdot\rangle\rangle^{lc}$ tends to be tight. By contrast, boolean guards such as $f_5^A = (x < 4?)$ are often a source of repair potential: e.g., $\mathit{repair}(f_5^A, [0, 9]) = \{[0, u] \mid 3 \le u \le 8, u \in \mathbb{Z}\}$ contains many intervals, making repair-by-composition more likely.

Overall, our choice of $\langle\langle\cdot\rangle\rangle^{lc}$ represents an explicit precision–efficiency trade-off: it is exact when repair-by-composition is absent (or unreachable), and it can become loose when later transfers have the potential to repair previous dropped outputs.

7.5 How Widening Affects Confidence (the *LC* Value)

The widening operator $\triangledown_A$ affects our confidence values through how the analyzer computes the abstract execution of loops. Concretely, $\triangledown_A$ appears only in the fixpoint computation that defines the loop invariant/summary as shown in Figure 3. Since *LC* is defined by the event $\alpha(O) = O_A$ (abstract output equals the abstraction of the concrete output), different widening strategies may yield different O_A and hence different *LC* values. In practice, *more aggressive widening typically decreases LC*. Intuitively, aggressive widening tends to produce a coarser abstract invariant O_A, which is less likely to be exactly matched by $\alpha(O)$ of the concrete execution. Importantly, a low *LC* in this case is not a problem of our framework, as it accurately reflects the reality that the abstract operator/invariant is too coarse to be locally complete for most concrete realizations.

Example. Consider the loop program $((x < 7)?; x := x + \mathtt{step})^*$ with input $x = 0$, and assume the only imprecision is a coarse interval estimate for `step`, namely $[1, 2]$, and hence the concrete loop-body distribution is uniform over $\mathtt{step} \in \{1, 2\}$. With a trivial $\triangledown_A$, the interval-domain fixpoint is $fp = [0, \infty]$. In this case, $LC = 0$ since no concrete execution can produce $[0, \infty]$. If instead we apply a standard precision-improving approach [35] and use $[0, 0] \sqcup_{\mathsf{Int}} \rho(fp) = [0, 8]$ as the loop invariant (still a sound fixpoint), then *LC* increases: local completeness holds *iff* $\mathtt{step} = 2$, since only then does the concrete output $\{0, 2, 4, 6, 8\}$ abstract to $[0, 8]$. This illustrates that widening choices can materially affect *LC*, and that improving loop invariants (via less aggressive widening, narrowing, or related heuristics) can directly improve the *LC* value.

8 Limitations and Future Direction of Improvement

Our confidence is grounded in local completeness and thus inherits several practical limitations which will be discussed in this part. Note that the complexity of an analyzer or the coarseness of a domain does not inherently imply a smaller *LC*; rather, it primarily affects the cost and tightness of our *computation* of confidence using $\langle\langle\cdot\rangle\rangle^{lc}$, and we will mainly focus on it.

Complex Analyzers and the Gap Between $\langle\langle\cdot\rangle\rangle$ and $\langle\langle\cdot\rangle\rangle^{lc}$. For analyzers with a large number of atomic transfers, the gap between $\langle\langle\cdot\rangle\rangle$ and $\langle\langle\cdot\rangle\rangle^{lc}$ may widen. Intuitively, longer abstract executions increase the chance of *repair-by-composition*: later transfers can "repair" earlier incompleteness, whereas $\langle\langle\cdot\rangle\rangle^{lc}$ may have already dropped those samples as discussed in Section 7.4. Consequently, in complex analyzers, $\langle\langle\cdot\rangle\rangle^{lc}$ can be conservative and sometimes low. We emphasize

that the magnitude of this gap depends more on the *design of abstract transfers* (how much later transfers can repair earlier losses) than on the abstract domain being coarse as in Section 7.4.

Domain Expressiveness vs. Analysis-Induced Imprecision. Our framework measures *analysis-induced* imprecision (e.g., widenings), not limitations of the abstract domain's expressive power. Therefore, it is best suited for *reasonably expressive* domains where the specification is expressible (the standard assumption stated in Section 4.2). If the specification is not expressible, false alarms may persist even when LC holds; addressing such cases requires domain enrichment.

A Future Direction: Partial Completeness. A promising way to reduce conservativeness is to relax strict LC to the notion of *ε-partial-completeness* [7, 8], allowing ε-bounded intermediate loss that may later be recovered. This connects naturally to the repair-set perspective in Section 7.4: instead of dropping all non-locally-complete samples, one can retain those whose error remains within the bound ε and discard only when the accumulated error exceeds ε, preserving compositionality while potentially tightening our confidence.

9 Related Work

Completeness in Abstract Interpretation. Completeness, a foundational concept in abstract interpretation and static program analysis, has long been studied [3, 4, 19] to characterize precise program behaviors under approximations. Recent work explores classes of complete programs for specific abstractions. Giacobazzi et al. [19] propose the idea of completeness class, defined as the collection of programs that achieve completeness with regard to a given abstract domain. To weaken the concept of completeness, the work [4] introduces the notion of local completeness, that is, completeness among certain program traces. Our approach can be considered as a further weakening of local completeness, as we calculate the probability of local completeness among all possible concrete programs to measure and reason about the imprecision induced by analysis.

Measuring the Imprecision of Abstract Interpretation. Numerous works have addressed the problem of measuring the imprecision of abstract interpretation and static analysis. By defining distances or metrics in logic programming domains [9] or numeric domains [33, 44], these approaches can estimate the loss of precision during the analysis process. A paper in 2022 [7] proposes a weaker notion of completeness (called partial completeness) and a general framework which can estimate an upper bound to the imprecision accumulated by the abstract interpreter. All the notions of distances defined in the above works can be plugged in their framework. While these works measure how far the analysis results are from precise results based on analysis-specific metrics, our approach measures how likely an analysis is complete from a probabilistic perspective.

Probabilistic Abstract Interpretation. The reader might confuse our work with probabilistic abstract interpretation [16, 17] due to the title, but the purposes of the two works are completely different. Probabilistic abstract interpretation aims to extend the conventional abstract interpretation to static program analysis on probabilistic programs, and provide theoretical foundations on reasoning about probabilistic properties. However, it still only provides soundness guarantees and cannot quantify the imprecision. It would be interesting to extend our approach to these analyses.

10 Conclusion

We address the challenge of quantifying confidence in abstract interpretation by introducing a theoretical framework. We formalize the problem as calculating the probability of an abstract interpreter being locally complete on a program sampled from the distribution of programs consistent with it. To solve it, we propose two compositional denotational semantics and prove their correctness.

Acknowledgments

We would like to thank the anonymous reviewers for their feedback and comments. This work was sponsored by the National Natural Science Foundation of China (NSFC) under Grant No. W2411051.

References

[1] Sam Blackshear and Shuvendu K Lahiri. 2013. Almost-correct specifications: A modular semantic framework for assigning confidence to warnings. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 209–218. doi:10.1145/2491956.2462188

[2] Marcel Böhme. 2022. Statistical reasoning about programs. In *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: New Ideas and Emerging Results*. 76–80. doi:10.1145/3510455.3512796

[3] Roberto Bruni, Roberto Giacobazzi, Roberta Gori, Isabel Garcia-Contreras, and Dusko Pavlovic. 2019. Abstract extensionality: on the properties of incomplete abstract interpretations. *Proceedings of the ACM on Programming Languages* 4, POPL (2019), 1–28. doi:10.1145/3371096

[4] Roberto Bruni, Roberto Giacobazzi, Roberta Gori, and Francesco Ranzato. 2021. A logic for locally complete abstract interpretations. In *Proceedings of the 36th Annual ACM/IEEE Symposium on Logic in Computer Science* (Rome, Italy) *(LICS '21)*. Association for Computing Machinery, New York, NY, USA, Article 52, 13 pages. doi:10.1109/LICS52264.2021.9470608

[5] Roberto Bruni, Roberto Giacobazzi, Roberta Gori, and Francesco Ranzato. 2022. Abstract interpretation repair. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (San Diego, CA, USA) *(PLDI 2022)*. Association for Computing Machinery, New York, NY, USA, 426–441. doi:10.1145/3519939.3523453

[6] Roberto Bruni, Roberto Giacobazzi, Roberta Gori, and Francesco Ranzato. 2022. Abstract interpretation repair. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. 426–441. doi:10.1145/3519939.3523453

[7] Marco Campion, Mila Dalla Preda, and Roberto Giacobazzi. 2022. Partial (In)Completeness in abstract interpretation: limiting the imprecision in program analysis. *Proc. ACM Program. Lang.* 6, POPL, Article 59 (Jan. 2022), 31 pages. doi:10.1145/3498721

[8] Marco Campion, Mila Dalla Preda, Roberto Giacobazzi, and Caterina Urban. 2026. A Logic for the Imprecision of Abstract Interpretations. *Proc. ACM Program. Lang.* 10, POPL, Article 65 (Jan. 2026), 29 pages. doi:10.1145/3776707

[9] Ignacio Casso, José F Morales, Pedro López-García, Roberto Giacobazzi, and Manuel V Hermenegildo. 2019. Computing abstract distances in logic programs. In *International Symposium on Logic-Based Program Synthesis and Transformation*. 57–72. arXiv:1907.13263 http://arxiv.org/abs/1907.13263

[10] Tianyi Chen, Kihong Heo, and Mukund Raghothaman. 2021. Boosting static analysis accuracy with instrumented test executions. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1154–1165. doi:10.1145/3468264.3468626

[11] Patrick Cousot. 2021. *Principles of abstract interpretation*. MIT Press. doi:10.1145/3546953

[12] Patrick Cousot and Radhia Cousot. 1977. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages* (Los Angeles, California) *(POPL '77)*. Association for Computing Machinery, New York, NY, USA, 238–252. doi:10.1145/512950.512973

[13] Patrick Cousot and Radhia Cousot. 1977. Static determination of dynamic properties of generalized type unions. *ACM SIGOPS Operating Systems Review* 11, 2 (1977), 77–94. doi:10.1145/800022.808314

[14] Patrick Cousot and Radhia Cousot. 1979. Systematic design of program analysis frameworks. In *Proceedings of the 6th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages* (San Antonio, Texas) *(POPL '79)*. Association for Computing Machinery, New York, NY, USA, 269–282. doi:10.1145/567752.567778

[15] Patrick Cousot and Radhia Cousot. 1992. Abstract Interpretation Frameworks. *J. Log. Comput.* 2 (1992), 511–547. https://api.semanticscholar.org/CorpusID:6384543

[16] Patrick Cousot and Michael Monerau. 2012. Probabilistic abstract interpretation. In *European Symposium on Programming*. Springer, 169–193. doi:10.1007/978-3-642-28869-2_9

[17] Alessandra Di Pierro and Herbert Wiklicky. 2000. Measuring the precision of abstract interpretations. In *International Workshop on Logic-Based Program Synthesis and Transformation*. Springer, 147–164.

[18] Dino Distefano, Manuel Fähndrich, Francesco Logozzo, and Peter W. O'Hearn. 2019. Scaling static analyses at Facebook. *Commun. ACM* 62, 8 (July 2019), 62–70. doi:10.1145/3338112

[19] Roberto Giacobazzi, Francesco Logozzo, and Francesco Ranzato. 2015. Analyzing program analyses. *ACM SIGPLAN Notices* 50, 1 (2015), 261–273. doi:10.1145/2676726.2676987

[20] Roberto Giacobazzi, Francesco Ranzato, and Francesca Scozzari. 2000. Making abstract interpretations complete. *J. ACM* 47, 2 (March 2000), 361–416. doi:10.1145/333979.333989

[21] Kihong Heo, Mukund Raghothaman, Xujie Si, and Mayur Naik. 2019. Continuously reasoning about programs using differential bayesian inference. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 561–575. doi:10.1145/3314221.3314616

[22] Yungbum Jung, Jaehwang Kim, Jaeho Shin, and Kwangkeun Yi. 2005. Taming false alarms from a domain-unaware C analyzer by a bayesian statistical post analysis. In *International Static Analysis Symposium*. Springer, 203–217. doi:10.1007/11547662_15

[23] Hyunsu Kim, Mukund Raghothaman, and Kihong Heo. 2022. Learning Probabilistic Models for Static Analysis Alarms. In *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022*. ACM, 1282–1293. doi:10.1145/3510003.3510098

[24] Ugur Koc, Parsa Saadatpanah, Jeffrey S Foster, and Adam A Porter. 2017. Learning a classifier for false positive error reports emitted by static code analysis tools. In *Proceedings of the 1st ACM SIGPLAN international workshop on machine learning and programming languages*. 35–42. doi:10.1145/3088525.3088675

[25] Dexter Kozen. 1997. Kleene algebra with tests. *ACM Trans. Program. Lang. Syst.* 19, 3 (May 1997), 427–443. doi:10.1145/256167.256195

[26] Ted Kremenek, Ken Ashcraft, Junfeng Yang, and Dawson Engler. 2004. Correlation exploitation in error ranking. *ACM SIGSOFT Software Engineering Notes* 29, 6 (2004), 83–93. doi:10.1145/1029894.1029909

[27] Ted Kremenek and Dawson Engler. 2003. Z-ranking: Using statistical analysis to counter the impact of static analysis approximations. In *Static Analysis: 10th International Symposium, SAS 2003 San Diego, CA, USA, June 11–13, 2003 Proceedings 10*. Springer, 295–315. web.stanford.edu/~engler/sas-camera-ready.pdf

[28] Wei Le and Mary Lou Soffa. 2010. Path-based fault correlations. In *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*. 307–316. doi:10.1145/1882291.1882336

[29] Seongmin Lee and Marcel Böhme. 2023. Statistical reachability analysis. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 326–337. doi:10.1145/3611643.3616268

[30] Woosuk Lee, Wonchan Lee, and Kwangkeun Yi. 2012. Sound non-statistical clustering of static analysis alarms. In *Verification, Model Checking, and Abstract Interpretation: 13th International Conference, VMCAI 2012, Philadelphia, PA, USA, January 22-24, 2012. Proceedings 13*. Springer, 299–314. doi:10.1145/3095021

[31] Tianchi Li and Xin Zhang. 2025. Combining Formal and Informal Information in Bayesian Program Analysis via Soft Evidences. *Proc. ACM Program. Lang.* 9, OOPSLA1, Article 143 (April 2025), 28 pages. doi:10.1145/3720508

[32] Benjamin Livshits and Shuvendu K Lahiri. 2014. In defense of probabilistic static analysis. APPROX.

[33] Francesco Logozzo, Corneliu Popeea, and Vincent Laviron. 2009. Towards a quantitative estimation of abstract interpretations. In *Workshop on Quantitative Analysis of Software*. Citeseer. www.microsoft.com/en-us/research/wp-content/uploads/2009/06/quantitative.ai_.pdf.pdf

[34] Ravi Mangal, Xin Zhang, Aditya V Nori, and Mayur Naik. 2015. A user-guided approach to program analysis. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. 462–473. doi:10.1145/2786805.2786851

[35] Antoine Miné. 2017. Tutorial on Static Inference of Numeric Invariants by Abstract Interpretation. *Found. Trends Program. Lang.* 4, 3–4 (Dec. 2017), 120–372. doi:10.1561/2500000034

[36] Peter W O'Hearn. 2018. Continuous reasoning: Scaling the impact of formal methods. In *Proceedings of the 33rd annual ACM/IEEE symposium on logic in computer science*. 13–25. doi:10.1145/3209108.3209109

[37] Peter W O'Hearn. 2019. Incorrectness logic. *Proceedings of the ACM on Programming Languages* 4, POPL (2019), 1–32. doi:10.1145/3371078

[38] Nikhil Parasaram, Earl T Barr, Sergey Mechtaev, and Marcel Böhme. 2023. Precise Data-Driven Approximation for Program Analysis via Fuzzing. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 611–623. doi:10.1109/ASE56229.2023.00185

[39] Mukund Raghothaman, Sulekha Kulkarni, Kihong Heo, and Mayur Naik. 2018. User-guided program reasoning using Bayesian inference. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 722–735. doi:10.1145/3192366.3192417

[40] Henry Gordon Rice. 1953. Classes of recursively enumerable sets and their decision problems. *Transactions of the American Mathematical society* 74, 2 (1953), 358–366. ww2.ams.org/journals/tran/1953-074-02/S0002-9947-1953-0053041-6/S0002-9947-1953-0053041-6.pdf

[41] Xavier Rival and Kwangkeun Yi. 2020. *Introduction to static analysis: an abstract interpretation perspective*. Mit Press.

[42] Caitlin Sadowski, Edward Aftandilian, Alex Eagle, Liam Miller-Cushon, and Ciera Jaspan. 2018. Lessons from building static analysis tools at google. *Commun. ACM* 61, 4 (2018), 58–66. doi:10.1145/3188720

[43] Yuanfeng Shi, Yifan Zhang, and Xin Zhang. 2025. On Abstraction Refinement for Bayesian Program Analysis. *Proc. ACM Program. Lang.* 9, OOPSLA2, Article 388 (Oct. 2025), 27 pages. doi:10.1145/3763166

[44] Pascal Sotin. 2010. Quantifying the precision of numerical abstract domains. (2010). https://inria.hal.science/inria-00457324/document

[45] Omer Tripp, Salvatore Guarnieri, Marco Pistoia, and Aleksandr Aravkin. 2014. Aletheia: Improving the usability of static security analysis. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. 762–774. doi:10.1145/2660267.2660339

[46] Glynn Winskel. 1993. *The formal semantics of programming languages: an introduction*. MIT Press, Cambridge, MA, USA.

[47] Xin Zhang, Radu Grigore, Xujie Si, and Mayur Naik. 2017. Effective interactive resolution of static analysis alarms. *Proceedings of the ACM on Programming Languages* 1, OOPSLA (2017), 1–30. doi:10.1145/3133881

[48] Xin Zhang, M. Naik, and Hongseok Yang. 2013. Finding optimum abstractions in parametric dataflow analysis. *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation* (2013). doi:10.1145/2499370.2462185

[49] Xin Zhang, Xujie Si, and Mayur Naik. 2017. Combining the logical and the probabilistic in program analysis. In *Proceedings of the 1st ACM SIGPLAN International Workshop on Machine Learning and Programming Languages* (Barcelona, Spain) *(MAPL 2017)*. Association for Computing Machinery, New York, NY, USA, 27–34. doi:10.1145/3088525.3088563

[50] Yifan Zhang, Yuanfeng Shi, and Xin Zhang. 2024. Learning Abstraction Selection for Bayesian Program Analysis. *Proceedings of the ACM on Programming Languages* 8, OOPSLA1 (2024), 954–982. doi:10.1145/3649845