A Debugger for Probabilistic Programs

Alexander Hoppen and Thomas Noll[0000−0002−1865−1798]

Software Modelling and Verification Group
RWTH Aachen University, 52056 Aachen, Germany
alexander.hoppen@rwth-aachen.de, noll@cs.rwth-aachen.de

Abstract. Debuggers play an integral role in modern software development workflows. They not only aid a software developer in finding and removing bugs in a faulty program but also allow to explore its semantics in an interactive way. This is especially useful for software that exhibits non-deterministic behaviour, such as probabilistic programs, for which debugging support is currently scarce. In this paper, we provide a theoretical foundation for recording-based debuggers for imperative probabilistic programs supporting randomised choice, conditioning, and loops. In order to handle different branches of execution, we take a semantics-based approach that employs weakest preexpectations and that introduces iteration bounds for approximating the behaviour of potentially non-terminating loops. Moreover, we present a prototype implementation of our framework.

Keywords: probabilistic programs · debugging · semantics-based methods · weakest preexpectations

1 Introduction

Debugging plays an integral role in modern software development workflows. In its broad sense, it is understood as the process of finding and resolving *bugs* (that is, defects or problems that prevent correct operation) within computer programs (or hardware). Debugging strategies can involve interactive debugging, various forms of testing, monitoring and profiling, and many more techniques. Here we are particularly interested in *interactive* debugging, which allows software to be inspected at the source code level by stepping through commands, setting breakpoints, and examining and changing variable values whenever execution is paused. Many programming languages and software development environments offer tool support to aid in this activity, known as *debuggers* [13]. While traditional debuggers only allow forward execution of a program, *recording-based* debuggers like Mozilla's `rr` debugger [4] also enable the user to jump to previous execution states.

In contrast to their name (and their original intention), debuggers not only aid a software developer in finding and removing bugs in faulty programs. Rather, the possibility of executing a program step-by-step and inspecting variable values at intermediate states allows a software engineer who is unfamiliar with a

program to explore its semantics in an interactive way. This is especially useful in situations where the program's semantics are non-obvious at first glance. Such situations can include programs that exhibit *non-deterministic* control flow due to, e.g., concurrency or stochastic behaviour. Here, it is crucial that a software developer is able to follow several paths of execution in order to cover all scenarios.

This, in particular, applies to *probabilistic programs*, which implement a programming paradigm in which probabilistic models are specified using a probabilistic programming language and inference for these models is performed automatically. It attempts to unify probabilistic modelling and traditional general-purpose programming in order to make the former easier and more widely applicable [7].

Use cases of probabilistic programs include both randomised variants of deterministic algorithms like randomised Quicksort, which compute a deterministic result with a non-deterministic runtime behaviour, as well as the description of complex probability distributions in a structured, algorithmic way. In this paper, we focus our attention on the latter. Such programs typically terminate almost surely, that is with probability one.

We note that debuggers for probabilistic programming languages, both traditional and recording-based, are currently scarce.

In this paper, we provide the theoretical foundations on which recording-based debuggers for probabilistic programming languages can be built. As indicated before, the key obstacle that needs to be overcome in order to build such a debugger is that in the presence of probabilistic branching operations, there no longer exists a unique execution path. To resolve this execution ambiguity, we introduce two new debugger commands *Step Into True* and *Step Into False* that allow the user to jump into one of the two execution branches in such probabilistic cases. These commands are supported by implicit *observations* that restrict the execution to exactly one of the two execution branches. In addition, we employ loop iteration bounds for approximating the behaviour of potentially non-terminating loops, thereby making the program's semantics computable.

The remainder of this paper is organised as follows. We continue in Section 2 with a discussion of related work, followed by the introduction of the probabilistic programming language employed in this paper in Section 3. Section 4 provides the semantic foundations of our debugging approach in terms of weakest pre-expectations, and Section 5 describes the approximation of loops by iteration bounds. Section 6 constitutes the core part of our paper, in which we formally investigate how to compute weakest preexpectations for intermediate states of a program's execution. Finally, Section 7 sketches a prototype implementation of our approach, and Section 8 concludes with some final remarks and possible directions for future work.

2 Related Work

Owing to the importance of debugging within the software development life cycle, there exists extensive literature on general debugging approaches [21, 24], debuggers [13, 1], and language-specific techniques for e.g. C++ [22, 9] and Java [2, 3].

However, as mentioned in the introduction, debugging support for probabilistic programming languages is currently scarce. The only approach we are aware of is the DePP tool [16], which employs a programming model that supports constructs for invoking inference in the language and that represents such inference operations using an extended form of Bayesian networks. This allows to automatically identify programming errors such as the assumption of independence between variables that are actually correlated, or premature inference operations. However, in contrast to our approach, analysing the detailed behaviour of probabilistic program by interactive, step-wise execution is not supported.

Another important development is the Storm framework [6]. It does not itself provide debugging features but allows the reduction of probabilistic programs by leveraging both generic code/data transformations from compiler testing and domain-specific, probabilistic transformations. This way, it enables simplified debugging and localisation of faults.

Moreover, probabilistic reasoning has been employed to improve classical debugging by computing a probabilistic model of the dependencies in the program, allowing to focus on the "most impacted" parts of a program [20], to localise faults more effectively [25, 17], or to combine human-like reasoning with program semantics-based analysis [23].

It is also worth mentioning that probabilistic programming is not the only domain where debugging activities are complicated by non-determinism. Debugging concurrent programs, for example, is extremely challenging due to non-determinism in thread scheduling [19, 14].

Weakest precondition reasoning was established in a classical setting by Dijkstra [5] and has been extended to provide semantic foundations for probabilistic programs by McIver and Morgan [15], who also coined the term *weakest preexpectations (WP)*. Their relation to operational models is studied in [8]. Moreover, weakest precondition reasoning has been shown to be useful for obtaining bounds on the expected resource consumption [18] and, in particular, the expected runtime [12] of probabilistic programs. To the best of our knowledge, the present paper is the first one to apply this approach as a semantic framework for debugging and to offer interactive debugging support for probabilistic programs.

3 Probabilistic Programs

To start, let us introduce the probabilistic programming language which we will be working with in this paper. It has a C-like syntax and supports the standard probabilistic constructs of probabilistic choice and `observe`-statements. An example of such a program can be seen in Figure 1. `prob`-statements behave

similar to `if`-statements but execute the `prob`-branch with the given probability and the (optional) `else`-branch otherwise. An `observe`-statement only continues program execution for those execution branches that satisfy its condition.

```
1  bool aliceInfectious = true
2  bool bobInfected = false
3  while aliceInfectious {
4    prob 0.1 {
5      bobInfected = true
6    }
7    prob 0.6 {
8      aliceInfectious = false
9    }
10 }
```

Fig. 1. A structured probabilistic program modelling the transmission of the SARS-CoV2 virus.

Example 1. Figure 1 displays a valid probabilistic program, modelling a hypothetical transmission of the SARS-CoV2 virus.[1] Suppose there are two people, Alice and Bob, who meet every day. Alice is initially infected with the virus. While she is infectious, every time they meet, there is a 10% chance that she infects Bob. The next day she continues to be infectious with a likelihood of 40%.

We would now like to know with which probability Bob also catches the virus, i.e. the likelihood that `bobInfected` is `true` at the end of program execution. Using weakest preexpectations as described in the following sections, one can show that this probability is 15.62%.

Definition 1 (Abstract syntax). *A program P is either*

- *the empty program $P = ()$ or*
- *a single statement s together with the line number[2] $l \in \mathbb{N}$ at which this statement started: $P = (l, s)$, or*
- *the sequential composition $P = (P_1, P_2)$ of two programs, where P_2 is executed after P_1.*

Let $\mathbb{Q}$ be the set of all valid programs and let $\mathcal{V}$ be the set of variables defined in a program $P \in \mathbb{Q}$. We refer to a program $P = (l, s)$ as $P = s$ if we do not care about the line number.

Definition 2 (Observe-removed program). *Given a program $P \in \mathbb{Q}$, let $\overline{P}$ be the program in which all* `observe`*-statements have been removed.*

[1] The example does not claim to be medically correct in any way.

[2] For simplicity's sake, we assume that there is only one statement per line. Otherwise information about the starting column needs to be included in l to make each statement-line-number-combination unique.

4 Weakest Preexpectations

A standard way of capturing the semantics of a probabilistic program is the computation of *weakest preexpectations*. For a thorough explanation of these, we refer to [15]. In the following, we give a short recap and apply the standard definitions to our language.

Definition 3 (Iverson brackets). *Given a boolean value $b \in \mathbb{B}$, we define the Iverson brackets as*

$$[\![b]\!] = \begin{cases} 0 & \text{if } b = \texttt{false} \\ 1 & \text{if } b = \texttt{true} \end{cases}$$

Definition 4 (Variable assignment function). *We define the set of partial functions that map declared variables to their values as $\mathbb{A} = \big(\mathcal{V} \rightharpoonup (\mathbb{B} \cup \mathbb{R})\big)$.*

Definition 5 (Expectation). *An* expectation *is a function that maps variable assignments to probabilities. We define $\mathbb{P} = \{\, p \in \mathbb{R} \mid 0 \leq p \leq 1 \,\}$ as the set of all probabilities and $\mathbb{E} = (\mathbb{A} \to \mathbb{P})$ as the set of all expectations.*[3]

In the following, we restrict ourselves to such expectations that can be expressed as finite arithmetic expressions over $\mathcal{V}$. An expectation refers to both the function and the arithmetic expression that describes it. It will become clear from the context which representation is meant.

Definition 6 (Variable replacement). *For an expectation $f \in \mathbb{E}$, we let $f[var \coloneqq expr]$ denote the expectation that results from replacing all occurrences of $var \in \mathcal{V}$ in the definition of f by the arithmetic expression $expr$ over $\mathcal{V}$.*

Example 2. For $f = [\![\mathsf{y} = 1]\!] \in \mathbb{E}$ we have $f[\mathsf{y} \coloneqq 2] = [\![2 = 1]\!] = 0$.

Lemma 1. *$\mathbb{E}$ forms a complete partial order and the lfp and gfp fixed-point operators are thus well-defined for continuous functions on $\mathbb{E}$ (cf. [15, p. 183]).*

Given these preliminaries, we can now define the weakest preexpectation operator wp.

Definition 7 (wp). *For a program $P \in \mathbb{Q}$ and an expectation $f \in \mathbb{E}$, the weakest preexpectation $\mathsf{wp}(P, f)$ of $f \in \mathbb{E}$ is defined according to Figure 2.*

Lemma 2. *The wp operator is Scott-continuous and thus also monotonic in the second component (cf. [8, Lemma 34]).*

[3] In [15, p. 16] expectations are defined as functions that map to $\mathbb{R}_{\geq 0}$. We are more restrictive by choosing the image $\mathbb{P}$ since we are only interested in computing the probability that a variable takes a certain value, i.e. the weakest preexpectation of an expectation $f = [\![x = \xi]\!]$ with $x \in \mathcal{V}$ and ξ being a constant. The estimation of the approximation error in Section 5.3 also requires this restricted image.

P	$\mathsf{wp}(P, f)$
$()$	f
(P_1, P_2)	$\mathsf{wp}(P_1, \mathsf{wp}(P_2, f))$
(`int` | `float` | `bool` | ε) *var* `=` *expr*	$f[var \coloneqq expr]$
`observe` *expr*	$[\![expr]\!] \cdot f$
`if` *expr* `{` P_{if} `}` `else` `{` P_{else} `}`	$[\![expr]\!] \cdot \mathsf{wp}(P_{\text{if}}, f) + [\![\neg expr]\!] \cdot \mathsf{wp}(P_{\text{else}}, f)$
`prob` *expr* `{` P_{if} `}` `else` `{` P_{else} `}`	$expr \cdot \mathsf{wp}(P_{\text{if}}, f) + (1 - expr) \cdot \mathsf{wp}(P_{\text{else}}, f)$
`while` *expr* `{` P_{body} `}`	$\mathsf{lfp}\ X.\big([\![expr]\!] \cdot \mathsf{wp}(P_{\text{body}}, X) + [\![\neg expr]\!] \cdot f\big)$

Fig. 2. Transformation functions of the wp operator.

In order to account for violated `observe`-statements, the weakest preexpectation returned by the wp operator needs to be normalised with respect to the probability that all `observe`-statements are satisfied during the run, which can be computed using the weakest liberal preexpectation operator as $\mathsf{wlp}(P, 1)$.

Definition 8 (wlp). *The definition of* wlp *is equal to the definition of* wp *(while recursing to* wlp*) with the only difference that we use the greatest fixed point* gfp *instead of the least fixed point* lfp *for loops.*

Lemma 3. wlp *is Scott-continuous and monotonic in the second component like* wp *(cf. [8, Lemma 34]).*

Definition 9 (Conditional weakest preexpectation). *The probability that a postexpectation f is satisfied after program execution, given that all* `observe`*-statements were satisfied during the run, is represented by the conditional weakest preexpectation $\frac{\mathsf{wp}(P,f)}{\mathsf{wlp}(P,1)}$. This is the value we are interested in computing.*

In Section 5.3 we will employ a new *weakest observe-ignoring preexpectation* operator woip as a correction term.

Definition 10 (woip). *The* woip *operator is defined like* wp *(recursing to* woip*) but ignores* `observe`*-statements, that is* $\mathsf{woip}(\texttt{observe}\ expr, f) = f$.

Observation 1. We have $\mathsf{woip}(P, f) = \mathsf{wp}(\overline{P}, f)$ where $\overline{P}$ is the observe-removed program defined in Definition 2.

Lemma 4. *The* woip *operator is Scott-continuous and thus also monotonic in the second component.*

Proof. Since $\mathsf{woip}(P, f) = \mathsf{wp}(\overline{P}, f)$, woip inherits the properties of wp described in Lemma 2.

5 Loop Iteration Bounds

While the definition of weakest preexpectations offers a well-founded basis for determining a probabilistic program's semantics, it does not directly provide an

algorithmic access because the least (resp. greatest) fixed point of a loop is not computable in general. To solve this problem, we introduce *loop iteration bounds*. Intuitively, an iteration bound for a loop in a program P is a non-negative integer $b \in \mathbb{N}$. Whenever the execution of P would traverse the loop more than b times, we declare that execution branch as having diverged and stop its execution.

Observation 2. We can view the loop iteration bounds as a syntactic transformation of the program. When imposing an iteration bound b on a loop, we are replacing the loop by a series of b `if`-statements as follows:

```
1 if expr {
2   P_body
3 }
4 // repeat if-statement b times
5
6 if expr {
7   while true {}
8 }
```

Observation 3. The weakest preexpectations of the code snippet in Observation 2 are computable because for the concluding loop $P =$ `while true {}`, we know that $\mathsf{wp}(P, f) = 0$, $\mathsf{wlp}(P, f) = 1$ and $\mathsf{woip}(P, f) = 0$ for all $f \in \mathbb{E}$.

Essentially, what we are doing when introducing loop iteration bounds is trading computability for correctness since we are now *approximating* loops. We are able to do so because we do not need to compute values that are correct to the last decimal digit. For applications like debugging, it is usually sufficient to compute approximate values for variables together with an approximation error. Therefore, we need to find suitable loop iteration bounds that are large enough to yield results with the desired accuracy while at the same time being as small as possible to ensure efficiency.

Definition 11 (Loop iteration bounds). *The loop iteration bounds of a program P are described by a partial function $\beta \colon \mathbb{Q} \rightharpoonup \mathbb{N}$ that assigns to each subprogram P of the form $(l,$* `while` *expr* `{` P_{body} `}`$)$ *an integer $\beta(P)$ that limits the number of times this loop can be traversed.*

5.1 Bounded Preexpectation Transformers

With this intuitive definition of loop iteration bounds at hand, let us define the wp_β, wlp_β and woip_β operators that compute the weakest (liberal, observe-ignoring) preexpectation of a postexpectation $f \in \mathbb{E}$ with respect to a program $P \in \mathbb{Q}$ when taking loop iteration bounds β into account.

Definition 12 (wp_β). *The weakest preexpectation transformer $\mathsf{wp}_\beta(P, f)$ of a program P while respecting loop iteration bounds β is analogous to that of* wp *in*

Definition 7 (while recursing to wp_β) for all non-loop statements P. For $P =$ `while` $expr$ `{` P_{body} `}`, we have:

$$\begin{aligned} &\mathsf{wp}_\beta(P, f) = \lambda^{\beta(P)}([\![\neg expr]\!] \cdot f) \\ &\textit{with} \quad \lambda(f) = [\![expr]\!] \cdot \mathsf{wp}_\beta(P_{\text{body}}, f) + [\![\neg expr]\!] \cdot f \end{aligned}$$

Observation 4. Putting the definition of wp_β into context with the code snippet in Observation 2, $([\![\neg expr]\!] \cdot f)$ evaluates lines 6–8. Applying the λ function $\beta(P)$ times evaluates the unrolled `if` $expr$ `{` P_{body} `}` statement $\beta(P)$ times.

Lemma 5. *wp_β is linear and monotonic in the second component.*

Proof. Follows by structural induction on the program. The non-loop cases are trivial. For loops, wp_β applies the λ function $\beta(P)$ times. Since λ is a linear combination of linear and monotonic functions (by induction), it enjoys the same properties, entailing the linearity and monotonicity of wp_β.

Definition 13 (wlp_β). *The operator wlp_β is defined analogously to wlp in Definition 8 (while recursing to wlp_β) for all non-loop statements P. For $P =$ `while` $expr$ `{` P_{body} `}`, we have:*

$$\begin{aligned} &\mathsf{wlp}_\beta(P, f) = \mu^{\beta(P)}([\![expr]\!] + [\![\neg expr]\!] \cdot f) \\ &\textit{with} \quad \mu(f) = [\![expr]\!] \cdot \mathsf{wlp}_\beta(P_{\text{body}}, f) + [\![\neg expr]\!] \cdot f \end{aligned}$$

Lemma 6. *wlp_β is monotonic in the second component.*

Proof. Follows by structural induction on the program. The non-loop cases have the same definition as wp_β and thus the proof carries over. For loops one proves the lemma by induction on the loop iteration bound. **Base case** $(\beta(P) = 0)$**.**

$$\begin{aligned} &\mathsf{wlp}_\beta(P, f) \leq \mathsf{wlp}_\beta(P, g) \\ \Leftrightarrow\ &\mu^0([\![e]\!] + [\![\neg e]\!] \cdot f) \leq \mu^0([\![e]\!] + [\![\neg e]\!] \cdot g) && \text{(Definition 13)} \\ \Leftarrow\ &f \leq g && \text{(simplify)} \end{aligned}$$

Induction case $(b = \beta(P) > 0)$**.**

$$\begin{aligned} &\mathsf{wlp}_\beta(P, f) \leq \mathsf{wlp}_\beta(P, g) \\ \Leftrightarrow\ &\mu^b([\![e]\!] + [\![\neg e]\!] \cdot f) \leq \mu^b([\![e]\!] + [\![\neg e]\!] \cdot g) && \text{(Definition 13)} \\ \Leftrightarrow\ &\mu(F) \leq \mu(G) && (F \coloneqq \mu^{b-1}([\![e]\!] + [\![\neg e]\!] \cdot f),\ G \text{ analogous}) \\ \Leftrightarrow\ &[\![e]\!] \cdot \mathsf{wlp}_\beta(P_{\text{b}}, F) + [\![\neg e]\!] \cdot F \leq [\![e]\!] \cdot \mathsf{wlp}_\beta(P_{\text{b}}, G) + [\![\neg e]\!] \cdot G && \text{(Definition 13)} \\ \Leftarrow\ &[\![e]\!] \cdot \mathsf{wlp}_\beta(P_{\text{b}}, F) \leq [\![e]\!] \cdot \mathsf{wlp}_\beta(P_{\text{b}}, G) && (F \leq G \text{ by induction}) \\ \Leftarrow\ &\mathsf{wlp}_\beta(P_{\text{b}}, F) \leq \mathsf{wlp}_\beta(P_{\text{b}}, G) && \text{(simplify)} \\ \Leftarrow\ &F \leq G && \text{(structural induction)} \\ \Leftarrow\ &f \leq g && \text{(induction)} \end{aligned}$$

Definition 14 (woip_β). *The operator woip_β is defined analogously to woip in Definition 10 (while recursing to woip_β) for all statements P other than loops. For $P = \texttt{while}\ expr\ \texttt{\{}\ P_{\text{body}}\ \texttt{\}}$ we have:*

$$\mathsf{woip}_\beta(P, f) = \gamma^{\beta(P)}([\![\neg expr]\!] \cdot f)$$
$$\textit{with} \quad \gamma(f) = [\![expr]\!] \cdot \mathsf{woip}_\beta(P_{\text{body}}, f) + [\![\neg expr]\!] \cdot f$$

Corollary 1. *The functions λ and γ are linear and monotonic. The function μ is monotonic.*

Lemma 7. *We have $\mathsf{wp}_\beta(P, f) \leq \mathsf{woip}_\beta(P, f)$.*

Proof. For all statements other than `observe`, the lemma holds trivially since the definitions match. For `observe`-statements, we have $[\![expr]\!] \cdot f \leq f \Leftarrow [\![expr]\!] \leq 1$.

Lemma 8. *When letting the loop iteration bounds tend towards infinity, the wp_β operator has the same semantics as wp. That is*

$$\lim_{\substack{\forall P \in \mathsf{Dom}(\beta): \\ \beta(P) \to \infty}} \mathsf{wp}_\beta(P, f) = \mathsf{wp}(P, f)$$

The same holds for wlp_β and wlp as well as woip_β and woip.

Proof. Note that when expanding the loop as described in Observation 2, each `if` $expr$ `{` $P_{\texttt{body}}$ `}` statement matches the execution of one loop iteration. If this `if`-statement is repeated arbitrarily often as the loop iteration bounds tend towards infinity, the corresponding semantics converges towards that of the unrestricted loop.

5.2 Finding Loop Iteration Bounds Through Sampling

Having seen how loop iteration bounds can make the weakest preexpectation operator computable, we need some heuristic that determines suitable loop iteration bounds. Assuming that we are given a program which almost surely terminates (i.e. terminates with probability one), as is typically the case for programs modelling probability distributions, a simple, yet effective, method to determine such loop iteration bounds is to use a *sampling-based execution* that counts the number of times each loop is traversed.

In this heuristic, the program is executed n times. Each of these n runs is deterministic in the sense that each variable is assigned a unique value. A probabilistic choice executes either of the two branches with the given probability. A violated `observe`-statement drops the sample. For each loop, we count how many times it is traversed in each of the deterministic runs and use the maximum iteration count as the loop iteration bound.

Observation 5. When loop iteration bounds of a program without `observe`-statements are determined using the heuristic described above, a fraction of $1 - \frac{1}{n}$ of all runs are expected to terminate within these bounds.

Thus, for any program P that terminates almost surely, even when removing `observe`-statements, the expected value of $\mathsf{woip}_\beta(P, 1)$ is $1 - \frac{1}{n}$.

5.3 Approximation Error

Loop iteration bounds allow us to compute weakest preexpectations by approximating loops. It is thus only natural to ask how large the approximation error is. To this aim, we need to establish bounds on the value of $\mathsf{wp}(P, f)$ and $\mathsf{wlp}(P, 1)$ based on the values computed with loop iteration bounds.

Lemma 9. *We have* $\mathsf{wlp}(P, f) = \mathsf{wp}(P, f) + \mathsf{wlp}(P, 0)$ *(cf. [11, Cor. 4.26]). Since we can view the loop iteration bounds as a syntactic transformation of the program by Observation 2, the lemma also holds when taking loop iteration bounds into account.*

Lemma 10. *Let* β^+ *result from the function* β *by increasing the loop iteration bound for each loop by one, that is* $\forall P \in \mathsf{Dom}(\beta)\colon \beta^+(P) = \beta(P) + 1$.

Note that if for a function φ *parameterised on* β *we have* $\varphi(\beta) \leq \varphi(\beta^+)$*, the relation also holds in the limit (analogously for "*$\geq$*"), i.e.*

$$\varphi(\beta) \leq \varphi(\beta^+) \leq \varphi(\beta^{++}) \leq \cdots \leq \lim_{\substack{\forall P \in \mathsf{Dom}(\beta)\colon \\ \beta(P) \to \infty}} \varphi(\beta)$$

Example 3. If $\varphi(\beta) = \mathsf{wp}_\beta(P, f)$ for some fixed $f \in \mathbb{E}$ and $P \in \mathbb{Q}$ and $\mathsf{wp}_\beta(P, f) \leq \mathsf{wp}_{\beta^+}(P, f)$, then

$$\mathsf{wp}_\beta(P, f) \leq \lim_{\substack{\forall P \in \mathsf{Dom}(\beta)\colon \\ \beta(P) \to \infty}} \mathsf{wp}_\beta(P, f) = \mathsf{wp}(P, f).$$

Theorem 1 (Bounds on wp and wlp). *We have*

$$\begin{aligned} \mathsf{wp}_\beta(P, f) \leq \mathsf{wp}(P, f) \ &\leq \mathsf{wp}_\beta(P, f) + \big(1 - \mathsf{woip}_\beta(P, 1)\big) \\ \mathsf{wlp}_\beta(P, f) - \mathsf{wlp}_\beta(P, 0) \leq \mathsf{wlp}(P, f) &\leq \mathsf{wlp}_\beta(P, f) \end{aligned}$$

Proof. We prove the bounds inequality by inequality:

Part 1: $\mathsf{wp}_\beta(P, f) \leq \mathsf{wp}(P, f)$

Part 2: $\mathsf{wp}(P, f) \leq \mathsf{wp}_\beta(P, f) + \big(1 - \mathsf{woip}_\beta(P, 1)\big)$

Part 3: $\mathsf{wlp}_\beta(P, f) - \mathsf{wlp}_\beta(P, 0) \leq \mathsf{wlp}(P, f)$

Part 4: $\mathsf{wlp}(P, f) \leq \mathsf{wlp}_\beta(P, f)$

Proof of Part 1 $(\mathsf{wp}_\beta(P, f) \leq \mathsf{wp}(P, f))$**.** We have

$$\begin{aligned} &\mathsf{wp}_\beta(P, f) \leq \mathsf{wp}(P, f) \\ \Leftarrow\ &\mathsf{wp}_\beta(P, f) \leq \mathsf{wp}_{\beta^+}(P, f) && \text{(Lemma 10) } (\star) \end{aligned}$$

Now show $(\star)$. If P contains no loop, the statement is trivially true since the definitions of wp_β and wp_{β^+} match. For a loop $P = \texttt{while}\ e\ \texttt{\{}\ P_\mathrm{b}\ \texttt{\}}$ we have:

Base case ($\beta(P) = 0$).

$(\star)$

$\Leftrightarrow \lambda^0([\![\neg e]\!] \cdot f) \leq \lambda^1([\![\neg e]\!] \cdot f)$ (Definition 12)

$\Leftrightarrow [\![\neg e]\!] \cdot f \leq [\![e]\!] \cdot \mathsf{wp}_\beta(P_\mathrm{b}, [\![\neg e]\!] \cdot f) + [\![\neg e]\!] \cdot f$ (Definition 12)

$\Leftrightarrow 0 \leq [\![e]\!] \cdot \mathsf{wp}_\beta(P_\mathrm{b}, [\![\neg e]\!] \cdot f)$ (simplify) ✓

Induction case ($b = \beta(P) > 0$).

$(\star)$

$\Leftrightarrow \lambda^b([\![\neg e]\!] \cdot f) \leq \lambda^{b+1}([\![\neg e]\!] \cdot f)$ (Definition 12)

$\Leftarrow \lambda^{b-1}([\![\neg e]\!] \cdot f) \leq \lambda^b([\![\neg e]\!] \cdot f)$ (Corollary 1)

(holds by induction) ✓

Proof of Part 2 ($\mathsf{wp}(P, f) \leq \mathsf{wp}_\beta(P, f) + (1 - \mathsf{woip}_\beta(P, 1))$)**.** We have

$\mathsf{wp}(P, f) \leq \mathsf{wp}_\beta(P, f) + \big(1 - \mathsf{woip}_\beta(P, 1)\big)$

$\Leftarrow \mathsf{wp}(P, f) + \big(1 - \mathsf{woip}(P, 1)\big) \leq \mathsf{wp}_\beta(P, f) + \big(1 - \mathsf{woip}_\beta(P, 1)\big)$ ($\mathsf{woip}(P, 1) \leq 1$)

$\Leftarrow \mathsf{wp}_{\beta^+}(P, f) + \big(1 - \mathsf{woip}_{\beta^+}(P, 1)\big) \leq \mathsf{wp}_\beta(P, f) + \big(1 - \mathsf{woip}_\beta(P, 1)\big)$

(Lemma 10) $(\star)$

Now show $(\star)$. If P contains no loop, the statement is trivially true. For loops we have:

Base case ($\beta(P) = 0$).

$(\star)$

$\Leftrightarrow \lambda^1([\![\neg e]\!] \cdot f) + \big(1 - \gamma^1([\![\neg e]\!])\big) \leq \lambda^0([\![\neg e]\!] \cdot f) + \big(1 - \gamma^0([\![\neg e]\!])\big)$ (Def. 12, 14)

$\Leftrightarrow [\![e]\!] \cdot \mathsf{wp}_\beta(P_\mathrm{b}, [\![\neg e]\!] \cdot f) + [\![\neg e]\!] \cdot f + \big(1 - [\![e]\!] \cdot \mathsf{woip}_\beta(P_\mathrm{b}, [\![\neg e]\!]) - [\![\neg e]\!]\big)$
$\leq [\![\neg e]\!] \cdot f + (1 - [\![\neg e]\!])$ (Definition 12, 14)

$\Leftrightarrow [\![e]\!] \cdot \mathsf{wp}_\beta(P_\mathrm{b}, [\![\neg e]\!] \cdot f) - [\![e]\!] \cdot \mathsf{woip}_\beta(P_\mathrm{b}, [\![\neg e]\!]) \leq 0$ (simplify)

$\Leftarrow \mathsf{wp}_\beta(P_\mathrm{b}, [\![\neg e]\!] \cdot f) \leq \mathsf{woip}_\beta(P_\mathrm{b}, [\![\neg e]\!])$ (rearrange)

(continues on next page)

$\Leftarrow \mathsf{wp}_\beta(P_\mathrm{b}, [\![\neg e]\!] \cdot f) \leq \mathsf{wp}_\beta(P_\mathrm{b}, [\![\neg e]\!])$ (Lemma 7)

$\Leftarrow [\![\neg e]\!] \cdot f \leq [\![\neg e]\!]$ (Lemma 5)

$\Leftarrow f \leq 1$ (simplify) ✓

Induction case ($b = \beta(P) > 0$). First reformulate the statement to show

$(\star)$

$\Leftrightarrow \lambda^{b+1}([\![\neg e]\!] \cdot f) + \big(1 - \gamma^{b+1}([\![\neg e]\!])\big) \leq \lambda^b([\![\neg e]\!] \cdot f) + \big(1 - \gamma^b([\![\neg e]\!])\big)$ (Def. 12)

$\Leftrightarrow \lambda^{b+1}([\![\neg e]\!] \cdot f) - \lambda^b([\![\neg e]\!] \cdot f) \leq \gamma^{b+1}([\![\neg e]\!]) - \gamma^b([\![\neg e]\!])$ (rearrange) $(\star\star)$

Now show $(\star\star)$ by induction.

$(\star\star)$

$\Leftrightarrow \lambda\big(\lambda^b([\![\neg e]\!]\cdot f) - \lambda^{b-1}([\![\neg e]\!]\cdot f)\big) \le \gamma\big(\gamma^b([\![\neg e]\!]) - \gamma^{b-1}([\![\neg e]\!])\big)$ (Corollary 1)

$\Leftrightarrow \lambda(\Lambda) \le \gamma(\Gamma)$ (abbreviate terms inside λ, γ)

$\Leftrightarrow [\![e]\!]\cdot \mathsf{wp}_\beta(P_\mathrm{b}, \Lambda) + [\![\neg e]\!]\cdot\Lambda \le [\![e]\!]\cdot \mathsf{woip}_\beta(P_\mathrm{b}, \Gamma) + [\![\neg e]\!]\cdot\Gamma$ (Def. 12, 14)

$\Leftarrow [\![e]\!]\cdot \mathsf{wp}_\beta(P_\mathrm{b}, \Lambda) \le [\![e]\!]\cdot \mathsf{woip}_\beta(P_\mathrm{b}, \Gamma)$ ($\Lambda \le \Gamma$ by induction)

$\Leftarrow \mathsf{wp}_\beta(P_\mathrm{b}, \Lambda) \le \mathsf{woip}_\beta(P_\mathrm{b}, \Gamma)$ (simplify)

$\Leftarrow \mathsf{wp}_\beta(P_\mathrm{b}, \Lambda) \le \mathsf{wp}_\beta(P_\mathrm{b}, \Gamma)$ (Lemma 7)

$\Leftarrow \Lambda \le \Gamma$ (Lemma 5)

(holds by induction) ✓

Proof of Part 3 $(\mathsf{wlp}_\beta(P,f) - \mathsf{wlp}_\beta(P,0) \le \mathsf{wlp}(P,f))$**.**

$\mathsf{wlp}_\beta(P,f) - \mathsf{wlp}_\beta(P,0) \le \mathsf{wlp}(P,f)$

$\Leftrightarrow \mathsf{wp}_\beta(P,f) \le \mathsf{wlp}(P,f)$ (Lemma 9)

$\Leftarrow \mathsf{wp}(P,f) \le \mathsf{wlp}(P,f)$ (Part 1) ✓

Proof of Part 4 $(\mathsf{wlp}(P,f) \le \mathsf{wlp}_\beta(P,f))$**.** We have

$\mathsf{wlp}(P,f) \le \mathsf{wlp}_\beta(P,f)$

$\Leftarrow \mathsf{wlp}_{\beta^+}(P,f) \le \mathsf{wlp}_\beta(P,f)$ (Lemma 10) $(\star)$

Now show $(\star)$. If P contains no loop, the statement is trivially true. For loops we have:

Base case $(\beta(P) = 0)$.

$(\star)$

$\Leftrightarrow \mu^1([\![e]\!] + [\![\neg e]\!]\cdot f) \le \mu^0([\![e]\!] + [\![\neg e]\!]\cdot f)$ (Definition 13)

$\Leftrightarrow [\![e]\!]\cdot \mathsf{wlp}_\beta(P_\mathrm{b}, [\![e]\!] + [\![\neg e]\!]\cdot f) + [\![\neg e]\!]\cdot f \le [\![e]\!] + [\![\neg e]\!]\cdot f$ (Definition 13)

$\Leftrightarrow [\![e]\!]\cdot \mathsf{wlp}_\beta(P_\mathrm{b}, [\![e]\!] + [\![\neg e]\!]\cdot f) \le [\![e]\!]$ (simplify)

$\Leftarrow \mathsf{wlp}_\beta(P_\mathrm{b}, [\![e]\!] + [\![\neg e]\!]\cdot f) \le 1$ (simplify) ✓

Induction case $(b = \beta(P) > 0)$.

$(\star)$

$\Leftrightarrow \mu^{b+1}([\![e]\!] + [\![\neg e]\!]\cdot f) \le \mu^b([\![e]\!] + [\![\neg e]\!]\cdot f)$ (Def. 13)

$\Leftrightarrow \mu^b([\![e]\!] + [\![\neg e]\!]\cdot f) \le \mu^{b-1}([\![e]\!] + [\![\neg e]\!]\cdot f)$ (Cor. 1)

(holds by induction) ✓

Corollary 2. *Combining the bounds on* $\mathsf{wp}(P,f)$ *and* $\mathsf{wlp}(P,1)$ *from Theorem 1, we obtain bounds on the conditional weakest preexpectation* $\frac{\mathsf{wp}(P,f)}{\mathsf{wlp}(P,1)}$*:*

$$\frac{\mathsf{wp}_\beta(P,f)}{\mathsf{wlp}_\beta(P,1)} \le \frac{\mathsf{wp}(P,f)}{\mathsf{wlp}(P,1)} \le \frac{\mathsf{wp}_\beta(P,f) + \big(1 - \mathsf{woip}_\beta(P,1)\big)}{\mathsf{wlp}_\beta(P,1) - \mathsf{wlp}_\beta(P,0)}$$

Theorem 2. *If a program P is almost surely terminating even when removing all* `observe`*-statements and the loop iteration bounds are being determined using the sampling technique described in Section 5.2 on $\overline{P}$, then the approximation error (that is, the difference between the upper and lower bound on the conditional weakest preexpectation) is expected to be less than $\frac{2}{n \cdot \mathsf{wlp}(P,1)}$. It is thus in $\mathcal{O}(\frac{1}{n})$.*

Proof.

$$
\begin{aligned}
&\frac{\mathsf{wp}_\beta(P,f) + \big(1 - \mathsf{woip}_\beta(P,1)\big)}{\mathsf{wlp}_\beta(P,1) - \mathsf{wlp}_\beta(P,0)} - \frac{\mathsf{wp}_\beta(P,f)}{\mathsf{wlp}_\beta(P,1)} \\
&\leq \frac{\mathsf{wp}_\beta(P,f) + \big(1 - \mathsf{woip}_\beta(P,1)\big)}{\mathsf{wlp}_\beta(P,1) - \big(1 - \mathsf{woip}_\beta(P,1)\big)} - \frac{\mathsf{wp}_\beta(P,f)}{\mathsf{wlp}_\beta(P,1)} && (\dagger) \\
&\leq \frac{\mathsf{wp}_\beta(P,f) + 2 \cdot \big(1 - \mathsf{woip}_\beta(P,1)\big)}{\mathsf{wlp}_\beta(P,1)} - \frac{\mathsf{wp}_\beta(P,f)}{\mathsf{wlp}_\beta(P,1)} && (\tfrac{a+c}{b+c} \geq \tfrac{a}{b} \text{ if } \tfrac{a}{b} \leq 1, c \geq 0) \\
&= 2 \cdot \frac{\big(1 - \mathsf{woip}_\beta(P,1)\big)}{\mathsf{wlp}_\beta(P,1)} && \text{(simplify)} \\
&\leq 2 \cdot \frac{\big(1 - \mathsf{woip}_\beta(P,1)\big)}{\mathsf{wlp}(P,1)} && \text{(Theorem 1)} \\
&\sim \frac{2}{n \cdot \mathsf{wlp}(P,1)} && \text{(Observation 5)}
\end{aligned}
$$

$\dagger\colon 1 - \mathsf{woip}_\beta(P,1) = 1 - \mathsf{wp}_\beta(\overline{P},1) = \mathsf{wlp}_\beta(\overline{P},0) \geq \mathsf{wlp}_\beta(P,0)$

6 Weakest Preexpectations at Intermediate Execution States

With the definition of computable weakest preexpectation operators at hand, let us turn towards the development of a debugger for probabilistic programs. The first thing we need to consider, is how the interaction features of a debugger for deterministic programs can be mapped to those for probabilistic programs. Common debuggers for deterministic languages essentially offer the following four interaction buttons:

- **Step Over:** Continue execution to the next statement in the current function.
- **Step Into:** If the current statement is a function call, continue execution inside the callee's body.
- **Step Out:** Continue execution until the end of the current function and step out to the caller.
- **Continue:** Continue execution until the end of the program.

Step Into and *Step Out* both deal with function calls which are not defined in our language and can thus be ignored in this paper. *Continue* can be thought of as executing the *Step Over* command infinitely often. Hence, these debuggers

essentially have a *single* interaction feature for programs without function calls, namely the *Step Over* command that continues execution to the next statement. This is possible since in deterministic programs each execution state has a *unique* successor state. Even for the branching statements `if` and `while`, the condition is either satisfied or not. Thus, execution either jumps into the `true`- or the `false`-branch.

In probabilistic programs, this is no longer the case. While non-branching statements like assignments still have a unique successor, the branching statements `if`, `prob` and `while` do not. For `if` *expr* `{` P_{if} `}` `else {` P_{else} `}` where *expr* is `true` with 60% and `false` with 40%, both branches are viable – it is up to the user to decide which branch he or she wants to jump into.[4] The possible options are:

- Only focus on those runs where *expr* is `true` and jump to the first statement in P_{if}.
- Only focus on those runs where *expr* is `false` and jump to the first statement in P_{else}.
- Do not focus on a particular value of *expr* but evaluate both branches simultaneously and jump to the statement after the `if`-statement.

This gives us the following three debugger commands for probabilistic programs:

- **Step Over:** Execute the current statement and jump to the next statement.[5]
- **Step Into True:** If execution is currently at a branching statement[6], only focus on runs that satisfy the condition and jump to the first statement of the `true`-branch. For `if`- and `prob`-statements, this means jumping into the `if`-branch. For `while`-statements, this means jumping into the loop's body. If the current statement is not branching, the semantics is equivalent to *Step Over*.
- **Step Into False:** Analogous to *Step Into True*. For `if`- and `prob`-statements jump into the `else`-branch, for `while`-statements terminate the loop, for non-branching statements equivalent to *Step Over*.

So far we have talked vaguely about focussing on certain runs. Fortunately, there already exists a construct that performs exactly the operation we require, namely `observe`-statements. We can thus view a *Step Into True* command as executing a virtual `observe (`*expr* `== true)` statement right before the branching statement (which afterwards has a unique successor). The *Step Into False* command analogously executes a virtual `observe (`*expr* `== false)` statement.

[4] The same logic also applies to `prob`- and `while`-statements. For loops, the user can jump into the loop's body or exit the loop.

[5] Here, `if`-, `prob` and `while`-statements are viewed as single statements that include their bodies.

[6] Branching statements are `if`-, `prob`- and `while`-statements. All other statements are non-branching.

6.1 Execution History of a Program

While the interactive execution of a program is very well suited for usability, we need a way to grasp the entire interaction procedure in a single object for formal analysis. For this, we describe an intermediate execution state by its execution history, which consists of all debugger commands that have been executed since the start of the program.

Definition 15 (Execution history). *An execution history $h = (h_0, \ldots, h_n)$ with $h_i \in \{\,\mathfrak{so}, \mathfrak{sit}, \mathfrak{sif}\,\}$ is the list of debugger commands that have been executed since the start of the program, h_0 being the first debugger command.* $\mathfrak{so}$ *stands for* Step Over*,* $\mathfrak{sit}$ *for* Step Into True *and* $\mathfrak{sif}$ *for* Step Into False*.*

Definition 16 (Augmented execution history). *For an execution history $h = (h_0, \ldots, h_n)$ on a program $P \in \mathbb{Q}$, the augmented execution history is a list $h^P = \big((h_0, P_0), \ldots, (h_n, P_n)\big)$ with $P_i \in \mathbb{Q}$ where each debugger command is augmented with the statement on which it was executed.*

Example 4. The execution history $(\mathfrak{so}, \mathfrak{so}, \mathfrak{sit}, \mathfrak{so}, \mathfrak{so})$ executes exactly one loop iteration of the program in Figure 1.

The corresponding augmented execution history has the following entries. Notice that executing the *Step Into True* command on the `while`-statement only deals with the condition. The body is executed by the two following *Step Over* commands.

- $(\mathfrak{so}, \texttt{bool aliceInfectious = true})$
- $(\mathfrak{so}, \texttt{bool bobInfected = false})$
- $(\mathfrak{sit}, \texttt{while aliceInfectious \{ ... \}})$
- $(\mathfrak{so}, \texttt{prob 0.1 \{ bobInfected = true \}})$
- $(\mathfrak{so}, \texttt{prob 0.6 \{ aliceInfectious = false \}})$

6.2 WP-Inference of Execution Histories

Now that we have defined execution histories and intuitively covered how the *Step Into True* and *Step Into False* commands should be handled, we can also define WP-inference on them.

Definition 17 (wph). *The* wph *operator performs WP-inference of an augmented execution history h^P and a postexpectation $f \in \mathbb{E}$.*

$$\mathsf{wph}(h^P, f) = \begin{cases} f & \text{if } h^P = () \\ \textit{see table below} & \text{if } h^P = \big((h_0, P_0)\big) \\ \mathsf{wph}\big((h_0^P, \ldots h_{n-1}^P), \mathsf{wph}(h_n^P, f)\big) & \\ & \text{if } h^P = (h_0^P, \ldots, h_n^P), n \geq 1 \end{cases}$$

P_0	h_0	$\mathsf{wph}\big((h_0, P_0), f\big)$
(`int` | `float` | `bool` | ε) *var* `=` *expr*	$\mathfrak{so}$, $\mathfrak{sit}$, $\mathfrak{sif}$	$\mathsf{wp}(P_0, f)$
`observe` *expr*	$\mathfrak{so}$, $\mathfrak{sit}$, $\mathfrak{sif}$	$\mathsf{wp}(P_0, f)$
`if` *expr* { P_{if} } `else` { P_{else} }	$\mathfrak{so}$	$\mathsf{wp}(P_0, f)$
	$\mathfrak{sit}$	$[\![expr]\!] \cdot f$
	$\mathfrak{sif}$	$[\![\neg expr]\!] \cdot f$
`prob` *expr* { P_{if} } `else` { P_{else} }	$\mathfrak{so}$	$\mathsf{wp}(P_0, f)$
	$\mathfrak{sit}$	$expr \cdot f$
	$\mathfrak{sif}$	$(1 - expr) \cdot f$
`while` *expr* { P_{body} }	$\mathfrak{so}$	$\mathsf{wp}(P_0, f)$
	$\mathfrak{sit}$	$[\![expr]\!] \cdot f$
	$\mathfrak{sif}$	$[\![\neg expr]\!] \cdot f$

Definition 18 (wlph, woiph). *The* wlph *operator for execution histories is analogous to* wph *with the only difference that it delegates to* wlp *where* wph *delegates to* wp*. Similarly the* woiph *operator delegates to* woip *instead of* wp*.*

Definition 19 ($\mathsf{wph}_\beta, \mathsf{wlph}_\beta, \mathsf{woiph}_\beta$). *The* wph_β*,* wlph_β *and* woiph_β *operators compute the weakest preexpectation of an execution history while taking loop iteration bounds* β *into account. They are defined analogously to* wph*,* wlph *and* woiph *but delegate to* wp_β*,* wlp_β *and* woip_β *instead of* wp*,* wlp *and* woip*.*

Given the definitions of the weakest preexpectation operators for execution histories, we can also place bounds on the unbounded conditional weakest preexpectation $\frac{\mathsf{wph}(h^P, f)}{\mathsf{wlph}(h^P, 1)}$ based on the loop-bounded values.

Theorem 3 (Approximation error of weakest preexpectations for execution histories). *We have*

$$\frac{\mathsf{wph}_\beta(h^P, f)}{\mathsf{wlph}_\beta(h^P, 1)} \leq \frac{\mathsf{wph}(h^P, f)}{\mathsf{wlph}(h^P, 1)} \leq \frac{\mathsf{wph}_\beta(h^P, f) + \big(1 - \mathsf{woiph}_\beta(h^P, 1)\big)}{\mathsf{wlph}_\beta(h^P, 1) - \mathsf{wlph}_\beta(h^P, 0)}$$

Proof. The proof is analogous to that of Theorem 1.

Example 5. For the program in Figure 1, the probability that Bob is infected after two encounters with an infectious Alice is given by the probability of `bobInfected` being `true` after the execution of exactly two loop iterations. This is modelled by the execution history $h = (\mathfrak{so}, \mathfrak{so}, \mathfrak{sit}, \mathfrak{so}, \mathfrak{so}, \mathfrak{sit}, \mathfrak{so}, \mathfrak{so})$. Figure 3 shows the computation of $\mathsf{wph}_\beta(h^P, [\![\texttt{bobInfected}]\!])$ and $\mathsf{wlph}_\beta(h^P, 1)$.

By Theorem 3 we get a lower bound of $\frac{0.076}{0.4} = 0.19$ on $\frac{\mathsf{wph}(h^P, f)}{\mathsf{wlph}(h^P, 1)}$. Since we did not employ loop iteration bounds in this example, we have $\mathsf{woiph}_\beta(h^P, 1) = 1$ and $\mathsf{wlph}_\beta(h^P, 0) = 0$. Thus the upper bound is also 0.19.

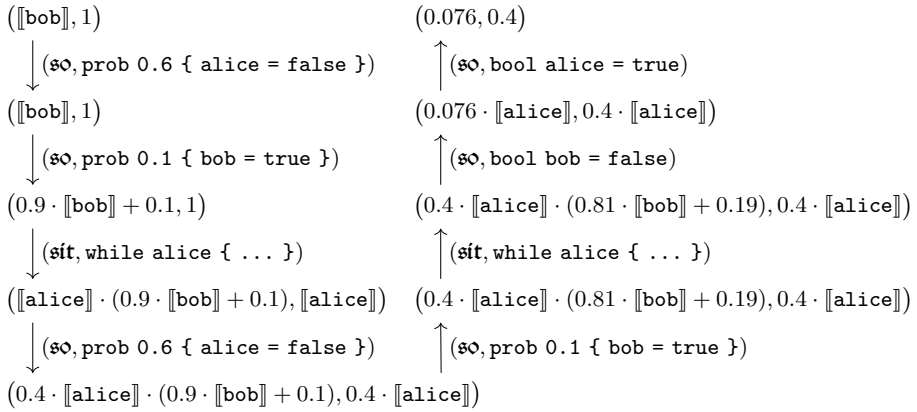

Fig. 3. Computation of $\mathsf{wph}_\beta(h^P, [\![\texttt{bobInfected}]\!])$ and $\mathsf{wlph}_\beta(h^P, 1)$ for $h = (\mathfrak{so}, \mathfrak{so}, \mathfrak{sit}, \mathfrak{so}, \mathfrak{so}, \mathfrak{sit}, \mathfrak{so}, \mathfrak{so})$ on the program in Figure 1 (where `alice` and `bob` respectively stand for `aliceInfectious` and `bobInfected`).

7 Implementation

Based on the ideas of the algorithms described in this paper, a debugger for probabilistic programs has been implemented. It operates on an *Intermediate Representation (IR)* in *Static Single Assignment (SSA)* form instead of the syntactic representation discussed in this paper.

The debugger is available both as a graphical user interface, which runs on macOS, as well as a command line tool that runs on both macOS and Linux. The source code can be found at [10].

Figure 4 shows a screenshot of the debugger's graphical user interface. Its left-hand side displays a structured outline of the program's execution. This execution outline is being generated during a sample-based execution of the program as described in Section 5.2. For each executed statement, a corresponding entry is being added to the execution outline. Clicking on one of the entries in the execution outline jumps to the corresponding execution state. When doing so, the statement that will be executed next is highlighted in the source code on the top right and the current variable values, together with their approximation error if necessary, are displayed on the bottom right. Should the user want to step through the program manually, he or she can use the three debugger buttons *Step Over*, *Step Into True* and *Step Into False* on top of the variables view.

By the ability to step to arbitrary execution states from the execution outline on the left-hand side, our software implements a recording-based debugger.

8 Conclusion

In this paper, we have provided the semantic foundations for debugging probabilistic programs. After some initial considerations on how a debugger for prob-

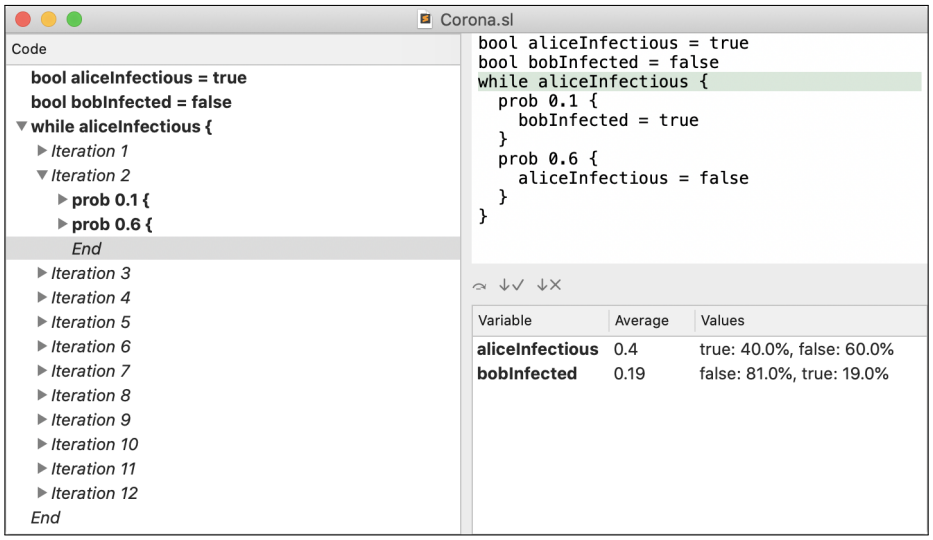

Fig. 4. Screenshot of the debugger's graphical user interface.

abilistic programs should operate, including the introduction of the new *Step Into True* and *Step Into False* debugger commands, we defined a weakest pre-expectation operator that is not only able to compute variable values after the execution of the entire program but can also compute these values at intermediate execution states by using execution histories.

To mitigate the problem that this operator is not computable due to its use of fixed point operators, we introduced loop iteration bounds that can be determined using sampling-based program execution. This yields a computable weakest preexpectation operator whose approximation error could be quantified by means of a careful analysis. A prototypical implementation of the approach shows that it offers a nice and intuitive way of exploring a program's semantics.

Regarding future work, there are several possible ways to improve both the expressivity of the programming language and the usability of the debugging framework. The first concerns the type of distributions: Our framework currently only supports discrete distributions generated through `prob`-statements. An extension to continuous distributions, which are also supported in the DePP approach [16], should be straightforward. Another possible enhancement are (possibly recursive) user-defined functions, which would require a limitation of the recursion depth similarly to loop iteration bounds as discussed in Section 5.

Moreover, a very common debugger feature is being able to place breakpoints that halt program execution at a certain statement [19]. In our setting, implementing breakpoints should be possible by generating all execution histories that hit the breakpoint and performing WP-inference (as described in Section 6.2) for all of them.

References

1. Aggarwal, S.K., Kumar, M.S.: Debuggers for programming languages. In: The Compiler Design Handbook, pp. 297–329. CRC Press (2002)
2. Caballero, R., Hermanns, C., Kuchen, H.: Algorithmic debugging of Java programs. Electronic Notes in Theoretical Computer Science **177**, 75–89 (2007). https://doi.org/10.1016/j.entcs.2007.01.005
3. Chesley, O.C., Ren, X., Ryder, B.G.: Crisp: a debugging tool for Java programs. In: 21st IEEE International Conference on Software Maintenance (ICSM'05). pp. 401–410. IEEE (2005). https://doi.org/10.1109/ICSM.2005.37
4. Corporation, M.: Mozilla rr debugger. https://rr-project.org (2020)
5. Dijkstra, E.W.: A discipline of programming. Prentice-Hall (1976)
6. Dutta, S., Zhang, W., Huang, Z., Misailovic, S.: Storm: program reduction for testing and debugging probabilistic programming systems. In: ESEC/FSE 2019. pp. 729–739. ACM (2019). https://doi.org/10.1145/3338906.3338972
7. Gordon, A.D., Henzinger, T.A., Nori, A.V., Rajamani, S.K.: Probabilistic programming. In: Future of Software Engineering (FOSE 2014). pp. 167–181. ACM (2014). https://doi.org/10.1145/2593882.2593900
8. Gretz, F., Katoen, J.P., McIver, A.: Operational versus weakest pre-expectation semantics for the probabilistic guarded command language. Performance Evaluation **73**, 110–132 (2014). https://doi.org/https://doi.org/10.1016/j.peva.2013.11.004
9. Grötker, T., Holtmann, U., Keding, H., Wloka, M.: The Developer's Guide to Debugging. Springer (2008)
10. Hoppen, A., Noll, T.: ppdb – a debugger for probabilistic programs (November 2020), https://github.com/ahoppen/probabilistic-debugger
11. Kaminski, B.L.: Advanced weakest precondition calculi for probabilistic programs. Ph.D. thesis, RWTH Aachen University (2019). https://doi.org/http://doi.org/10.18154/RWTH-2019-01829
12. Kaminski, B.L., Katoen, J.P., Matheja, C., Olmedo, F.: Weakest precondition reasoning for expected runtimes of randomized algorithms. J. ACM **65**(5) (2018). https://doi.org/10.1145/3208102
13. Law, R.: An overview of debugging tools. SIGSOFT Softw. Eng. Notes **22**(2), 43–47 (1997). https://doi.org/10.1145/251880.251926
14. Lopez, C.T., Singh, R.G., Marr, S., Boix, E.G., Scholliers, C.: Multiverse debugging: Non-deterministic debugging for non-deterministic programs. In: 33rd European Conference on Object-Oriented Programming (ECOOP 2019). LIPIcs, vol. 134, pp. 27:1–27:30. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik (2019). https://doi.org/10.4230/LIPIcs.ECOOP.2019.27
15. McIver, A., Morgan, C.: Abstraction, refinement and proof for probabilistic systems. Springer Science & Business Media (2005)
16. Nandi, C., Grossman, D., Sampson, A., Mytkowicz, T., McKinley, K.S.: Debugging probabilistic programs. In: MAPL 2017. pp. 18–26. ACM (2017). https://doi.org/10.1145/3088525.3088564
17. Nath, A., Domingos, P.M.: Learning tractable probabilistic models for fault localization. CoRR **abs/1507.01698** (2015), http://arxiv.org/abs/1507.01698
18. Ngo, V.C., Carbonneaux, Q., Hoffmann, J.: Bounded expectations: Resource analysis for probabilistic programs. SIGPLAN Not. **53**(4), 496–512 (2018). https://doi.org/10.1145/3296979.3192394
19. Park, C.S., Sen, K.: Concurrent breakpoints. SIGPLAN Not. **47**(8), 331–332 (2012). https://doi.org/10.1145/2370036.2145880

20. Santelices, R., Harrold, M.J.: Probabilistic slicing for predictive impact analysis. Tech. Rep. GIT-CERCS-10-10, Georgia Institute of Technology (2010), http://hdl.handle.net/1853/36917
21. Telles, M., Hsieh, Y.: The Science of Debugging. Coriolis (2001)
22. Teorey, T.J., Ford, A.R.: Practical Debugging in C++. Prentice Hall (2002)
23. Xu, Z., Ma, S., Zhang, X., Zhu, S., Xu, B.: Debugging with intelligence via probabilistic inference. In: ICSE 2018. pp. 1171–1181. ACM (2018). https://doi.org/10.1145/3180155.3180237
24. Zeller, A.: Why Programs Fail: A Guide to Systematic Debugging. Elsevier Science (2009)
25. Zhang, Y., Santelices, R.: Prioritized static slicing and its application to fault localization. Journal of Systems and Software **114**, 38–53 (2016). https://doi.org/10.1016/j.jss.2015.10.052