Automated Expected Value Analysis of Recursive Programs

Martin Avanzini, Georg Moser, Michael Schaper

Automated Expected Value Analysis of Recursive Programs

MARTIN AVANZINI, INRIA Sophia Antipolis Méditerranée, France
GEORG MOSER, Universität Innsbruck, Austria
MICHAEL SCHAPER, Build Informed, Austria

In this work, we study the fully automated inference of expected result values of probabilistic programs in the presence of natural programming constructs such as procedures, local variables and recursion. While crucial, capturing these constructs becomes highly non-trivial. The key contribution is the definition of a term representation, denoted as $\mathsf{infer}[\cdot]$, translating a pre-expectation semantics into first-order constraints, susceptible to automation via standard methods. A crucial step is the use of logical variables, inspired by previous work on Hoare logics for recursive programs. Noteworthy, our methodology is not restricted to tail-recursion, which could unarguably be replaced by iteration and wouldn't need additional insights. We have implemented this analysis in our prototype `ev-imp`. We provide ample experimental evidence of the prototype's algorithmic expressibility.

CCS Concepts: • **Theory of computation** → **Program analysis**; *Automated reasoning*.

Additional Key Words and Phrases: probabilistic programming, expected value analysis, weakest pre-expectation semantics, automation

ACM Reference Format:
Martin Avanzini, Georg Moser, and Michael Schaper. 2023. Automated Expected Value Analysis of Recursive Programs. *Proc. ACM Program. Lang.* 7, PLDI, Article 149 (June 2023), 27 pages. https://doi.org/10.1145/3591263

1 INTRODUCTION

The *verification* of the *quantitative* behaviour of probabilistic programs is a highly active field of research, motivated partly by the recent success of machine learning methodologies (see eg. [Batz et al. 2019; Eberl et al. 2020; Vasilenko et al. 2022]). Without verification, bugs may hide in correctness arguments and subsequent implementations, in particular, as reasoning about probabilistic programs (or data structures for that matter) is highly non-trivial and error prone.

Instead of verifying quantitative program behaviour semi-automatically, it would be desirable to fully automatically *infer* such estimates. For example, the goal of inference could be an approximate but still precise computation of *expected (amortised) costs* (eg. [Avanzini et al. 2020; Leutgeb et al. 2022; Wang et al. 2020]) or *quantitative invariants* (eg. [Bao et al. 2022; Wang et al. 2018]).[1] The goal of inference is to compute approximations that are as precise as possible, which requires the use of optimisation techniques. As there are now powerful optimising constraint solvers available, such as Z3 [de Moura and Bjørner 2008] or OptiMathSAT [Sebastiani and Trentin 2020], we may

[1]The need for automated techniques that analyse eg. the computational cost of code has also been recognised in large software companies. For example at Facebook, one routinely runs a cost analysis on the start-up routines in order to ensure a fast loading of the Facebook web page [Distefano et al. 2019].

Authors' addresses: Martin Avanzini, INRIA Sophia Antipolis Méditerranée, Route des Lucioles - BP 93, France, martin.avanzini@inria.fr; Georg Moser, Universität Innsbruck, Innsbruck, Austria, georg.moser@uibk.ac.at; Michael Schaper, Build Informed, Innsbruck, Austria, mschaper@posteo.net.

2475-1421/2023/6-ART149
https://doi.org/10.1145/3591263

even hope to automatically conduct optimisations that earlier required intricate analysis with pen and paper, cf. [Leutgeb et al. 2021, 2022].

The central contribution of our work is the (fully) automated inference of expected result values of probabilistic programs in the presence of natural programming constructs such as procedures, local variables and recursion. While crucial, capturing these constructs becomes highly non-trivial. To analyse recursive procedures successfully, it is required to properly model the call-stack. In particular, for each (recursive) procedure call the program context may change, that is, the analysis of (general) recursion requires some form of parametricity.[2] To the best of our knowledge, prior work fails in general to deal with these natural constructs.

Upper invariants (on the expected value) constitute a liberalisation of exact quantitative invariants. This approximated rendering of invariants allows for easier (and thus more powerful) automation. Such an analysis may also act as a stepping stone towards the incorporation of support for automation into ITPs. To wit, a recent and partly motivating work is Vasilenko et al. [2022]. We take up one of their motivating examples—see Listing 1(a) below—and show how a variant of this example becomes susceptible to full automation.

Our analysis is based on a weakest pre-expectation semantics [McIver and Morgan 2005] (see also [Gretz et al. 2014; Kaminski et al. 2018])—an axiomatic semantic in Dijkstra's spirit—for a simple imperative language PWhile endowed with sampling instructions, non-deterministic choice, lexically scoped local variables, and, crucially, recursive procedures. On a conceptual level, this semantics—denoted as $\mathsf{et}[\![\mathsf{P}]\!]$ for a program P—capture the expected behaviour of probabilistic programs.

Automation of a weakest pre-expectation semantics in the context of recursive procedures is highly non-trivial.[3] Technically, this is due to the fact that well-definedness of the semantics requires the existence of *higher-order* fixed-points. Full automation, however, requires the inference of (upper) bounds on closed-forms of such fixed-points. We overcome this challenge through the definition of a suitable *term representation*—denoted as $\mathsf{infer}[\mathsf{P}]$—of the aforementioned pre-expectation semantics $\mathsf{et}[\![\mathsf{P}]\!]$. This syntactic representation translates the pre-expectation semantics into first-order constraints, susceptible to automation via standard methods. To provide for the aforementioned parametricity in this analysis, the use of logical variables—inspired by the work on Hoare logics for recursive programs [Kleymann 1999]—is essential. We make use of a template approach and employ Z3 as suitable optimising SMT solver. Perhaps the closest comparison is to work on amortised cost analysis of functional languages (eg. [Leutgeb et al. 2022; Wang et al. 2020]).

Contributions. In sum, we present a novel methodology for the *automated expected value* analysis of non-deterministic, probabilistic and recursive programs. Our starting point is a natural *weakest pre-expectation semantics* for PWhile in the form of an *expectation transformer* $\mathsf{et}[\![\mathsf{P}]\!]$, capturing also natural program constructs such as lexically scoped local variables, procedure parameters, unrestricted return statements, etc. Our main contribution lies in a novel *term representation* of the aforementioned expectation transformer, denoted as $\mathsf{infer}[\mathsf{P}]$. Its definition follows the pattern of $\mathsf{et}[\![\mathsf{P}]\!]$, but notably differs in the definition of procedure calls and loops, where we replace the underlying fixed-point constructions via suitably constrained first-order templates. Through this step, we manage to tightly over-approximate the precise, but higher-order semantics via first-order constraint generation susceptible to automation. Second, we establish a new and original *automation* of this quantitative analysis in our prototype implementation `ev-imp`.

[2] In the context of automated cost analysis this problem has been partly addressed through the notion of *resource parametricity* [Hoffmann 2011].

[3] Automation is here understood here as "push-button" automation; no user-interaction is required.

Fig. 1. Textbook Examples on Expected Value Analysis

```
def balls(n):
  var b := 0
  if (n > 0) {
    b := balls(n-1);
    if (Bernoulli(1/5)) {b := b + 1}
  };
  return b
```

(a) Balls in a single bin.

```
def throws():
  if (Bernoulli(1/5)) {
    return 1
  } else {
    return (1 + throws())
  }
```

(b) Throws for a hit.

```
def every(i):
  if (0 < i <= 5) {
    if (Bernoulli(1/5)) {i := i - 1};
    return (1 + every(i))
  } else {
    return 0
  }
```

(c) Every bin contains at least one ball.

Outline. In Section 2 we provide a bird's eye view on the contributions of this work. In Section 3 we detail the syntactic structure of our language PWhile and provide the definition of the aforementioned weakest pre-expectation semantics. Section 4 constitutes the main technical part of the work, detailing the conception and definition of a term representation of the weakest pre-expectation semantics, susceptible to automation. Conclusively in Section 5 we discuss implementation choices for our prototype implementation, the chosen benchmark suites and the experimental evaluation. Finally, we conclude with related works and future work in Sections 6 and 7, respectively. Omitted proofs can be found in the extended version [Avanzini et al. 2023].

2 EXPECTED VALUE ANALYSIS, AUTOMATED

We motivate the central contribution of our work: an *automated* expected result value analysis of non-deterministic, probabilistic programs, featuring *recursion*. First, we present three textbook examples (and their encoding in PWhile) motivating the interest and importance of an *expected value* analysis, in contrast to eg. an expected cost analysis (see eg. [Avanzini et al. 2020; Kaminski et al. 2018; Leutgeb et al. 2022; Ngo et al. 2018; Wang et al. 2020, 2019]) or the inference of quantitative invariants (see eg. [Bao et al. 2022; Katoen et al. 2010; Wang et al. 2018]). Second, we emphasise the need for formal verification techniques going beyond pen-and-paper proofs and provide the intuition behind the thus chosen methodology of so-called *expectation transformers*. Third, we detail challenges posed by *recursive procedures* for the formal definition of our expectation transformers. Finally, we highlight the challenges of automated verification techniques in this context.

Textbook examples. To motivate expected value analysis in general, we study corresponding textbook examples—taken from Cormen et al. [Cormen et al. 2009]—and suitable code representations thereof, depicted in Figure 1. Consider an unspecified number of bins and suppose we throw balls towards the bins. Let us attempt to answer the following three questions: (i) *How many balls will fall in a given bin?* (ii) *How many balls must we toss, on average, until a given bin contains a ball?* and finally (iii) *How many balls must we toss until every bin contains at least one ball?*

The first two questions can be easily answered given some mild background in probability theory. If we throw say n balls, the number of balls per bin follows a binomial distribution. Assuming we will always hit at least one bin, then the success probability is $1/bins$, where $bins$ denotes the number of bins. Thus, the expected value of the number of balls in a single bin is given by $1/bins \cdot n$. Success in the second problem follows a geometric distribution. Thus the answer is $\frac{1}{1/bins} = bins$. But the last one is more tricky and involves a more intricate argumentation. Note that this problem and thus the procedure `every` is equivalent to the *Coupon Collector* problem, cf. [Avanzini et al. 2020; Cormen et al. 2009; Mitzenmacher and Upfal 2005]. Following the argument in [Cormen et al. 2009, Chapter 5.4], we say that we have a "hit", if we have successfully hit a specific bin. Using the notion of hits, we can split the task into stages, each corresponding to a completed hit. If all stages are complete, we are done. To complete stage k, we need to hit the k^{th} bin. Thus, the success probability

to complete this stage is given as ${}^{bins-k+1}/_{bins}$. Now let n_k denote the number of throws in the k^{th} stage. Thus, the total number required to fill the bins is $n = \sum_{k=1}^{bins} n_k$. As $\mathbb{E}[n_k] = {}^{bins}/_{bins-k+1}$, we obtain from linearity of expectations

$$\mathbb{E}[n] = \mathbb{E}[\textstyle\sum_{k=1}^{bins} n_k] = \sum_{k=1}^{bins} \mathbb{E}[n_k] = \sum_{k=1}^{bins} \frac{bins}{bins-k+1} = bins \cdot \sum_{k=1}^{bins} \frac{1}{k} \,,$$

Further, we obtain $bins \cdot \sum_{k=1}^{bins} {}^1/_k \in \Theta(bins \cdot \log(bins))$, utilising that the *harmonic number* $H_{bins} = \sum_{k=1}^{bins} {}^1/_k$ is asymptotically bounded by $\log(bins)$, cf. [Graham et al. 1994].

Probabilistic, recursive procedures. As depicted in Listings 1(a–c), it is easy to encode the above questions as (probabilistic, recursive) procedures. The encoding fixes the number of bins to five. The command `Bernoulli(`p`)` draws a value from a Bernoulli distribution, returning 1 with probability p and 0 with probability $(1-p)$.[4] Our simple imperative language PWhile follows the spirit of Dijkstra's *Guarded Command Language*, see Section 3 for the formal details. Apart from randomisation and recursion, the encoding makes—we believe natural—use of local variables, formal parameters and return statements.

The textbook questions above now become (possibly intricate) questions on program behaviours, that is, on the *expected value* of program variables wrt. the memory distributions in the final program state. Expected value analysis encompasses expected cost analysis, as we can often, that is, for almost surely terminating programs, represent program costs through a dedicated counter. Similarly, upper bounds to expected values as considered here, form approximations of quantitative invariants. In the experimental validation of our prototype implementation, we will see that wrt. the latter often our bounds are in fact exact, that is, constitute *invariants* (see Section 5).

It is well-known that in general pen-and-paper analyses of the expected value of programs are not an easy matter—as we have for example seen in the answer to the last question above—and more principled methodologies are essential, in particular if our focus is on (full) automation. To this avail, we build upon earlier work on *weakest pre-expectation semantics* of (probabilistic) programs and develop an *expectation transformer* for PWhile.

Expectation Transformers. Predicate transformer semantics—introduced in the seminal works of Dijkstra [1975]—map each program statement to a function between two predicates on the state-space of the program. Their semantics can be viewed as a reformulation of Floyd–Hoare logic [Hoare 1969]. Subsequently, this methodology has been extended to randomised programs by replacing predicates with so-called expectations—real valued functions on the program's state-space $\mathsf{Mem}\,V$—leading to the notion of *expectation transformers* (see [Kaminski et al. 2018; McIver and Morgan 2005]) and the development of *weakest pre-expectation semantics* [Gretz et al. 2014].

In the following, we impose a pre-expectation semantics on programs in PWhile, thereby providing a formal definition of the expected value of program values. Adequacy of such semantics is well understood in the literature, for example wrt. operational semantics based on Markov Decision Processes [Kaminski et al. 2018] or probabilistic abstract rewrite systems [Avanzini et al. 2020]. The difference to our work seems incremental to us. Thus, our starting point are pre-expectation transformer semantics, permitting us to focus on inference.

For a given command `C`, its *expectation transformer* $\mathsf{et}[\![\mathtt{C}]\!]$ (detailed in Figure 2 in Section 3) maps a post-expectation f to a pre-expectation, measuring the expected value that f takes on the distribution of states resulting from running `C`, as a function in the initial state of `C`. Slightly simplifying, the expectation transformer assigns semantics to *commands*

$$\mathsf{et}[\![\mathtt{C}]\!] : (\mathsf{Mem}\,V \to \mathbb{R}^{+\infty}) \to (\mathsf{Mem}\,V \to \mathbb{R}^{+\infty}) \,.$$

[4] For ease of encoding, Listing 1(c) employs the sampling from `Bernoulli(`${}^i/_5$`)`. Setting $i = bins - k + 1$ and $bins = 5$, we recover the textbook argumentation.

In the special case where f is a predicate, ie. a $\{0,1\}$ valued function on memories, $\mathsf{et}[\![\mathtt{C}]\!]\, f\, \sigma$ yields the probability that f holds after completion of C for any initial state σ; thereby generalising Djikstra's and Nielson's weakest-precondition transformer, cf. [Kaminski et al. 2018; Nielson 1987]. This definitions is probably most easily understood as a denotational semantics in continuation passing style, with post-expectation f interpreted as *continuation*, cf. [Avanzini et al. 2021; Friedman and Wand 2008].

In this reading, $f\colon \mathsf{Mem}\, V \to \mathbb{R}^{+\infty}$ measures a quantity on the continuations outcome given its inputs—in expectation— while $\mathsf{et}[\![\mathtt{C}]\!]\, f$ performs a backward reasoning lifting f to a function on initial states of C. (In the following, we use "continuation" and "expectation" interchangingly.)

Let us consider the procedure `balls` depicted in Figure 1(a). As mentioned, the final value of b, the number of balls hitting a dedicated bin, follows a binomial distribution. In each recursive call the success probability is $1/5$. Thus, the expected value of the number of balls in a single bin is $1/5 \cdot n$. To provide a bird's view on the working of expectation transformers in this context, we verify this simple equality in the sequel. Calculating the pre-expectation from the continuation f is straightforward, as long as the program under consideration contains neither recursive calls nor loops. To wit, let $\langle b \rangle$ be the real-valued function that measures (the positive value of) variable `b` in a given memory. Then

$$\mathsf{et}[\![\texttt{if (Bernoulli(1/5)) \{b=b+1\}}]\!]\, \langle b \rangle = 1/5 \cdot \mathsf{et}[\![\texttt{b=b+1}]\!]\, \langle b \rangle + 4/5 \cdot \langle b \rangle = 1/5 \cdot \langle b+1 \rangle + 4/5 \cdot \langle b \rangle\ .$$

Here, addition and scaling by a constant on expectations should should be understood point-wise. Notice how this expression captures the fact `b` is incremented only with probability $1/5$. *Composability* of the transformer extends this kind of reasoning to sequences of command, in the sense that command composition is interpreted as the composition of the corresponding expectation transformers. For straight-line programs C, $\mathsf{et}[\![\mathtt{C}]\!]\, f$ can be computed in a bottom up fashion.

Loops and Recursive Procedures. When dealing with loops or recursive procedures though, the definition of $\mathsf{et}[\![\mathtt{C}]\!]$ becomes more involved. As usual in giving denotational program semantics, the definition of the expectation transformer of these self-referential program constructs is based on a fixed-point construction, cf. [Avanzini et al. 2020; Bao et al. 2022; Kaminski et al. 2018; Katoen et al. 2010; McIver and Morgan 2005; Wang et al. 2018]. This permits attributing precise semantics to programs. In our setting, *procedures* p are interpreted via an expectation transformer

$$\mathsf{et}[\![\mathtt{p}]\!] : (\mathbb{Z} \times \mathsf{GMem} \to \mathbb{R}^{+\infty}) \to (\mathbb{Z}^{\mathsf{ar}(\mathtt{p})} \times \mathsf{GMem} \to \mathbb{R}^{+\infty})\ ,$$

where GMem the set of global memories. Following the definition of p, $\mathsf{et}[\![\mathtt{p}]\!]$ turns a continuation, parameterised in the return value and global memory, into a pre-expectation in the formal parameters of p and the global memory before execution of p. The definition (formalised in Section 3) is slightly technical to permit recursive definitions and to ensure proper passing of arguments and lexical scoping. For a procedure declared as $\texttt{def p}(\vec{x})\ \{\mathtt{C}\}$ and a post-expectation f, $\mathsf{et}[\![\mathtt{p}]\!]\, f$ is given by the transformer $\mathsf{et}[\![\mathtt{C}]\!]$ associated to its body C, initialising the parameters $\vec{x}$ accordingly and with the post-expectation f applied whenever C leaves its scope through a return-statement.

Let us re-consider the procedure `balls` from Listing 1(a). We use *Iverson brackets* $[\cdot]$ to lift predicates on memories to expectations, that is, $[\mathtt{B}](\sigma) := 1$, if B holds in memory σ, and $[\mathtt{B}](\sigma) := 0$, otherwise. Since the program does not make use of global variables, we elide representation of the empty global memory for brievity. Thus, for an arbitrary continuation $f : \mathbb{Z} \to \mathbb{R}^{+\infty}$, the expectation transformer of `balls` is given by the least functional satisfying:[5]

$$\mathsf{et}[\![\texttt{balls}]\!]\, f = \lambda n.\, [n > 0] \cdot \mathsf{et}[\![\texttt{balls}]\!]\, \big(\lambda b.\, 1/5 \cdot f(b+1) + 4/5 \cdot f(b)\big)\, (n-1) + [n \leq 0] \cdot f(0)\ , \quad (\dagger)$$

[5]Note that the functional is also ordered point-wise. Well-definedness of the employed fixed-point construction rests on the observations that expectation transformers form ω-*CPOs* [Winskel 1993].

where the continuation f passed to the recursive call of $\mathsf{et}[\![\texttt{balls}]\!]$ is computed as above. Thus, the continuation to the call to `balls` probabilistically updates the returned value. Arguing inductively, we see that $\mathsf{et}[\![\texttt{balls}]\!]\ f_r = \lambda n.\, {}^1\!/_5 \cdot n + r$ for any continuation of the form $f_r := \lambda b.\, b + r$, where r denotes a non-negative real. Since f_0 measures the return value, we conclude that on argument n, $\mathsf{et}[\![\texttt{balls}]\!]$ returns a value of ${}^1\!/_5 \cdot n$ in expectation. Conclusively, we re-obtain that a fifth of the thrown balls will fall in the considered bin.

In the same vein, the analysis of the return value of `throws` and `every`—in expectation—is performed for Listings 1(b) and 1(c), respectively. In this fashion, the translation of the above pen-and-paper analysis for the textbook examples into the formalism of an expectation transformer results into a rigorous and principled methodology. This builds a sound (and suitable) foundation for subsequent automation.

Automation. The above calculation of the expected return value of procedure `balls` gives evidence on the advantages of a formal methodology over an ad-hoc pen-and-paper analysis. The crux of turning such a calculus into a fully automated analysis lies in automatically deriving closed forms for expected values of loops and recursive procedures. Related problems have been extensively studied in the literature, eg. [Contejean et al. 2005; Fuhs et al. 2007; Podelski and Rybalchenko 2004; Sinn et al. 2014]. One prominent approach lies in assigning *templates* to expected value functions, by which the definition of the expectation transformer can be reduced to a set of constraints treatable with off-the-shelf SMT solvers, like Z3 [de Moura and Bjørner 2008].

Following this approach, we express values as linear combination $\sum_i c_i \cdot b_i$ of *base functions* (or *norms*) b_i with variable coefficients c_i, mapping program valuations to (non-negative) real numbers. Norms serve as a numerical abstraction of memories and encompass a variety of common abstractions, for example the absolute value of a variable, the difference between two variables and more generally arbitrary polynomial combinations thereof. Often, expected values can be computed symbolically on such *value expressions*. Concerning recursive procedures or loops, the definition of the expectation transform is in essence recursive itself. Rather than computing this fixed-point directly, we make use of Park's theorem [Kaminski et al. 2018; Wechler 1992] and seek an upper bound in closed-form. In the context of recursive procedure, we also have to model the call-stack appropriately.

Concerning recursive procedures, our solution is to represent every higher-order functional $\mathsf{et}[\![\mathsf{p}]\!]$ through a pair of first-order templates $\langle \mathsf{H}_\mathsf{p}, \mathsf{K}_\mathsf{p} \rangle$, representing pre- and post-expectations respectively. Concretely, such a template signifies $\mathsf{et}[\![\mathsf{p}]\!]\ \mathsf{K}_\mathsf{p} \le \mathsf{H}_\mathsf{p}$. Inspired by Hoare-calculi for recursive programs (cf. [Kleymann 1999]), we index templates by *logical variables* $\mathbf{x}$, kept implicit in the sequel, but emphasised here for clarity of exposition. Thus indeed, a template represents a *families of* pre-/post-expectations. To wit, let us revisit procedure `balls` in Listing 1(a) once more. We take the templates:

$$\mathsf{H}^{\mathbf{x}}_{\texttt{balls}} := \lambda n.\, c_0 + c_1 \cdot \langle n \rangle + c_2 \cdot \mathbf{x} \qquad\qquad \mathsf{K}^{\mathbf{x}}_{\texttt{balls}} := \lambda b.\, \langle b \rangle + \mathbf{x}\,,$$

Notice that the logical variable $\mathbf{x}$ should only be instantiated non-negatively, so as to ensure that these templates indeed capture (non-negative) expectations. Thus, the indended meaning of this template for `balls` is captured by the constraint

$$\forall \mathbf{x} \ge 0.\, \forall n.\, \mathsf{et}[\![\texttt{balls}]\!]\ \mathsf{K}^{\mathbf{x}}_{\texttt{balls}}\ n \le \mathsf{H}^{\mathbf{x}}_{\texttt{balls}}\ n\,, \tag{‡}$$

Symbolic evaluation of the left-hand side—making use of the template for calls to `balls`—can now be used to sufficiently constrain the undetermined coefficients. Concretely (‡) is representable via the following three constraints, where the variables b, n and $\mathbf{x}$ are universally quantified, while the

unknowns c_0, c_1, d_0 and d_1 are existentially quantified.

$$0 \leq \mathbf{x} \vDash {}^1\!/_5 \cdot \langle b+1 \rangle + {}^4\!/_5 \cdot \langle b \rangle + \mathbf{x} \leq \langle b \rangle + (d_0 + d_1 \cdot \mathbf{x}) \tag{c1}$$

$$0 \leq \mathbf{x} \vDash 0 \leq d_0 + d_1 \cdot \mathbf{x} \tag{c2}$$

$$0 \leq \mathbf{x} \vDash [n > 0] \cdot (c_0 + c_1 \cdot \langle n-1 \rangle + c_2 \cdot (d_0 + d_1 \cdot \mathbf{x})) + [n \leq 0] \cdot \mathbf{x} \leq c_0 + c_1 \langle n \rangle + c_2 \cdot \mathbf{x}\,. \tag{c3}$$

To see how these constraints have been formed, recall the fixed-point equation for $\mathsf{et}[\![\texttt{balls}]\!]$ in (†), instantiating $f = \mathsf{K}^{\mathbf{x}}_{\texttt{balls}}$. Constraint (c1) expresses that the post-expectation

$$\lambda b.\, {}^1\!/_5 \cdot \mathsf{K}^{\mathbf{x}}_{\texttt{balls}}(b+1) + {}^4\!/_5 \cdot \mathsf{K}^{\mathbf{x}}_{\texttt{balls}}(b) = \lambda b.\, {}^1\!/_5 \cdot \langle b+1 \rangle + {}^4\!/_5 \cdot \langle b \rangle + \mathbf{x}\,,$$

passed to the call of $\mathsf{et}[\![\texttt{balls}]\!]$ in (†) is over-approximated by instance $\mathsf{K}^{d_1+d_1\cdot\mathbf{x}}_{\texttt{balls}}$, taking a (to be further determined) instantiation $\mathbf{x} \mapsto d_1 + d_1 \cdot \mathbf{x}$ of the logical variable $\mathbf{x}$. This substitution is guaranteed non-negative (and thus well-defined) by constraint (c2). Thus taking (c1) into account, y (‡) e have

$$\mathsf{et}[\![\texttt{balls}]\!]\ (\lambda b.\, {}^1\!/_5 \cdot \mathsf{K}^{\mathbf{x}}_{\texttt{balls}}(b+1) + {}^4\!/_5 \cdot \mathsf{K}^{\mathbf{x}}_{\texttt{balls}}(b))\ (n-1) \leq \mathsf{H}^{d_0+d_1\cdot\mathbf{x}}_{\texttt{balls}}\ (n-1)\,.$$

Making use of this over-approximation in (†), it is now evident that the final constraint (c3) witnesses satisfiability of the main constraint (‡). These three constraints can be met by taking $c_1, d_0 = {}^1\!/_5$, $c_2, d_1 = 1$, and $c_0 = 0$.

We re-obtain (unsurprisingly) that the expected return value of procedure `balls` is given as ${}^1\!/_5 \cdot n$. We emphasise that parameterising templates through logical variables is crucial, even for relatively simple examples such as `balls`. Since the program is not tail-recursive, the continuation changes after each recursive call. Parametricity allows us to vary templates across recursive calls, as we have done above through instantiating $\mathbf{x}$ by $d_0 + d_1 \cdot \mathbf{x}$.

We have successfully automated this approach in our prototype implementation `ev-imp`, see Section 5. Automation is based on our first contribution, the development of a *term representation* of the expectation transformer inducing an inference method that describes the generation of constraints. In the context of procedure `balls`, this representation reduces the task of finding a functional satisfying (†) to the definition of a set of constraints like (‡), whose solution yields over-approximations of the function graph of $\mathsf{et}[\![\texttt{balls}]\!]$. (See Section 4 for the details.) Based on this formal development, our tool `ev-imp` proceeds with an analysis as outlined, and integrates a dedicated constraint solver for solving constraints of the form (c1)–(c3). Apart from handling recursive procedures, we have incorporated the constraints for handling loop programs. Here, we take inspiration from [Avanzini et al. 2020] to guarantee a *modular* (and thus *scalable*) inference of upper bounding functions.

Wrt. the three motivating examples in Figure 1, our tool derives (upper-bounds to) the corresponding expected return values in milliseconds. The bounds for the procedures in Listings 1(a) and (b) are precise, respectively. For the procedure `every` we, however, only manage to derive a (sound) upper bound. The latter is not surprising. As shown in the textbook proof the expected number of throws is given as $\Theta(bins \cdot \log(bins))$ in general. Our template approach does not (yet) support logarithmic functions. Hence, we cannot hope to derive the precise bound fully automatically. To the best of our knowledge, our prototype `ev-imp` is the only existing tool able to fully automatically analyse the expected result value (or expected cost for that matter) of probabilistic, recursive programs, if formulated in an imperative language. The complete evaluation of our prototype implementation is given in Section 5.

3 AN IMPERATIVE PROBABILISTIC LANGUAGE

We consider a small *imperative language* in the spirit of Dijkstra's *Guarded Command Language*. In particular, the language features (i) dynamic sampling instructions; (ii) non-deterministic choice, formalised via a non-deterministic choice operator (`<>`); (iii) nested loops; and (iv) crucially

recursive procedure declarations. In this section, we first formalise the syntax and then endow it with axiomatic, pre-expectation semantics.

Syntax. Let $\mathsf{Var} = \{x, y, \dots\}$ be a set of (integer-valued) program *variable*. We fix three syntactic categories of *Boolean* valued *expressions* $\mathsf{BExpr}\,V$, *(integer valued) expressions* $\mathsf{Expr}\,V$, and *sampling instructions* $\mathsf{SExpr}\,V$ over (finitely many) variables $V \subseteq \mathsf{Var}$, respectively. In the following, $\mathtt{B}$ will range over Boolean, $\mathtt{E}, \mathtt{F}$ over integer valued expressions and $\mathtt{G}$ over sampling instructions. Furthermore, let $\mathsf{Proc} = \{\mathtt{p}, \mathtt{q}, \dots\}$ be a set of *procedure (symbols)*. The set of commands $\mathsf{Cmd}\,V$ over program variables $x \in V$ is given by

$$\begin{aligned}\mathtt{C}, \mathtt{D} ::= {} & \mathtt{skip} \mid x :\approx \mathtt{G} \mid x :\approx \mathtt{p}(\mathtt{E}_1, \dots, \mathtt{E}_{\mathsf{ar}(\mathtt{p})}) \mid \mathtt{return}(\mathtt{E}) \mid \mathtt{var}\ x \leftarrow \mathtt{E}\ \mathtt{in}\ \{\mathtt{C}\} \\ & \mid \mathtt{C};\mathtt{D} \mid \mathtt{if}\ (\mathtt{B})\ \{\mathtt{C}\}\ \mathtt{else}\ \{\mathtt{C}\} \mid \mathtt{while}\ (\mathtt{B})\ \{\mathtt{C}\} \mid \mathtt{C} <> \mathtt{D}\end{aligned}$$

The interpretation of these commands is fairly standard. The command $\mathtt{skip}$ acts as a no-op. In an assignment $x :\approx \mathtt{R}$, the right-hand side $\mathtt{R}$ evaluates to a distribution, from which a value is sampled and assigned to x. As right-hand side $\mathtt{R}$, we permit either built-in sampling expressions $\mathtt{G} \in \mathsf{SExpr}$ such as $\mathtt{unif}(lo, hi)$ for sampling an integer uniformly between constants lo and hi, or calls to user-defined procedures $\mathtt{p} \in \mathsf{Proc}$. A command $\mathtt{var}\ x \leftarrow \mathtt{E}\ \mathtt{in}\ \{\mathtt{C}\}$ declares a local variable x, initialised by $\mathtt{E}$, within command $\mathtt{C}$. Further, commands can be defined by *composition*, via a conditional *conditional*, via a *while-loop* construct or via the *non-deterministic* choice operator $\mathtt{C} <> \mathtt{D}$, interpreted in a demonic way. For ease of presentation, we elide a probabilistic choice command and probabilistic guards (see eg [Kaminski et al. 2016, 2018]), since they do not add to the expressiveness of the language. In examples, however, we make use of mild syntactic sugaring to simplify readability, as we did already above.

A *program* $\mathtt{P}$ is given as a finite sequence of procedure definitions. Each procedure $\mathtt{p}$ expects a number of integer-valued arguments and returns an integer upon termination. The number of arguments of a procedure $\mathtt{p}$ is called its *arity* and denoted as $\mathsf{ar}(\mathtt{p})$. The body of $\mathtt{p}$, denoted as $\mathtt{Bdy}_{\mathtt{p}}$, consists of a command $\mathtt{C} \in \mathsf{Cmd}\,V$ that may refer both to formal parameters, locally and globally declared variables. Note that since $\mathtt{p}$ may trigger a sampling instruction, its evaluation evolves probabilistically, yielding a final value and modifying the global state with a certain probability. Formally, a program is a tuple $\mathtt{P} = (\mathtt{GVar}, \mathtt{Decl})$ where $\mathtt{GVar} \subseteq \mathsf{Var}$ is a finite set of *global variables*, and where $\mathtt{Decl}$ is a finite sequence of *procedure declarations* of the form

$$\mathtt{def}\ \mathtt{p}(x_1, \dots, x_{\mathsf{ar}(\mathtt{p})})\ \{\mathtt{C}\}\ .$$

To avoid notational overhead due to variable shadowing, we assume that the *formal parameters* $\mathtt{Args}_{\mathtt{p}} := x_1, \dots, x_{\mathsf{ar}(\mathtt{p})}$ and global variable are all pairwise different, and distinct from variables locally bound within $\mathtt{C}$. Using α-renaming, this can always been guaranteed. Throughout the following, we keep the program $\mathtt{P} = (\mathtt{GVar}, \mathtt{Decl})$ fixed.

Weakest Pre-Expectations Semantics. A *memory* (or *state*) over finite variables $V \subseteq \mathsf{Var}$ is a mapping $\sigma \in \mathsf{Mem}\,V := V \to \mathbb{Z}$ from variables to integers. We write $\mathsf{GMem} := \mathsf{Mem}\,\mathtt{GVar}$ for the set of *global memories*, and $\sigma{\restriction_V}$ for its restriction to variables in V. Let $x \in V$ be a variable and let $v \in \mathbb{Z}$. Then we write $\sigma[x \mapsto v]$ for the memory that is as σ except that x is mapped to v. As short-forms, we write σ_{g} for the *global memory* $\sigma{\restriction_{\mathtt{GVar}}}$, and dual, σ_{l} for the *local memory* $\sigma{\restriction_{\mathsf{Var}\setminus\mathtt{GVar}}}$. Let $\mathbb{R}^{+\infty}$ denote the set of non-negative real numbers extended with ∞, ie. $\mathbb{R}^{+\infty} := \mathbb{R}_{\geq 0} \cup \{\infty\}$. A (discrete) *subdistribution* over A is a function $\delta \colon A \to \mathbb{R}_{\geq 0}$ so that $\sum_{a \in A}(\delta\, a) \leq 1$, and a *distribution*, if $\sum_{a \in A}(\delta\, a) = 1$. We may write (sub)distributions δ as $\{\!\{\delta\, a : a\}\!\}_{a \in A}$. The set of all subdistributions over A is denoted by $\mathsf{D}\,A$. We restrict to distributions over countable sets A. The *expectation of* a function $f \colon A \to \mathbb{R}^{+\infty}$ wrt. a distribution δ is given by $\mathbb{E}_\delta\ f := \sum_{a \in A}(\delta\, a) \cdot (f\, a)$. We suppose that expressions $\mathtt{E} \in \mathsf{Expr}\,V$, Boolean expressions $\mathtt{B} \in \mathsf{BExpr}\,V$ and sampling expressions $\mathtt{G} \in \mathsf{SExpr}\,V$

Fig. 2. Expectation Transformer Semantics of PWhile.

$$
\begin{aligned}
&\mathsf{et}[\![\mathsf{p}]\!]^{\eta} : (\mathbb{Z} \to \mathsf{GMem} \to \mathbb{R}^{+\infty}) \to (\mathbb{Z}^{\mathsf{ar}(\mathsf{p})} \to \mathsf{GMem} \to \mathbb{R}^{+\infty}) \\
&\mathsf{et}[\![\mathsf{p}]\!]^{\eta}\, f && := \lambda \vec{v}\, \sigma.\, \mathsf{et}[\![\mathsf{Bdy}_{\mathsf{p}}]\!]^{\eta}_{f}\, (\lambda \tau.\, f\, 0\, \tau_{\mathrm{g}})\, (\sigma \uplus \{\mathtt{Args}_{\mathsf{p}} \mapsto \vec{v}\}) \\
&\mathsf{et}[\![\mathsf{C}]\!]^{\eta}_{f} \colon (\mathsf{Mem}\, V \to \mathbb{R}^{+\infty}) \to (\mathsf{Mem}\, V \to \mathbb{R}^{+\infty}) \\
&\mathsf{et}[\![\mathtt{skip}]\!]^{\eta}_{f}\, g && := g \\
&\mathsf{et}[\![x :\approx \mathsf{G}]\!]^{\eta}_{f}\, g && := \lambda \sigma.\, \mathbb{E}_{[\![\mathsf{G}]\!]\,\sigma}\, (\lambda v.\, f[x \mapsto v]) \\
&\mathsf{et}[\![x :\approx \mathsf{p}(\vec{\mathsf{E}})]\!]^{\eta}_{f}\, g && := \lambda \sigma.\, \eta\, \mathsf{p}\, (\lambda v\, \tau.\, g\, (\sigma_{\mathrm{l}} \uplus \tau)[x \mapsto v])\, ([\![\vec{\mathsf{E}}]\!]\, \sigma)\, \sigma_{\mathrm{g}} \\
&\mathsf{et}[\![\mathtt{return}(\mathsf{E})]\!]^{\eta}_{f}\, g && := \lambda \sigma.\, f\, ([\![\mathsf{E}]\!]\, \sigma)\, \sigma_{\mathrm{g}} \\
&\mathsf{et}[\![\mathtt{var}\ x \leftarrow \mathsf{E}\ \mathtt{in}\ \{\mathsf{C}\}]\!]^{\eta}_{f}\, g && := \lambda \sigma.\, \mathsf{et}[\![\mathsf{C}]\!]^{\eta}_{f}\, g\, \sigma[x \mapsto [\![\mathsf{E}]\!]\, \sigma] \\
&\mathsf{et}[\![\mathsf{C};\mathsf{D}]\!]^{\eta}_{f}\, g && := \lambda \sigma.\, \mathsf{et}[\![\mathsf{C}]\!]^{\eta}_{f}\, (\mathsf{et}[\![\mathsf{D}]\!]^{\eta}_{f}\, g)\, \sigma \\
&\mathsf{et}[\![\mathtt{if}\ (\mathsf{B})\ \{\mathsf{C}\}\ \mathtt{else}\ \{\mathsf{D}\}]\!]^{\eta}_{f}\, g && := \lambda \sigma.\, [\![\mathsf{B}]\!]\, \sigma] \cdot \mathsf{et}[\![\mathsf{C}]\!]^{\eta}_{f}\, g\, \sigma + [\![\neg \mathsf{B}]\!]\, \sigma] \cdot \mathsf{et}[\![\mathsf{D}]\!]^{\eta}_{f}\, g\, \sigma \\
&\mathsf{et}[\![\mathtt{while}\ (\mathsf{B})\ \{\mathsf{C}\}]\!]^{\eta}_{f}\, g && := \lambda \sigma.\, \mathsf{lfp}\left(\lambda G.\, \lambda \tau.\, [\![\mathsf{B}]\!]\, \tau] \cdot \mathsf{et}[\![\mathsf{C}]\!]^{\eta}_{f}\, G\, \tau + [\![\neg \mathsf{B}]\!]\, \tau] \cdot g\, \tau\right) \sigma \\
&\mathsf{et}[\![\mathsf{C} <> \mathsf{D}]\!]^{\eta}_{f}\, g && := \lambda \sigma.\, \max(\mathsf{et}[\![\mathsf{C}]\!]^{\eta}_{f}\, g, \mathsf{et}[\![\mathsf{D}]\!]^{\eta}_{f}\, g)\, \sigma
\end{aligned}
$$

are equipped with interpretations $[\![\mathsf{E}]\!] : \mathsf{Mem}\, V \to \mathbb{Z}$, $[\![\mathsf{B}]\!] : \mathsf{Mem}\, V \to \mathbb{B}$ and $[\![\mathsf{G}]\!] : \mathsf{Mem}\, V \to \mathsf{D}\, \mathbb{Z}$, respectively. Functions in $A \to \mathbb{R}^{+\infty}$ are called *expectation* (over a set A) in the literature and are usually denoted by f, g, h etc. We equip expectations and transformers with the point-wise ordering, that is, $f \leq g$ if $f\, a \leq g\, a$ for all $a \in A$. We also extend functions over A point-wise to expectations and denote these extensions in `typewriter` font, eg., $f + g := \lambda a.\, f\, a + g\, a$ etc. In particular, $\mathtt{0} = \lambda a.0$ and $\infty = \lambda a.\infty$. Equipped with this ordering, expectations form an ω-*CPO* [Winskel 1993], whose least element is the constant zero function $\mathtt{0}$.

Expectation transformer have the shape $F : (A \to \mathbb{R}^{+\infty}) \to (B \to \mathbb{R}^{+\infty})$ (for some sets A and B respectively). Ordered point-wise these again form an ω-CPO and have thus enough structure to give a denotational model to programs. In particular, when $A = B$ the least fixed-point $\mathsf{lfp}\,(F)$ of F is well-defined and given by $\sup_n (F^n \perp_A)$, for F^n the n-fold composition of F [Winskel 1993]. Following Dijkstra [Dijkstra 1976], expectation transformers can be seen as giving rise to a (denotational) semantics. [6]

The transformer $\mathsf{et}[\![\mathsf{P}]\!]$ is defined in terms of an *expectation transformer* $\mathsf{et}[\![\mathsf{p}]\!]$, $\mathsf{p} \in \mathsf{P}$, for *procedures*, which in turn is mutual recursively defined via an *expectation transformer* $\mathsf{et}[\![\mathsf{C}]\!]$ on *commands*, cf. Figure 2. These transformers are parameterised in a *procedure environment* η, of type

$$\mathsf{p}{:}\mathsf{Proc} \to (\mathbb{Z} \times \mathsf{GMem} \to \mathbb{R}^{+\infty}) \to (\mathbb{Z}^{\mathsf{ar}(\mathsf{p})} \times \mathsf{GMem} \to \mathbb{R}^{+\infty})\ ,$$

associating an expectation semantic to each $\mathsf{p} \in \mathsf{Proc}$, which is used in the case of procedure calls $x :\approx \mathsf{p}(\vec{\mathsf{E}})$. As already alluded to, the definitions given in Figure 2 are best understood as a denotational semantics in continuation passing style, with post-expectations $f \colon \mathbb{Z} \times \mathsf{GMem} \to \mathbb{R}^{+\infty}$

[6] Such a transformer constitutes a standard denotational semantic, where effects are interpreted in the continuation monad $\mathsf{Cont}_{\mathbb{R}^{+\infty}}(-) = ((-) \to \mathbb{R}^{+\infty}) \to \mathbb{R}^{+\infty}$, cf. [Avanzini et al. 2021]. To observe this, flip the arguments f and σ in Figure 2.

Fig. 3. Expectation Transformer Laws.

monotonicity	$\eta \leq \chi \wedge f_1 \leq f_2 \wedge g_1 \leq g_2 \implies \mathsf{et}[\![\mathtt{C}]\!]^{\eta}_{f_1}\, g_1 \leq \mathsf{et}[\![\mathtt{C}]\!]^{\chi}_{f_2}\, g_2$
linearity	$\mathsf{et}[\![\mathtt{C}]\!]^{\eta}_{\sum_i \mathsf{R}_i \cdot f_i}\,(\sum_i \mathsf{R}_i \cdot g_i) = \sum_i \mathsf{R}_i \cdot \mathsf{et}[\![\mathtt{C}]\!]^{\eta}_{f_i}\, g_i$
loop-invariant	$[\neg \mathtt{B}] \cdot g_1 \leq g_2 \wedge [\mathtt{B}] \cdot \mathsf{et}[\![\mathtt{C}]\!]^{\eta}_{f}\, g_2 \leq g_2 \implies \mathsf{et}[\![\mathtt{while}\ (\mathtt{B})\ \{\mathtt{C}\}]\!]^{\eta}_{f}\, g_1 \leq g_2$
procedure-invariant	$\forall \mathtt{p} \in \mathsf{Proc}.\ \mathsf{et}[\![\mathtt{p}]\!]^{\chi}_{f} \leq \chi\, \mathtt{p} \implies \mathsf{et}[\![\mathtt{P}]\!] \leq \chi$

and $g : \mathsf{Mem}\, V \to \mathbb{R}^{+\infty}$ being interpreted as continuations, respectively. In short, the functional

$$\mathsf{et}[\![\mathtt{p}]\!] : (\mathbb{Z} \times \mathsf{GMem} \to \mathbb{R}^{+\infty}) \to (\mathbb{Z}^{\mathsf{ar}(\mathtt{p})} \times \mathsf{GMem} \to \mathbb{R}^{+\infty})\,,$$

turns a continuation in the return-value and (possibly modified within the body of p) global memory, and lifts it to a function in the formal parameters of p and the initial global memory. To this end, $\mathsf{et}[\![\mathtt{p}]\!]$ links formal parameters $\mathtt{Args}_{\mathtt{p}}$ to the input values $\vec{v}$, and yields to the transformer $\mathsf{et}[\![\mathtt{Bdy}_{\mathtt{p}}]\!]^{\eta}_{f}$ associated to its body. As denoted, this auxiliary transformer on commands is parameterised by a procedure environment η, and continuation f of the procedure p. The latter is necessary to model evaluation when $\mathtt{Bdy}_{\mathtt{p}}$ pre-maturely leaves scope through a return statement. To capture the sitation where evaluation completes without encountering a return, the continuation to $\mathsf{et}[\![\mathtt{Bdy}_{\mathtt{p}}]\!]^{\eta}_{f}$ supplies to f a default return value of zero.

To give some intuitions on the transformer of commands, it is again advisive to think of $\mathsf{et}[\![\mathtt{C}]\!]^{\eta}_{f}\, g\, \sigma$ as running the command C on memory σ, and then proceeding with continuation g. The only exception to this reading lies in the treatment of sampling instructions.

For a sampling instructions $x :\approx \mathtt{G}$, $\mathsf{et}[\![\mathtt{C}]\!]^{\eta}_{f}\, g\, \sigma$ computes the expected value of the continuation g on the distribution of stores obtained by updating x with elements sampled from G. (As a simplified and rough intuition, think of the assignment rule in Hoare logics.) For a procedure call $x :\approx \mathtt{p}(\vec{\mathtt{E}})$, we make use of the semantics $\eta\, \mathtt{p}$ of p, which is applied to the evaluated arguments and the current global memory σ_{g}. The continuation passed to $\eta\, \mathtt{p}$ runs the continuation g on the combination of global memory τ yielded by p and the local pre-memory σ_{l}, with x updated by the return value yielded by p. When C is a return statement, the transformer skips continuation g and jumps directly to the continuation f defined by the enscoping procedure, supplying the returned value. For a local variable declarations, $\mathsf{et}[\![\mathtt{var}\ x \leftarrow \mathtt{E}\ \mathtt{in}\ \{\mathtt{C}\}]\!]^{\eta}_{f}\, g$ implement lexical scope, updating variable x in σ by E. Due to the variable convention—as emphasised on page 8—the *local* variable x is fresh and thus cannot conflict with any variable in V.

The transformer for composed commands is given by the composition of the corresponding transformers. In the case of conditionals, we use Iverson bracket $[\cdot]$ to interpret Boolean values `false` and `true` as integers 0 and 1, respectively. Thus the transformer of conditionals effectively recurses on one of the two branches of the conditional, depending on the condition B. The expectation transformer for loops can be seen as (the least fixed-point) satisfying

$$\mathsf{et}[\![\mathtt{while}\ (\mathtt{B})\ \{\mathtt{C}\}]\!]^{\eta}_{f}\, g\, \sigma = \begin{cases} \mathsf{et}[\![\mathtt{C;while}\ (\mathtt{B})\ \{\mathtt{C}\}]\!]^{\eta}_{f}\, g\, \sigma & \text{if } [\![\mathtt{B}]\!]\, \sigma = \mathtt{true}, \\ g\, \sigma & \text{if } [\![\mathtt{B}]\!]\, \sigma = \mathtt{false}, \end{cases}$$

running C once followed by the while loop in case the guard holds; and calling the continuation g otherwise. Finally, non-determinism is modelled as the *maximum* of the pre-expectations obtained from the alternatives. This is motivated by the fact that we are interested in worst-case bounds (on

expected values and costs) and follows the treatment of non-determinism in the context of program analysis, cf. [Nielson et al. 1999]. We note that our treatment of non-determinism constitutes a conceptual difference to the *weakest pre-expectation calculus* of [McIver and Morgan 2005]. There the focus is on (quantitative) program behaviours and thus a lower-bound to the pre-expectation is sought which results in choosing the *minimum* of the pre-expectations to handle non-deterministic choice.

What remains is to set up the procedure environment η according to the declarations in P. To this end, we associate the semantics $\mathrm{et}[\![\mathsf{P}]\!]$ with the procedure environment η that makes each $\mathsf{p} \in \mathsf{Proc}$ adhere to the semantics $\mathrm{et}[\![\mathsf{p}]\!]^\eta$, that is, that satisfies the (least) fixed point $\eta\,\mathsf{p}\,f = \mathrm{et}[\![\mathsf{p}]\!]^\eta\,f$. More precisely, $\mathrm{et}[\![\mathsf{P}]\!] := \mathsf{lfp}\left(\lambda\Xi.\,\lambda\mathsf{p}.\,\mathrm{et}[\![\mathsf{p}]\!]^\Xi\right)$. Again, this least (higher-order) fixed-point exists as expectation transformers form an ω-CPO.

We emphasise that all the individual transformers are defined mutually, thus permitting mutual recursion on procedure declarations. In the following, we write $\mathrm{et}[\![\cdot]\!]$ instead of $\mathrm{et}[\![\cdot]\!]^\eta_f$ when $\eta = \mathrm{et}[\![\mathsf{P}]\!]$ and caller expectation f is clear from context. Table 3 lists laws, tacitly employed above, for the expectation transformers $\mathrm{et}[\![\mathsf{p}]\!]$ and $\mathrm{et}[\![\mathsf{C}]\!]$, respectively.

4 INFERENCE

In this section we present the key contribution of this work, the development of a *term representation* of the expectation transformer $\mathrm{et}[\![\mathsf{P}]\!]$, see Figure 4. This forms the crucial basis of the automated inference of upper bounds to $\mathrm{et}[\![\mathsf{P}]\!]$, as implemented in our prototype `ev-imp`.

To a great extent, the definition of $\mathsf{infer}[\mathsf{p}]$, p a procedure, follows the pattern of the definition of $\mathrm{et}[\![\mathsf{p}]\!]$. Notably, however, it differs in the definition of procedure calls, where we employ the templates $\langle \mathsf{H_p}, \mathsf{K_p}\rangle$ outlined in Section 2 and the definition of loops, where we replace the fixed-point construction via a suitably constrained template. Theorem 4.1 verifies that this approximation is sound; in proof we make essential use of the invariant laws depicted in Figure 3. We represent expectations syntactically as terms, denoting linear combinations of norms:

$$\mathsf{Norm}\,V\,Z \ni \mathsf{N} ::= [\mathsf{B}]\cdot\mathsf{E} \quad (\textit{norms}) \qquad \mathsf{Term}\,V\,Z \ni \mathsf{T},\mathsf{S},\mathsf{U} ::= \sum_i \mathsf{R}_i\cdot\mathsf{N}_i \quad (\textit{terms})$$

$\mathsf{E} \in \mathsf{Expr}\,(V \uplus Z)$ denotes an arbitrary expression over *program* variables V and *logical* variables $Z = \{\mathbf{x}, \mathbf{y}, \dots\}$. $\mathsf{B} \in \mathsf{BExpr}\,(V \uplus Z)$ denotes a Boolean expression over program and logical variables. Coefficients R denote terms yielding non-negative real numbers.

We require that for any norm N, E is non-negative, whenever B holds. A norm abstracts an expression over a program variable as a non-negative real number. For instance, $\max(x, 0) = [x \geq 0]\cdot x$, or $[x \geq y]\cdot(x - y)$ gives the distance from x to y. For brevity, we set $\langle x\rangle := [x \geq 0]\cdot x$. Following the semantics, we set $[\mathsf{B}]\cdot(\sum_i \mathsf{R}_i\cdot([\mathsf{B}_i]\cdot\mathsf{E}_i)) = \sum_i \mathsf{R}_i\cdot([\mathsf{B}\wedge\mathsf{B}_i]\cdot\mathsf{E}_i)$. Note that, since norms are non-negative, a term $\mathsf{T} \in \mathsf{Term}\,V\,\varnothing$ can be interpreted as an expectation $[\![\mathsf{T}]\!] : \mathsf{Mem}\,V \to \mathbb{R}^{+\infty}$ over the state-space. Let $\mathsf{G} \in \mathsf{SExpr}$ denote a sampling instruction and let $\mathsf{T} \in \mathsf{Term}\,V\,\varnothing$, then $\mathsf{E}_{x\leftarrow\mathsf{G}}\,\mathsf{T}$ denotes the term representation of the expectation $\mathbb{E}_{[\![\mathsf{G}]\!]\,\sigma}\,[\![\mathsf{T}[x \mapsto v]]\!]$ of the continuation $[\![\mathsf{T}[x \mapsto v]]\!]$ wrt. the distribution $[\![\mathsf{G}]\!]$ applied to the current memory σ.

The definition of $\mathsf{infer}[\cdot]$ (see Figure 4) is best understand as a syntactic representation of the expectation transformer $\mathrm{et}[\![\cdot]\!]$. It translates this denotational semantics into first-order constraints, susceptible to automation. Apart from returning a term, representing a pre-expectation, it generates a set of *side-conditions* of the form $\Gamma \vdash \mathsf{S} \leq \mathsf{T}$. Such a constraint is *valid*, if for all logical variables $\vec{\mathbf{x}}$ occurring in the expressions B, S and T, respectively, we have $[\![\mathsf{B}[\vec{\mathbf{x}} := \vec{v}]]\!] \vDash [\![\mathsf{S}[\vec{\mathbf{x}} := \vec{v}]]\!] \leq [\![\mathsf{T}[\vec{\mathbf{x}} := \vec{v}]]\!]$ for all $\vec{v}$. In this reformulation, the procedure environment η is kept implicit. The semantics of each $\mathsf{p} \in \mathsf{Proc}$ is thus representable as a pair of terms $\mathsf{K_p} \in \mathsf{Term}\,\mathsf{GVar} \uplus \{\mathbf{r}\}\,Z$, $\langle \mathsf{H_p}, \mathsf{K_p}\rangle$, with

Fig. 4. Term Representation of Expectation Transformer Semantics.

$$
\begin{array}{lll}
\mathsf{infer}[\mathtt{p}] : \mathsf{Term}\,\mathsf{GVar}\,\{\mathbf{r}\} \to \mathsf{Term}\,\mathsf{GVar}\,\{\vec{\mathbf{a}}_\mathtt{p}\} & & \\
\mathsf{infer}[\mathtt{p}]\;\mathsf{S} & := (\mathsf{infer}[\mathsf{Bdy}_\mathtt{p}]_\mathsf{S}\,\mathsf{S}[\mathbf{r} \mapsto 0])[\mathtt{Args}_\mathtt{p} \mapsto \vec{\mathbf{a}}_\mathtt{p}] & \\
\mathsf{infer}[\mathtt{C}]_\mathsf{S} : \mathsf{Term}\,V\,\{\vec{\mathbf{z}}_\mathtt{p}\} \to \mathsf{Term}\,V\,\{\vec{\mathbf{z}}_\mathtt{p}\} & & \\
\mathsf{infer}[\mathtt{skip}]_\mathsf{S}\,\mathsf{T} & := \mathsf{T} & \\
\mathsf{infer}[x :\approx \mathtt{G}]_\mathsf{S}\,\mathsf{T} & := \mathsf{E}_{x \leftarrow \mathtt{G}}\,\mathsf{T} & \\
\mathsf{infer}[x :\approx \mathtt{q}(\vec{\mathsf{E}})]_\mathsf{S}\,\mathsf{T} & := \mathsf{H}_\mathtt{q}[\vec{\mathbf{a}}_\mathtt{q} \mapsto \vec{\mathsf{E}}, \vec{\mathbf{z}}_\mathtt{q} \mapsto \vec{\mathsf{U}}] \;\textit{where}\; \vec{\mathsf{U}} \in \mathsf{Term}\,(V \setminus \mathsf{GVar})\,\{\vec{\mathbf{z}}_\mathtt{p}\} & \\
 & \qquad \Gamma_\mathtt{q} \vdash \Gamma_\mathtt{q}[\vec{\mathbf{z}}_\mathtt{q} \mapsto \vec{\mathsf{U}}]; \quad \Gamma_\mathtt{q} \vdash \mathsf{T}[x \mapsto \mathbf{r}] \leq \mathsf{K}_\mathtt{q}[\vec{\mathbf{z}}_\mathtt{q} \mapsto \vec{\mathsf{U}}] & \\
\mathsf{infer}[\mathtt{return}(\mathtt{E})]_\mathsf{S}\,\mathsf{T} & := \mathsf{S}[\mathbf{r} \mapsto \mathsf{E}] & \\
\mathsf{infer}[\mathtt{var}\;x \leftarrow \mathtt{E}\;\mathtt{in}\;\{\mathtt{C}\}]_\mathsf{S}\,\mathsf{T} & := (\mathsf{infer}[\mathtt{C}]_\mathsf{S}\,\mathsf{T})[x \mapsto \mathsf{E}] & \\
\mathsf{infer}[\mathtt{C};\mathtt{D}]_\mathsf{S}\,\mathsf{T} & := \mathsf{infer}[\mathtt{C}]_\mathsf{S}\,(\mathsf{infer}[\mathtt{D}]_\mathsf{S}\,\mathsf{T}) & \\
\mathsf{infer}[\mathtt{if}\;(\mathtt{B})\;\{\mathtt{C}\}\;\mathtt{else}\;\{\mathtt{D}\}]_\mathsf{S}\,\mathsf{T} & := [\mathtt{B}] \cdot \mathsf{infer}[\mathtt{C}]_\mathsf{S}\,\mathsf{T} + [\neg\mathtt{B}] \cdot \mathsf{infer}[\mathtt{D}]_\mathsf{S}\,\mathsf{T} & \\
\mathsf{infer}[\mathtt{while}\;(\mathtt{B})\;\{\mathtt{C}\}]_\mathsf{S}\,\mathsf{T} & := \mathsf{U} & \mathtt{B} \vdash \mathsf{infer}[\mathtt{C}]_\mathsf{S}\,\mathsf{U} \leq \mathsf{U}; \quad \neg\mathtt{B} \vdash \mathsf{T} \leq \mathsf{U} \\
\mathsf{infer}[\mathtt{C} <> \mathtt{D}]_\mathsf{S}\,\mathsf{T} & := \mathsf{U} & \vdash \mathsf{infer}[\mathtt{C}]_\mathsf{S}\,\mathsf{T} \leq \mathsf{U}; \quad \vdash \mathsf{infer}[\mathtt{D}]_\mathsf{S}\,\mathsf{T} \leq \mathsf{U}
\end{array}
$$

$\mathsf{H}_\mathtt{p} \in \mathsf{Term}\,\mathsf{GVar} \uplus \{\vec{\mathbf{a}}_\mathtt{p}\}\,Z$ and where $\vec{\mathbf{a}}_\mathtt{p} = \mathbf{a}_1, \ldots, \mathbf{a}_{\mathsf{ar}(\mathtt{p})}$ and $\mathbf{r}$ are dedicated variables, referring to the formal parameters and the return value of $\mathtt{p}$.

For each $\mathtt{p}$, the terms $\mathsf{H}_\mathtt{p}$ and $\mathsf{K}_\mathtt{p}$ can be understood as families of terms, parameterised in the substitution of logical variables. To wit, let θ denote an arbitrary substitution of logical variables for values $\vec{v}$, then $[\![\mathsf{H}_\mathtt{p}\theta]\!]$ and $[\![\mathsf{K}_\mathtt{p}\theta]\!]$ denote pre- and post-expectations $h_{\vec{v}} \colon \mathbb{Z}^n \times \mathsf{GMem} \to \mathbb{R}^{+\infty}$ and $k_{\vec{v}} \colon \mathbb{Z} \times \mathsf{GMem} \to \mathbb{R}^{+\infty}$, respectively. Conceptually, the logical variables model *resource parametricity* [Hoffmann 2011], which is required to model the call-stack in recursive calls suitably. To improve upon the expressiveness of templates, we require that the terms $\mathsf{H}_\mathtt{p}$ and $\mathsf{K}_\mathtt{p}$ are non-negative only under an *associated constraint* $\Gamma_\mathtt{p}$ on logical variables.

For instance, in Section 2, we implicitly used the logical context $\Gamma_{\mathtt{balls}} = (0 \leq \mathbf{x})$ to ensure that templates $\mathsf{H}_{\mathtt{balls}} = c_0 + c_1 \cdot \langle \mathbf{a} \rangle + c_2 \cdot \mathbf{x}$ and $\mathsf{K}_{\mathtt{balls}} = \mathbf{r} + \mathbf{x}$ are non-negative. To ensure that all the pairs $\langle \mathsf{H}_\mathtt{p}, \mathsf{K}_\mathtt{p} \rangle_{\mathtt{p} \in \mathsf{Prog}}$ adhere to the semantics of P, wrt. to their associated contexts $\Gamma_\mathtt{p}$, we finally require (for all $\mathtt{p} \in \mathsf{P}$)

$$\Gamma_\mathtt{p} \vdash (\mathsf{infer}[\mathtt{p}]\,\mathsf{K}_\mathtt{p})[\mathtt{Args}_\mathtt{p} \mapsto \vec{\mathbf{a}}] \leq \mathsf{H}_\mathtt{p}\,. \tag{1}$$

The left-hand side of the inequality in (1) may reference the pair $\langle \mathsf{H}_\mathtt{p}, \mathsf{K}_\mathtt{p} \rangle$, namely when $\mathtt{p}$ calls itself recursively.

As we have already seen in the informal description in Section 2 this recourse may be performed for an *instantiation* $[\vec{\mathbf{z}}_\mathtt{p} \mapsto \vec{\mathsf{U}}]$ of logical variables. In Section 5 we detail how this instance is chosen. Under the intended meaning of $\langle \mathsf{H}_\mathtt{p}, \mathsf{K}_\mathtt{p} \rangle$, any application of $\langle \mathsf{H}_\mathtt{p}, \mathsf{K}_\mathtt{p} \rangle$ to an expectation T (an alternation of $\mathsf{K}_\mathtt{p}$) can safely be replaced by $\mathsf{H}_\mathtt{p}$, provided the passed expectation T is bounded again from above by $\mathsf{K}_\mathtt{p}$, viz, the corresponding constraint $\Gamma_\mathtt{q} \vdash \mathsf{T}[x \mapsto \mathbf{r}] \leq \mathsf{K}_\mathtt{q}[\vec{\mathbf{z}}_\mathtt{q} \mapsto \vec{\mathsf{U}}]$ in Figure 4.

Theorem 4.1 (Soundness Theorem). *If for all $\mathtt{p} \in \mathsf{P}$ the constraint (1) is fulfilled and all side-conditions in Figure 4 are met, then for all $\mathtt{p} \in \mathsf{P}$, $\mathsf{et}[\![\mathtt{p}]\!] \leq [\![\mathsf{infer}[\mathtt{p}]]\!]$, that is, the inference algorithm is sound.*

Fig. 5. Studies in Expected Value Analysis

```
def biased_coin(x1,x2):
if (*) {
 if (Bernoulli(1/2)) {
   x1 := 2 · x1; x2 := 2 · x2;
   if (x2 + 1 ≤ x1) {x2 := x2 + 1}
   else {
     if (x2 + 1/2 ≤ x1) {x1 := 1; x2 := 0}
     else {x1 := 0; x2 := 0}}}};
return x1
```

(a) Generating a biased coin [Wang et al. 2018].

```
def binomial_update(N):
  var x := 0;
  var n := 0;
  while (n < N) {
    x := x + Bernoulli(1/2);
    n := n + 1
  };
  return x
```

(b) Binomial update [Katoen et al. 2010].

```
def hire(n):
   var hires := 0;
   if (n > 0) {
     hires := hire(n-1);
     hires := hires+Bernoulli(1/n)
   };
   return hires
```

(c) Hire a new assistant [Cormen et al. 2009].

Templating approach. In the inference of upper invariants, we follow the *templating approach* standard in the literature (see eg. [Avanzini et al. 2020; Leutgeb et al. 2022; Wang et al. 2020]) in which the functions to be synthesised—in our case a closed form of infer[P]—are given as linear combination $\sum_i c_i \cdot b_i$ of pre-determined *base functions* b_i, with variable coefficient c_i. We emphasise that base functions can be *linear* or *non-linear*. Concretely, for straight-line commands C this is captured by the definition of the term representation infer[C] in the context of the continuation T. In Section 2 we have seen an informal account of this recipe. As a slightly more involved example, consider procedure `biased_coin` depicted in Listing 5(a).[7] The procedure incorporates a *non-deterministic choice*, denoted by the conditional with unspecified guard (∗).

We focus on the *non-deterministic choice* and the *sampling instruction* incorporated. Wrt. C <> D the term representation eludes an explicit representation of the maximum function max. Instead non-determinism is resolved by asserting the constraints (i) $\vdash \mathsf{infer}[\mathsf{C}]_{\mathsf{S}}\,\mathsf{T} \leq \mathsf{U}$ and (ii) $\vdash \mathsf{infer}[\mathsf{D}]_{\mathsf{S}}\,\mathsf{T} \leq \mathsf{U}$, respectively. Wrt. sampling instructions, $\mathsf{infer}[x :\approx \mathsf{G}]_{\mathsf{S}}\,\mathsf{T}$ is defined as the term $\mathsf{E}_{x \leftarrow \mathsf{G}}\,\mathsf{T}$, representing the computation of the expected value of the continuation T symbolically on the distribution of the memory obtained by sampling elements according to the instruction G.

To illustrate, let C denote the body of the procedure `biased_coin`. By default, we choose the return value x_1 as continuation, which we abstracted by the norm $\langle x_1 \rangle$. Restricting to the non-trivial branch of the non-deterministic choice, we obtain by symbolic calculation that $\mathsf{infer}[\mathsf{C}]\,\langle x_1 \rangle = \langle x_1 \rangle + [x_1 \geq 1/2 + x_2]$. Solely by symbolic execution, w obtain the constraint

$$\vDash \langle x_1 \rangle + [x_1 \geq 1/2 + x_2] \leq c_0 \cdot [x_1 \geq 1/2 + x_2] \cdot \langle x_1 \rangle + c_1 \cdot [x_2 \geq x_1] \cdot \langle x_1 \rangle + 1/2 \cdot c_2 \cdot [x_1 \geq 1/2 + x_2]\,,$$

solvable with $c_0 = c_1 = c_2 = 1$, yielding $\langle x_1 \rangle + [x_1 \geq 1/2 + x_2]$ as the desired expected return value.

Loop programs. Considering loops, we employ the *loop-invariant law* to derive a closed form, cf. Figure 3. Conclusively, for a loop statement `while (B) {C}`, $\mathsf{infer}[\texttt{while (B) \{C\}}]_{\mathsf{S}}\,\mathsf{T}$ asserts the constraints (i) $\mathsf{B} \vdash \mathsf{infer}[\mathsf{C}]_{\mathsf{S}}\,\mathsf{U} \leq \mathsf{U}$ and (ii) $\neg\mathsf{B} \vdash \mathsf{T} \leq \mathsf{U}$, respectively. Ie. U represents an upper bound I_{T}, parameterised in the post-expectation T. Due to its reminiscence with a loops invariant, the function I_{T} is called an *upper invariant* in the literature [Kaminski et al. 2018].

To illustrate, consider the procedure `binomial_update` from Figure 5(b).[8] Again we approximate the return value by the norm $\langle x \rangle$. The above constraints on the term U induce the following constraints on an upper invariant $I_{\langle x \rangle}$ (i) $N \leq n \vDash \langle x \rangle \leq I_{\langle x \rangle}$ and (ii) $n < N \vDash 1/2 \cdot \mathsf{et}[\![x :\approx x + 1; n :\approx n + 1]\!]\, I_{\langle x \rangle} \leq$

[7] Listing 5(a) is taken from Wang et al. [Wang et al. 2018] and—making use of a non-deterministic abstraction—constitutes a variant of an example considered by Katoen et al. [Katoen et al. 2010]. The latter example uses a stream of fair coin flips to generate a (single) biased coin.

[8] The code sets variable x to a value between 0 and N, following a binomial distribution, cf. [Katoen et al. 2010].

$I_{\langle x\rangle}$, respectively. Making use of linear template based on base functions 1, $\langle x\rangle$ and $\langle N-n\rangle$ this can be instantiated as the following constraints.

$$N \leq n \vDash \langle x\rangle \leq c_0 + c_1 \cdot \langle N-n\rangle + c_2 \cdot \langle x\rangle$$
$$n < N \vDash c_0 + c_1 \cdot \langle N-(n+1)\rangle + 1/2 \cdot c_2 \cdot \big(\langle x\rangle + \langle x+1\rangle\big) \leq c_0 + c_1 \cdot \langle N-n\rangle + c_2 \cdot \langle x\rangle\,,$$

solvable with $c_0 = c_1 = 0$ and $c_2 = 1/2$, yielding the *quantitative invariant* $1/2 \cdot x$, that is, the derived upper invariant is optimal, cf. [Katoen et al. 2010].

Recursive procedures. Since we represent $\mathsf{et}[\![\mathsf{p}]\!]$ through the term representation $\mathsf{infer}[\mathsf{p}]$, a similar approach can be suited to recursively defined procedures $\mathtt{def}\ \mathsf{p}(\vec{x})\ \{\mathsf{Bdy}_\mathsf{p}\}$. Instead of the *loop-invariant law*, we implicitly employ the law on *procedure-invariants* to derive a closed form, though (see Figure 3). Thus, the definition of $\mathsf{infer}[x :\approx \mathsf{q}(\vec{\mathsf{E}})]_\mathsf{S}\,\mathsf{T}$ asserts the constraints (i) $\Gamma_\mathsf{q} \vdash \Gamma_\mathsf{q}[\vec{\mathsf{z}}_\mathsf{q} \mapsto \vec{\mathsf{U}}]$ and (ii) $\Gamma_\mathsf{q} \vdash \mathsf{T}[x \mapsto \mathbf{r}] \leq \mathsf{K}_\mathsf{q}[\vec{\mathsf{z}}_\mathsf{q} \mapsto \vec{\mathsf{U}}]$, respectively. Here, the latter constraint guarantees that the continuation $\mathsf{T}[x \mapsto \mathbf{r}]$ of the procedure call is bounded by an instance of the bounding function K_q. On the other hand the first constraint guarantees that the substitution $[\vec{\mathsf{z}}_\mathsf{q} \mapsto \vec{\mathsf{U}}]$ of logical variables employed is properly represented in the context information. Consider procedure `hire` depicted in Listing 5(c).[9] Following the recipe employed for the procedure `balls`, we generate the templates (i) $\mathsf{K}_\mathtt{hire} := \langle \mathbf{r}\rangle + \mathbf{x}$ and (ii) $\mathsf{H}_\mathtt{hire} := c_0 + c_1 \cdot \langle \mathbf{n}\rangle + c_2 \cdot \mathbf{x}$ where $\mathbf{n}$ refers to the parameter taken by `hire`, and $\mathbf{x}$ is a logical variable subject to the constraint $\Gamma_\mathtt{hire} := (0 \leq \mathbf{x})$. This results in the following three constraints

$$0 < \mathbf{n}, \Gamma_\mathtt{hire} \vDash {}^{\langle 1+\mathbf{r}\rangle}/_\mathbf{n} + (1 - {}^1/_\mathbf{n})\langle \mathbf{r}\rangle + \mathbf{x} \leq \langle \mathbf{r}\rangle + (d_0 + d_1 \cdot \mathbf{x})$$
$$\Gamma_\mathtt{hire} \vDash 0 \leq d_0 + d_1 \cdot \mathbf{x}$$
$$\Gamma_\mathtt{hire} \vDash [0 < \mathbf{n}] \cdot (c_0 + c_1 \cdot \langle \mathbf{n}-1\rangle + c_2 \cdot (d_0 + d_1 \cdot \mathbf{x})) + [\mathbf{n} \leq 0] \cdot \mathbf{x} \leq c_0 + c_1 \cdot \langle \mathbf{n}\rangle + c_2 \cdot \mathbf{x}\,,$$

solvable with $c_0 = 0$, $c_1 = c_2 = d_0 = d_1 = 1$. Here, the additional premise $0 < \mathbf{n}$ in the first constraint stems from a (forward) analysis of the conditional governing the call to `hire`. This yields $\langle \mathbf{n}\rangle$ as (sub-optimal) upper bound to expected number of hires in `hire`. Note that the expected value for the number of expected hires is given as harmonic number $H_k \in \Theta(\log(n))$. So linear is the best we can do with the templates provided in our prototype implementation `ev-imp`.

5 AUTOMATION

We have implemented the outlined procedures, proven sound in Theorem 4.1, in a prototype implementation, dubbed `ev-imp`.[10] Our tool `ev-imp` estimates upper-bounds on the expected (normalised) return value of procedures p, as a function of the inputs, that is, `ev-imp` computes an upper-bound to $\mathsf{et}[\![\mathsf{p}]\!]\ (\lambda v\_.\ \langle v\rangle)$. We note that the restriction to measurements of the expected return value does not constitute a real restriction but rather constitutes a slight design simplification, as long as the estimated continuation f is representable as a return expression E. The term representation from Figure 4 forms the basis of our prototype's implementation `ev-imp`. Below, we highlight the main design choices that lead us from the term representation to a concrete algorithm and provide ample experimental evidence of the algorithmic expressivity of our prototype implementation.

Assignments. Our implementation supports sampling from *finite*, *discrete* distributions G, assigning probabilities p_i to values e_i ($0 \leqslant i \leqslant k$ for constant k), where probabilities p_i are expressed as rationals and values e_i as integer expressions. Both, probabilities and expressions can possibly

[9] The procedure represents a hiring process, encoded as probabilistic program, cf. [Cormen et al. 2009].

[10] Our prototype implementation employs libraries of the open-source tool, freely available at https://gitlab.inria.fr/mavanzin/ecoimp. In particular, we rely on its auxiliary functionality, such as program parsers etc., but also on its underlying constraint solver.

depend on the current memory, thus our tool natively supports *dynamic sampling instructions* such as in `Bernoulli(1/n)`, used for example in our rendering of the textbook example Listing 5(c). Note that the probabilities depend on the value of n taken in the memory under which the distribution is evaluated. Such dynamic distributions are also required to represent our variant of the Coupon Collector's problem—procedure `every`, cf. Listing 1(c). Overall, `ev-imp` natively implements in this way a variety of standard distributions, in particular it supports sampling *uniform distributions with bounded support*, *Bernoulli*, *binomial* and *hypergeometric* distributions. Note, that through recursion more involved distributions can be build, with unbounded and dynamic support. Practical extensions, eg. further support for dynamic sampling are (in our opinion) engineering tasks that do not require novel theoretical insights.

Template selection and instantiation. The overall approach rests on templating to over-approximate the behaviour of procedures and loops. As indicated earlier, we describe these templates via linear combinations $\sum_i c_i \cdot \mathsf{N}_i$ of pre-determined *norms* N_i and undetermined coefficients c_i. To determine these templates, our implementation selects a set of candidate *base functions* from post-expectations, program invariants (determined through a simple forward-analysis) and loop-guards, loosely following the heuristics of Sinn et al. [2016] and Avanzini et al. [2020]. Specifically, an established invariant or guard $x \leq y$ gives rise to the base function $\langle x - y \rangle$, modelling the (non-negative) distance between x and y. This heuristic is extended from variables to expressions, and to arbitrary Boolean formulas, and turns out to work well in practice. These base functions are then combined to an overall *linear*, or *simple-mixed* template, cf. Contejean et al. [2005]. of base functions. For instance, $c_1\langle x \rangle + c_2\langle y \rangle + c_3 x^2 + c_4 y^2 + c_5[x \geq 0 \wedge y \geq 0](x \cdot y) + c_6$ is a simple-mixed template over base functions $\langle x \rangle$ and $\langle y \rangle$. In a similar spirit, we make use of templates for the instantiations $\vec{\mathsf{U}} \in \mathsf{Term}\,(V \setminus \mathtt{GVar})\,\{\vec{\mathsf{z}}_\mathsf{p}\}$, as linear functions in the local and logical variables, in the same vain as we have already done when presenting examples. Our prototype implements caching and backtracking to test different templates when operating in a modular setting (see the paragraph on modularity below). In particular, non-linear base functions are employed only if the linear ones fail. Here, we follow similar approaches in the literature, cf. [Avanzini et al. 2020; Leutgeb et al. 2021, 2022; Wang et al. 2018, 2020].

Constraint solving. Evaluation of $\mathsf{infer}[\mathsf{p}]$ for a procedure p results in a set of constraints, whose solution is then used to assess bounding functions. In effect, these constraints are inequalities over real-valued polynomial expressions over unknown coefficients, enriched with conditionals (through Iverson's bracket). We emphasise that via our definition of $\mathsf{infer}[\cdot]$, the computational very complicated problem of computing the pre-expectation $\mathsf{et}[\![\mathsf{p}]\!]\, f$ for a given continuation f (finding solutions to second-order fixed point equations to deal with general recursion) has been transformed into a problem suitable to be expressed in constraints, susceptible to automation. In particular this permit us to use the constraint solver implemented within Avanzini et al. [2020] tool `eco-imp` to reason about such constraints. In brief, the solver resolves conditionals through case analysis, resulting in a set of equivalent constraints over unconditional polynomials, and then makes essential use of Handelman's theorem [Handelman 1988] to turn these into constraints over undetermined coefficients. This in turn enables the use of off-the-shelf SMT solver supporting `QF_NRA`, in our case Z3.

Improving upon modularity of the analysis. As in many denotational semantics, the expectation transformer $\mathsf{et}[\![\cdot]\!]$ is compositional, which facilitates reasoning. Unfortunately, this compositionality does not directly give rise to modularity of inference. Indeed, in the context of our expected result value analysis, such a modular inference is unsound in general. Recursive procedures and loops cause cyclic dependencies that hinder modularity. In our setting, these cyclic dependencies are

reflected within the constraints generated by $\mathsf{infer}[\cdot]$. Templates assigned to nested loops, or potentially mutually recursive procedures, are thus defined through cyclic constraints. As a result, constraints cannot be solved in isolation, effectively rendering the approach described so far a whole program analysis.

In some cases, however, stratification present in the program can be exploited to run our machinery in an iterative way, thereby greatly improving upon modularity, and in consequence improving upon the performance of the overall implementation. To illustrate, eg. one case where our implementation departs from the definition of $\mathsf{infer}[\cdot]$ lies within the analysis of nested loops. Here our implementation follows ideas originally proposed by Avanzini et al. [2020]. In particular, pre-expectations of nested loops can be determined in isolation. In essence, this modular treatment of loops depends on the linear shape of templates, and the linearity of expectations law (see Figure 3). While the approach can in general not be lifted to our setting with procedure calls, for a sizeable class of loops—for example those that do not contain recursive calls—the approach extends to our setting. In this specialised case the constraints imposed by the treatment of the loop are free of unknowns expect those mentioned in the template U over-approximating the expectation of the loop. Thus, the added two constraints can be solved independently, and consequently, a concrete upper-bound rather than a template term U can be substituted for $\mathsf{infer}[\texttt{while (B) \{C\}}]_{\mathsf{S}}\,\mathsf{T}$. Soundness of this specialisation follows by a straightforward adaption of [Avanzini et al. 2020, Theorem 7.5]. These efforts yield that our prototype implementation can analyse the entirety of our 53 benchmarks examples in around 5 minutes on a standard desktop. Accepting a slight deterioration of strength (loosing one example), this benchmark can be handled in less than 5 seconds.

Evaluation. To the best our knowledge there is currently no tool providing a *fully automated* expected value analysis of programs available, regardless whether the considered programming language is imperative or not, admits recursive programs or not. On the other hand, there is ample work on (automated) generation of *quantitative invariants*, see eg. [Bao et al. 2022; Chakarov and Sankaranarayanan 2014; Katoen et al. 2010; McIver and Morgan 2005; Wang et al. 2018], employing a variety of techniques. Furthermore, there is a large body work on *expected cost analysis*, see eg. [Avanzini et al. 2020; Kaminski et al. 2018; Ngo et al. 2018; Wang et al. 2019]. Thus we have chosen examples from these seminal works as basis of the developed benchmark suite. In addition, we have added a number of examples of our own, detailed below. In sum this amounts to a test-suite of 53 examples. The results of these evaluations are given in Tables 1 and 2. In short, we can handle $^{49}/_{53}$ of the benchmark suite, without any recourse to user interaction.

Selection and evaluation results on examples on invariant generation. We haven chosen challenging examples from [Chakarov and Sankaranarayanan 2014; Katoen et al. 2010], detailed in benchmarks (a) and (b) in Table 1. Our tool can handle all of these examples, with the exception of `uniform-dist`, where we can only handle a restrictive instance (denoted as `uniform-dist-100` in the benchmark). We also note that the expectations employed in [Katoen et al. 2010] are more sophisticated than ours. On the other hand only semi-automation is achieved. Further, we consider examples from [Bao et al. 2022; Wang et al. 2018], establishing fully automated methodologies. Wang et al. employ a templating approach similar to ours, while the very recent work by Bao et al. employs a conceptually highly interesting learning approach. The results are described in Table 1 (c), (d) as well as in Table 2 (d), respectively. To suit to the expressivity of `ev-imp`, we instantiate the probabilities by constant once in the majority of the benchmarks in (d). Variable probabilities are not (yet) expressible as upper invariants in our prototype.

We can handle all expect one example from these benchmarks. In the one example `eg` (benchmark (c)) that we cannot handle the templates chosen are not precise enough, which is explained by the more liberal construction we use to handle non-determinism in the definition of $\mathsf{infer}[\cdot]$.

Wrt. precision, the bounds generated by `ev-imp` are often as precise as those generated by the tools from Wang et al. [2018] and Bao et al. [2022]. Wrt. to speed, our prototype implementation `ev-imp` is typically able to handle the benchmark examples in milliseconds. As already mentioned the whole testsuite can be handled in around 5 minutes on a standard desktop.

Selection and evaluation results on examples on expected costs. As argued by Kaminski et al. [2018] an expected value analysis is in general unsound for expected cost analysis. However, if the program under consideration is *almost-surely terminating* (*AST* for short), then an expected cost analysis can be recovered from an expected value analysis by counter instrumentation. In this context, an expected value analysis constitutes a strict extension of an expected cost analysis. Note that all benchmarks considered in Table 2 (f) and (g) are AST (even *positive almost-sure terminating*).

We have incorporated recursive examples from Kaminski et al. [2018], suited to our possibilities, cf. Table 2 (e). While procedure `geo` can be handled instantly with `ev-imp`, the growth rate of the (expected) value of `faulty_factorial` is non-polynomial. We thus suited a variant—dubbed `faulty_sum`—to our benchmarks replacing the multiplication by a sum, thus featuring polynomial growth but similar algorithmic complexity. In addition, we considered challenging—newly introduced examples from Avanzini et al. [2020]—if expressible in our prototype. Wrt. both benchmarks, we adapted the original expected cost analysis to an expected return value analysis, as proposed above. The evaluation results are given in Table 2 (f) and (g). Remarkably, we can handle both recursive examples from Kaminski et al. and also make significant inroad into challenging non-linear example from [Avanzini et al. 2020]. Unsurprisingly, we cannot handle all the selected benchmarks due to the greater generality of our prototype `ev-imp`.

For example, the crucial motivating example by Avanzini et al.—*Coupon Collector*—cannot (yet) be handled by our prototype implementation `ev-imp`, as support for dynamic uniform distributions is lacking. This is left for future work. We can however express (and handle) concrete instances of this benchmark. (We included `coupon-10`, `coupon-50` and `coupon-100` as examples.) Wrt. precision, the bounds generated by `ev-imp` are on the same order of magnitude as those generated by the tools from `eco-imp`, while wrt. speed the generality of our tool takes its toll.

Programs considered in this work. We have already discussed the examples `balls`, `throws`, `every-5` and benchmark example `hire` in Section 2 and 4, respectively. Benchmark `every` constitutes the general case to procedure `every`, where we do not restrict the number of bins to five, while `every-while` constitutes an encoding of this problem as loop program. For the moment, `ev-imp` cannot handle the general encoding of the question how many balls have to be thrown in average until all bins are filled with at least one ball (compare Section 2) due to complexity of the example. The remaining examples constitute recursive variants of standard examples and are given in full in [Avanzini et al. 2023].

6 RELATED WORK

Very briefly, we refer to the extensive literature of analysis methods for (non-deterministic) probabilistic programs introduced in the last years. These have been provided in the form of *abstract interpretations* [Chakarov and Sankaranarayanan 2014]; *martingales*, eg., ranking supermartingales [Agrawal et al. 2018; Takisaka et al. 2018; Wang et al. 2019]; or equivalently *Lyapunov ranking functions* [Bournez and Garnier 2005]; *model checking* [Katoen 2016]; *program logics* [Bao et al. 2022; Kaminski and Katoen 2017; Kaminski et al. 2018; Kaminski and Katoen 2015; McIver and Morgan 2005; McIver et al. 2018; Ngo et al. 2018; Wang et al. 2018]; *proof assistants* [Barthe et al. 2009]; *recurrence relations* [Sedgewick and Flajolet 1996]; methods based on *program analysis* [Celiku and McIver 2005; Katoen et al. 2010; Kozen 1985]; or *symbolic inference* [Gehr et al.

Table 1. Automatically Derived Bounds on the Expected Value via our Prototype ev-imp.

Program	Return Value	Inferred Bound	Time (sec)
(a) examples from Katoen et al. [2010]			
biased-coin	$bool$	$1/2$	0.029
binom-update	x	$1/2 \cdot \langle N \rangle$	0.012
uniform-dist	g	—	2.173
uniform-dist-100	g	99	0.09
(b) examples from Chakarov and Sankaranarayanan [2014]			
mot-ex	$count$	$44/3$	0.02
(c) benchmark from Wang et al. [2018]			
2d-walk	$count$	$\langle 1 + count \rangle$	0.012
aggregate-rv	x	$1/2 \cdot [x \geq -1 \wedge 499 \geq i] \cdot (1 + x) + 1/2 \cdot [499 \geq i] \cdot \langle x \rangle + [i \geq 500] \cdot \langle x \rangle$	0.009
biased-coin	x_1	$1/2 \cdot \langle x_1 \rangle + 1/2 \cdot [x_1 > x_2]$	0.014
binom-update	x	$1/4 \cdot [x \geq -1 \wedge 99 \geq n] \cdot (1 + x) + 3/4 \cdot [99 \geq n] \cdot \langle x \rangle + [n \geq 100] \cdot x$	0.009
coupon5	$count$	$[count \geq -1 \wedge 4 \geq i] \cdot \langle 1 + count \rangle + [i \geq 5] \cdot \langle count \rangle$	0.009
eg-tail	x	$\langle x \rangle + 19/16 \cdot \langle z \rangle + 7/4$	0.056
eg	x	—	0.18
hare-turtle	h	$[h \geq -1 \wedge t \geq h] \cdot 1/22 \cdot \sum_{i=0}^{10} (h + i) + [h > t] \cdot \langle h \rangle + [h \leq t] 6/11 \cdot \langle h \rangle$	0.010
hawk-dove	$count$	1	0.013
mot-ex	$count$	$\langle 1 + count \rangle$	0.009
recursive	x	$\langle x \rangle + 10$	0.03
uniform-dist	g	$1/2 \cdot ([g \geq -1/2 \wedge 9 \geq n] \cdot (1 + 2 \cdot y) + [9 \geq n] \cdot 2 \cdot \langle y \rangle) + [n \geq 10] \cdot \langle g \rangle$	0.009
(d) benchmark from Bao et al. [2022]			
biasdir	x	$\mathsf{N}1 - x + \langle x \rangle$	0.022
bin0	x	$1/2 \cdot \langle n \rangle \cdot \langle y \rangle + \langle x \rangle$	0.075
bin1	x	$1/2 \cdot \langle M - n \rangle + \langle x \rangle$	0.012
bin2	x	$1/4 \cdot (\langle n \rangle + n^2) + 1/2 \cdot \langle n \rangle \cdot \langle y \rangle + \langle x \rangle$	0.141
deprv	z	$\langle z \rangle$	0.011
detm	$count$	$\langle 11 - x \rangle + \langle count \rangle$	0.012
duel	$turn$	$\langle turn \rangle + 2 \cdot \langle continuing \rangle$	0.036
fair	$count$	$\langle count \rangle + 2 \cdot \langle 1 - c_2 \rangle$	0.86
gambler0	z	$[x \geq 0 \wedge y \geq x] \cdot (x \cdot y - x^2) + \langle z \rangle$	0.063
geo0	z	$\langle 1 - flip \rangle + \langle z \rangle$	0.014

2016]; and finally *type systems* [Avanzini et al. 2019; Leutgeb et al. 2022; Vasilenko et al. 2022; Wang et al. 2020].

Invariant generation. Generally, invariant generation is more challenging than the expected result value analysis that we study. Still, often ev-imp derives exact bounds, thus establishing invariants. On the other hand, our methodology is applicable in a more general framework, for example to expected cost analysis. Further, our methodology encompasses recursive (imperative) programs, which is—to the best of our knowledge—not the case for any of the approaches on invariant generation (or expected cost analysis, for that matter). Katoen et al. [2010] provide constraint-based methods for the semi-automated generation of linear quantitative invariants, based on sophisticated proof-based methods. The studied examples are highly interesting and have been integrated into our benchmarks (see Section 5). The form of expectations considered can be very expressive. We emphasise, however, that our method is fully automated, while the approach in [Katoen et al. 2010] is only partly automated, in particular nested loops require user-interaction. Related results have been reported by Chakarov and Sankaranarayanan [2014], suitable adapting an

Table 2. Automatically Derived Bounds on the Expected Value via our Prototype ev-imp.

Program	Return Value	Inferred Bound	Time (sec)
(d) benchmark from Bao et al. [2022] (cont'd)			
geo1	z	$\langle 1 - flip \rangle + \langle z \rangle$	0.014
geo2	z	$\langle 1 - flip \rangle + \langle z \rangle$	0.015
geoar0	x	$^3/_2(1 + \langle 1 + y \rangle + \langle x \rangle)$	0.09
linexp	z	$\langle z \rangle + {}^{21}/_8 \cdot \langle n \rangle$	0.014
mart	$rounds$	$\langle rounds \rangle + 3 \cdot \langle b \rangle$	0.012
prinsys	$bool$	1	0.013
revbin	z	$\langle z \rangle + 2 \cdot \langle x \rangle$	0.011
sum0	x	$^1/_4 \cdot (\langle n \rangle + n^2) + \langle x \rangle$	0.034
(e) examples from [Kaminski et al. 2018]			
faulty_sum	x	$^3/_7 \cdot \langle x \rangle^2 + {}^4/_7 \cdot \langle x \rangle + 1$	0.04
geo	0	0	0.009
(f) benchmark from [Avanzini et al. 2020]			
bridge	$count$	$[b \geq x \wedge x \geq a] \cdot (-a \cdot b + a \cdot x + b \cdot x - x^2)$	0.067
coupon-10	$count$	110	0.145
coupon-50	$count$	2550	2.691
coupon-100	$count$	10100	10.63
nest-1	$count$	$4 \cdot \langle n \rangle$	0.014
nest-2	$count$	—	100.00
trader-5	$\langle price \rangle$	$^{20}/_3 \cdot [price \geq -1] \cdot (1 + 2 \cdot price + price^2) + {}^{35}/_6 \cdot \langle 1 + price \rangle$	
(g) examples from this work			
balls	b	$^1/_5 \cdot \langle n \rangle$	0.012
throws	$throws$	5	0.012
every	$number$	—	100.0
every-5	$number$	20	0.017
every-while	$number$	25	0.973
double_recursive	y	0	0.011
hire	$hire$	$\langle n \rangle$	0.504
rdwalk	n	$2 \cdot \langle n \rangle$	0.016
rec1	n	$^1/_2 \cdot (1 + \langle n \rangle)$	0.014

abstract interpretation framework to the notion of invariant generation. Their motivating example can be handled by ev-imp fully automatically, establishing a slightly worse constant bound than the exact bound (see Section 5). In contrast to [Chakarov and Sankaranarayanan 2014; Katoen et al. 2010], Wang et al. [2018] and the very recent Bao et al. [2022] provide fully automated (linear) invariant generation methodologies. Wang et al. [2018] provide a compelling algebraic framework for the program analysis of probabilistic programs. Apart from the here relevant linear invariant generation, interprocedural Bayesian inference analysis and the Markov decision problem are conducted. Wrt. linear invariant generation, we have considered all provided examples in our benchmarks and obtained comparable results, while improving the speed of the analysis by a magnitude (in comparison to the analysis times reported in [Wang et al. 2018]), cf. Section 5. Automation of the method developed in [Wang et al. 2018] is based, like ours, on a template approach. To overcome the dependency on templates, Bao et al. [2022] have developed a highly interesting learning approach. Their approach is data-driven. Apart from invariants, Bao et al. also consider *sub-invariants* which are dual to our upper invariants, establishing lower bounds on the pre-expectations. In both cases, however, analysis times are high. We have incorporated

the benchmark examples from [Bao et al. 2022] into our test suites. In all cases our tool `ev-imp` provides precise invariants, cf. Section 5. This is remarkable, as reported in [Bao et al. 2022], their prototype implementation `Exist` handles only 12/18 of their benchmark suite fully automatically.

Expected cost analysis. Kaminski et al. [2018] establishes an expected cost analysis of recursive programs and we have suited the corresponding two example to our benchmark suite. Both examples can be handled fully automatically. Technically our development of recursive programs constitutes an extension as our language (and methodology) admits local variables, formal parameters and (unrestricted) return values. This allows a more natural representation of programs (see Section 3). Still the definition of our expectation transformer $\mathsf{et}[\![\mathsf{p}]\!]$ is closely related to the corresponding definition in [Kaminski et al. 2018, Chapter 7]. We emphasise, however, that in [Kaminski et al. 2018] automation is discussed only superficially. Avanzini et al. [2020], on the other hand, take automation very seriously. As mentioned above, we have taken inspiration from their work in the modular analysis of recursion-free programs. Despite our efforts, however, our prototype implementation `ev-imp` lacks scalability and speed in comparison to their tool `eco-imp`. Wrt. precision, however, we often derive the same bounds on expected costs. Further, and as argued, our methodology focusing on expected value analysis is more general and our implementation incorporated the highly non-trivial handling of recursive programs. Technically the closest comparison is to work on amortised cost analysis of functional languages (eg. [Leutgeb et al. 2022; Wang et al. 2020]), as resource parametricity [Hoffmann 2011] has been studied in this context. In comparison to Wang et al. [2020], our tool `ev-imp` cannot infer higher-order moments but concerning expectations seems to have better support for recursion. For instance, we were not able to reproduce an expected cost analysis of the `balls` procedure in their RaML prototype. Similarly, `ev-imp` is no match to Leutgeb et al. [2022]'s ATLAS, when it comes to the precise analysis of (probabilistic) data structures. Remarkably, however, the `balls` benchmark can only be expressed convolutedly in their language. (Due to the lack of support for general, inductive, data structures.) Further, their automated analysis does not derive an optimal bound.

Expected value analysis. Finally, we briefly remark on very recent and partly motivating work by Vasilenko et al. [2022]. In [Vasilenko et al. 2022] a refinement type system—Liquid Haskell, cf. [Handley et al. 2020; Vazou 2016]—is updated, to reason about relational properties of probabilistic computations. Eg. Vasilenko et al. [2022] provide a semi-automated proof that for (a variant of) procedure `balls`, cf. Listing 1(a), the expected return value of b is given as $p \cdot n$.

7 CONCLUSION

We established a fully automated expected result value analysis for probabilistic programs in the presence of natural programming constructs, in particular recursion. Our analysis is in the form of an *expectation transformer* $\mathsf{et}[\![\cdot]\!]$ and its syntactic representation $\mathsf{infer}[\cdot]$. As argued, automated inference of upper invariants is challenging for PWhile, due to the presence of recursion. We have overcome these challenges and implemented the established methodology in our novel prototype implementation `ev-imp`. In future work, we aim to incorporate (i) *more program features* and , like eg. support for *dynamic* uniform distributions; (ii) improve the *constraint solving* capabilities of our prototype implementation, to handle the analysis of further natural probabilistic programs and data structures fully automatically.

ACKNOWLEDGMENTS

We would like the thank the annoymous reviewers for their work and invaluable suggestions, which greatly improved our presentation. This work is partly supported by the INRIA Associate Team TC(Pro)^3 and by the ANR Project PPS: "Probabilistic Program Semantics".

8 AVAILABILITY OF DATA AND SOFTWARE

Our prototype implementation `ev-imp` is publicly available via the following Zenodo link: https://doi.org/10.5281/zenodo.7706691. The artifact is given as a Docker image running a minimal Linux distribution, containing the sources in directory /evimp, as well as pre-compiled executables. A second, larger, image contains the source distribution as well as the complete toolchain to compile `ev-imp`. We have successfully tested the images under Linux, MacOS and Windows, as long as these were non-ARM architectures. For ARM64 architectures, the artifact additionaly provides instructions to install from source. Detailed instructions on the use of `ev-imp` are provided as well.

REFERENCES

S. Agrawal, K. Chatterjee, and P. Novotný. 2018. Lexicographic Ranking Supermartingales: An Efficient Approach to Termination of Probabilistic Programs. *PACMPL* 2, POPL (2018), 34:1–34:32. https://doi.org/10.1145/3385412.3386002

M. Avanzini, G. Barthe, and U. Dal Lago. 2021. On Continuation-Passing Transformations and Expected Cost Analysis. *PACM on Programming Languages* 5, ICFP (2021), 1–30. https://doi.org/10.1145/3473592

M. Avanzini, U. Dal Lago, and A. Ghyselen. 2019. Type-Based Complexity Analysis of Probabilistic Functional Programs. In *Proc. of 34th LICS*. IEEE, 1–13. https://doi.org/10.1109/LICS.2019.8785725

Martin Avanzini, Georg Moser, and Michael Schaper. 2020. A modular cost analysis for probabilistic programs. *Proc. ACM Program. Lang.* 4, OOPSLA (2020), 172:1–172:30. https://doi.org/10.1145/3428240

Martin Avanzini, Georg Moser, and Michael Schaper. 2023. Automated Expected Value Analysis of Recursive Programs. arXiv:2304.01284 [cs.PL]

Jialu Bao, Nitesh Trivedi, Drashti Pathak, Justin Hsu, and Subhajit Roy. 2022. Data-Driven Invariant Learning for Probabilistic Programs. In *Proc. of 34th CAV (LNCS, Vol. 13371)*. 33–54. https://doi.org/10.1007/978-3-031-13185-1_3

G. Barthe, B. Grégoire, and S. Z. Béguelin. 2009. Formal Certification of Code-based Cryptographic Proofs. In *Proc. of 36th POPL*. ACM, 90–101. https://doi.org/10.1145/1480881.1480894

Kevin Batz, Benjamin Lucien Kaminski, Joost-Pieter Katoen, Christoph Matheja, and Thomas Noll. 2019. Quantitative separation logic: a logic for reasoning about probabilistic pointer programs. *PACM on Programming Languages* 3, POPL (2019), 34:1–34:29. https://doi.org/10.1145/3290347

O. Bournez and F. Garnier. 2005. Proving Positive Almost-Sure Termination. In *Proc. of 16th RTA (LNCS, Vol. 3467)*. Springer, 323–337. https://doi.org/10.1142/S0129054112400588

O. Celiku and A. McIver. 2005. Compositional Specification and Analysis of Cost-Based Properties in Probabilistic Programs. In *Proc. of FM 2005 (LNCS, Vol. 3582)*. Springer, 107–122. https://doi.org/10.1007/11526841_9

A. Chakarov and S. Sankaranarayanan. 2014. Expectation Invariants for Probabilistic Program Loops as Fixed Points. In *Proc. of 21th SAS (LNCS)*. Springer, 85–100. https://doi.org/10.1007/978-3-319-10936-7_6

E. Contejean, C. Marché, A.-P. Tomás, and X. Urbain. 2005. Mechanically Proving Termination Using Polynomial Interpretations. *JAR* 34, 4 (2005), 325–363. https://doi.org/10.1007/s10817-005-9022-x

Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2009. *Introduction To Algorithms* (3rd ed.). MIT Press. http://mitpress.mit.edu/books/introduction-algorithms

Leonardo Mendonça de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Proc. 14th TACAS (LNCS, Vol. 4963)*. 337–340.

E. W. Dijkstra. 1975. Guarded Commands, Nondeterminacy and Formal Derivation of Programs. *ACM* 18, 8 (1975), 453–457.

E. W. Dijkstra. 1976. *A Discipline of Programming*. Prentice-Hall.

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

Manuel Eberl, Max W. Haslbeck, and Tobias Nipkow. 2020. Verified Analysis of Random Binary Tree Structures. *J. Autom. Reason.* 64, 5 (2020), 879–910. https://doi.org/10.1007/s10817-020-09545-0

Daniel P. Friedman and Mitchell Wand. 2008. *Essentials of Programming Languages* (3rd ed.). The MIT Press.

C. Fuhs, J. Giesl, A. Middeldorp, P. Schneider-Kamp, R. Thiemann, and H. Zankl. 2007. SAT Solving for Termination Analysis with Polynomial Interpretations. In *Proc. of 10th SAT (LNCS, Vol. 4501)*. Springer, 340–354. https://doi.org/10.1007/978-3-540-72788-0_33

T. Gehr, S. Misailovic, and M. Vechev. 2016. PSI: Exact Symbolic Inference for Probabilistic Programs. In *Proc. of 28th CAV (LNCS, Vol. 9779)*. Springer, 62–83. https://doi.org/10.1007/978-3-319-41528-4_4

Ronald L. Graham, Donald E. Knuth, and Oren Patashnik. 1994. *Concrete Mathematics: A Foundation for Computer Science* (2nd ed.). Addison-Wesley. https://www-cs-faculty.stanford.edu/%7Eknuth/gkp.html

Friedrich Gretz, Joost-Pieter Katoen, and Annabelle McIver. 2014. Operational versus weakest pre-expectation semantics for the probabilistic guarded command language. *Performance Evaluation* 73 (2014), 110–132.

D. Handelman. 1988. Representing Polynomials by Positive Linear Functions on Compact Convex Polyhedra. *PJM* 132, 1 (1988), 35–62. https://doi.org/10.2140/pjm.1988.132.35

Martin A. T. Handley, Niki Vazou, and Graham Hutton. 2020. Liquidate your assets: reasoning about resource usage in liquid Haskell. *PACMPL* 4, POPL (2020), 24:1–24:27. https://doi.org/10.1145/3371092

C. A. R. Hoare. 1969. An Axiomatic Basis for Computer Programming. *Commun. ACM* 12, 10 (1969), 576–580. https://doi.org/10.1145/363235.363259

Jan Hoffmann. 2011. *Types with potential: polynomial resource bounds via automatic amortized analysis.* Ph. D. Dissertation. Ludwig Maximilians University Munich. http://edoc.ub.uni-muenchen.de/13955/

B. L. Kaminski and J.-P. Katoen. 2017. A Weakest Pre-expectation Semantics for Mixed-sign Expectations. In *Proc. of 32nd LICS*. IEEE, 1–12. https://doi.org/10.1109/LICS.2017.8005153

B. Lucien Kaminski, J.-P. Katoen, C. Matheja, and F. Olmedo. 2016. Weakest Precondition Reasoning for Expected Run-Times of Probabilistic Programs. In *Proc. of 25th ESOP (LNCS, Vol. 9632)*. Springer, 364–389. https://doi.org/10.1007/978-3-662-49498-1_15

B. L. Kaminski, J.-P. Katoen, C. Matheja, and F. Olmedo. 2018. Weakest Precondition Reasoning for Expected Runtimes of Randomized Algorithms. *JACM* 65, 5 (2018), 30:1–30:68. https://doi.org/10.1145/3208102

B. L. Kaminski and J.-P. Katoen. 2015. On the Hardness of Almost-Sure Termination. In *MFCS 2015, Part I (LNCS)*. Springer, 307–318. https://doi.org/10.1007/978-3-662-48057-1_24

J.-P. Katoen. 2016. The Probabilistic Model Checking Landscape. In *Proc. of 31nd LICS*. ACM, 31–45. https://doi.org/10.1145/2933575.2934574

J.-P. Katoen, A. McIver, L. Meinicke, and C.C. Morgan. 2010. Linear-Invariant Generation for Probabilistic Programs: - Automated Support for Proof-Based Methods. In *Proc. of 17th SAS (LNCS, Vol. 6337)*. Springer, 390–406. https://doi.org/10.1007/978-3-642-15769-1_24

Thomas Kleymann. 1999. Hoare Logic and Auxiliary Variables. *Formal Aspects Comput.* 11, 5 (1999), 541–566. https://doi.org/10.1007/s001650050057

D. Kozen. 1985. A Probabilistic PDL. *JCSC* 30, 2 (1985), 162 – 178. https://doi.org/10.1016/0022-0000(85)90012-1

Lorenz Leutgeb, Georg Moser, and Florian Zuleger. 2021. ATLAS: Automated Amortised Complexity Analysis of Self-adjusting Data Structures. In *Proc. of 33th CAV (LNCS, Vol. 12760)*. 99–122. https://doi.org/10.1007/978-3-030-81688-9_5

Lorenz Leutgeb, Georg Moser, and Florian Zuleger. 2022. Automated Expected Amortised Cost Analysis of Probabilistic Data Structures. In *Proc. of 34th CAV (LNCS, Vol. 13372)*. 70–91. https://doi.org/10.1007/978-3-031-13188-2_4

Annabelle McIver and Carroll Morgan. 2005. *Abstraction, refinement and proof for probabilistic systems.* Springer Science & Business Media.

A. McIver, C. Morgan, B. L. Kaminski, and J-P Katoen. 2018. A New Proof Rule for Almost-sure Termination. *PACMPL* 2, POPL (2018), 33:1–33:28. https://doi.org/10.1145/3158121

M. Mitzenmacher and E. Upfal. 2005. *Probability and Computing: Randomized Algorithms and Probabilistic Analysis.* Cambridge University Press. https://doi.org/10.1017/CBO9780511813603

N. C. Ngo, Q. Carbonneaux, and J. Hoffmann. 2018. Bounded Expectations: Resource Analysis for Probabilistic Programs. In *Proc. of 39th PLDI*. ACM, 496–512. https://doi.org/10.1145/3296979.3192394

Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. 1999. *Principles of program analysis.* Springer. https://doi.org/10.1007/978-3-662-03811-6

Hanne Riis Nielson. 1987. A Hoare-Like Proof System for Analysing the Computation Time of Programs. *Sci. Comput. Program.* 9, 2 (1987), 107–136. https://doi.org/10.1016/0167-6423(87)90029-3

A. Podelski and A. Rybalchenko. 2004. A Complete Method for the Synthesis of Linear Ranking Functions. In *Proc. of 5th VMCAI (LNCS, Vol. 2937)*. Springer, 239–251. https://doi.org/10.1007/978-3-540-24622-0_20

E. Schlechter. 1996. *Handbook of Analysis and Its Foundations.* Elsevier.

Roberto Sebastiani and Patrick Trentin. 2020. OptiMathSAT: A Tool for Optimization Modulo Theories. *J. Autom. Reason.* 64, 3 (2020), 423–460. https://doi.org/10.1007/s10817-018-09508-6

R. Sedgewick and P. Flajolet. 1996. *An Introduction to the Analysis of Algorithms.* Addison-Wesley-Longman. https://doi.org/10.1142/10875

M. Sinn, F. Zuleger, and H. Veith. 2014. A Simple and Scalable Static Analysis for Bound Analysis and Amortized Complexity Analysis. In *Proc. of 26th CAV (LNCS, Vol. 8559)*. Springer, Heidelberg, DE, 745–761.

M. Sinn, F. Zuleger, and H. Veith. 2016. A Simple and Scalable Static Analysis for Bound Analysis and Amortized Complexity Analysis. In *Proc. of 26th CAV (LNCS, Vol. 8559)*. Springer, 745–761. https://doi.org/10.1007/978-3-319-08867-9_50

T. Takisaka, Y. Oyabu, N. Urabe, and I. Hasuo. 2018. Ranking and Repulsing Supermartingales for Reachability in Probabilistic Programs. In *Proc. of 16th ATVA (LNCS, Vol. 11138)*. Springer, 476–493. https://doi.org/10.1007/978-3-030-01090-4_28

Elizaveta Vasilenko, Niki Vazou, and Gilles Barthe. 2022. Safe couplings: coupled refinement types. *Proc. ACM Program. Lang.* 6, ICFP (2022), 596–624. https://doi.org/10.1145/3547643

Niki Vazou. 2016. *Liquid Haskell: Haskell as a Theorem Prover.* Ph. D. Dissertation. University of California, San Diego, USA. http://www.escholarship.org/uc/item/8dm057ws

D. Wang, J. Hoffmann, and T. W. Reps. 2018. PMAF: An Algebraic Framework for Static Analysis of Probabilistic Programs. In *Proc. of 39th PLDI.* ACM, 513–528. https://doi.org/10.1145/3192366.3192408

Di Wang, David M. Kahn, and Jan Hoffmann. 2020. Raising expectations: automating expected cost analysis with types. *PACM on Programming Languages* 4, ICFP (2020), 110:1–110:31. https://doi.org/10.1145/3408992

P. Wang, H. Fu, A. K. Goharshady, K. Chatterjee, X. Qin, and W. Shi. 2019. Cost Analysis of Nondeterministic Probabilistic Programs. In *Proc. of 40th PLDI.* ACM, 204–220. https://doi.org/10.1145/3314221.3314581

Wolfgang Wechler. 1992. *Universal Algebra for Computer Scientists.* EATCS Monographs on Theoretical Computer Science, Vol. 25. Springer. https://doi.org/10.1007/978-3-642-76771-5

G. Winskel. 1993. *The Formal Semantics of Programming Languages.* MIT Press. https://doi.org/10.7551/mitpress/3054.003.0004

A MATHEMATICAL BACKGROUND

Proposition A.1 (Function Lifting of ω-CPOs [Winskel 1993, Section 8.3.3]). *Let $(D, \sqsubseteq)$ be an ω-CPO. Then $(A \to D, \sqsubseteq)$ where $\sqsubseteq$ extends $\sqsubseteq$ point-wise forms an ω-CPO, with the supremum on $A \to D$ given point-wise. If $\sqsubseteq$ has least and greatest elements $\bot$ and $\top$, then $\bot$ and $\top$ are the least and greatest elements of $\sqsubseteq$, respectively.*

Theorem A.2 (Kleene's Fixed-Point Theorem for ω-CPOs, [Winskel 1993, Theorem 5.11]). *Let $(D, \sqsubseteq)$ be a ω-CPO with least element $\bot$. Let $\chi \colon D \to D$ be continuous (thus monotone). Then χ has a least fixed-point given by*

$$\mathsf{lfp}\,(\chi) = \sup_{n \in \mathbb{N}} \chi^n(\bot)\ .$$

Lemma A.3 (Continuity of Expectation). $\mathbb{E}_\mu \sup_{n\in\mathbb{N}} f_n = \sup_{n\in\mathbb{N}} \mathbb{E}_\mu\, f_n$ *for all ω-chains $(f_n)_{n\in\mathbb{N}}$.*

Proof. This is the discrete version of Lebesgue's Monotone Convergence Theorem [Schlechter 1996, Theorem 21.38]. □

B OMITTED PROOFS

Proposition B.1 (Expectation Tranformer Laws). *For any program P, any procedure environment η, any command C and any expectations f, f_1, f_2, g, g_1, g_2, ... the laws in Figure 3 hold.*

Proof. The proofs follow the pattern of the proof of continuity of the expected cost transformer in [Avanzini et al. 2020, Lemma 6.2]. □

We define finite approximations of procedure environments $\mathsf{et}[\![\mathsf{P}]\!]^{(i)}$ inductively, so that $\mathsf{et}[\![\mathsf{P}]\!]^{(0)} := \lambda\mathsf{p}.\,\lambda f\,\vec{v}\,\sigma.\,0$ and $\mathsf{et}[\![\mathsf{P}]\!]^{(i+1)} := \lambda\mathsf{p}.\,\mathsf{et}[\![\mathsf{p}]\!]^{\mathsf{et}[\![\mathsf{P}]\!]^{(i)}}$, where $\mathsf{p} \in \mathsf{P}$. In this way $\mathsf{et}[\![\mathsf{P}]\!]^{(0)}$ represents the poorest approximation by the constant zero function, while approximations are refined iteratively.

Proposition B.2 (Finite Approximations of Procedure Environments). *For any program P, we have $\mathsf{et}[\![\mathsf{P}]\!] = \sup_{i\geqslant 0} \mathsf{et}[\![\mathsf{P}]\!]^{(i)}$.*

Proof. Direct consequence of the continuity of $\mathsf{et}[\![\mathsf{p}]\!]$ and Knaster-Tarski fixed-point theorem. □

Theorem 4.1 (Soundness Theorem). *If for all $\mathsf{p} \in \mathsf{P}$ the constraint (1) is fulfilled and all side-conditions in Figure 4 are met, then for all $\mathsf{p} \in \mathsf{P}$, $\mathsf{et}[\![\mathsf{p}]\!] \leq [\![\mathsf{infer}[\mathsf{p}]]\!]$, that is, the inference algorithm is sound.*

Proof. Let $\mathsf{S}, \mathsf{T} \in \mathsf{Term}\,V\,Z$ and $\theta \colon Z \to \mathbb{Z}$. Then we prove for all commands C that

$$\mathsf{et}[\![\mathsf{C}]\!]^{\eta}_{[\![\mathsf{S}\theta]\!]}\,[\![\mathsf{T}\theta]\!] \leq [\![(\mathsf{infer}[\mathsf{C}]_{\mathsf{S}}\,\mathsf{T})\theta]\!]\ . \tag{2}$$

Let $s := [\![\mathsf{S}\theta]\!]$ and $t := [\![\mathsf{T}\theta]\!]$. In order to prove (2), we prove the following, for all $i \in \mathbb{N}$:

$$\mathsf{et}[\![\mathsf{C}]\!]^{\eta_i}_{s}\,t \leq [\![(\mathsf{infer}[\mathsf{C}]_{\mathsf{S}}\,\mathsf{T})\,\theta]\!]\ ,$$

where (i) $\eta_0\,\mathsf{p} := \lambda\mathsf{p}.0$; (ii) $\eta_{i+1}\,\mathsf{p} := \mathsf{et}[\![\mathsf{p}]\!]^{\eta_i}$; and $\eta := \sup_{i\geqslant 0} \eta_i$.

In proof, we elide the logical context Γ_{p}, employed in the definition of $\mathsf{infer}[\mathsf{p}]$ for notational convenience. As the essence of this context information is that logical variables are always instantiated non-negatively, this can be guaranteed globally.

We proceed by main induction on i and side induction on C, where we focus on the (main) step case, as the case for $i = 0$ is similar, but simpler. Thus let $i > 0$ and we proceed by case induction on C, considering the most interesting cases, only.

- Case `skip`. By unfolding of definitions, we easily obtain

$$\mathrm{et}[\![\texttt{skip}]\!]_s^{\eta_{i+1}}\, t = [\![\mathsf{T}\,\theta]\!] = [\![(\mathsf{infer}[\texttt{skip}]_{\mathsf{S}}\,\mathsf{T})\,\theta]\!]\ ,$$

  which concludes the case.
- Case $x :\approx \mathsf{p}(\vec{\mathsf{E}})$. Suppose $\mathsf{p} \in \mathsf{Prog}$. By unfolding of definitions, we obtain for $\sigma \in \mathsf{Mem}\,V$

$$\begin{aligned}
\mathrm{et}[\![x :\approx \mathsf{p}(\vec{\mathsf{E}})]\!]_s^{\eta_{i+1}}\, t\,\sigma &= \eta_{i+1}\,\mathsf{p}\ \underbrace{(\lambda v\,\tau.\, t\,(\sigma_{\mathsf{l}} \uplus \tau_{\mathsf{g}})[x \mapsto v])}_{:=f}\ ([\![\vec{\mathsf{E}}]\!]\,\sigma)\,\sigma_{\mathsf{g}} \\
&= \mathrm{et}[\![\mathsf{p}]\!]^{\eta_i}\, f\,([\![\vec{\mathsf{E}}]\!]\,\sigma)\,\sigma_{\mathsf{g}} \\
&= \mathrm{et}[\![\mathsf{Bdy}_{\mathsf{p}}]\!]_f^{\eta_i}\,(\lambda\tau.\, t\,(\sigma_{\mathsf{l}} \uplus \tau_{\mathsf{g}})[x \mapsto 0]\,\sigma_{\mathsf{g}} \uplus \{\mathsf{Args}_{\mathsf{p}} \mapsto [\![\vec{\mathsf{E}}]\!]\,\sigma\}) \\
&\leq \mathrm{et}[\![\mathsf{Bdy}_{\mathsf{p}}]\!]_{[\![\mathsf{K}_{\mathsf{p}}\,\theta]\!]}^{\eta_i}\,[\![\mathsf{K}_{\mathsf{p}}[\mathbf{r} \mapsto 0]\,\theta]\!]\,\sigma_{\mathsf{g}} \uplus \{\mathsf{Args}_{\mathsf{p}} \mapsto [\![\vec{\mathsf{E}}]\!]\,\sigma\} \\
&\leq [\![(\mathsf{infer}[\mathsf{Bdy}_{\mathsf{p}}]_{\mathsf{K}_{\mathsf{p}}}\,\mathsf{K}_{\mathsf{p}}[\mathbf{r} \mapsto 0])\,\theta]\!]\,\sigma_{\mathsf{g}} \uplus \{\mathsf{Args}_{\mathsf{p}} \mapsto [\![\vec{\mathsf{E}}]\!]\,\sigma\} \\
&= [\![(\mathsf{infer}[\mathsf{Bdy}_{\mathsf{p}}]_{\mathsf{K}_{\mathsf{p}}}\,\mathsf{K}_{\mathsf{p}}[\mathbf{r} \mapsto 0])[\mathsf{Args}_{\mathsf{p}} \mapsto \vec{\mathbf{a}}_{\mathsf{p}}]\,\theta]\!]\,\sigma_{\mathsf{g}} \uplus \{\vec{\mathbf{a}}_{\mathsf{p}} \mapsto [\![\vec{\mathsf{E}}]\!]\,\sigma\} \\
&= [\![(\mathsf{infer}[\mathsf{p}]\,\mathsf{K}_{\mathsf{p}})\,\theta]\!]\,\sigma_{\mathsf{g}} \uplus \{\vec{\mathbf{a}}_{\mathsf{p}} \mapsto [\![\vec{\mathsf{E}}]\!]\,\sigma\} \\
&\leq [\![\mathsf{H}_{\mathsf{p}}\,\theta]\!]\,\sigma_{\mathsf{g}} \uplus \{\vec{\mathbf{a}}_{\mathsf{p}} \mapsto [\![\vec{\mathsf{E}}]\!]\,\sigma\} \\
&= [\![\mathsf{H}_{\mathsf{p}}[\vec{\mathbf{a}}_{\mathsf{p}} \mapsto \vec{\mathsf{E}}]\,\theta]\!]\,\sigma = [\![(\mathsf{infer}[x :\approx \mathsf{p}(\vec{\mathsf{E}})]_{\mathsf{S}}\,\mathsf{T}\,\theta]\!]\,\sigma\ .
\end{aligned}$$

  Here, we have used in conjunction with reduction to definitions (i) two instances of the assumed side-condition $\vdash \mathsf{T}[x \mapsto \mathbf{r}] \leq \mathsf{K}_{\mathsf{q}}\,\theta$ in line three; (ii) together with monotonicity of the expectation transformer $\mathrm{et}[\![\mathsf{C}]\!]$, cf. Figure 3; (iii) induction hypothesis in line four; and finally (iv) in the pre-ultimate line, the main constraint on the soundness of the inference mechanisms (1). This concludes the case.
- Case `return`(E). By unfolding of definitions, we obtain for $\sigma \in \mathsf{Mem}\,V$

$$\begin{aligned}
\mathrm{et}[\![\texttt{return}(\mathsf{E})]\!]_s^{\eta_{i+1}}\, t\,\sigma &= [\![\mathsf{S}\,\theta]\!]\,([\![E]\!]\,\sigma)\,\sigma_{\mathsf{g}} \\
&= [\![\mathsf{S}[\mathbf{r} \mapsto \mathsf{E}]\,\theta]\!]\,\sigma \\
&= [\![(\mathsf{infer}[\texttt{return}(\mathsf{E})]_{\mathsf{S}}\,\mathsf{T})\,\theta]\!]\,\sigma\ .
\end{aligned}$$

  This concludes the case.
- Case `var` $x \leftarrow \mathsf{E}$ `in` $\{\mathsf{C}\}$. By unfolding of definitions in conjunction with application of the induction hypothesis, we obtain for $\sigma \in \mathsf{Mem}\,V$

$$\begin{aligned}
\mathrm{et}[\![\texttt{var}\ x \leftarrow \mathsf{E}\ \texttt{in}\ \{\mathsf{C}\}]\!]_s^{\eta_{i+1}}\, t\,\sigma &= \mathrm{et}[\![\mathsf{C}]\!]_s^{\eta_{i+1}}\, t\,\sigma[x \mapsto [\![\mathsf{E}]\!]\,\sigma] \\
&\leq [\![(\mathsf{infer}[\mathsf{C}]_{\mathsf{S}}\,\mathsf{T})\,\theta]\!]\,\sigma[x \mapsto [\![\mathsf{E}]\!]\,\sigma] \\
&= [\![(\mathsf{infer}[\mathsf{C}]_{\mathsf{S}}\,\mathsf{T})[x \mapsto \mathsf{E}]\,\theta]\!]\,\sigma \\
&= [\![(\mathsf{infer}[\texttt{var}\ x \leftarrow \mathsf{E}\ \texttt{in}\ \{\mathsf{C}\}]_{\mathsf{S}}\,\mathsf{T})\,\theta]\!]\,\sigma
\end{aligned}$$

  In the third line, we employ that due to the variable condition the local variable x is distinct from all (global) variables in the domain of σ. This concludes the case.

Fig. 6. Additional Benchmark Examples

```
def f(x):
  var y
  if (x ≥ 0) {
    if (Bernoulli(\frac{1}{2})) {
      y :≈ f(x) }
    if (Bernoulli(\frac{1}{3})) {
      y :≈ f(y) }
    };
  return y
```

(a) `double_recursive` example.

```
def rdwalk(n):
  var c := 0
  if (n > 1) {
    if (Bernoulli(\frac{1}{2})) {
      c :≈ rdwalk(n-2) }
    else {
      c :≈ rdwalk(n+1) }
    return c+1 }
  else { return c }
```

(b) `rdwalk` benchmark example.

- Case `C;D`. By unfolding of definitions in conjunction with two applications of the induction hypothesis, we obtain for $\sigma \in \mathsf{Mem}\, V$

$$\begin{aligned}\mathsf{et}[\![\mathtt{C;D}]\!]_s^{\eta_{i+1}}\, t\, \sigma &= \mathsf{et}[\![\mathtt{C}]\!]_s^{\eta_{i+1}}\, (\mathsf{et}[\![\mathtt{D}]\!]_s^{\eta_{i+1}}\, t)\, \sigma \\ &\leq [\![(\mathsf{infer}[\mathtt{C}]_{\mathtt{S}}\, \mathtt{T})\, \theta]\!]\, [\![(\mathsf{infer}[\mathtt{D}]_{\mathtt{S}}\, \mathtt{T})\, \theta]\!]\, \sigma \\ &= [\![\mathsf{infer}[\mathtt{C}]_{\mathtt{S}}\, (\mathsf{infer}[\mathtt{D}]_{\mathtt{S}}\, \mathtt{T})\, \theta]\!]\, \sigma \\ &= [\![(\mathsf{infer}[\mathtt{C;D}]_{\mathtt{S}}\, \mathtt{T})\, \theta]\!]\, \sigma\end{aligned}$$

  Apart from applications of the induction hypothesis, in line two, we have employed monotonicity, cf. Figure 3. This concludes the case.
- Case `if (B) {C} else {D}`. By unfolding of definitions in conjunction with two applications of the induction hypothesis, we obtain for $\sigma \in \mathsf{Mem}\, V$

$$\begin{aligned}\mathsf{et}[\![\mathtt{if\ (B)\ \{C\}\ else\ \{D\}}]\!]_s^{\eta_{i+1}}\, t\, \sigma &= [[\![\mathtt{B}]\!]\, \sigma] \cdot \mathsf{et}[\![\mathtt{C}]\!]_s^{\eta_{i+1}}\, t\, \sigma + [[\![\neg \mathtt{B}]\!]\, \sigma] \cdot \mathsf{et}[\![\mathtt{D}]\!]_s^{\eta_{i+1}}\, t\, \sigma \\ &\leq [[\![\mathtt{B}]\!]\, \sigma] \cdot [\![(\mathsf{infer}[\mathtt{C}]_{\mathtt{S}}\, \mathtt{T})\, \theta]\!] + [[\![\neg \mathtt{B}]\!]\, \sigma] \cdot [\![(\mathsf{infer}[\mathtt{D}]_{\mathtt{S}}\, \mathtt{T})\, \theta]\!] \\ &= [\![([\mathtt{B}] \cdot \mathsf{infer}[\mathtt{C}]_{\mathtt{S}}\, \mathtt{T}) + [\neg \mathtt{B}] \cdot \mathsf{infer}[\mathtt{D}]_{\mathtt{S}}\, \mathtt{T})\, \theta]\!]\, \sigma \\ &= [\![(\mathsf{infer}[\mathtt{if\ (B)\ \{C\}\ else\ \{D\}}]_{\mathtt{S}}\, \mathtt{T})\, \theta]\!]\, \sigma\end{aligned}$$

  This concludes the case.
- Case `while (B) {D}`. In this case, $(\mathsf{infer}[\mathtt{C}]_{\mathtt{S}}\, \mathtt{T})\, \theta = \mathtt{U}\, \theta$ for some term $\mathtt{U}$, satisfying (i) $\mathtt{B} \vdash \mathsf{infer}[\mathtt{C}]_{\mathtt{S}}\, \mathtt{T} \leq \mathtt{U}$ and (ii) $\neg \mathtt{B} \vdash \mathtt{T} \leq \mathtt{U}$. By induction hypothesis and monotonicity of $\mathsf{et}[\![\mathtt{D}]\!]_{\mathtt{S}}^{\eta}$, this says nothing more than that $[\![\mathtt{U}\, \theta]\!]$ is a loop-invariant for the while loop, wrt. $[\![\mathtt{T}\, \theta]\!]$ (see Figure 3).
- Case `C <> D`. In this case, $(\mathsf{infer}[\mathtt{C}]_{\mathtt{S}}\, \mathtt{T})\, \theta = \mathtt{U}\, \theta$ for some term $\mathtt{U}$, satisfying (i) $\vdash \mathsf{infer}[\mathtt{C}]_{\mathtt{S}}\, \mathtt{N} \leq \mathtt{U}$ and (ii) $\vdash \mathsf{infer}[\mathtt{D}]_{\mathtt{S}}\, \mathtt{N} \leq \mathtt{U}$. The case follows by of two applications of induction hypothesis.

□

ADDITIONAL BENCHMARKS

We detail in Figures 6 and 7 the benchmarks from this paper, whose evaluation results are given in Table 2 (h).

Received 2022-11-10; accepted 2023-03-31

Fig. 7. Additional benchmark examples (cont'd)

```
def f(n):
 var m
 if (n > 0) {
   m :≈ f(n - 1)
 };
 m :≈ m + Bernoulli(1/2);
 return m
```

(a) rec1 benchmark example.

```
def every(bins):
   var d, number, k
   k := 1
   while (k ≤ 5) {
      if (Bernoulli((5−k+1)/5) {k := k + 1}
      number := number + 1
   };
   return number
```

(b) every-while benchmark example.