A Modular Cost Analysis for Probabilistic Programs

Martin Avanzini, Georg Moser, Michael Schaper

A Modular Cost Analysis for Probabilistic Programs*

MARTIN AVANZINI, INRIA Sophia Antipolis, France
GEORG MOSER, University of Innsbruck, Austria
MICHAEL SCHAPER, University of Innsbruck, Austria

We present a novel methodology for the automated resource analysis of non-deterministic, probabilistic imperative programs, which gives rise to a *modular approach*. Program fragments are analysed in full independence. Moreover, the established results allow us to incorporate sampling from *dynamic distributions*, making our analysis applicable to a wider class of examples, for example the *Coupon Collector's problem*. We have implemented our contributions in the tool `eco-imp`, exploiting a constraint-solver over iterative refineable cost functions facilitated by off-the-shelf SMT solvers. We provide ample experimental evidence of the prototype's algorithmic power. Our experiments show that our tool runs typically at least one *order of magnitude faster* than comparable tools. On more involved examples, it may even be the case that execution times of seconds become milliseconds. At the same time we retain the precision of existing tools. The extensions in applicability and the greater efficiency of our prototype, yield scalability of sorts. This effects into a wider class of examples, whose expected cost analysis can be thus be performed fully automatically.

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

Additional Key Words and Phrases: probabilistic programs, average complexity, automation, modularity

ACM Reference Format:
Martin Avanzini, Georg Moser, and Michael Schaper. 2020. A Modular Cost Analysis for Probabilistic Programs. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 172 (November 2020), 30 pages. https://doi.org/10.1145/3428240

1 INTRODUCTION

Resource analysis is a subfield of static analysis, studying a non-functional property of programs, namely the use of *resources* (see [Cohen and Zuckerman 1974; Wegbreit 1975, 1976] for early references). Resource analysis impacts on the *correctness* or *safety* of programs. Programs that over-exceed available computing resources are most likely to fail, and hence cannot run correctly. See eg. Albert et al. [2019] for an application of resource analysis on the safety of *smart contracts*.

In the last decades there has been significant progress in the area of *fully automated* resource analysis, where no user interaction is required. This resulted in significant success stories showing that resource analysis can be practicable and scalable, cf. [Frohn and Giesl 2017; Gulwani et al. 2009; Hoffmann et al. 2017; Wilhelm et al. 2008; Wilhelm and Grund 2014]. *Modularity* of the analysis turned out as a key ingredient to the scalability of automated resource analysis, as modularity allows for code fragments to be analysed in full independence, so that, whole-program analyses can be overcome. This may significantly speeds up the analysis without affecting the precision of

*This work is partially supported by the French ANR: "Agence National de Recherche" under Grant "PPS: Probabilistic Program Semantics", No. ANR-19-CE48-0014, and the Inria associated team TC(Pro)^3.

Authors' addresses: Martin Avanzini, martin.avanzini@inria.fr, INRIA Sophia Antipolis, France; Georg Moser, georg.moser@uibk.ac.at, University of Innsbruck, Austria; Michael Schaper, michael.schaper@student.uibk.ac.at, University of Innsbruck, Austria.

2475-1421/2020/11-ART172
https://doi.org/10.1145/3428240

the analysis. See for example [Avanzini et al. 2016; Brockschmidt et al. 2016; Frohn and Giesl 2017; Gulwani and Zuleger 2010; Moser and Schaper 2018; Sinn et al. 2016, 2017] for references to the literature on fully automated resource analysis methods of imperative programs.

Apart from modularity, the study of *non-deterministic* programs has proven immensely useful. Non-determinism appears naturally through *program abstraction*. Program abstraction allows to focus on those program behaviours essential for resource analysis, while ignoring other aspects. For example, in a conditional statement typically the resource usage of both branches needs to be analysed, while the condition's guard is not essential. Thus, the conditional can be abstracted by allowing for non-deterministic choice. In particular, in the analysis of imperative programs, program abstractions form an integral part. They provide the stepping stone for the modularity of static program analysis itself, for example through modular combinations of abstract interpretations, cf. [Cousot and Cousot 2002; Dillig 2011; Gulwani and Tiwari 2006]. For instance, it is due to program abstractions that sophisticated, fully automated analyses of complex, pointer-based programs have become possible. (See Fiedor et al. [2018] also for further pointers to the literature.)

Interrelations between computer science and *probability theory* are well-studied. Indeed, probabilistic variations on well-known models like automata, Turing machines or the λ-calculus have been studied since the early days of theoretical computer science (see Kozen [1981] for an early reference). Generalising the model of computation further and allowing for *probabilistic, non-deterministic* programs, induces new challenges to an automated resource analysis.

First, *non-deterministic* probabilistic programs have not received much attention in the literature. Predominately, the semantics of languages with non-deterministic choice are modelled in terms of *Markov Decision Processes* so that schedulers resolve non-deterministic choices based on the current history of states. These constructions are technical and heavy handed; in our opinion a new foundation is essential. Second, automation, in particular automation of the push-button variety is not yet firmly established. Only few prototypes exist and the literature is lacking clear experimental comparisons. Further, intuitive textbook examples, like the *Coupon Collector's problem* (see Figure 1c) cannot be represented properly in existing tools. This is due to the lack of support for *dynamic distributions*, ie. for sampling from distributions whose support or value depends on the current environment. Third, and most crucially, *modularity* of the analysis is not provided for. Instead, the automated analysis in all existing prototypes degenerates to a whole-program analysis. Indeed, in a probabilistic setting this is not too straight forward, as running a command results in a distribution of possible end states. The established results in this paper overcome all these issues.

We are concerned with an automated *average runtime analysis* of a prototypical *imperative language* pWhile in the spirit of Dijkstra's *Guarded Command Language*. This language is endowed with primitives for *sampling* and *non-deterministic choice* so that *randomised algorithms* can be expressed. Further, a dedicated command `consume`(e) allows for the representation of *unbounded non-negative* costs given by the expression e. Non-negativity is required so as to ensure that the expected cost of a program is well-defined. Thus our automated analysis provides an *expected cost analysis*. By instrumenting the program so that every program statement is attributed cost one, this analysis can be used to assess the mean-time to termination and, as such, assesses that a program is *(positive) almost-surely terminating* Bournez and Garnier [2005].

Contributions. We present a novel *modular* methodology for the *automated* resource analysis of non-deterministic, probabilistic programs. Precisely,

- we present a novel *structural operational semantics* in terms of *probabilistic abstract reduction systems* originally due to Bournez and Garnier—significantly simplifying the formal development; following Avanzini and Yamada [2020], we refine this model by attributing costs to rules, so as neatly express the expected cost of computations.

```
while (p > min ≥ 0) {
  b ≔ Bernoulli(1/4);
  if (b) {p ≔ p + 1} {p ≔ p - 1};
  n ≔ Uniform(0,10);
  while (n > 0) {
    consume(p); n ≔ n - 1 }}
```

(a) Trader C_{trader}.

```
b ≔ 1;
x ≔ 1;
while (b = 1) {
  consume(1);
  x ≔ x * 2;
  b ≔ Bernoulli(1/2) }
```

(b) Geometric C_{geo}.

```
coupons ≔ 0;
while (0 ≤ coupons < n) {
  consume(1);
  draw ≔ Uniform(1,n);
  if (draw > coupons) {
    coupons ≔ coupons + 1 }}
```

(c) Coupon Collector C_{coupons}.

Fig. 1. Motivating Examples.

- we generalise the *expected runtime transformer* of Kaminski et al. [2016] to a cost transformer and formally prove it sound wrt. our novel operational semantics;
- we establish a novel alternating *expected cost* and *expected value* analysis, which gives rise to the first *fully modular resource analysis* of non-deterministic, probabilistic programs;
- we demonstrate how sampling from *dynamic distributions* can be incorporated, making the analysis applicable to a wider class of programs; specifically, we allow for uniform sampling from an interval dependent on program states;
- we have implemented our method in the tool `eco-imp` and show *efficiency* and *scalability* of this automation by comparison with existing prototypes in the literature; while `eco-imp` parallels other tools in precision, its analysis times are significantly faster;
- finally, we detail our resource analysis algorithm underlying our prototype `eco-imp`, together with an in-depth discussion on the employed constraint solving mechanism.

Our prototype facilitates off-the-shelf SMT solvers for the iteratively refineable synthesis of cost expressions. In benchmarking, we focus on complex and novel scenarios, thus emphasising the algorithmic power and scalability of our approach. We validate its effectiveness on a set of 66 challenging probabilistic programs taken from the literature. In particular, for programs with non-linear bounds and nested loop structure our analysis is upto *three orders of magnitude faster* than existing prototypes. At the same time we retain the precision.

Outline. We start with motivating our quest for a *modular* framework in Section 2, where we also provide a high-level introduction to automated expected cost analysis. Our novel, modular approach is outlined in Section 3. Probabilistic abstract reduction systems, which form the theoretical underpinning, and our imperative language are presented in Section 4 and 5, respectively. Section 6 details a weakest precondition calculus due to Kaminski et al. [2018], which is generalised in Section 7 to serve our needs of a modular analysis. In Section 8, the actual implementation is described and ample experimental evidence is given. In this section, we also explain our dedicated constraint solver and highlight our methodology on a handful of interesting examples. Section 9 discusses related work and we conclude in Section 10.

2 AUTOMATED EXPECTED COST ANALYSIS

We motivate the central contribution of our work: an *automated* and *modular* expected cost analysis of non-deterministic, probabilistic programs. First, we present a classic analysis of an algorithm incorporating a one-dimensional random walk, cf. Figure 1a. Second, we highlight the need for a more refined analysis technique, when it comes to compositionality of the analysis. Third, we detail the challenges of automated verification techniques. Finally, we motivate the need for *modularity* of the analysis in this context. Our contributions to this respect are highlighted in Section 3 below.

Recurrence equations. A classical way to analyse the runtime of a program is to set up a set of recurrence equations, whose closed-form provides the sought runtime. Wegbreit [1975, 1976]

was the first to automate this approach for deterministic programs. This idea generalises straight forward to probabilistic programs.

Consider the program $\mathtt{C}_{\mathsf{trader}}$ due to Ngo et al. [2018] in Figure 1a, which illustrates the behaviour of a stock trader. While the stock price (p) is above the minimum (min), the trader decides to buy shares. The stock price is governed by a *one-dimensional random walk*. With probability $1/4$ the price increases (by one), and with probability $3/4$ the price decreases. To this end, an integer b is sampled from the Bernoulli distribution with parameter $1/4$, that is, the distribution assigns 1 to b with probability $1/4$, and 0 with probability $3/4$. After the change of the stock price takes effect, the trader decides to buy upto 10 shares, all with equal probability. In the program, this is modelled by sampling the number of shares n from a uniform distribution of values from the interval $[0, 10]$. The cost for the trader is given as the total amount invested. This is indicated by the command $\mathtt{consume}(p)$, which signals that p units of resources are required.

Clearly, the total cost of bought shares is unbounded in some scenarios, namely, when the price p stays above the minimum price. However, the combined probability of all such cases is 0. Worst case bounds are thus not informative in this context. A more informative measure, and the one we are interested in, is given by the *expected cost*. Ie. the average cost emitted on all the computational branches, weighted by their probability. Simplifying the analysis by assuming $min = 0$ for now, the expected cost of $\mathtt{C}_{\mathsf{trader}}$ is expressed by the recurrence

$$\begin{aligned} T(p) &= 1/4 \cdot \textstyle\sum_{n=0}^{10} 1/11 \cdot \big(n \cdot (p+1) + T(p+1)\big) + 3/4 \cdot \sum_{n=0}^{10} 1/11 \cdot \big(n \cdot (p-1) + T(p-1)\big) \\ &= 5 \cdot p - 5/2 + 1/4 T(p+1) + 3/4 T(p-1) \ . \end{aligned}$$

Here, the two sums, weighted by probabilities $1/4$ and $3/4$, respectively, give the expected cost of an increasing and decreasing stock price. The terms $n \cdot (p+1)$ and $n \cdot (p-1)$, with n taking a value between 0 and 10 with probability $1/11$, account for the cost incurred by the inner loop. This recurrence T is a non-homogeneous linear second-order recurrence, whose closed-form can be computed as $5 \cdot p \cdot (p+1)$. While the general solution of the corresponding homogeneous recurrence can be derived directly (see Levitin [2007]), the manual computation of the closed-form of T requires some work. It is well-known that in general such a manual analysis—even for small and simple programs as in the case of $\mathtt{C}_{\mathsf{trader}}$—is tedious, error prone and fragile to small changes of the recurrence equations.

Compositionality of the Analysis. A central observation in the seminal work by Kaminski et al. [2018] is that an expected runtime analysis is inherently *non-compositional*, that is, from finite expected runtimes of program parts, we cannot conclude finite expected runtime of the whole program. A related issue has been encountered—and overcome—in the context of unbounded updates of non-deterministic (but non-probabilistic) programs, cf. [Ben-Amram 2011; Ben-Amram 2015; Ben-Amram and Hamilton 2019; Ben-Amram and Kristiansen 2012; Hirokawa and Moser 2008; Jones and Kristiansen 2009].

To illustrate, let us first consider the program $\mathtt{C}_{\mathsf{geo}}$ depicted in Figure 1b. In this example, the final value of x follows a geometric distribution, more precisely, the loop will exit after $i = 1, 2, \ldots$ iterations with probability $1/2^{i-1}$, in which case the variable x will hold the value 2^i. While also this example potentially diverges, its expected cost—the number of loop iterations—is given as:

$$1 + 1/2 + 1/4 + \cdots = \textstyle\sum_{i=1}^{\infty} 1/2^{i-1} = 2 \ .$$

Second, consider the composition of $\mathtt{C}_{\mathsf{geo}}$ with a program $\mathtt{D}$, whose expected cost is given by $f(x)$. Then the expected cost of $\mathtt{C}_{\mathsf{geo}}; \mathtt{D}$, that is, running the two programs in sequence, becomes

$$\textstyle\sum_{i=1}^{\infty} 1/2^{i-1} + \sum_{i=1}^{\infty} 1/2^{i-1} \cdot f(2^i) = \sum_{i=1}^{\infty} 1/2^{i-1} \cdot (1 + f(2^i)) \ . \tag{1}$$

When f grows at least linearly, this sum is infinite. Conclusively, while the expected costs of $\texttt{C}_{\texttt{geo}}$ and $\texttt{D}$ are finite, the expected cost of their composition is not. This is in contrast to non-probabilistic programs, where the sequential composition of two programs with bounded runtime yields again a program with bounded runtime.

Generalising a weakest precondition calculus à la Dijkstra's to an *expected runtime transformer* ert, Kaminski et al. overcome this issue through the expression of the expected runtime in continuation passing style, cf. [Kaminski and Katoen 2017; Kaminski et al. 2016, 2018; Olmedo et al. 2016]. As already observed by Kaminski et al.—and as we will see later in Section 6—this transformer in turn generalises seamlessly to an *expected cost transformer*

$$\mathrm{ect}[\texttt{C}]\colon (\Sigma \to \mathbb{R}^{\infty}_{\geq 0}) \to (\Sigma \to \mathbb{R}^{\infty}_{\geq 0})\ ,$$

for reasoning about expected costs. Informally, when $f\colon \Sigma \to \mathbb{R}^{\infty}_{\geq 0}$ measures the expected cost of a continuation in terms of a valuation $\sigma \in \Sigma$, $\mathrm{ect}[\texttt{C}](f)$ yields the overall cost of first executing $\texttt{C}$, followed by the continuation. The expected cost of a program $\texttt{C}$ is then given by $\mathrm{ecost}[\texttt{C}] \triangleq \mathrm{ect}[\texttt{C}](\lambda\sigma.0)$, ie. by attributing the continuation a cost of 0.

Formalising the expected cost in terms of a cost transformer facilitates a recursive definition of program costs. If e evaluates to a natural i under a given store, we set $\mathrm{ect}[\texttt{consume}(e)](f) \triangleq \lambda\sigma.i + f(\sigma)$, which accounts for the fact that $\texttt{consume}(e)$ incurs an additional cost of i. If d evaluates to a distribution assigning probabilities p_i to integers i under a given store σ, we let $\mathrm{ect}[x \coloneqq d](f) \triangleq \lambda\sigma.\ \sum_{i\in\mathbb{Z}} p_i \cdot f(\sigma[x/i])$. Ie. the expected cost is set as the mean value of f on stores σ with x updated by the sampled value i. Expected cost of two sequentially composed commands $\texttt{C};\texttt{D}$ becomes expressible via composition of the transformer: $\mathrm{ect}[\texttt{C};\texttt{D}] \triangleq \mathrm{ect}[\texttt{C}] \circ \mathrm{ect}[\texttt{D}]$. For straight line programs, expected costs can thus be derived simply by unfolding definitions. On the other hand, the expected cost of a loop $\texttt{while}\ (\phi)\ \{\texttt{C}\}$, wrt. the cost of a continuation f, is definable as

$$\phi \vDash \mathrm{ect}[\texttt{while}\ (\phi)\ \{\texttt{C}\}](f) = \mathrm{ect}[\texttt{C};\texttt{while}\ (\phi)\ \{\texttt{C}\}](f) \quad \neg\phi \vDash \mathrm{ect}[\texttt{while}\ (\phi)\ \{\texttt{C}\}](f) = f\ , \tag{2}$$

where $\phi \vDash f = g$ indicates that f coincides with g on inputs satisfying ϕ. Following standard semantics, the expected cost thus equals the cost of unrolling the loop once in case the guard ϕ holds, or the cost f of the continuation when the guard does not hold. To illustrate this, re-consider the program $\texttt{C}_{\texttt{geo}}$ from Figure 1b. Let us denote by $C(b,x)$ the expected cost of the while-loop wrt. a cost of the continuation f, measured in the program variable x. Then (2) translates to

$$b = 1 \vDash C(b,x) = 1 + 1/2 \cdot C(1, 2\cdot x) + 1/2 \cdot C(0, 2\cdot x) \qquad b \neq 1 \vDash C(b,x) = f(x)\ .$$

Since b and x are initialized to one, $C(1,1)$ then gives the cost (1) of $\texttt{C}_{\texttt{geo}}$. To derive closed forms for the expected cost, we can again resort to methods for solving such (conditional) recurrences, unlike above however, recurrences can be extracted in a systematic, compositional way.

To sum up, the ert-calculus and its expected cost derivative, are more refined technical tools for the analysis of probabilistic programs than the classical analysis through recurrence relations. The analysis becomes clean and is less fragile. Kaminski et al. have shown that the ert-calculus allows an optimal analysis of a variety of interesting case studies, like eg. the *Coupon Collector's Problem* formulated with the program $\texttt{C}_{\texttt{coupons}}$ from Figure 1c. However, Kaminski et al. [2018] do not provide an automation and also only superficially concepts automation. So the method remains tedious.

Automation. The crux of turning such a calculus into a fully automated analysis lies in deriving closed forms for expected costs of loops. Related problems have been extensively studied in the literature, eg. [Contejean et al. 2005; Fuhs et al. 2007; Podelski and Rybalchenko 2004]. One prominent approach lies in assigning templates to cost functions, by which the recurrences can be reduced to a set of constraints treatable with off-the-shelf SMT solvers.

The Absynth prototype provided by Ngo et al. [2018] has been the first *automated* expected resource analysis of probabilistic programs, implementing such an approach, thereby demonstrating the feasibility of an automated analysis. For instance, Absynth infers the manual bound on the expected cost of $\mathsf{C}_{\mathsf{trader}}$ shown above, fully automatically. Conceptually, this tool can be seen as a specialised automation of Kaminski's ert-calculus. The transformer is coached into a Hoare-style calculus and expected cost functions are formalised as *potential functions* [Ngo et al. 2018]. Automation is achieved by specialising these potential functions to linear combinations of *base functions*, the latter abstracting stores as non-negative numbers.

Recently, Wang et al. [2019] provides a resource analysis, based on martingale theory. Notably, their methodology is the first to also account for *negative costs*. The main challenge in accommodating negative costs lies in ensuring that the overall expected cost remains well-defined. Wang et al. ensures this by requiring that in the presence of negative costs, program updates are generally bounded and almost-sure termination is required.

Modularity of the Analysis. It is perhaps important to emphasise that composability of the *analysis* of the expected cost of sequential commands does not induce composability of the *synthesis* of the corresponding bounding functions. We illustrate the difference wrt. $\mathsf{C}_{\mathsf{trader}}$. Concretely, as $\mathsf{C}_{\mathsf{trader}}$ features a nested loop, its expected cost is driven by the following inter-dependent constraints. Below, $I(n, p, min)$ and $O(n, p, min)$ stands for the cost before entering the inner and outer loop, respectively.

$$
\begin{aligned}
p > min \geq 0 &\vDash O(n,p,min) = {}^1\!/_4 \cdot \textstyle\sum_{n=1}^{10} {}^1\!/_{11} \cdot I(n,p+1,min) + {}^3\!/_4 \cdot \sum_{n=1}^{10} {}^1\!/_{11} \cdot I(n,p-1,min) \\
\neg(p > min \geq 0) &\vDash O(n,p,min) = 0 \\
n \geq 0 &\vDash I(n,p,min) = p + I(n-1,p,min) \\
\neg(n \geq 0) &\vDash I(n,p,min) = O(n,p,min)\ .
\end{aligned}
$$

Here, through the first equation, the cost of the outer loop $O(n, p, min)$ depends on the cost of the inner loop. Vice verse, since the execution of the inner loop eventually gives back control, the cost of the inner loop $I(n, p, min)$ is also dependent on the outer loop. Therefore, the costs $I(n, p, min)$ and $O(n, p, m)$ cannot be considered in isolation. In general, synthesis degenerates to a *whole-program analysis*, ie. program parts cannot be treated in isolation.

Indeed, the aforementioned tool Absynth [Ngo et al. 2018] and the prototype described by Wang et al. perform such a whole-program analysis, which is not modular and results in issues with scalability even on small, handcrafted examples. To whit, consider a more realistic variant of $\mathsf{C}_{\mathsf{trader}}$, where the stock trader is choosing uniformly upto 100.000 shares. In this setting, Absynth is no longer able to compute a bound, while the prototype established by Wang et al. [2019] requires upto 20 seconds of runtime—additionally a significant amount of user guidance is required. Similar issues hold wrt. multiple nested loops (see Figure 8). In contrast, we provide a novel modular and efficient automation, inspired by the continuation-passing style analysis method formalised in the ert-calculus, which is free of the aforementioned scalability issues. In the following, we refer to the composability of the synthesis of upper invariants as *modularity* of the (expected) cost analysis. In the next section we provide a high-level overview of the crucially needed concepts to provide such a modular analysis.

3 A MODULAR EXPECTED COST ANALYSIS

Modularity requests that the cost analysis for a program can be broken into an independent cost analysis of program parts. This establishes a crucial stepping stone for the *scalability* of the analysis. For that, we suite the approach outlined in the preceding section to one that interleaves a *value analysis*.

It is well known that a modular cost analysis can be obtained by combining cost with an analysis on how values evolve within a program [Avanzini et al. 2016; Brockschmidt et al. 2016; Frohn and Giesl 2017; Gulwani and Zuleger 2010; Moser and Schaper 2018; Sinn et al. 2016, 2017]. Consider for instance a sequential command $\texttt{C};\texttt{D}$. To determine the cost of this command from the costs of $\texttt{C}$ and $\texttt{D}$, given as functions in their input states, it is necessary to determine how the program's state evolves to the point where the second command is executed. A *value analysis* provides such additional information. The case is similar for loops, that sequentially execute a given command several times.

Combining Cost with Value Analysis. Since it is clearly infeasible to reason how program values evolve in all—possibly uncountable many—execution paths, we focus on incorporating an *expected*, or *mean value analysis*. To this end, we start with an *expected cost transformer* as outlined above. As in previous approaches, we express program costs as linear combination of *base functions* b_i, mapping program valuations to (positive) real numbers. These base functions serve as a numerical abstraction of program stores. Base functions 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.

In the majority of cases, expected costs can be computed symbolically on such *cost expressions*. Particularly, we express the cost of a loop $\texttt{while}\ (\phi)\ \{\texttt{C}\}$ as a linear combination $\kappa(b_1,\dots,b_n)$ of base functions. Rather than directly subjecting this symbolic bound to recurrences (2) however, we seek for an upper-bound satisfying

$$\phi \vDash \kappa(b_1,\dots,b_n) \geq \mathrm{ecost}[\texttt{C}] + \kappa(\mathrm{evalue}[\texttt{C}](b_1),\dots,\mathrm{evalue}[\texttt{C}](b_n)) \quad \neg\phi \vDash \kappa(b_1,\dots,b_n) \geq 0 \tag{3}$$

Here, $\mathrm{ecost}[\texttt{C}]$ amounts to the cost of executing the loop's body $\texttt{C}$ once; $\mathrm{evalue}[\texttt{C}](b_i)$ gives the mean value of measures b_i after executing $\texttt{C}$. In other words, $\mathrm{evalue}[\texttt{C}](b_i)$ amounts to the *expected value of the base function* b_i on the distribution of final states obtained from executing the loop's body $\texttt{C}$. For instance,

$$\mathrm{evalue}[x \coloneqq x + 1;\ \ b \coloneqq \texttt{Bernoulli}(1/2)](\lambda\sigma.\sigma(b)\cdot\sigma(x)) = \lambda\sigma.1/2\cdot(\sigma(x)+1)\ .$$

The constraints (3) thus witnesses that the inferred cost $\kappa(b_1,\dots,b_n)$ of the loop $\texttt{while}\ (\phi)\ \{\texttt{C}\}$ dominates the cost of an iteration ($\mathrm{ecost}[\texttt{C}]$), plus the cost of re-iterating the loop, in terms of measures b_i after the execution ($\kappa(\mathrm{evalue}[\texttt{C}](b_1),\dots,\mathrm{evalue}[\texttt{C}](b_n))$).

In contrast to the constraints given by (2), in (3), the cost $\kappa(b_1,\dots,b_n)$ is not passed down along the transformer given by the body $\texttt{C}$. Indeed, $\mathrm{ecost}[\texttt{C}]$, as well as the expected values $\mathrm{evalue}[\texttt{C}](b_i)$ of base functions b_i can be pre-computed and substituted into the above upper bounds, and then offloaded to a constraint solver to determine a concrete bound, *in isolation*. To whit, reconsider the trader example $\texttt{C}_{\texttt{trader}}$ from Figure 1a. Rather than proceeding top-down to extract the recurrences in the preceding section, we proceed inside out and start with the analysis of the inner loop

```
while (n > 0) { consume(p); n := n - 1 }
```

While the body is iterated $n > 0$ times, the cost of the loop's body is $p \geq 0$. This suggests n and p play a role in the cost of this loop. Let us thus choose the template $\kappa(\langle n\rangle, \langle p\rangle, \langle n\rangle\cdot\langle p\rangle)$, a simple-mixed template [Contejean et al. 2005] over (non-negative) base-functions $\langle n\rangle = \max(n,0)$ and $\langle p\rangle = \max(p,0)$. Since n is decremented in the loop body while p remains unchanged, the (expected) value of the three given base functions is given by $\langle n-1\rangle$, $\langle p\rangle$ and $\langle n-1\rangle\cdot\langle p\rangle$, respectively. Based on these, the upper bounds (3) translates to

$$n > 0 \vDash \kappa(\langle n\rangle, \langle p\rangle, \langle n\rangle\cdot\langle p\rangle) \geq p + \kappa(\langle n-1\rangle, \langle p\rangle, \langle n-1\rangle\cdot\langle p\rangle) \quad n \leq 0 \vDash \kappa(\langle n\rangle, \langle p\rangle, \langle n\rangle\cdot\langle p\rangle) \geq 0\ .$$

The inequalities hold by taking $\kappa(x, y, z) \triangleq z$ as indeed, $n > 0$ entails $\langle n \rangle \cdot \langle p \rangle \geq p + \langle n-1 \rangle \cdot \langle p \rangle$, and $\langle n \rangle \cdot \langle p \rangle$ is always non-negative. The cost of the program is thus (tightly) upper-bounded by $\langle n \rangle \cdot \langle p \rangle$. One can now proceed with the outer loop, by substituting this bound in the recurrence (3) and derive the bound

$$10 \cdot \langle min + 1 \rangle \cdot \langle p - min \rangle + 5 \cdot \langle p - min \rangle^2 \,. \tag{4}$$

See Section 7.1 for further details.

Proving that (3) yields indeed an upper-bound $\kappa(b_1, \ldots, b_n)$ on the cost of loops is non-trivial in the presence of probabilistic sampling. Indeed, our main theorem witnessing the soundness of this approach, Theorem 7.5, relies essentially on the fact that κ is assumed a linear, more generally concave, function.

Computing Expected Values. A crucial insight is that the ect-calculus keeps track of expected values on cost functions f. As it turns out, this machinery can be turned into one for reasoning about expected values. Indeed, we have $\mathrm{ect}[\mathtt{C}](f) = \mathrm{ecost}[\mathtt{C}] + \mathrm{evalue}[\mathtt{C}](f)$ for probabilistic programs[1] C. By removing all cost emitting statements $\mathtt{consume}(e)$ from C, or alternatively, by setting $\mathrm{evalue}[\mathtt{consume}(n)](f) \triangleq f$, but otherwise defining the expected value along the way of the expected cost transformer, we obtain a recursive definition that can be inferred by exactly the same procedure we employ to reason about expected costs.

Automating our Approach. We have successfully automated this approach in our tool `eco-imp`, see Section 8. This tool proceeds with an analysis as just outlined, and integrates a dedicated constraint solver for solving constraints of the form (3). Our tool derives the bound (4) in 25ms. This constitutes a significant speedup to the `Absynth` prototype—which can be considered among the most efficient implementations of such an analysis to date—which requires more than 3 seconds, and the prototype of Wang et al. [2019] which requires 10 seconds to analyse this example. Our implementation significantly benefits from the fact that program fragments can be considered in isolation. For one, the recurrences handled are considerably simpler, as they describe a single loop, rather than the whole program's cost behaviour. But also heuristics, such as for constructing suitable templates from a given program can be made more precise, and simpler. Indeed, as it turned out, we do not even require the (brittle) use of invariant generation tools [Ngo et al. 2018] or manual annotations for this task [Wang et al. 2019]. Unsurprisingly, the approach is particularly effective in the analysis of nested loops, which has direct consequences for the scalability and speed of the analysis. Benefits are, of course, most pronounced when the program under consideration contains nested loops, see Section 8.2 where we present our experimental evaluation.

An Intuitive Textbook Example. We conclude this section with a non-trivial example that is beyond the scope of pre-existing tools. Following the formulation of Mitzenmacher and Upfal [2005], the above mentioned *Coupon Collector's problem* states as follows. Given a box with n different coupons, one is interested in the expected number of draws (with replacement) that are needed before having drawn each coupon at least once. The corresponding code $\mathtt{C}_{\mathsf{coupons}}$ is depicted in Figure 1c. Here, *coupons* represents the number of unique coupons collected; *draw* is sampled uniformly from the interval $[1, n]$, with n an input parameter. The chosen cost model reflects the number of trials. Note, that the probability of collecting a new coupon drops in proportion of the collected coupons, that is, in proportion $1/\mathit{coupons}+1$. In particular, the expected cost is finite.

The full pen-and-paper analysis of $\mathtt{C}_{\mathsf{coupons}}$ given by Kaminski et al. [2018]—which provides the optimal expected time bound $\mathsf{O}(n \log n)$—spans several pages and requires non-trivial estimations. Thus, non surprisingly, automation poses significant challenges. So far, to the best of our

[1] When C features also non-deterministic choice, as we do later on in this work, this equality turns into an inequality, see Lemma 7.2.

knowledge, the Coupon Collector's problem has been elusive to an automated analysis. As already mentioned, no existing tool can even *represent* the corresponding code $\mathtt{C}_{\mathtt{coupons}}$. To date support for *dynamic probabilistic branching*—when sampling draws a number of samples not statically bounded—is lacking. When sampling is in contrast static, expectations are finite, weighted sums, and consequently can be encoded by their unfolding. When sampling is dynamic though, such as within the program $\mathtt{C}_{\mathtt{coupons}}$ through the assignment `draw ≔ Uniform(1,n)`, such an unfolding is no longer possible. To illustrate this, observe

$$\mathrm{ect}[\mathtt{draw} \coloneqq \mathtt{Uniform}(1,n)](f) = \lambda\sigma.\, \textstyle\sum_{i=1}^{\sigma(n)} {}^{1}\!/_{\sigma(n)} \cdot f(\sigma[draw/i])\ .$$

To reason about such costs, we define the sum in terms of a constraint of the form (3), and make use of our dedicated constraint solver to find an upper bound. See Section 8 for the details.

Through this encoding, our prototype `eco-imp` is able to derive the (non-optimal) bound $n+{}^{1}\!/_{2}\cdot n^2$ fully automatically in *a fraction of a second* for the expected cost of $\mathtt{C}_{\mathtt{coupons}}$. If the distribution is set statically, eg. to a uniform distribution of 10 coupons, the example becomes expressible by existing tools through unrolling. Alas, only the `Absynth` tool can provide a (non-optimal) bound (see Section 8.2). Still the employed potential functions are not amenable to express the subtle dependency of the expected cost of $\mathtt{C}_{\mathtt{coupons}}$ on the (now) static distribution governing the draw. Ie. the generated constraint grows linearly to the number of coupons. Wrt. tool execution time this implies that our prototype `eco-imp` handles a uniform distribution of upto 100 coupons in seconds. On the other hand the `Absynth` tool and the prototype by [Wang et al. 2019] cannot provide bounds for larger numbers, even after several minutes of runtime, cf. Table 2 (d).

4 PROBABILISTIC REDUCTION SYSTEMS

Probabilistic abstract reduction systems are due to Bournez and Garnier [2005] and form a generalisation of *abstract reduction systems*, accounting for probabilistic choice. Avanzini and Yamada [2020] extend probabilistic abstract reduction systems with *weights*, allowing for the formulation of a suitable *cost model*. We refer to the weighted variant again as *probabilistic abstract reduction systems* (*PARSs*).

Let $\mathbb{R}^{\infty}_{\geq 0}$ denote the set of non-negative real numbers extended with ∞, ie. $\mathbb{R}^{\infty}_{\geq 0} \triangleq \mathbb{R}_{\geq 0} \cup \{\infty\}$. 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 subdistributions δ as $\{\!\{\delta(a) : a\}\!\}_{a\in A}$. The set of all subdistributions over A is denoted by $\mathcal{D}(A)$. We restrict to distributions over countable sets A. The *expectation of* a function $f\colon A \to \mathbb{R}^{\infty}_{\geq 0}$ wrt. a distribution δ is given by $\mathbb{E}_{\delta}(f) \triangleq \sum_{a\in A} \delta(a)\cdot f(a)$.

Probabilistic abstract reduction systems. A PARS over A is a set of *rules* $a \to \delta$ indicating that $a \in A$ reduces to $b \in A$ with probability $\delta(b)$ if $\delta(b) \neq 0$. Operationally, a reduction step on a involves first picking a rule $a \to \delta$ (among possibly many) and then sampling the reduct from δ. To allow for non-uniform costs, we endow each rule with a *weight* $w \in \mathbb{R}_{\geq 0}$, accounting for the *cost* of the corresponding reduction step, cf. [Avanzini and Yamada 2020]. Thus formally, a PARS on A is given by ternary relation $\cdot \dot{\to} \cdot \subseteq A \times \mathbb{R}_{\geq 0} \times \mathcal{D}(A)$, where in a rule $a \xrightarrow{w} \delta$, the weight w indicates the cost of the rule application. Objects $a \in A$ with no rule $a \xrightarrow{w} \delta$ are called *terminal*, in notation $a{\downarrow}$. PARSs constitute a universal probabilistic model of computation well-suited to model small-step operational semantics of our probabilistic language. For instance, the program $\mathtt{C}_{\mathtt{geo}}$ from Figure 1b is modelled by the PARS $\xrightarrow[\mathtt{geo}]{}$, defined by

$$\mathtt{geo}(x) \xrightarrow[\mathtt{geo}]{1} \{\!\{{}^{1}\!/_{2} : 2\cdot x, {}^{1}\!/_{2} : \mathtt{geo}(2\cdot x)\}\!\} \quad \text{for all } x \in \mathbb{Z}.$$

Following [Avanzini et al. 2020; Avanzini and Yamada 2020], we define the dynamics of a PARS $\to$ in terms of a *(weight-indexed) reduction relation* $\cdot \dot{\twoheadrightarrow} \cdot \subseteq \mathcal{M}(A) \times \mathbb{R}_{\geq 0} \times \mathcal{M}(A)$ over *multidistributions*

$\mathcal{M}(A)$. These are countable *multisets* $\{\!\{p_i : a_i\}\!\}_{i\in I}$ over pairs $p_i : a_i$ of *probabilities* $0 < p_i \leq 1$ and *objects* $a_i \in A$ with $\sum_{i\in I} p_i \leq 1$. Multidistributions are denoted by $\mu, \nu, \ldots$. The notion of *expectation* of a function $f\colon A \to \mathbb{R}^\infty_{\geq 0}$ is extended to multidistributions μ in the natural way: $\mathbb{E}_\mu(f) \triangleq \sum_{(p:a)\in\mu} p \cdot f(a)$. Finally, the reduction relation is inductively find by

$$\frac{}{\mu \xrightarrow{0}\!\!\!\!\rightarrow \mu} \qquad \frac{a \xrightarrow{w} \delta}{\{\!\{1 : a\}\!\} \xrightarrow{w}\!\!\!\!\rightarrow \delta} \qquad \frac{\mu_i \xrightarrow{w_i}\!\!\!\!\rightarrow \nu_i}{\biguplus_{i\in I} p_i \cdot \mu_i \xrightarrow{w}\!\!\!\!\rightarrow \biguplus_{i\in I} p_i \cdot \nu_i} \ .$$

In the second rule, distributions are lifted to multidistributions in the obvious way. In the last rule, $w = \sum_{i\in I} p_i \cdot w_i$ and $p_i > 0$ are probabilities with $\sum_{i\in I} p_i \leq 1$. Scalar multiplication is performed component-wise on probabilities, $\uplus$ refers to the usual notion of multiset union. Ie. $\mu \xrightarrow{w}\!\!\!\!\rightarrow \nu$ indicates that some elements a_i with associated probability p_i are replaced by δ_i, re-weighted with probability p_i, according to rules $a_i \xrightarrow{w_i} \delta_i$. The overall cost w is given by the sum $\sum p_i \cdot w_i$, corresponding to the expected cost of the single reduction step $\mu \xrightarrow{w}\!\!\!\!\rightarrow \nu$. Eg. the PARS $\xrightarrow[\mathsf{geo}]{}$ gives rise to the reduction

$$\{\!\{1 : \mathsf{geo}(1)\}\!\} \xrightarrow[\mathsf{geo}]{1}\!\!\!\!\rightarrow \{\!\{1/2 : 2, 1/2 : \mathsf{geo}(2)\}\!\} \xrightarrow[\mathsf{geo}]{1/2}\!\!\!\!\rightarrow \{\!\{1/2 : 2, 1/4 : 4, 1/4 : \mathsf{geo}(4)\}\!\} \xrightarrow[\mathsf{geo}]{1/4}\!\!\!\!\rightarrow \cdots \ .$$

We write $\mu \xrightarrow{w}\!\!\!\!\rightarrow^n \nu$ if $\mu \xrightarrow{w_1}\!\!\!\!\rightarrow \cdots \xrightarrow{w_n}\!\!\!\!\rightarrow \nu$ for $w = \sum_{i=1}^n w_i$.

Expected cost. A reduction from $a \in A$ is a, generally infinite, sequence $\Delta : \{\!\{1 : a\}\!\} \xrightarrow{w_0}\!\!\!\!\rightarrow \mu_1 \xrightarrow{w_1}\!\!\!\!\rightarrow \mu_2 \xrightarrow{w_2}\!\!\!\!\rightarrow \cdots$. The infinite sum $w = \sum_{i\in\mathbb{N}} w_i$ gives the *expected cost of this specific reduction*. Since the weight w_i are non-negative, this sum is always well-defined (although it may be infinite). Throughout the following, the set of *costs* $\mathbb{C}$ is given by $\mathbb{R}^\infty_{\geq 0}$. Due to the presence of non-determinism, expected costs are in general not unique. Taking a demonic view on non-determinism, we define the *expected cost function*, denoted as $\mathrm{ecost}[\to]\colon A \to \mathbb{C}$, as the function that associates each $a \in A$ with the maximal expected cost of reductions starting from $\{\!\{1 : a\}\!\}$. Concisely, via the monotone convergence theorem of real numbers this can be defined as

$$\mathrm{ecost}[\to](a) \triangleq \sup\{w \mid \{\!\{1 : a\}\!\} \xrightarrow{w}\!\!\!\!\rightarrow^n \mu\} \ .$$

For instance, we have $\mathrm{ecost}[\xrightarrow[\mathsf{geo}]{}](\mathsf{geo}(x)) = \sup\{\sum_{i=0}^n 1/2^i \mid n \in \mathbb{N}\} = \sum_{i=0}^\infty 1/2^i = 2$.

Remark. In the literature, the operational semantics of *purely probabilistic programs*, are usually modelled as Markov chains over program states. Here, the i-th random variable in this chain gives the probability of being in a state after i reduction steps. On the other hand, the operational semantics of languages with non-deterministic choice are modelled in terms of *Markov Decision Processes* (*MDPs*) (see, eg. [Agrawal et al. 2018; Chakarov and Sankaranarayanan 2013; Kaminski et al. 2018; Olmedo et al. 2016]). Schedulers (aka policies) resolve non-deterministic choices based on the current history of states, thereby breaking down reductions again to Markov chains. Such MDPs M are naturally modelled as PARSs $\to_M$, so that the Markov chains induced by the schedulers are in one-to-one correspondence with reductions wrt. $\to_M$. The distribution underlying the i-th random variable is represented by the i-th multidistribution in the corresponding reduction. The use of multidistributions, rather than distributions, eliminates the need for schedulers, intuitively, because multidistributions within a reduction implicitly encode histories, cf. Avanzini et al. [2020]. Particularly, Avanzini et al. [2020] proves the following correspondence.

Proposition 4.1 ([Avanzini et al. 2020; Avanzini and Yamada 2020]). *Suppose each rule in the PARS $\to$ is attributed weight one. Then* $\mathrm{ecost}[\to]$ *coincides with the expected time to termination under the standard MDP semantics of Bournez and Garnier [2005].*

While the proposition assumes a unitary cost model, where each reduction step is attributed cost one, the proposition generalises to arbitrary (non-negative) costs as employed in the definition of $\mathrm{ecost}[\to]$.

4.1 Expected Cost Transformers for PARSs

In this subsection, we define an *expected cost transformer* $\mathrm{ect}[\to]$ for arbitrary PARSs $\to$. This transformer, generalising the expected cost function, serves as a technical tool to prove soundness and completeness of our methods.

To this end, we quickly recap notions from program semantics, cf. Winskel [1993]. A poset $(A, \sqsubseteq)$ is called an *ω-CPO*, if every *ω-chain* $a_0 \sqsubseteq a_1 \sqsubseteq \cdots$ has a supremum $\sup\{a_n \mid n \in \mathbb{N}\} \in A$. A function $f\colon A \to B$ between two ω-CPOs is called *(Scott)-continuous* if $f(\sup_{n\in\mathbb{N}} a_n) = \sup_{n\in\mathbb{N}} f(a_n)$ holds for all ω-chains $(a_n)_{n\in\mathbb{N}}$. Recall that a continuous function is monotone, wrt. the underlying orders of the poset. Kleene's Fixed-Point Theorem asserts that, if $f\colon A \to A$ is continuous and A features a least element $\bot$, the *least fixed-point* $\mathsf{lfp}(f)$ of f is given $\mathsf{lfp}(f) \triangleq \sup_{n\in\mathbb{N}} f^n(\bot)$ for f^n the n-fold composition of f. Let $\mathbb{C}^A \triangleq \{f \mid f\colon A \to \mathbb{C}\}$ denote the set of *cost functions* over A. We endow this set with the order $\leq$ defined by $f \leq g$ if $f(a) \leq g(a)$ for all $a \in A$. We also extend functions over $\mathbb{C}$ point-wise to cost functions and denote these extensions in **bold face** font, as we already did above, eg., $f + g \triangleq \lambda a.f(a) + g(a)$ for $f, g \in \mathbb{C}^A$ etc. In particular, $0 = \lambda a.0$ and $\infty = \lambda a.\infty$. The proof of the following is standard.

Proposition 4.2 (Cost functions form an ω-CPO). *For any A, $(\mathbb{C}^A, \leq)$ is an ω-CPO, with least and greatest element 0 and ∞, respectively. The supremum of ω-chains $(f_n)_{n\in\mathbb{N}}$ is given point-wise: $\sup_{n\in\mathbb{N}} f_n \triangleq \lambda a.\sup_{n\in\mathbb{N}} f_n(a)$.*

The following definition introduces the expected cost transformer of a PARS $\to$. Informally, this transformer associates each $a \in A$ with its expected cost, plus the expected value of a given cost function f on terminal objects.

Definition 4.3 (Expected Cost Transformer for PARSs). The *expected cost transformer* $\mathrm{ect}[\to]\colon \mathbb{C}^A \to \mathbb{C}^A$ *for a PARS* $\to$ is given by $\mathrm{ect}[\to](f) \triangleq \mathsf{lfp}(\chi_f)$, where the functional $\chi_f\colon \mathbb{C}^A \to \mathbb{C}^A$ is defined as

$$\chi_f(g) \triangleq \lambda a. \begin{cases} f(a) & \text{if } a\!\downarrow, \\ \sup\{w + \mathbb{E}_\delta(g) \mid a \xrightarrow{w} \delta\} & \text{else.} \end{cases}$$

It can be shown that for any f, $\chi_f(g)$ is a continuous functional. Hence $\mathrm{ect}[\to](f)$ is well-defined. Moreover, it is *continuous* and hence *monotone*.

Lemma 4.4 (Central Properties of $\mathrm{ect}[\to]$).

(1) continuity*:* $\mathrm{ect}[\to](\sup_{n\in\mathbb{N}} f_n) = \sup_{n\in\mathbb{N}} \mathrm{ect}[\to](f_n)$ *for all ω-chains $(f_n)_{n\in\mathbb{N}}$;*
(2) monotonicity*:* $f \leq g \implies \mathrm{ect}[\to](f) \leq \mathrm{ect}[\to](g)$.

A standard induction reveals that $\chi_f^n(0) = \lambda a.\sup\{w \mid a \xrightarrow{w}\!\!\!\!\to^n \mu\}$, where χ_f^n denotes the n-fold composition of χ_f. Consequently, the next result follows by Kleene's Fixed-Point Theorem.

Theorem 4.5 (Expected Cost via Cost Transformer).

$$\mathrm{ecost}[\to] = \mathrm{ect}[\to](0)\ .$$

5 A PROBABILISTIC LANGUAGE

We consider an *imperative language* pWhile in the spirit of Dijkstra's *Guarded Command Language*, endowed with a non-deterministic choice operator <> and where the assignment statement is generalised to one that can sample from a distribution. A command `consume`(e) signals the consumption of $e \geq 0$ resource units.

We fix a finite set of integer-valued *variables* Var. *Stores* are denoted by $\sigma \in \Sigma \triangleq \mathsf{Var} \to \mathbb{Z}$. With $\phi \in \mathsf{BExp} \triangleq \Sigma \to \mathbb{B}$, $e \in \mathsf{Exp} \triangleq \Sigma \to \mathbb{Z}$, and $d \in \mathsf{DExp} \triangleq \Sigma \to \mathcal{D}(\mathbb{Z})$ we denote, *Boolean*, *Integer*,

$$\frac{}{\sigma \triangleright x \coloneqq d \xrightarrow{0} \{\!\{ p_i : \sigma[x/i] \mid p_i = d(\sigma)(i) > 0 \}\!\}_{i\in\mathbb{Z}}}\ [\textsc{Assign}]$$

$$\frac{}{\sigma \triangleright \texttt{skip} \xrightarrow{0} \sigma}\ [\textsc{Skip}] \qquad \frac{}{\sigma \triangleright \texttt{abort} \xrightarrow{0} \varnothing}\ [\textsc{Abort}] \qquad \frac{}{\sigma \triangleright \texttt{consume}(e) \xrightarrow{\langle e(\sigma)\rangle} \sigma}\ [\textsc{Consume}]$$

$$\frac{\sigma \vDash \phi}{\sigma \triangleright \texttt{if } (\phi)\ \{\texttt{C}\}\ \{\texttt{D}\} \xrightarrow{0} \sigma \triangleright \texttt{C}}\ [\textsc{IfT}] \qquad \frac{\sigma \nvDash \phi}{\sigma \triangleright \texttt{if } (\phi)\ \{\texttt{C}\}\ \{\texttt{D}\} \xrightarrow{0} \sigma \triangleright \texttt{D}}\ [\textsc{IfF}]$$

$$\frac{\sigma \vDash \phi}{\sigma \triangleright \texttt{while } (\phi)\ \{\texttt{C}\} \xrightarrow{0} \sigma \triangleright \texttt{C}; \texttt{while } (\phi)\ \{\texttt{C}\}}\ [\textsc{WhileT}] \qquad \frac{\sigma \nvDash \phi}{\sigma \triangleright \texttt{while } (\phi)\ \{\texttt{C}\} \xrightarrow{0} \sigma}\ [\textsc{WhileF}]$$

$$\frac{i \in \{1,2\}}{\sigma \triangleright \{\texttt{C}_1\} <> \{\texttt{C}_2\} \xrightarrow{0} \sigma \triangleright \texttt{C}_i}\ [\textsc{Choice}] \qquad \frac{\sigma \triangleright \texttt{C} \xrightarrow{r} \{\!\{ p_i : \sigma_i \triangleright \texttt{C}_i \}\!\}_{i\in I} \uplus \{\!\{ q_j : \sigma_j \}\!\}_{j\in J}}{\sigma \triangleright \texttt{C};\texttt{D} \xrightarrow{r} \{\!\{ p_i : \sigma_i \triangleright \texttt{C}_i;\texttt{D} \}\!\}_{i\in I} \uplus \{\!\{ q_j : \sigma_j \triangleright \texttt{D} \}\!\}_{j\in J}}\ [\textsc{Compose}]$$

Fig. 2. Small-step operational semantics as a PARS.

and *Integer-valued distribution expression* over Var, respectively. The syntax of *program commands* Cmd is given as follows.

$$\texttt{C},\texttt{D} ::= x \coloneqq d \mid \texttt{skip} \mid \texttt{abort} \mid \texttt{consume}(e) \mid \texttt{C};\texttt{D} \mid \texttt{if } (\phi)\ \{\texttt{C}\}\ \{\texttt{D}\} \mid \texttt{while } (\phi)\ \{\texttt{C}\} \mid \{\texttt{C}\} <> \{\texttt{D}\}\ .$$

Commands are fairly standard. The assignment statement $x \coloneqq d$ samples a value from d, ie. an expression that evaluates to a distribution over integers. This command generalises the usual non-probabilistic assignment $x \coloneqq e$. The command $\texttt{consume}(e)$ consumes e resource units but acts as a no-op otherwise. Here e is a non-negative but otherwise arbitrary integer-valued expression. Particularly, the incurred cost can depend on the programs state rather than being constant. The non-deterministic choice operator $\{\texttt{C}\}$ <> $\{\texttt{D}\}$ executes either C or D. For brevity, we omit a probabilistic choice command and probabilistic guards as by Kaminski et al. [2016], since they do not add to the expressiveness of our language.

Semantics. We model reduction semantics of our language as a PARS over *configurations* $\mathsf{Conf} \triangleq (\mathsf{Cmd} \times \Sigma) \cup \Sigma$. Elements $(\texttt{C}, \sigma) \in \mathsf{Conf}$ are denoted by $\sigma \triangleright \texttt{C}$ and signal that the command C is to be executed under the current store σ, whereas $\sigma \in \mathsf{Conf}$ indicates that the computation has halted with final store σ. The (infinite) PARS is depicted in Figure 2. Rules $\sigma \triangleright \texttt{C} \xrightarrow{w} \{\!\{1 : \gamma\}\!\}$ without probabilistic effect are written as $\sigma \triangleright \texttt{C} \xrightarrow{w} \gamma$ for brevity. For $\phi \in \mathsf{BExp}$ and $\sigma \in \Sigma$ we denote by $\sigma \vDash \phi$ that ϕ evaluates on σ to true. Note that only in Rule (Consume) resources are consumed. Here, $\langle z \rangle \triangleq \max(0, z)$ denotes *Macaulay's bracket*. As we had for PARSs, incurred costs are thus always non-negative.

6 EXPECTED COST AND EXPECTED VALUE TRANSFORMERS

We now suite the ert-transformer of Kaminski et al. [2016] to an *expected cost transformer*. To this end, for $\phi \in \mathsf{BExp}$, we lift *Iverson brackets* $[\cdot]$ to stores, resulting in the cost function $[\phi](\sigma) \triangleq 1$ if $\sigma \vDash \phi$, and $[\phi](\sigma) \triangleq 0$ otherwise. In particular, $[\phi] \cdot f + [\neg\phi] \cdot g$ evaluates to $f(\sigma)$ on stores $\sigma \vDash \phi$, and to $g(\sigma)$ if $\sigma \vDash \neg\phi$. We denote by $f[x/v]$ the cost function that applies f on the modified store where x takes value v.

Our *expected cost transformer* $\mathrm{ect}[\cdot](\cdot) \colon \mathsf{Cmd} \to \mathbb{C}^\Sigma \to \mathbb{C}^\Sigma$ operates on cost functions over stores, that is, elements of $\mathbb{C}^\Sigma$. In the literature $f \in \mathbb{C}^\Sigma$ are also referred to as *expectations* [Kaminski et al. 2016]. As already indicated in Section 2, the cost $\mathrm{ect}[\texttt{C}](f)$ should be seen as the cost of evaluating C wrt. to a continuation of expected cost f.

C	$\mathrm{ect}[\mathtt{C}](f)$	$\mathrm{evalue}[\mathtt{C}](f)$
$\mathtt{consume}(e)$	$\langle e\rangle + f$	f
$\mathtt{skip}$	f	f
$\mathtt{abort}$	0	0
$x \coloneqq d$	$\lambda\sigma.\mathbb{E}_{d(\sigma)}(\lambda v.f[x/v](\sigma))$	$\lambda\sigma.\mathbb{E}_{d(\sigma)}(\lambda v.f[x/v](\sigma))$
$\mathtt{C};\mathtt{D}$	$\mathrm{ect}[\mathtt{C}](\mathrm{ect}[\mathtt{D}](f))$	$\mathrm{evalue}[\mathtt{C}](\mathrm{evalue}[\mathtt{D}](f))$
$\mathtt{if}\ (\phi)\ \{\mathtt{C}\}\ \{\mathtt{D}\}$	$[\phi]\cdot\mathrm{ect}[\mathtt{C}](f) + [\neg\phi]\cdot\mathrm{ect}[\mathtt{D}](f)$	$[\phi]\cdot\mathrm{evalue}[\mathtt{C}](f) + [\neg\phi]\cdot\mathrm{evalue}[\mathtt{D}](f)$
$\mathtt{while}\ (\phi)\ \{\mathtt{C}\}$	$\mathsf{lfp}(\lambda F.[\phi]\cdot\mathrm{ect}[\mathtt{C}](F) + [\neg\phi]\cdot f)$	$\mathsf{lfp}(\lambda F.[\phi]\cdot\mathrm{evalue}[\mathtt{C}](F) + [\neg\phi]\cdot f)$
$\{\mathtt{C}\}\ \diamond\ \{\mathtt{D}\}$	$\max(\mathrm{ect}[\mathtt{C}](f), \mathrm{ect}[\mathtt{D}](f))$	$\max(\mathrm{evalue}[\mathtt{C}](f), \mathrm{evalue}[\mathtt{D}](f))$

Fig. 3. Definition of expected cost transformer $\mathrm{ect}[\mathtt{C}]$ and expected value function $\mathrm{evalue}[\mathtt{C}]$. Notice that their definition coincides up to the case $\mathtt{C} = \mathtt{consume}(e)$.

Definition 6.1 (Expected Cost Transformer). The *expected cost transformer* $\mathrm{ect}[\cdot](\cdot)\colon \mathsf{Cmd} \to \mathbb{C}^{\Sigma} \to \mathbb{C}^{\Sigma}$ for *commands* $\mathtt{C}$ is defined through the rules given in the second column in Figure 3. For $\mathtt{C} \in \mathsf{Cmd}$, we set $\mathrm{ecost}[\mathtt{C}] \triangleq \mathrm{ect}[\mathtt{C}](0)$ and call $\mathrm{ecost}[\mathtt{C}]$ the expected cost of $\mathtt{C}$.

Remark. The earlier defined expected cost transformer $\mathrm{ect}[\to]$ *wrt. PARS* $\to$ (see Section 4) abstracts the above definition of the expected cost transformer *wrt. programs* $\mathtt{C}$. As we see in in the next subsection, these are in correspondence. Hence, we take the liberty to employ the same notations.

With the above intuition in mind, most of the cases are straight forward to derive from the operational semantics given in Figure 2. In the case of a statement $\mathtt{consume}(e)$, a cost of e is incurred (provided $e(\sigma) \geq 0$) in addition to the cost of the continuation. While $\mathtt{skip}$ is a no-op, consequently $\mathrm{ect}[\mathtt{skip}](f) = f$, the command $\mathtt{abort}$ immediately aborts the computation and thus $\mathrm{ect}[\mathtt{abort}](f) = 0$. Running an assignment $x \coloneqq d$ followed by a continuation amounts to first sampling a value v for x according to d, say with probability p_v, updating x in the given store σ to v, and then running the continuation on the updated store. The overall expected resource consumption is thus given by the sum of expected costs $f[x/v]$ of all such runs, weighted by their probability p_v. For instance, if $\mathtt{Bernoulli}(p)$ denotes the Bernoulli distribution with parameter p, (yielding 1 with probability p and 0 with probability $p-1$), then $\mathrm{ect}[x \coloneqq \mathtt{Bernoulli}(1/3)](f) = 1/3 \cdot f[x/1] + 2/3 \cdot f[x/0]$. For sequential commands $\mathtt{C};\mathtt{D}$, $\mathrm{ect}[\mathtt{C};\mathtt{D}]$ is given by the composition $\mathrm{ect}[\mathtt{C}] \circ \mathrm{ect}[\mathtt{D}]$, transforming the cost f of a continuation to that of running $\mathtt{C}$, then $\mathtt{D}$ followed by the continuation. For conditionals $\mathtt{if}\ (\phi)\ \{\mathtt{C}\}\ \{\mathtt{D}\}$, the transformer perform a case analysis on the guard. Considering loops $\mathtt{while}\ (\phi)\ \{\mathtt{C}\}$, the expected cost transformer is defined as the least fixed-point of the functional $\lambda F.[\phi]\cdot\mathrm{ect}[\mathtt{C}](F) + [\neg\phi]\cdot f$. We will see below that this fixed-point is always defined. The transformer $\mathrm{ect}[\mathtt{while}\ (\phi)\ \{\mathtt{C}\}]$ thus in particular enjoys

$$\begin{aligned}\mathrm{ect}[\mathtt{while}\ (\phi)\ \{\mathtt{C}\}](f) &= [\phi]\cdot\mathrm{ect}[\mathtt{C}](\mathrm{ect}[\mathtt{while}\ (\phi)\ \{\mathtt{C}\}](f)) + [\neg\phi]\cdot f\\ &= [\phi]\cdot\mathrm{ect}[\mathtt{C};\mathtt{while}\ (\phi)\ \{\mathtt{C}\}](f) + [\neg\phi]\cdot f\ ,\end{aligned}$$

that is, it evaluates to the cost of unfolding the loop when the loop's guard ϕ is satisfied, and otherwise returns the cost f, in correspondence to the semantics. Finally, the transformer on non-deterministic choices maximises over expected costs along the two branches.

6.1 Well-definedness and Soundness

Before we prove soundness, let us mention that as for the expected cost transformer on PARSs, $\mathrm{ect}[\mathtt{C}]$ is continuous, and consequently also monotone, in the following sense:

Lemma 6.2 (Central Properties of $\mathrm{ect}[\texttt{while}\ (\phi)\ \{\texttt{C}\}]$).

(1) continuity: $\mathrm{ect}[\texttt{C}](\sup_{n\in\mathbb{N}} f_n) = \sup_{n\in\mathbb{N}} \mathrm{ect}[\texttt{C}](f_n)$ *for all* ω*-chains* $(f_n)_{n\in\mathbb{N}}$*;*
(2) monotonicity: $f \preceq g \implies \mathrm{ect}[\texttt{C}](f) \preceq \mathrm{ect}[\texttt{C}](g)$.

In particular, continuity ensures that in the case for loop $\texttt{C} = \texttt{while}\ (\phi)\ \{\texttt{D}\}$, the least fixed point underlying $\mathrm{ect}[\texttt{C}]$ is well-defined. Now that we have established that $\mathrm{ect}[\texttt{C}](f)$, and hence $\mathrm{ecost}[\texttt{C}]$, is well-defined, the remaining objective of this section is to prove that $\mathrm{ecost}[\texttt{C}](\sigma)$ indeed gives the expected cost of running C on store σ. To this end, we show that $\mathrm{ect}[\texttt{C}](f)(\sigma)$ coincides with $\mathrm{ect}[\to](f)(\sigma \triangleright \texttt{C})$ defined in terms of the underlying PARS in Section 4. Let us denote by $\mathrm{ect}[\to](f)(\bullet \triangleright \texttt{C})$ the function $\lambda\sigma.\mathrm{ect}[\to](f)(\sigma \triangleright \texttt{C})$. The correspondence thus becomes $\mathrm{ect}[\texttt{C}](f) = \mathrm{ect}[\to](f)(\bullet \triangleright \texttt{C})$.

Since the definition of $\mathrm{ect}[\texttt{C}](f)$ is guided by the semantics, for most commands C this equality is immediate. The only non-trivial cases are that of composition and loops. The following lemma links the corresponding cases.

Lemma 6.3 (Composition and Loop Lemma).

(1) $\mathrm{ect}[\to](f)(\bullet \triangleright \texttt{C};\texttt{D}) = \mathrm{ect}[\to](\mathrm{ect}[\to](f)(\bullet \triangleright \texttt{D}))(\bullet \triangleright \texttt{C})$*;*
(2) $\mathrm{ect}[\to](f)(\bullet \triangleright \texttt{while}\ (\phi)\ \{\texttt{C}\}) = \mathsf{lfp}(F.[\phi] \cdot \mathrm{ect}[\to](F)(\bullet \triangleright \texttt{C}) + [\neg\phi] \cdot f)$.

Note that the right-hand side in (2) is well-defined by Lemma 4.4(1). Relying on this auxiliary lemma, our central soundness result follows by a standard induction on C.

Theorem 6.4 (Soundness & Completeness). *For every command* $\texttt{C} \in \mathsf{Cmd}$*, we have*

(1) $\mathrm{ect}[\to](f)(\bullet \triangleright \texttt{C}) = \mathrm{ect}[\texttt{C}](f)$*; and consequently*
(2) $\mathrm{ecost}[\to](\bullet \triangleright \texttt{C}) = \mathrm{ecost}[\texttt{C}]$.

The theorem thus witnesses that the expected cost transformer of this section gives a sound and complete method for reasoning about the expected cost of programs C.

6.2 Upper Invariants

To find closed-forms for the runtime of loops, Kaminski et al. [2016] propose to search for *upper invariants*, ie. prefix points of the loops characteristic function. The following constitutes a straight forward generalisation of Kaminski et al. [2016, Theorem 3]. For a Boolean expression $\phi \in \mathsf{BExp}$, and two cost functions $f, g \in \mathbb{C}^\Sigma$, as before let us denote by $\phi \vDash f \preceq g$ that $f(\sigma) \leq g(\sigma)$ holds for all stores σ with $\phi \vDash \sigma$.

Proposition 6.5 (Upper Invariant). *For every* $I \in \mathbb{C}^\Sigma$*,*

$$\phi \vDash \mathrm{ect}[\texttt{C}](I) \preceq I \wedge \neg\phi \vDash f \preceq I \Rightarrow \mathrm{ect}[\texttt{while}\ (\phi)\ \{\texttt{C}\}](f) \preceq I\,.$$

The two premises are equivalent to $[\phi] \cdot \mathrm{ect}[\texttt{C}](I) + [\neg\phi] \cdot f \preceq I$, that is, I is a pre-fixed point of the functional $\lambda F.[\phi] \cdot \mathrm{ect}[\texttt{C}](F) + [\neg\phi] \cdot f$; through which the expected cost transformer of the loop $\texttt{while}\ (\phi)\ \{\texttt{C}\}$ is defined. The theorem is thus a reformulation of the fact that the least fixed-point is the least among all its pre-fixed points. Via this proposition, the problem of computing $\mathrm{ecost}[\texttt{C}]$ can be reduced to a set of inequalities whose solution gives an *upper bound* on the expected cost of C, compare the discussion at the end of Section 2. Let us illustrate this also on a simple example.

Example 6.6. Reconsider the program $\texttt{C}_{\mathsf{geo}}$ from Figure 1b. By Proposition 6.5, the expected cost of the loop within $\texttt{C}_{\mathsf{geo}}$ is bounded by I subject to the upper invariant conditions: (i) $b = 1 \vDash \mathrm{ect}[\texttt{D}](I) \preceq I$; and (ii) $b \neq 1 \vDash 0 \preceq I$, for D the loop's body. Define $I \triangleq [b = 1] \cdot 2$. Unfolding the

definition of $\mathrm{ect}[\mathtt{D}](I)$ yields

$$\begin{aligned}\mathrm{ect}[\mathtt{D}](I) &= 1 + 1/2 \cdot I[b/1, x/2 \cdot x] + 1/2 \cdot I[b/0, x/2 \cdot x] \\ &= 1 + 1/2 \cdot [1 = 1] \cdot 2 + 1/2 \cdot [0 = 1] \cdot 2 = 2\,.\end{aligned}$$

It is then not difficult to see that the upper invariant conditions hold. Since the variables b and x are initialized to one in $\mathtt{C}_{\mathsf{geo}}$, by Theorem 6.4(2) we conclude $\mathrm{ecost}[\mathtt{C}_{\mathsf{geo}}] = I[b/1, x/1] = 2$.

Proposition 6.5 suggests the following two stage approach towards an automated cost analysis of a program $\mathtt{C}$ via Theorem 6.4. (i) evaluate $\mathrm{ecost}[\mathtt{C}] = \mathrm{ect}[\mathtt{C}](0)$ symbolically and generating a constraint corresponding to the the upper invariant condition; and (ii) synthesise concrete upper invariants via the collection of generated constraints.

This is akin to the common approach, where cost functions are specified as linear combination over an (arbitrary but fixed) finite vector of *base functions* $(b_1, \ldots, b_k)$, itself cost functions, abstracting stores as non-negative numbers. Such cost expressions take the form $\boldsymbol{\kappa}(b_1, \ldots, b_k)$, where $\kappa(r_1, \ldots, r_k) = \sum_i q_i \cdot r_i$ for rational (in particular real) numbers q_i. Upper invariants then take the form $\phi \vDash \mathrm{ect}[\mathtt{C}](\boldsymbol{\kappa_I}(b_1, \ldots, b_k)) \leq \boldsymbol{\kappa_I}(b_1, \ldots, b_k)$ and $\neg\phi \vDash \boldsymbol{\kappa_f}(b_1, \ldots, b_k) \leq \boldsymbol{\kappa_I}(b_1, \ldots, b_k)$.

By treating κ_I and κ_f as undetermined, these constraints can be reduced to inequalities over expressions on coefficients in such a way that the resulting set of constraints is eg. amenable to Linear Programming. A solution to these constraints, viz, particularly concrete values for the coefficients q_i occurring in κ_I, then yields a concrete upper invariant. Consequently, an upper bound to the expected cost of $\mathtt{while}\ (\phi)\ \{\mathtt{C}\}$ in terms of the base functions $\vec{b}$ is inferred.

7 ALTERNATING EXPECTED COST AND VALUE ANALYSIS

While the calculus developed in the previous section is indeed compositional, the cost analysis via Proposition 6.5, as described above, is not. This is most apparent in the case of nested loops, abstractly represented as follows: $\mathtt{while}\ (\phi)\ \{\mathtt{while}\ (\psi)\ \{\mathtt{C}\}\}$. The synthesis of upper invariants I and O for the inner and outer loops, respectively, wrt. a cost function f, is driven by the following inter-dependent constraints, compare the exposition of the trader example in Section 2.

$$\phi \vDash I \leq O \quad \wedge \quad \neg\phi \vDash f \leq O \quad \wedge \quad \psi \vDash \mathrm{ect}[\mathtt{C}](I) \leq I \quad \wedge \quad \neg\psi \vDash O \leq I\,.$$

These constraints cannot be considered in isolation. In particular, if we express I and O in terms of linear combinations $\boldsymbol{\kappa}_I(b_1, \ldots, b_k)$ and $\boldsymbol{\kappa}_O(b_1, \ldots, b_k)$ of base functions b_i, the undetermined combinators κ_I and κ_O cannot be synthesised separately. Thus, the analysis degenerates to a *whole-program analysis*.

As emphasised in Section 3, in the *non-probabilistic* setting modularity of a cost analysis is facilitated through combining cost analysis with an analysis on how stores evolve through the execution of program parts. In the following, we suit this conceptual simple idea to an expected cost analysis.

The result of a program $\mathtt{C}$, run on an initial store σ, is conceivable as a subdistribution μ over stores (many, in the case $\mathtt{C}$ is also non-deterministic). Then $\mathrm{ect}[\mathtt{C}](f)(\sigma)$ yields the expected cost of $\mathtt{C}$ plus the *expected value* $\mathbb{E}_\mu(f)$ of the cost function f on the distribution of states μ. When f coincides with the cost of a continuation, we re-obtain the initial intuition that $\mathrm{ect}[\mathtt{C}](f)$ yields the cost of running $\mathtt{C}$ followed by the corresponding continuation. Otherwise, if the program $\mathtt{C}$ is non-deterministic, then $\mathtt{C}$ may yield many subdistributions μ. In this case, $\mathrm{ect}[\mathtt{C}](f)$ accounts for the supremum of the expectations of f. We make this correspondence precise. First, we define the expected value transformer $\mathrm{evalue}[\mathtt{C}]$ of a command $\mathtt{C}$ in correspondence to $\mathrm{ect}[\mathtt{C}]$, ignoring costs.

Definition 7.1 (Expected Value Transformer). The *expected value transformer* $\mathrm{evalue}[\mathtt{C}] \colon \mathbb{C}^\Sigma \to \mathbb{C}^\Sigma$ for *commands* $\mathtt{C}$ is defined through the rules given in the third column in Figure 3.

Concerning the program $\mathtt{C}_{\mathsf{geo}}$ from Figure 1b for instance, we obtain $\mathrm{evalue}[\mathtt{C}_{\mathsf{geo}}](f) = \lambda\sigma.\, \sum_{i=0}^{\infty} {}^1\!/_{2^i} \cdot f(\{b \mapsto 1, x \mapsto 2^{i+1}\})$. Note that $\mathrm{evalue}[\mathtt{C}](f) = \mathrm{ect}[\mathsf{costFree}(\mathtt{C})](f)$ holds, where the *cost-free program* $\mathsf{costFree}(\mathtt{C})$ is obtained from $\mathtt{C}$ by dropping all cost annotations $\mathtt{consume}(e)$. Particularly, Proposition 6.5 remains intact if we substitute $\mathrm{evalue}[\cdot]$ for $\mathrm{ect}[\cdot]$.

The next lemma establishes a separation of the expected cost and value computation, which underlies the expected cost transformer $\mathrm{ect}[\mathtt{C}]$.

Lemma 7.2 (Separating Expected Cost and Value).

$$\mathrm{ect}[\mathtt{C}](f) \leq \mathrm{ecost}[\mathtt{C}] + \mathrm{evalue}[\mathtt{C}](f) \ .$$

A similar result, albeit restricted to purely probabilistic programs, has been observed by Kaminski et al. [2018, Thm 8.1]. Notice that if $\mathtt{C}$ features non-determinism, the right-hand side may over-approximate $\mathrm{ect}[\mathtt{C}](f)$.

Example 7.3. Consider the non-deterministic program $\mathtt{C} = \{\mathtt{consume}(1)\} <> \{x \coloneqq x + 1\}$. Then

$$\mathrm{ect}[\mathtt{C}](f) = \max(1 + f, f[x/x+1]) \leq 1 + \max(f, f[x/x+1]) = \mathrm{ecost}[\mathtt{C}] + \mathrm{evalue}[\mathtt{C}](f) \ .$$

When f depends on the value of x, the two sides are thus in general not equal.

In this example, we exploit that one branch is costly in terms of consumed resources, while the other one in terms of the expected value of f. When $\mathtt{C}$ doesn't contain non-deterministic choice, the inequality in Lemma 7.2 can be replaced by an equality.

Consider now a while loop, $\mathtt{while}\ (\phi)\ \{\mathtt{C}\}$. As above, we express the upper invariant I as the combination $\boldsymbol{\kappa}(b_1, \ldots, b_k)$ of base functions b_i. Reconsider the first premise of Proposition 6.5: $\phi \vDash \mathrm{ect}[\mathtt{C}](\boldsymbol{\kappa}(b_1, \ldots, b_k)) \leq \boldsymbol{\kappa}(b_1, \ldots, b_k)$. By Lemma 7.2, this constraint is entailed by

$$\phi \vDash \mathrm{ecost}[\mathtt{C}] + \mathrm{evalue}[\mathtt{C}](\boldsymbol{\kappa}(b_1, \ldots, b_k)) \leq \boldsymbol{\kappa}(b_1, \ldots, b_k) \ , \tag{5}$$

Moreover, when the upper invariant κ is linear in all its arguments, it distributes over expectations and hence over the expected value transformer $\mathrm{evalue}[\mathtt{C}]$. Consequently (5) becomes

$$\phi \vDash \mathrm{ecost}[\mathtt{C}] + \boldsymbol{\kappa}(\mathrm{evalue}[\mathtt{C}](b_1), \ldots, \mathrm{evalue}[\mathtt{C}](b_k)) \leq \boldsymbol{\kappa}(b_1, \ldots, b_k) \ . \tag{6}$$

First, this decomposes the analysis of the cost of the loop's body $\mathtt{C}$ from the analysis of how $\boldsymbol{\kappa}(b_1, \ldots, b_k)$ changes in expectation during one iteration. Second, none of the calls to $\mathrm{evalue}[\mathtt{C}]$ do reference the combinator κ. These can be determined in independence of the analysis of $\mathtt{while}\ (\phi)\ \{\mathtt{C}\}$. We generalise this observation. As usual, we call κ *concave* if $p \cdot \kappa(\vec{r}) + (1-p) \cdot \kappa(\vec{s}) \leq \kappa(p \cdot \vec{r} + (1-p) \cdot \vec{s})$ (where $0 \leq p \leq 1$) and (weakly) *monotone* if $\vec{r} \leq \vec{s}$ implies $\kappa(\vec{r}) \leq \kappa(\vec{s})$. Together, the two conditions yield the following sufficient criterion to distribute κ over expectations, which can be proven by a standard induction on $\mathrm{evalue}[\mathtt{C}](f) = \mathrm{evalue}[\mathtt{C}](f)$.

Lemma 7.4 (Distribute over Expected Values). *Suppose* $\kappa \colon \mathbb{C}^k \to \mathbb{C}$ *is monotone. Furthermore, suppose* κ *is concave if the command* $\mathtt{C}$ *is probabilistic. Then* $\mathrm{evalue}[\mathtt{C}](\boldsymbol{\kappa}(b_1, \ldots, b_k)) \leq \boldsymbol{\kappa}(\mathrm{evalue}[\mathtt{C}](b_1), \ldots, \mathrm{evalue}[\mathtt{C}](b_k))$.

Note that concavity is only needed to distribute κ over the expected value given by *probabilistic* statements, that is, probabilistic assignments from non-dirac distributions. The above lemmas in conjunction with Proposition 6.5 yields the main result of this section.

Theorem 7.5 (Modular Upper-Invariants). *Let* $\mathtt{C}$ *and* κ *be as in Lemma 7.4. Then*

1. $\phi \vDash \mathrm{ecost}[\mathtt{C}] + \boldsymbol{\kappa}(\mathrm{evalue}[\mathtt{C}](b_1), \ldots, \mathrm{evalue}[\mathtt{C}](b_k)) \leq \boldsymbol{\kappa}(b_1, \ldots, b_k)$
   $\Rightarrow \mathrm{ecost}[\mathtt{while}\ (\phi)\ \{\mathtt{C}\}] \leq \boldsymbol{\kappa}(b_1, \ldots, b_k)$*; and*

2. $\phi \vDash \boldsymbol{\kappa}(\mathrm{evalue}[\mathtt{C}](b_1),\ldots,\mathrm{evalue}[\mathtt{C}](b_k)) \preceq \boldsymbol{\kappa}(b_1,\ldots,b_k) \wedge \neg\phi \vDash f \preceq \boldsymbol{\kappa}(b_1,\ldots,b_k)$
$\Rightarrow \mathrm{evalue}[\mathtt{while}\ (\phi)\ \{\mathtt{C}\}](f) \preceq \boldsymbol{\kappa}(b_1,\ldots,b_k)$.

Concerning (2), we exploit $\mathrm{evalue}[\mathtt{while}\ (\phi)\ \{\mathtt{C}\}](f) = \mathrm{ect}[\mathtt{while}\ (\phi)\ \{\mathsf{costFree}(\mathtt{C})\}](f)$ and $\mathrm{ecost}[\mathsf{costFree}(\mathtt{C})] = 0$. The theorem gives a *modular recursive procedure* to infer upper bounds on costs of loops, as follows, cf. Section 8.

(1) Estimate the cost of one iteration of the body C.
(2) Analyse how the expected value of the base functions b_i is changed by one iteration of the body changes, as formalises in the expected value transformer.
(3) Solve the recurrence, given by the implication, where $\boldsymbol{\kappa}$ is to be determined.

Crucially the loop's body C becomes treatable in complete independence of the loop itself. Via Lemma 7.2 and Lemma 7.4, sequential composition can be treated similarly to Theorem 7.5.

Theorem 7.6 (Modular Sequential Analysis). *Let* C *and* κ *be as in Lemma 7.4. Then*

1. $\mathrm{ecost}[\mathtt{D}](f) \preceq \boldsymbol{\kappa}(b_1,\ldots,b_k) \Rightarrow \mathrm{ecost}[\mathtt{C};\mathtt{D}] \preceq \mathrm{ecost}[\mathtt{C}]+\boldsymbol{\kappa}(\mathrm{evalue}[\mathtt{C}](b_1),\ldots,\mathrm{evalue}[\mathtt{C}](b_k))$;
2. $\mathrm{evalue}[\mathtt{D}](f) \preceq \boldsymbol{\kappa}(b_1,\ldots,b_k) \Rightarrow \mathrm{evalue}[\mathtt{C};\mathtt{D}](f) \preceq \boldsymbol{\kappa}(\mathrm{evalue}[\mathtt{C}](b_1),\ldots,\mathrm{evalue}[\mathtt{C}](b_k))$.

7.1 Analysis of Examples

We conclude the section, by discussing the analysis of the running examples of Section 2 using our methodology. Further, we present an interesting example by Wang et al. [2019] modelling a *fork-join queues*. Noteworthy, all displayed bounds are derived via our implementation `eco-imp` as detailed in the following section.

Geo. Reconsider the program $\mathtt{C}_{\mathsf{geo}}$ (Figure 1b), where the cost is given by the number of loop iterations. Since loop-iterations depend on the value of b, we choose the base function $\langle b\rangle$. This is sufficient to handle $\mathtt{C}_{\mathsf{geo}}$ via Theorem 7.5(1). The body D of the loop changes $\langle b\rangle$ to $1/2{\cdot}\langle 0\rangle{+}1/2{\cdot}\langle 1\rangle = 1/2$ in expectation. Moreover, the cost of a single loop iteration $\mathrm{ecost}[\mathtt{D}]$ is one, which follows by unfolding the definition. By Theorem 7.5(1), $\boldsymbol{\kappa}(\langle b\rangle)$ is an upper bound to the cost of the loop if

$$b = 1 \vDash \mathrm{ecost}[\mathtt{D}] + \boldsymbol{\kappa}(\mathrm{evalue}[\mathtt{D}](\langle b\rangle)) \preceq \boldsymbol{\kappa}(\langle b\rangle) \quad \Leftrightarrow \quad b = 1 \vDash 1 + \boldsymbol{\kappa}(1/2) \preceq \boldsymbol{\kappa}(\langle b\rangle)\ .$$

A case analysis on $b = 1$ shows that the constraint is satisfied with $\kappa(x) \triangleq 2 \cdot x$. Finally, since $b \coloneqq 1$ before entering the loop we derive the optimal bound $\kappa(\langle 1\rangle) = 2$ for $\mathtt{C}_{\mathsf{geo}}$.

Trader. Recall program $\mathtt{C}_{\mathsf{trader}}$ from Figure 1a. As we have seen in Section 3, the cost of the inner loop is given by $\langle n\rangle \cdot \langle p\rangle$. This then yields

$$\mathrm{ecost}[\mathtt{D}] = 1/44 \cdot \textstyle\sum_{n=0}^{10} n \cdot \langle p+1\rangle + 3/44 \cdot \sum_{n=0}^{10} n \cdot \langle p-1\rangle\ ,$$

as bound the cost of the body D of the outer loop. Wrt. the outer loop itself, consider the base functions $b_1 \triangleq \langle min+1\rangle \cdot \langle p - min\rangle$ and $b_2 \triangleq \langle p-min\rangle^2$. Applying Theorem 7.5(1), the cost of $\mathtt{C}_{\mathsf{trader}}$ is given by $\boldsymbol{\kappa}(b_1,b_2)$ subject to the constraint

$$0 \preceq min \preceq p-1 \vDash \mathrm{ecost}[\mathtt{D}] + \boldsymbol{\kappa}(\mathrm{evalue}[\mathtt{D}](b_1),\mathrm{evalue}[\mathtt{D}](b_2)) \preceq \boldsymbol{\kappa}(b_1,b_2)\ .$$

Note that the loop body D increments and decrements the price with probabilities $1/4$ and $3/4$, respectively, whereas *min* remains unchanged. Formally,

$$\mathrm{evalue}[\mathtt{D}](b_1) = 1/4 \cdot \langle min+1\rangle \cdot \langle p+1-min\rangle + 3/4 \cdot \langle min+1\rangle \cdot \langle p-1-min\rangle\ ,$$
$$\mathrm{evalue}[\mathtt{D}](b_2) = 1/4 \cdot \langle p+1-min\rangle^2 + 3/4 \cdot \langle p-1-min\rangle^2\ ,$$

which can be derived by unfolding $\mathrm{evalue}[\mathtt{D}]$ and applying Theorem 7.5(2). In turn, the above constraint holds with $\kappa(x,y) \triangleq 10 \cdot x + 5 \cdot y$, yielding an upper bound

$$\boldsymbol{\kappa}(b_1,b_2) = 10 \cdot \langle min+1\rangle \cdot \langle p-min\rangle + 5 \cdot \langle p-min\rangle^2\ ,$$

```
l1 := 0; l2 := 0; i := 1;
while (i ≤ n) {
  n := n - 1;
  if (l1 ≥ 1) { l1 := l1 - 1 };
  if (l2 ≥ 1) { l2 := l2 - 1 };
  if (Bernoulli(1/50)) {
    if (Bernoulli(1/5))
      { l1 := l1 + 3 }
      { if (Bernoulli(1/2))
          { l2 := l2 + 2 }
          { l1 := l1 + 2; l2 := l2 + 1 }}};
  if (l1 ≥ l2) { consume(l1) } { consume(l2) } }
```

(a) Fork-join Queue $\mathtt{C}_{\text{fj-queue}}$.

```
while (n > 0) {
  k := 1;
  while (k > 0) {
    x := Bernoulli(1/2);
    y := Bernoulli(1/2);
    if (x == y) { k := 1 } { k := 0 };
    consume(1)
  };
  n := n - 1;
}
```

(b) Rejection sampling $\mathtt{C}_{\text{rejection-sampling}}$.

Fig. 4. Further examples from the literature.

on the cost of $\mathtt{C}_{\text{trader}}$. Note that this upper bound can be derived with our implementation in 25ms.

Coupon Collector. Recall program $\mathtt{C}_{\text{coupons}}$ of Figure 1c, and let D denote the main loop's body. While $\mathrm{ecost}[\mathtt{D}] = 1$ is straight forward to derive, the challenge of this example lies in calculating the expected value of base functions, due to the sampling instruction. Particularly, for a base function b, $\mathrm{evalue}[\mathtt{D}](b)$ is given by

$$[1 \le n] \cdot \sum_{draw=1}^{n} \big([coupons < draw] \cdot b[coupons/coupons + 1] + [draw \le coupons] \cdot b\big)/n \,.$$

Our implementation selects, among others, the base functions $b_1 \triangleq \langle n - coupons\rangle^2$ and $b_2 \triangleq \langle n - coupons\rangle \cdot \langle coupons + 1\rangle$ from the loop guard, for which it derives bounds

$$b_1' \triangleq \langle n - coupons\rangle \cdot (\langle n - coupons - 1\rangle^2 + \langle n - coupons\rangle \cdot \langle coupons\rangle)/\langle n\rangle$$
$$b_2' \triangleq \langle n - coupons\rangle \cdot (\langle n - coupons - 1\rangle \cdot \langle coupons + 2\rangle + \langle coupons + 1\rangle \cdot \langle coupons\rangle)/\langle n\rangle \,,$$

for $\mathrm{ect}[\mathtt{D}](b_i)$. The constraint $0 \le i \le n - 1 \vDash 1 + \boldsymbol{\kappa}(b_1', b_2') \le \boldsymbol{\kappa}(b_1, b_2)$, describing the cost of the loop, is then shown satisfiable by taking $\kappa(x, y) \triangleq x + 1/2 \cdot y$. Substituting 0 for *coupons* in the bound $\boldsymbol{\kappa}(b_1, b_2)$ due to the initial assignments yields the overall estimate $\langle n\rangle + 1/2 \cdot \langle n\rangle^2$ for the expected cost of $\mathtt{C}_{\text{coupons}}$. This analysis was performed in 195ms with our tool.

Queuing Network. Fork-join queues (see Kim and Agrawala [1989]) have been used to model various systems [Dean and Ghemawat 2008; Hill and Marty 2008; Menascé 2004]. They consist of n processors, each of which is equipped with a dedicated queue. On arrival at the fork point, a job is probabilistically split into n sub-jobs, which are then served by each of the n servers commanding the respective queues. After completion, the sub-jobs wait until all other sub-jobs have also been processed. The sub-jobs are then rejoined and the job is completed.

Following Wang et al. [2019], we represent fork-join queues for 2 servers as a probabilistic program with a unitary cost model, that measures progress of the sub-jobs in the respective queues, cf. Figure 4a on page 18. A fixed probability ($1/50$) is employed to model the arrival of new jobs, while n jobs are processes by the two servers. With probability $1/5$ the job is handled by the first server. Otherwise, with equal probability the job is handled either by the second server, or, passed to both servers. Server 1 takes 3 time units for completion, while Server 2 takes 2 units. If the job is split on the servers, the Server 1 and 2 require 2 and 1 units, respectively. This is modelled by enlarging the queues respectively. Each server can handle one queue entry per time unit. The cost for the fork-join queue is given by the expected time of completion of each job, equally representable as the length of the longest queue. Our prototype computes the (asymptotically) optimal bound $113/741\langle n + 1\rangle + 125/243 + 125/244$ in 2.215s.

function ecost[C]
  switch C **do**
    case D; E **where** D contains a loop:
      $\boldsymbol{\kappa}(\vec{b}) \leftarrow$ ecost[E]
      $\vec{b'} \leftarrow$ **foreach** $b \in \vec{b}$: evalue[D](b)
      return ecost[D] $+ \boldsymbol{\kappa}(\vec{b'})$
    case `while` (ϕ) {D}:
      $g \leftarrow$ ecost[D]
      $\vec{b} \leftarrow$ SelectBases(ϕ, g)
      $\vec{b'} \leftarrow$ **foreach** $b \in \vec{b}$: evalue[D](b)
      $\kappa \leftarrow$ LinearTemplate(length$(\vec{b})$)
      $\theta \leftarrow$ Solve$(\phi \vDash g + \boldsymbol{\kappa}(\vec{b'}) \preceq \boldsymbol{\kappa}(\vec{b}))$
      return $\theta(\boldsymbol{\kappa})(\vec{b})$
    default: return SymbolicCost(C)

function evalue[C](f)
  switch C **do**
    case D; E **where** D contains a loop:
      $\boldsymbol{\kappa}(\vec{b}) \leftarrow$ evalue[E]
      $\vec{b'} \leftarrow$ **foreach** $b \in \vec{b}$: evalue[D](b)
      return $\boldsymbol{\kappa}(\vec{b'})$
    case `while` (ϕ) {D}:
      $\vec{b} \leftarrow$ SelectBases(ϕ, f)
      $\vec{b'} \leftarrow$ **foreach** $b \in \vec{b}$: evalue[D](b)
      $\kappa \leftarrow$ LinearTemplate(length$(\vec{b})$)
      $\theta \leftarrow$ Solve$(\phi \vDash \boldsymbol{\kappa}(\vec{b'}) \preceq \boldsymbol{\kappa}(\vec{b}) \wedge \neg\phi \vDash f \preceq \boldsymbol{\kappa}(\vec{b}))$
      return $\theta(\boldsymbol{\kappa})(\vec{b})$
    case $x \coloneqq d$: **return** Expectation$(d, \lambda v.f[x/v])$
    default: return SymbolicValue(C, f)

Fig. 5. Pseudocode detailing cost-inference.

8 IMPLEMENTATION

In this section, we give an overview of the implementation of our methodology within our tool `eco-imp`. The core of our tool is given by the algorithms ecost[C] and evalue[C] outlined in Figure 5, computing for a given program C and cost function f, an upper bound to ecost[C] and evalue[C](f), respectively. As for the transformers, the two algorithms are structurally similar.

In the majority of cases, the default branch of the algorithm is executed and the implementation symbolically executes the corresponding transformer from Figure 3. Conclusively, a precise bounds is computed in these cases. Exceptions to this are the cases of loops and assignments, detailed below. Here, possibly imprecise upper bounds are derived. Due to monotonicity (Lemma 4.4(2)) soundness of the overall algorithm remains intact.

Sequential composition. For a command C = D; E, our implementation relies on Theorem 7.5, except when D does not contain a loop. In this case, the algorithms proceed by unfolding according to the rules in Figure 5 and (recursion on D). Here, modularity is not necessary; the straight-line program D can be analysed directly.

While loops. The main novel aspect from which our implementation derives its strength lies in the treatments of loops, providing an almost literal implementation of Theorem 7.5, following to recipe given on page 17. The subprocedure SelectBases, outlined below, selects a vector of base functions $\vec{b}$ from the loops guard ϕ, and the recursively computed cost g of the loop's body or the cost function f. For each such base function b, an upper bound b' to its expectation evalue[C](b) is computed and a linear template $\kappa(\vec{x}) = \sum_i q_i \cdot x_i$ with undetermined coefficients $q_i \in \mathbb{R}_{\geq 0}$ is prepared. Based on these ingredients, the subprocedure Solve is applied to the constraint dictated by Theorem 7.5. Should this procedure establish a solution θ, ie. an assignment to the undetermined coefficients q_i underlying κ that validates the supplied constraint, the instantiated cost function $\theta(\boldsymbol{\kappa})(\vec{b}) = \sum_i \theta(q_i) \cdot \vec{b}_i$ is returned. The dedicated constraint solver underlying the procedure Solve is detailed below in Section 8.1.

Selection of base functions. Our implementation selects a set of candidate base functions $\vec{b}$ as a *linear* or *non-linear* combination of base functions $\vec{b}_\phi$, $\vec{b}_g$ and $\vec{b}_f$ extracted from the loop guard ϕ, the expected cost of the loop body g and the continuation f, loosely following the heuristics of Sinn et al. [2016]. Specifically, for a loop guard containing $e_1 \preceq e_2$ we take $\langle e_2 - e_1 + 1 \rangle \in \vec{b}_\phi$. Further, $\vec{b}_g$ and $\vec{b}_f$ contain all maximal non-max subexpressions of g and f, respectively. Given $\vec{b}_\phi$,

$\vec{b}_g$ and $\vec{b}_f$ we then construct different linear and non-linear combinations, such as $\vec{b} = \vec{b}_\phi + \vec{b}_g + \vec{b}_f$ and $\vec{b} = \vec{b}_\phi \times \vec{b}_g + \vec{b}_f$ where $\times$ and $+$ extend multiplication and addition to sets of base functions element-wise. Since base functions can be extracted from a given context rather than the whole program, the number of base functions is usually low. This is one crucial aspect to the efficiency of our algorithm. The prototype implements caching and backtracking to test different base functions. In particular, non-linear base functions are employed only if the linear ones fail. This is another crucial aspect to efficiency, in contrast to a monolithic procedure. Constraint solving over linear arithmetical expressions is significantly more efficient; non-linear expressions are employed only in the analysis of those program fragments where necessary.

Assignments. Our implementation supports, on the one hand, assignments with sampling from *finite*, *discrete* distributions, of the form $d(\sigma) = \{\!\{p_i(\sigma) : e_i(\sigma)\}\!\}_{0 \le i \le k}$ for a fixed constant k and $p_i = e_i/f_i$, such as $\{\!\{{}^{i+1}/_{10} : x + 2i\}\!\}_{0 \le i \le 9}$. Noteworthy, probabilities of probabilistic branches are not necessarily constant. In this case, the expected cost of a probabilistic assignment $x \coloneqq d$ is given by

$$\mathrm{ect}[x \coloneqq d](f)(\sigma) = \textstyle\sum_{1 \le i \le k} p_i(\sigma) \cdot f[x/e_i(\sigma)] \ .$$

Thus our system encompasses a variety of standard distributions where probabilistic branching is *static*, in the sense that the degree does not depend on program variables. Thus our implementation supports a variety of standard distributions, noteworthy sampling from *uniform distributions with bounded support*, *Bernoulli*, *binomial* and *hypergeometric* distributions.

On the other hand, we are able to natively support *dynamic probabilistic branching*, facilitated by our constraint solver for Upper Invariants. In particular, we have added support for sampling uniform distributions $\texttt{Uniform}(e_1, e_2)$ for $e_1 \le e_2$. This, for example, is crucial to represent the Coupon Collector's problem properly. Note, that in this case

$$\mathrm{ect}[x \coloneqq d](f) = [e_1 \le e_2] \cdot \big(\textstyle\sum_{i=e_1}^{e_2} f[x/i]\big)/(e_2 - e_1 + 1) \ .$$

To find a closed form for the bounded sum $\sum_{i=e_1}^{e_2} f[x/i]$, our implementation seeks for an upper bound $\boldsymbol{\kappa}(\vec{b})$ subject to the constraint $e_1 \le i \wedge i \le e_2 \vDash f[x/i] + \boldsymbol{\kappa}(\vec{b}[i/i+1]) \le \boldsymbol{\kappa}(\vec{b})$.

8.1 Constraint Solving Mechanism

In this section, we highlight the constraint solving mechanism as implemented in our prototype `eco-imp`, consisting of two stages: the first translates constraints over costs (with undetermined coefficients) to tests of non-negativity over polynomials; the second stage then treats this syntactically simpler form of constraints.

First, we fix two syntactic categories of (non-linear) integer expressions Exp and *cost expressions* CExp as follows.

$$\mathsf{Exp} \ni e, f ::= \boldsymbol{i} \mid x \mid e + f \mid e - f \mid e \cdot f \quad \mathsf{CExp} \ni c, d ::= e/f \mid [\phi] \cdot c \mid c + d \mid \max(c, d) \ .$$

In Exp, we use integer variables $x \in \mathsf{Var}$ to denote cost functions $\lambda\sigma.\sigma(x)$. Cost expressions denote fractions over integer expressions, closed under guards, sum and maximum. Furthermore, we restrict BExp to Boolean formulas over atoms $e \le f$, in disjunctive normal form. This still permits the usual comparison operators over integer expressions, eg., $e < f$ can be represented as $e + 1 \le f$ or $e = f$ as $e \le f \wedge f \le e$. We tacitly employ such equalities below. When denoting cost expressions, we usually write e instead of $e/1$. Base functions are given as a combination of expressions $\langle e \rangle \triangleq [0 \le e] \cdot e$. Not every cost expression is an expectation in $\mathbb{C}^\Sigma$ though, as cost expression need not be well-formed.

We say $c \in \mathsf{CExp}$ is *well-formed* wrt. store $\sigma \in \Sigma$ if (i) $c = e/f$ such that $e(\sigma) \ge 0$ and $f(\sigma) > 0$; (ii) $c = [\phi] \cdot d$ such that $\sigma \vDash \phi$ and d is well-formed wrt. σ; (iii) $c = c_1 + c_2$ or $c = \max(c_1, c_2)$ such

Simplification Rules

$$\frac{\phi \wedge \psi \vDash c_1 + c_2 \preceq d \quad \phi \wedge \neg\psi \vDash c_2 \preceq d}{\phi \vDash [\psi] \cdot c_1 + c_2 \preceq d}\,[\textsc{condL}] \qquad \frac{\phi \wedge \psi \vDash c \preceq d_1 + d_2 \quad \phi \wedge \neg\psi \vDash c \preceq d_2}{\phi \vDash c \preceq [\psi] \cdot d_1 + d_2}\,[\textsc{condR}]$$

$$\frac{\phi \vDash e + c \star f \preceq d \star f}{\phi \vDash e/f + c \preceq d}\,[\textsc{divL}] \qquad \frac{\phi \vDash c \star f \preceq e + d \star f}{\phi \vDash c \preceq e/f + d}\,[\textsc{divR}] \qquad \frac{\phi \vDash c_1 + c_3 \preceq d \quad \phi \vDash c_2 + c_3 \preceq d}{\phi \vDash \max(c_1, c_2) + c_3 \preceq d}\,[\textsc{maxL}]$$

Logical Rules

$$\frac{}{\phi \vDash 0 \preceq d}\,[\textsc{triv}] \qquad \frac{\phi \vDash \bot}{\phi \vDash c \preceq d}\,[\textsc{contr}] \qquad \frac{\forall i.(\phi_i \vDash c \preceq d)}{(\bigvee_i \phi_i) \vDash c \preceq d}\,[\textsc{conj}]$$

Fig. 6. Constraint simplification rules.

that both c_1 and c_2 are well-formed wrt. σ. Well-formedness of c under σ guarantees that $c(\sigma)$ is defined and non-negative. We call c well-formed wrt. a Boolean expression ϕ, if c is well-formed for all stores σ such that $\sigma \vDash \phi$. Apart from the lack of fixed points, the syntax of cost expression is rich enough to express our expectation transformers. Division is included to express the built-in distributions outlined above. Cost expressions form a commutative semigroup under $+$, with 0 the unit element. Thus, it is justified to consider cost expressions equal up to associativity and commutativity of $+$, as well as the unit law $0 + c = c + 0 = c$.

For $c \in \mathsf{CExp}$ and $e \in \mathsf{CExp}$, we define *multiplication* $c \star (e_1/e_2)$ with a fraction $q = e_1/e_2$ by recursion on c in the natural way as follows:

$$(f_1/f_2) \star (e_1/e_2) \triangleq (f_1 \cdot e_1)/(f_2 \cdot e_2) \qquad ([\phi] \cdot c) \star q \triangleq [\phi] \cdot (c \star q)$$
$$(c_1 + c_2) \star q \triangleq c_1 \star q + c_2 \star q \qquad \max(c_1, c_2) \star q \triangleq \max(c_1 \star q, c_2 \star q)\ .$$

Note that this operation preserves well-formedness. Due to the last clause, $c \star (e_1/e_2)$ does in general not equal $c \cdot (e_1/e_2)$, namely, on stores σ where the fraction is negative. However, for well-formed cost expression e_1/e_2, the equality holds.

Constraint simplification. In Figure 6, we present an inference system over *constraints* $\phi \vDash c \preceq d$, where $c, f \in \mathsf{CExp}$ and $\phi, \psi \in \mathsf{BExp}$. This system eliminates max, guards and disjunctions by case analysis. We elide a rule for $\max$ occurring in right-hand sides, as this can be easily prevented in the constraint solving setup. We say a constraint $\phi \vDash c \preceq d$ is *well-formed*, if c, d are well-formed under ϕ. It is *valid* if for all stores $\sigma \in \Sigma$ with $\sigma \vDash \phi$, we have $c(\sigma) \leq d(\sigma)$. It is easy to see that the rules preserve well-formedness. The system is sound and complete in the following sense.

Theorem 8.1. *Let Δ be a derivation of a well-formed constraint $\phi \vDash c \preceq d$. Then this constraint is valid if and only if all premises in Δ are.*

This statement is proven by induction on the derivation Δ. The rules concerning division, rule (divL) and (divR), are sound because the divisor f is positive on all stores σ satisfying ϕ, due to the well-formedness assumption.

Example 8.2. Recall that the analysis of program $\mathsf{C}_{\mathsf{geo}}$ in Section 7.1 required solving the constraint $b = 1 \vDash 1 + \boldsymbol{\kappa}(1/2) \preceq \boldsymbol{\kappa}(\langle b \rangle)$. Fix $\kappa(x) \triangleq q \cdot x$ for undetermined $q \in \mathbb{R}_{\geq 0}$ and note that $\langle b \rangle = [0 \preceq b] \cdot b$ by definition. By unfolding κ, the cost of the loop in $\mathsf{C}_{\mathsf{geo}}$ is given by $\boldsymbol{\kappa}(\langle b \rangle)$ if $b = 1 \vDash 1 + \boldsymbol{q}/2 \preceq [0 \preceq b] \cdot \boldsymbol{q} \cdot b$, is valid for some $q \in \mathbb{R}_{\geq 0}$. Eliminating guards yield several constraints, but only the constraint $b = 1 \wedge 0 \preceq b \vDash 1 + \boldsymbol{q}/2 \preceq \boldsymbol{q} \cdot b$, is not subject to rule (triv) or rule (contr). Rule (divL) then yields

$$b = 1 \wedge 0 \preceq b \vDash 2 + \boldsymbol{q} \preceq 2 \cdot \boldsymbol{q} \cdot b\ . \tag{7}$$

By Theorem 8.1, this constraint is equi-satisfiable with the initial one above, in the sense that any value for q for the latter makes also the former valid.

Constraint solving. Given a constraint $\phi \vDash c \leq d$, the simplification rules of Figure 6 can be seen as a way to eliminate max, guards, division and disjunctions. Thus, they translate constraints over cost-expressions to a set of equivalent constraints over integer-valued expressions. Wlog., these constraints are of the form $\bigwedge_i 0 \leq e_i \vDash 0 \leq e$, by employing additionally the identity $e \leq f \Leftrightarrow 0 \leq f - e$. Any expression e can be understood as a polynomial $p(\vec{x})$ in the program variables $\vec{x}$. Validity of constraints of the above form becomes representable as a test of non-negativity. To this end, we make use of known approximations for certifying non-negativity of polynomial expressions, cf. [Chatterjee et al. 2016; Fuhs et al. 2007; Wang et al. 2019]. Let p and $p_{1 \leqslant i \leqslant m}$, respectively, denote real-valued polynomials over the integers. The following proposition expresses a (gross) simplification of Handelman's theorem [Handelman 1988], which turns out to be quite effective in our context.

Proposition 8.3. *Let $p(\vec{x})$ be a real-valued polynomial over the integers. Suppose there exists multi-indices $(i_1, \dots, i_m) \in \mathbb{N}^m$, polynomials $p_{1 \leqslant i \leqslant m} \colon \mathbb{Z}^n \to \mathbb{R}$ and $c_{(i_1,\dots,i_m)} \in \mathbb{R}_{\geq 0}$ so that $p(\vec{x}) = \sum_{(i_1,\dots,i_m) \in \mathbb{N}^m} c_{(i_1,\dots,i_m)} \cdot p_1^{i_1}(\vec{x}) \cdots p_m^{i_m}(\vec{x})$. Then $\bigwedge_{i=1}^m 0 \leq p_i(\vec{x})$ implies $0 \leq p(\vec{x})$.*

Recall that overall, we are concerned with synthesising concrete values for the undetermined $q_i \in \mathbb{R}_{\geq 0}$ stemming from templates $\kappa(\vec{b}) \triangleq \sum q_i \cdot b_i$, so that all the generated constraints become valid. By reducing inequalities over polynomials to inequalities of the constituting coefficients and fixing indices I, via Proposition 8.3 this synthesis can be formulated as a satisfiability problem over arithmetical expressions over variables $q_i \geq 0$. In turn, this problem can be solved with an off-the-shelf SMT solver supporting `QF_NRA`. We illustrate this with our running example.

Example 8.4 (Example 8.2 continued). Unfolding the first premise $b = 1$ to $0 \leq b - 1 \wedge 0 \leq 1 - b$ as discussed above, the final remaining constraint (7) from the running example simplifies to

$$0 \leq b - 1 \wedge 0 \leq 1 - b \wedge 0 \leq b \implies 0 \leq 2 \cdot q \cdot b - q - 2 \,.$$

We restrict to the multi-indices $(1,0,0)$, $(0,1,0)$ and $(0,0,1)$, that is, we intend to express $2 \cdot q \cdot b - q - 2$ as a linear combination of the three functions $b-1$, $1-b$ and b, respectively, represent the conjuncts in the assumption. Proposition 8.3 yields the following obligation

$$2 \cdot q \cdot b - q - 2 = c_{(1,0,0)} \cdot (b-1) + c_{(0,1,0)} \cdot (1-b) + c_{(0,0,1)} \cdot b \,,$$

which, subject to the side-condition $0 \leq q$, holds with $q = 2$, $c_{(1,0,0)} = 4$ and $c_{(0,1,0)} = c_{(0,0,1)} = 0$. Theorem 8.1 now translates this solution to a solution $\kappa(x) \triangleq 2 \cdot x$ for the initial constraint in Example 8.2. The constraint solving mechanism established this way thus derives the (optimal) bound given by hand in Section 7.1.

8.2 Experimental Evaluation

To assess our implementation, we have conducted an experimental evaluation of `eco-imp` on the benchmarks of [Ngo et al. 2018; Wang et al. 2019]. We have elided example `recursive` from [Ngo et al. 2018] as we have not yet incorporated support for recursive definitions into `eco-imp`. This is planned for a future release of the prototype. The prototype of Wang et al. [2019] can also deal with some programs featuring negative costs, corresponding examples have been also removed from our testbed. We have extended the resulting testbed with the various examples from this work: In Table 2 (d) we list the program from Figure 1c as `coupons-n` and also instantiate n to constants 10, 50 and 100. Similar, the example from Figure 1a has been parameterised in the number of shares, denoted as `trader-n`, respectively. Examples `nest-i` perform i nested random walks. The example `bridge` performs an unbiased random walk within an interval $[a, b]$.

In Tables 1 and 2, we compare our implementation (column **A**) to the prototypes `Absynth` of Ngo et al. [2018] (column **B**) and Wang et al. [2019] (column **C**). On some of the examples, the

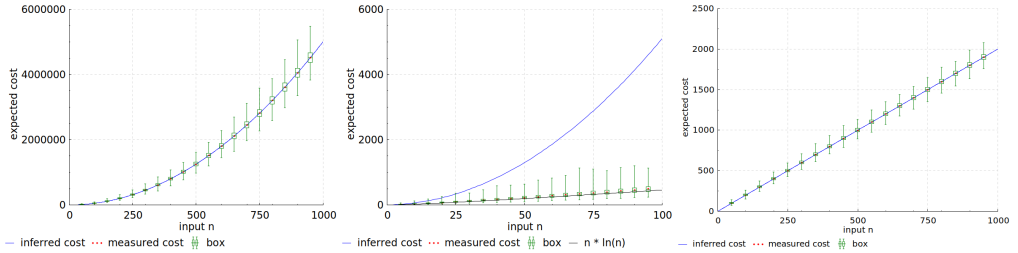

Fig. 7. Comparison of inferred expected cost and measured expected cost using a sample size of 10000, for (from left to right) examples (a) `trader` 2, (b) coupons and (c) `rejection_sampling`. The box plot depicts the minimum, lower- and upper quartile, and maximum cost measured. For `trader` we fix *min* to 100.

tool `Absynth` does not provide an upper bound in a reasonable amount of time; as timeout we use 5 minutes; corresponding examples have been marked with —. Wrt. the tool by Wang et al. [2019], we reproduce the experimental evaluation reported in Wang et al. [2019] and slightly extend them by results on the example `bridge`. Further, we provide timing information on the accessible part of the benchmark in [Ngo et al. 2018]. However, the tool requires extensive manual invariant annotations and various examples were not amendable to an experimental evaluation. Corresponding rows are marked with —. Execution times are obtained on an Intel i7-7600U CPU with 2.8GHz and 16GB RAM.

Precision. The bounds reported in Table 1 and 2 are to a great extent overlapping with those inferred by the `Absynth` tool from Ngo et al. [2018] and, where available, by the prototype of Wang et al. [2019]. In particular, the bounds for all but three example are on the same order of magnitude. In example `prnes`, `eco-imp` infers a quadratic rather than a linear bound. This example uses non-determinism in a way non-beneficial to a separated cost and value analysis, similar to Example 7.3. Concerning example `C4B_t30` our tool infers an asymptotic optimal linear bound, while `Absynth` yields a quadratic one. Whereas `Absynth` uses a combination of linear and non-linear base functions, our backtracking-based algorithm for selecting base functions succeeds on this example using purely linear base functions. A similar observation carries over to example `queueing-network` (depicted in Figure 4a). While the bound derived by `eco-imp` equals the one from [Wang et al. 2019] upto a constant, it is significantly more precise than the bound obtained by the `Absynth` tool.[2] These results are remarkable, as the precision of `Absynth` is outstanding. It is well-understood that a modular analysis, like ours, often comes with the drawback of less precision. Noteworthy is our analysis of `rejection_sampling` depicted in Figure 4b, due to Kaminski et al. [2016], where `eco-imp`'s precision equals the measured expected cost, cf. Figure 7.

Speed. Across the benchmark, where we have annotated the speedup in parenthesis besides execution times, `eco-imp` outperforms the other tools. While our tool achieves already significant speedups in the linear benchmark from [Ngo et al. 2018], the gains are most pronounced in examples with nested loops and non-linear bounds (Table 2 (b)—(c)). The only exception to this assessment are the examples `queueing-network` and `prnes`, both single-loop programs. Concerning the former, our tool performs several backtracking steps to derived the asymptotic optimal bound mentioned above. Also the use of large constants within the `SMT`-solving stage explains the longer execution time. Concerning `prnes`, while our tool performs significantly faster than the implementation

[2]Note that the bound reported in [Wang et al. 2019] and also produced by their tool contains terms depending on constant values $l1$, $l2$, and i, representing the length of the queues and the loop counter, respectively. In our presentation, however, we have set these values to their initial values to ease comparability.

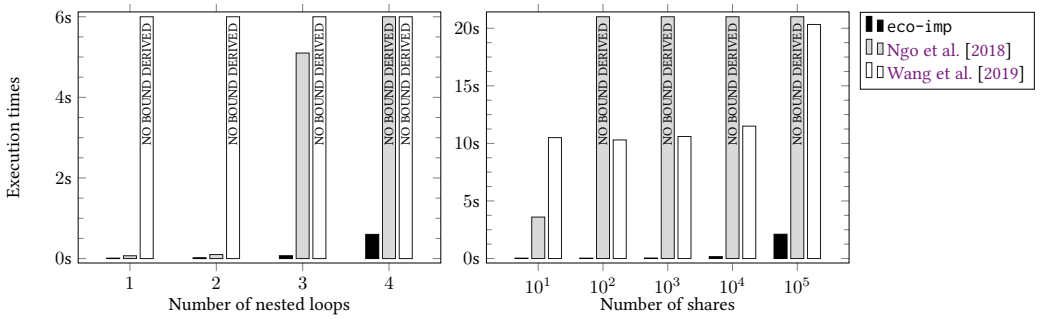

Fig. 8. Execution times of the three analysers on parametric examples (a) `nest-i` consisting of i nested loops, each performing a biased random walk and (b) `trader-n` from Figure 1a with number of shares fixed to n.

of Wang et al., it is slightly less performant than `Absynth`. Due to non-determinism underlying the example, as outlined above, `eco-imp` has to resort to non-linear bounding functions, making the constraint solving stage more involved.

This assessment remains intact when comparing to the implementation of Wang et al. on the subset of examples treated by Wang et al. [2019]. Note however that differences in execution times are even stronger pronounced.

Scalability. In Figure 8, we show the results of `eco-imp` on parametric examples in comparison to the prototypes of Ngo et al. [2018] and Wang et al. [2019]. The left figure plots execution times in relation to the number of nested loops. Similarly, the figure on the right depicts the effects on extending the number of shares in $\texttt{C}_{\texttt{trader}}$. See also Table 1 (d). Our results are competitive and highlight sharply the prime benefit of our implementation, viz, its modularity. As clear indication of the scalability of our tool, the execution times of `eco-imp` are barely visible in the charts in contrast to runtime of existing tools, if the tools can handle the examples at all.

Deterministic Programs. Finally, we also point out that `eco-imp` is competitive with regards to automated complexity analysis tools on deterministic programs. For illustration, we complemented the experimental evaluation from Carbonneaux et al. [2015], comparing their tools C^4B, `KoAT` [Brockschmidt et al. 2014], `Rank` [Alias et al. 2010] and `Speed` [Gulwani et al. 2009], on the 34 deterministic integer programs from their benchmark.[3] Here, C^4B outperforms the mentioned tools by a great margin, as C^4B can solve at least 10 more examples in each comparison. In comparison `eco-imp` can derive bounds for all examples but `t62`. However, since C^4B is constrained to linear bounds and consequently the testbed features only examples with linear cost, `eco-imp` cannot benefit from its modularity on this benchmark.

9 RELATED WORKS

We restrict our focus on related work concerned with the analysis of *bounded expected resource usage* of (non-deterministic) probabilistic imperative programs.

Very briefly, we refer to the extensive literature of analysis methods for (*bounded*) or *almost-sure termination* for (non-deterministic) probabilistic programs that have been introduced in the last years. These have been provided in the form of *abstract interpretations* [Chakarov and Sankaranarayanan 2014; Monniaux 2001]; *martingales*, eg., ranking super-martingales [Agrawal et al. 2018; Brázdil et al. 2015; Chakarov and Sankaranarayanan 2013; Chatterjee et al. 2016, 2017a,b; Esparza et al. 2005; Takisaka et al. 2018; Wang et al. 2019]; or equivalently *Lyapunov ranking*

[3]Full experimental evidence is available from the homepage of `eco-imp`.

functions [Bournez and Garnier 2005]; *model checking* [Katoen 2016]; *program logics* [Kaminski et al. 2018; 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. 2016]; and finally *type systems* [Avanzini et al. 2019; Breuvart and Dal Lago 2018].

Probabilistic resource analysis. Kaminski et al. generalises Dijkstra's wp-calculus to an expected runtime transformer ert. The obtained resource analysis expresses is smoothly applicable in a variety of case studies, in particular to the program $\texttt{C}_{\texttt{coupons}}$. Our expected cost transformer constitutes a generalisation of the ert-calculus. In particular, note that evalue[C] coincides with the weakest precondition transformer wp[C] by Kaminski et al. [2018] on *fully probabilistic programs*, ie. those without non-deterministic choice. In the presence of mixed-sign, unbounded updates, the ert-calculus has been extended by exploiting an absolute convergence criterion, cf. Kaminski and Katoen [2017]. A bulk of research in the literature concentrates on specific forms of martingales or Lyapunov ranking functions. Central to our work is the observation, that these approaches can be suited to a variety of quantitative program properties, see Takisaka et al. [2018] for an overview. We also mention Chatterjee et al. [2017a] where inference procedures to solve recurrences stemming from probabilistic programs are proposed.

Automation. Notably, the Absynth prototype by Ngo et al. [2018], implements a weakest precondition calculus for reasoning about expected costs. Our tool eco-imp generalises this implementation and provides a modular and thus a more efficient and scalable alternative. Also martingale based techniques have been implemented, eg., by Chatterjee et al. [2016] and more recently by Wang et al. [2019]. A novelty of Wang et al. [2019] is the established polynomial-time analysis and incorporation of a cost model which allows for unbounded *negative* and *positive* costs, as long as the variable updated are bounded, while demands (almost-sure) termination as prerequisite. Exploiting the approach by Kaminski and Katoen [2017], we expect the extensibility of our approach to negative costs as well. Our work overcomes the whole-program analysis established by these tools and provides a modular and scalable framework. Further, dynamic distributions are supported. As vindicated by the above provided experimental assessment, modularity significantly speeds up the analysis (see Section 8).

10 CONCLUSION

We established an automated expected resource analysis of non-deterministic, probabilistic imperative programs. As indicated by the formal hardness of this problem by Kaminski and Katoen [2015], this is challenging. While previous approaches only feature a whole-program analysis to synthesis precise cost functions, we present a novel modular framework for the automated analysis. Our prototype eco-imp is more efficient and scales better than existing tools. Its algorithmic superiority is eg. attested by the fact that the motivating example by Ngo et al. [2018] can be handled *three orders of magnitudes* faster.

To establish these highlights, (i) we exploit a novel operational semantics in terms of *weighted probabilistic abstract reduction systems* Avanzini et al. [2020]; Avanzini and Yamada [2020]; (ii) we establish modularity by generalising a weakest precondition calculus pioneered by Kaminski et al. [2018] to an *alternating expected cost and value analysis*; and (iii) painstaking attention to detail in the implementation with a focus on efficiency.

Future theoretical advances are needed to incorporate the treatment of presence of mixed-sign, unbounded updates, in order to properly deal with negative costs. Practical goals are the extension of the synthesis capabilities of our prototype towards *lower bounds* (following Ngo et al. [2017]) and further, real-world examples.

Table 1. Automatically derived bounds on the expected cost and execution times via eco-imp (**A**), [Ngo et al. 2018] (**B**), and [Wang et al. 2019] (**C**).

Program	Inferred Bound A	Inferred Bound B	Inferred Bound C	Runtime in secs. (speedup) A	Runtime in secs. (speedup) B	Runtime in secs. (speedup) C
(a) linear benchmark from [Ngo et al. 2018]						
2drwalk	$2\langle n-d+1\rangle$	$2\langle n-d+1\rangle$	–	0.026	0.286 (11)	–
bayesian_network	$5\langle n\rangle$	$5\langle n\rangle$	–	0.002	0.127 (63)	–
ber	$2\langle n-x\rangle$	$2\langle n-x\rangle$	$2(n-x)$	0.001	0.014 (14)	6.684 (6684)
bin	$^{1.024}/_{1.023}\langle n-x\rangle$	$0.2\langle 9+n\rangle$	$0.91(n-x)-0.91$	0.006	0.152 (25)	6.303 (1050)
C4B_t09	$41\langle x\rangle$	$8.27\langle x\rangle$	–	0.006	0.079 (13)	–
C4B_t13	$^{5}/_{4}\langle x\rangle+\langle y\rangle$	$1.25\langle x\rangle+\langle y\rangle$	$1.25x+y-1.25$	0.005	0.025 (5)	8.527 (1705)
C4B_t15	$2\langle x+1\rangle$	$\langle x-1\rangle+\langle x\rangle$	–	0.006	0.026 (4)	–
C4B_t19	$2\langle i-100\rangle+\langle i+k+51\rangle$	$2\langle i\rangle+\langle i+k+51\rangle$	–	0.005	0.022 (4)	–
C4B_t30	$\langle y+2\rangle+\langle x\rangle$	$0.33\langle x+2\rangle+0.167\langle 2+x\rangle\langle 2+y\rangle$	–	0.003	0.240 (80)	–
C4B_t61	$7+^{160}/_{19}\langle l-7\rangle$	$0.06\langle l-1\rangle+\langle l\rangle$	–	0.005	0.036 (7)	–
condand	$2\langle m\rangle$	$2\langle m\rangle$	$m+n-1$	0.002	0.019 (9)	8.037 (4018)
cooling	$^{21}/_{10}\langle 1+t\rangle+\langle mt-st\rangle$	$0.42\langle 5+t\rangle+\langle mt-st\rangle$	–	0.015	0.083 (6)	–
coupon	25	15	–	0.003	0.045 (15)	–
cowboy_duel	$^{6}/_{5}$	1.2	–	0.001	0.033 (33)	–
cowboy_duel_3way	$^{83}/_{24}$	2.083	–	0.004	0.149 (37)	–
fcall	$2\langle n-x\rangle$	$2\langle n-x\rangle$	–	0.001	0.021 (21)	–
filling_vol	$^{3}/_{5}\langle 1+vTF\rangle+^{3}/_{5}\langle vTF\rangle$	$0.33\langle 10+vTF\rangle+0.33\langle vTF\rangle$	–	0.006	0.230 (38)	–
geo	1	5	–	0.001	0.018 (18)	–
hyper	$^{145}/_{28}\langle n-x-1\rangle$	$5\langle n-x\rangle$	–	0.002	0.034 (17)	–
linear01	$\langle x-1\rangle$	$0.6\langle x\rangle$	$0.6x$	0.001	0.011 (11)	6.392 (6392)
no_loop	5	5	–	0.001	0.010 (10)	–
prdwalk	$^{8}/_{3}\langle n-x\rangle$	$1.14\langle 4+n-x\rangle$	$1.16(n-x)-1.16$	0.002	0.027 (13)	6.650 (3325)
prnes	$^{50}/_{9}\langle y-99\rangle\langle -n\rangle+^{250.000}/_{81}\langle -n\rangle^2$	$68.48\langle -n\rangle+0.05\langle y\rangle$	$0.05y-68.48n$	0.126	0.034 (0)	7.789 (62)
prseq	$6\langle x-y-2\rangle+3\langle y-9\rangle$	$1.65\langle x-y\rangle+0.15\langle y\rangle$	–	0.010	0.030 (3)	–
prseq_bin	$^{32}/_{5}\langle x-y-2\rangle+3\langle y-9\rangle$	$1.65\langle 1+x-y\rangle+0.15\langle y\rangle$	–	0.016	0.038 (2)	–
prspeed	$^{4}/_{3}\langle n-x-2\rangle+2\langle m-y\rangle$	$0.067\langle n-x\rangle+2\langle m-y\rangle$	–	0.013	0.033 (3)	–
race	$^{11}/_{6}\langle t-h+3\rangle$	$0.67\langle t-h+9\rangle$	$0.571(t-h)$	0.010	0.048 (5)	6.936 (694)
rdseql	$^{9}/_{4}\langle x\rangle+\langle y\rangle$	$2.25\langle x\rangle+\langle y\rangle$	$2.25x+y$	0.007	0.032 (5)	7.396 (1057)
rdspeed	$^{4}/_{3}\langle n-x-2\rangle+2\langle m-y\rangle$	$0.67\langle n-x\rangle+2\langle m-y\rangle$	–	0.012	0.096 (8)	–
rdwalk	$2\langle n-x+1\rangle$	$2\langle n-x+1\rangle$	$2(n-x)-2$	0.003	0.058 (19)	6.497 (2166)
rejection_sampling	$2\langle n\rangle$	$2\langle n\rangle$	–	0.002	1.221 (610)	–
rfind_lv	2	2	–	0.001	0.015 (15)	–
rfind_mc	$\langle k\rangle$	$\langle k\rangle$	–	0.002	0.019 (9)	–
robot	$^{5}/_{4}\langle n\rangle$	$0.38\langle 6+n\rangle$	–	0.006	1.546 (258)	–
roulette	$^{74}/_{15}\langle 10.011-n\rangle$	$4.933\langle 10.010-n\rangle$	–	0.049	0.139 (3)	–

Table 2. Automatically derived bounds on the expected cost and execution times via eco-imp (**A**), [Ngo et al. 2018] (**B**), and [Wang et al. 2019] (**C**).

Program	Inferred Bound A	Inferred Bound B	Inferred Bound C	Runtime in secs. (speedup) A	Runtime in secs. (speedup) B	Runtime in secs. (speedup) C
(a) linear benchmark from [Ngo et al. 2018] (continued)						
simple_game	$2\langle x+1\rangle$	$2\langle x+1\rangle$	–	0.006	0.045 (7)	–
sprdwalk	$2\langle n-x\rangle$	$2\langle n-x\rangle$	$2(n-x)$	0.001	0.019 (19)	6.155 (6155)
trapped_miner	$^{15}/_{2}\langle n\rangle$	$7.5\langle n\rangle$	–	0.002	0.052 (26)	–
(b) Non-linear benchmark from [Ngo et al. 2018]						
complex	$\langle w\rangle + 18\langle M\cdot N+N\rangle\langle N\rangle + 3\langle y\rangle$	$\langle w\rangle + 18\langle M\rangle\langle N\rangle + 9\langle N\rangle + 3\langle y\rangle$	–	0.029	0.809 (28)	–
multirace	$\langle m+1\rangle\langle n\rangle$	$2\langle m\rangle\langle n\rangle + 4\langle n\rangle$	–	0.013	2.190 (168)	–
pol04	$^{15}/_{2}\langle x\rangle + {}^{9}/_{2}\langle x\rangle^2$	$7.5\langle x\rangle + 4.5\langle x\rangle^2$	$3.125x^2 + 5.31x$	0.007	0.261 (37)	8.626 (1232)
pol05	$^{3}/_{4}\langle x\rangle + {}^{3}/_{2}\langle x\rangle^2$	$\langle x\rangle + \langle x\rangle^2$	$0.5x^2 + 2.5x$	0.007	0.150 (21)	8.480 (1211)
pol06	$\langle 1+m\rangle\langle 1+p\rangle + \langle p-m\rangle\langle 1+p\rangle$	$0.5\langle p-m\rangle + \langle p-m\rangle\langle p\rangle + \langle p-m\rangle^2$	–	0.040	1.567 (39)	–
pol07	$^{3}/_{2}\langle n-1\rangle^2$	$1.5\langle n-1\rangle\langle n-2\rangle$	–	0.015	0.561 (37)	–
rdbub	$3\langle n\rangle^2$	$3\langle n\rangle^2$	n^2+n	0.006	0.092 (15)	9.374 (1562)
trader(-20)	$20\langle 1+m\rangle\langle p-m\rangle + 10\langle p-m\rangle^2$	$20\langle m\rangle\langle p-m\rangle + 10\langle p-m\rangle + 10\langle p-m\rangle^2$	$10(p^2-m^2-2p+4m+1)$	0.030	119.464 (3982)	10.420 (347)
(c) additional examples from [Wang et al. 2019]						
2drobot	$^{1}/_{2}\langle 1+a-b\rangle\langle a-b\rangle + 2\langle 1+a-b\rangle^2 + {}^{1267}/_{18}\langle 6+a-b\rangle$	–	$1.73a^2 - 3.46ab + 31.45a + 1.73b^2 - 31.45a + 126.52$	1.760	–	11.621 (7)
queueing-network	$\frac{1.132.981}{7.411.500}\langle n+1\rangle + {}^{125}/_{243} + {}^{125}/_{244}$	$0.05\langle n+1\rangle + 0.014\langle n+1\rangle^2$	$0.049n$	2.215	1.286 (1)	78.191 (35)
(d) examples from this work						
bridge	$\langle x-a\rangle\langle b-x\rangle$	–	$ax+bx+a-x^2-b-ab+1$	0.005	–	8.173 (1635)
nest-1	$4\langle n\rangle$	$2\langle n+1\rangle$	–	0.011	0.070 (6)	–
nest-2	$4\langle n\rangle + 16\langle n\rangle^2$	$2\langle 1+n\rangle + 4\langle 1+n\rangle^2$	–	0.019	0.097 (5)	–
nest-3	$2\langle 2+n\rangle + 16\langle n\rangle^2 + 64\langle n\rangle^3$	$2\langle 1+n\rangle + 4\langle 1+n\rangle^2 + 8\langle 1+n\rangle^3$	–	0.068	5.130 (75)	–
nest-4	$144\langle 2+n\rangle\langle n\rangle + 10\langle 2+n\rangle + 254n^4$	–	–	0.554	–	–
trader-10	$10\langle 1+m\rangle\langle p-m\rangle + 5\langle p-m\rangle^2$	$10(\langle m\rangle\langle p-m\rangle + \langle p-m\rangle^2) + 5\langle p-m\rangle$	$5(p^2-m^2+p+m-2)$	0.025	3.638 (146)	10.460 (418)
trader-100	$100\langle 1+m\rangle\langle p-m\rangle + 50\langle p-m\rangle^2$	–	$50(p^2-m^2+4m-2p+1)$	0.027	–	10.312 (382)
trader-1000	$1.000\langle 1+m\rangle\langle p-m\rangle + 500\langle p-m\rangle^2$	–	$500(p^2-m^2+4m-2p+1))$	0.039	–	10.624 (272)
trader-10000	$10.000\langle 1+m\rangle\langle p-m\rangle + 5.000\langle p-m\rangle^2$	–	$5.000(p^2-m^2+4m-2p+1)$	0.163	–	11.489 (70)
trader-100000	$100.000\langle 1+m\rangle\langle p-m\rangle + 50.000\langle p-m\rangle^2$	–	$50.000(p^2-m^2+4m-2p+1)$	2.113	–	20.332 (10)
coupons-10	110	1550	–	0.015	27.011 (1801)	–
coupons-50	2250	–	–	0.304	–	–
coupons-100	10100	–	–	1.141	–	–
coupons-n	$\langle n\rangle + {}^{1}/_{2}\langle n\rangle^2$	not supported	not supported	0.195	–	–

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

E. Albert, P. Gordillo, A. Rubio, and I. Sergey. 2019. Running on Fumes - Preventing Out-of-Gas Vulnerabilities in Ethereum Smart Contracts Using Static Resource Analysis. In *Proc. of 13th VECoS (LNCS, Vol. 11847)*. Springer, 63–78. https://doi.org/10.1007/978-3-030-35092-5_5

C. Alias, A. Darte, P. Feautrier, and L. Gonnord. 2010. Multi-dimensional Rankings, Program Termination, and Complexity Bounds of Flowchart Programs. In *Proc. of 17th SAS (LNCS, Vol. 6337)*. Springer, 117–133. https://doi.org/10.1007/978-3-642-15769-1_8

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

M. Avanzini, U. Dal Lago, and A. Yamada. 2020. On Probabilistic Term Rewriting. *SCP* 185 (2020), 102338. https://doi.org/10.1016/j.scico.2019.102338

M. Avanzini, G. Moser, and M. Schaper. 2016. TcT: Tyrolean Complexity Tool. In *Proc. of 2nd TACAS (LNCS)*. Springer, 407–423. https://doi.org/10.1007/978-3-662-49674-9_24

M. Avanzini and A. Yamada. 2020. *Weighted Rewriting*. Technical Report. NII and INRIA.

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

A. Ben-Amram. 2011. Monotonicity Constraints for Termination in the Integer Domain. *LMCS* 7, 3 (2011), 1–43. https://doi.org/10.2168/LMCS-7(3:4)2011

A. Ben-Amram. 2015. Mortality of Iterated Piecewise Affine Functions over the Integers: Decidability and Complexity. *Computability* 4, 1 (2015), 19–56. https://doi.org/10.3233/COM-150032

A. Ben-Amram and G. Hamilton. 2019. Tight Worst-Case Bounds for Polynomial Loop Programs. In *Proc. of 22th FOSSACS (LNCS, Vol. 11425)*. Springer, 80–97. https://doi.org/10.1007/978-3-030-17127-8_5

A. M. Ben-Amram and L. Kristiansen. 2012. On the Edge of Decidability in Complexity Analysis of Loop Programs. *JFCS* 23, 7 (2012), 1451–1464. https://doi.org/10.1142/S0129054112400588

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

T. Brázdil, S. Kiefer, A. Kucera, and I.H. Vareková. 2015. Runtime Analysis of Probabilistic Programs with Unbounded Recursion. *JCSS* 81, 1 (2015), 288–310. https://doi.org/10.1016/j.jcss.2014.06.005

F. Breuvart and U. Dal Lago. 2018. On Intersection Types and Probabilistic Lambda Calculi. In *Proc. of 20th PPDP*. ACM, 8:1–8:13. https://doi.org/10.1145/3236950.3236968

M. Brockschmidt, F. Emmes, S. Falke, C. Fuhs, and J. Giesl. 2014. Alternating Runtime and Size Complexity Analysis of Integer Programs. In *Proc. of 20th TACAS (LNCS, Vol. 8413)*. Springer, 140–155. https://doi.org/10.1007/978-3-642-54862-8_10

M. Brockschmidt, F. Emmes, S. Falke, C. Fuhs, and J. Giesl. 2016. Analyzing Runtime and Size Complexity of Integer Programs. *TOPLAS* 38, 4 (2016), 13:1–13:50. https://doi.org/10.1145/2866575

Q. Carbonneaux, J. Hoffmann, and Z. Shao. 2015. Compositional Certified Resource Bounds. In *Proc. of 36th PLDI*. ACM, 467–478. https://doi.org/10.1145/2813885.2737955

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. 2013. Probabilistic Program Analysis with Martingales. In *Proc. of 25th CAV (LNCS, Vol. 8044)*. Springer, 511–526. https://doi.org/10.1007/978-3-642-39799-8_34

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

K. Chatterjee, H. Fu, and A. K. Goharshady. 2016. Termination Analysis of Probabilistic Programs Through Positivstellensatz's. In *Proc. of 28th CAV (LNCS, Vol. 9779)*. Springer, 3–22. https://doi.org/10.1007/978-3-319-41528-4_1

K. Chatterjee, H. Fu, and A. Murhekar. 2017a. Automated Recurrence Analysis for Almost-Linear Expected-Runtime Bounds. In *Proc. of 29th CAV (LNCS, Vol. 10426)*. Springer, 118–139. https://doi.org/10.1007/978-3-319-63387-9_6

K. Chatterjee, P. Novotný, and D. Zikelic. 2017b. Stochastic Invariants for Probabilistic Termination. In *Proc. of 44th POPL*. ACM, 145–160. https://doi.org/10.1145/3093333.3009873

J. Cohen and C. Zuckerman. 1974. Two Languages for Estimating Program Efficiency. *Comm. ACM* 17, 6 (1974), 301–308. https://doi.org/10.1145/355616.361015

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

P. Cousot and R. Cousot. 2002. Modular Static Program Analysis. In *Proc. of 11th CC (LNCS, Vol. 2304)*. Springer, 159–178. https://doi.org/10.1007/3-540-45937-5_13

J. Dean and S. Ghemawat. 2008. MapReduce: Simplified Data Processing on Large Clusters. *Commun. ACM* 51, 1 (2008), 107–113. https://doi.org/10.1145/1327452.1327492

T. Dillig. 2011. *A Modular and Symbolic Approach to Static Program Analysis.* Ph.D. Dissertation. Stanford University.

J. Esparza, A. Kucera, and R. Mayr. 2005. Quantitative Analysis of Probabilistic Pushdown Automata: Expectations and Variances. In *Proc. of 20th LICS*. IEEE, 117–126. https://doi.org/10.1109/LICS.2005.39

T. Fiedor, L. Holík, A. Rogalewicz, M. Sinn, T. Vojnar, and F. Zuleger. 2018. From Shapes to Amortized Complexity. In *Proc. of 19th VMCAI (LNCS, Vol. 10747)*. Springer, 205–225. https://doi.org/10.1007/978-3-319-73721-8_10

F. Frohn and J. Giesl. 2017. Complexity Analysis for Java with AProVE. In *Proc. of the 13th IFM (LNCS, Vol. 10510)*. Springer, 85–101. https://doi.org/10.1007/978-3-319-66845-1_6

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

S. Gulwani, K.K. Mehra, and T.M. Chilimbi. 2009. SPEED: Precise and Efficient Static Estimation of Program Computational Complexity. In *Proc. of 36th POPL*. ACM, 127–139. https://doi.org/10.1145/1594834.1480898

S. Gulwani and A. Tiwari. 2006. Combining Abstract Interpreters. In *Proc. of PLDI'06*. ACM, 376–386. https://doi.org/10.1145/1133981.1134026

S. Gulwani and F. Zuleger. 2010. The Reachability-Bound Problem. In *Proc. of PLDI'10*. ACM, 292–304. https://doi.org/10.1145/1809028.1806630

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

M. D. Hill and M.R. Marty. 2008. Amdahl's Law in the Multicore Era. *Computer* 41, 7 (2008), 33–38. https://doi.org/10.1109/MC.2008.209

N. Hirokawa and G. Moser. 2008. Automated Complexity Analysis Based on the Dependency Pair Method. In *Proc. of 4th IJCAR (LNAI, Vol. 5195)*. Springer, 364–380. https://doi.org/10.1007/978-3-540-71070-7_32

J. Hoffmann, A. Das, and S.-C. Weng. 2017. Towards Automatic Resource Bound Analysis for OCaml. In *Proc. of 44th POPL*. ACM, 359–373. https://doi.org/10.1145/3009837.3009842

N. D. Jones and L. Kristiansen. 2009. A Flow Calculus of *mwp*-Bounds for Complexity Analysis. *TOCL* 10, 4 (2009), 28:1–28:41. https://doi.org/10.1145/1555746.1555752

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

C. Kim and A. K. Agrawala. 1989. Analysis of the Fork-join Queue. *IEEE TC* 38, 2 (1989), 250–255. https://doi.org/10.1109/12.16501

D. Kozen. 1981. Semantics of Probabilistic Programs. *J. Comput. Syst. Sci.* 22, 3 (1981), 328–350.

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

A. Levitin. 2007. *The Design and Analysis of Algorithms.* Pearson.

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

D. A. Menascé. 2004. Response-Time Analysis of Composite Web Services. *IEEE IC* 8, 1 (2004), 90–92. https://doi.org/10.1109/MIC.2004.1260710

M. Mitzenmacher and E. Upfal. 2005. *Probability and Computing: Randomized Algorithms and Probabilistic Analysis.* Cambridge University Press.

D. Monniaux. 2001. An Abstract Analysis of the Probabilistic Termination of Programs. In *Proc. of 8th SAS (LNCS, Vol. 2126)*. Springer, 111–126. https://doi.org/10.1007/3-540-47764-0_7

G. Moser and M. Schaper. 2018. From Jinja Bytecode to Term Rewriting: A Complexity Reflecting Transformation. *IC* 261, Part (2018), 116–143. https://doi.org/10.1016/j.ic.2018.05.007

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

V. Chan Ngo, M. Dehesa-Azuara, M. Fredrikson, and J. Hoffmann. 2017. Verifying and Synthesizing Constant-Resource Implementations with Types. In *Proc. of 38th S&P*. 710–728.

F. Olmedo, B. L. Kaminski, J.-P. Katoen, and C. Matheja. 2016. Reasoning about Recursive Probabilistic Programs. In *Proc. of 31nd LICS*. ACM, 672–681. https://doi.org/10.1145/2933575.2935317

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

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. 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

M. Sinn, F. Zuleger, and H. Veith. 2017. Complexity and Resource Bound Analysis of Imperative Programs Using Difference Constraints. *JAR* 59, 1 (2017), 3–45. https://doi.org/10.1007/s10817-016-9402-4

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

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

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

B. Wegbreit. 1975. Mechanical Program Analysis. *Comm. ACM* 18, 9 (1975), 528–539. https://doi.org/10.1145/361002.361016

B. Wegbreit. 1976. Verifying Program Performance. *JACM* 23, 4 (1976), 691–699. https://doi.org/10.1145/321978.321987

R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenstrom. 2008. The Worst Case Execution Time Problem - Overview of Methods and Survey of Tools. *TECS* 7, 3 (2008), 1–53. https://doi.org/10.1145/1347375.1347389

R. Wilhelm and D. Grund. 2014. Computation Takes Time, But How Much? *Comm. ACM* 57, 2 (2014), 94–103. https://doi.org/10.1145/2500886

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