A Modular Cost Analysis for Probabilistic Programs

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

We present a novel methodology for the automated resource analysis of non-deterministic, probabilistic imperative programs, which gives rise to a unique modular approach. Program fragments are analysed in full independence. Further, the results established allow us to incorporate sampling from dynamic distributions, making our analysis applicable to realistic examples.

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 prototypes’ algorithmic superiority. Our experiments show that our tool runs typically at least one order of magnitude faster than comparable tools. On realistic examples, it is even 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 the tool. This effects into more realistic examples, whose expected cost analysis can be thus performed fully automatically. In particular, our tool is the first establishing an automated analysis of the Coupon Collector’s problem.

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

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). Resources are broadly construed and encompass runtime, memory usage, stack and heap sizes, etc. 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, for example 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, that is, whole-program analyses can be overcome. This significantly speeds up the analysis without affecting the precision of the analysis. See for example [Avanzini et al. 2016; Brockschmidt et al. 2016; Frohn and Giesl 2017;
Apart from modularity, the study of non-deterministic programs has proven immensely useful. Non-determinism appears naturally through program abstraction. In particular, in the analysis of imperative programs, program abstractions form an integral part. It is due to program abstractions—sometimes also referred to as program transformations—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 $\lambda$-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. The provided theoretical constructions, namely the resolution of non-determinism through dedicated scheduling complicates the principally beautiful foundation via Markov chains. 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 primitives for dynamic distributions, i.e., 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. The results established in this paper overcome all these issues.

We are concerned with an automated average runtime analysis of a prototypical imperative language $\text{pWhile}$ 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 $\text{consume}(e)$ 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]. On the other hand, almost-sure termination is not a pre-condition to our analysis.

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;
- 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 our analysis more realistic; specifically, we allow for uniform sampling from an interval dependent on program states;
A Modular Cost Analysis for Probabilistic Programs

1. while \((p > \min \geq 0)\) {
2. \(b \leftarrow \text{Bernoulli}(\frac{1}{4})\);
3. if \((b)\) \{ \(p \leftarrow p + 1\); \}
4. \(n \leftarrow \text{Uniform}(0, 10)\);
5. while \((n > 0)\) {
6. consume \((p)\); \(n \leftarrow n - 1\) }
7. }

(a) Trader \(C_{\text{trader}}\).

1. \(b \leftarrow 1\);
2. \(x \leftarrow 1\);
3. while \((b = 1)\) {
4. consume \((1)\);
5. \(x \leftarrow x \times 2\);
6. \(b \leftarrow \text{Bernoulli}(\frac{1}{2})\)
7. }

(b) Geometric \(C_{\text{geo}}\).

1. \(coupons \leftarrow 0\);
2. while \((0 \leq coupons < n)\) {
3. consume \((1)\);
4. \(draw \leftarrow \text{Uniform}(1, n)\);
5. if \((draw > coupons)\) {
6. coupons \leftarrow coupons + 1
7. }
}

(c) Coupon Collector \(C_{\text{coupons}}\).

Fig. 1. Motivating Examples.

• 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 refi-able synthesis of cost expressions. In benchmarking, we focus on complex and realistic scenarios, thus emphasising the algorithmic prowess 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 up to 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, later detailed in Section 6. Probabilistic abstract reduction systems, which form the theoretical underpinning, and our imperative language are presented in Section 3 and 4, respectively. Section 5 recalls a weakest precondition calculus due to Kaminski et al. [2018], which is generalised in Section 6 to serve our needs of a modular analysis. In Section 7, 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 8 discusses related work and we conclude in Section 9. The Supplementary Material to this paper contains information on the mathematical background and full proofs of all our results.

2 AUTOMATED EXPECTED COST ANALYSIS AND MODULARITY

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 emphasise the need for modularity in this context.

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 straightforward to probabilistic programs.

Consider the program \(C_{\text{trader}}\) 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 \(\frac{1}{4}\) the price increases (by one), and with probability \(\frac{3}{4}\) the price decreases. After the change of the stock price takes effect, the trader decides to buy up to 10 shares, all with equal probability. The cost for...
the trader is given as the total amount invested. Clearly, this cost is unbounded in some scenarios, namely, when the price \( p \) stays above the minimum price. However, the combined probability of all such scenarios is 0. Worst case bounds are thus not informative for such programs. 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 \( C_{\text{trader}} \) becomes expressible by the recurrence

\[
T(\text{price}) = \frac{1}{44} \cdot \sum_{n=0}^{10} (n \cdot (\text{price} + 1) + T(\text{price} + 1)) + \frac{3}{44} \cdot \sum_{n=0}^{10} (n \cdot (\text{price} - 1) + T(\text{price} - 1)) \\
= 5 \cdot \text{price} - \frac{5}{2} + \frac{1}{44}T(\text{price} + 1) + \frac{3}{44}T(\text{price} - 1).
\]

This is a non-homogeneous linear second-order recurrence, whose closed-form can be computed as \( 5 \cdot \text{price} \cdot (\text{price} + 1) \). While the general solution of the corresponding homogeneous recurrence can be derived directly (see Levi\( \text{t} \) [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 \( C_{\text{trader}} \)—is tedious, error prone and fragile to small changes of the recurrence equations. On the other hand our prototype implementation computes the (asymptotically) optimal upper bound of \( 5 \cdot \text{price} \cdot (\text{price} + 2) \) in milliseconds.

**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 \( C_{\text{geo}} \) depicted in Figure 1b. While also this example potentially diverges, its expected cost—the number of loop iterations—is 2. Second, consider the composition of \( C_{\text{geo}} \) with a program \( D \), whose expected cost is given by \( f(x) \). Then the expected cost of \( C_{\text{geo}} \circ D \), that is, running the two programs in sequence, becomes

\[
\sum_{i=0}^{\infty} \frac{1}{2^i} \cdot f(2^{i+1}) .
\]

When \( f \) grows at least linearly, this sum is infinite. Conclusively, while the expected costs of \( C_{\text{geo}} \) and \( D \) are finite, the expected cost of their composition is not. Generalising a weakest precondition calculus à la Dijkstra’s to an expected runtime transformer \( \text{ert} \), 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 observed by Kaminski et al., this transformer in turn generalises seamlessly to one for reasoning about expected costs. Wrt. a program \( D \), whose expected cost is \( f \), \( \text{ert}[C](f) \) computes the time of first executing \( C \) and then \( D \). In particular, \( \text{ert}[C_{\text{geo}}](f) \) is precisely (\( \dagger \)). The cost of two sequentially composed commands \( C; D \) thus becomes expressible via composition of the transformer: \( \text{ert}[C; D](f) = (\text{ert}[C] \circ \text{ert}[D])(f) \). Similarly, consider the program \( C_{\text{trader}} \). Composability of the analysis allow us to describe the expected costs of the outer loop’s body independently from the expected cost of the outer loop. Hence, the analysis becomes compositional in theory.

To sum up, the \( \text{ert} \)-calculus is a more refined technical tool for the analysis of probabilistic programs than the classical analysis through recurrence relations. The analysis becomes clean and is less fragile. Further, it allows an optimal analysis of a variety of interesting case studies, like eg. the aforementioned program \( C_{\text{coupons}} \) 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 Absynth prototype provided by Ngo et al. [2018] has been the first *automated* expected resource analysis of probabilistic programs. For instance, Absynth infers the manual bound on the expected cost of $C_{\text{trader}}$ shown above, fully automatically. The tool provides 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, 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. Here (almost-sure) termination is required. Moreover, their implementation yields a polytime algorithm.

Both tools provide a *whole-program analysis*. Conclusively, neither of these tools is modular and both tools have issues with scalability. To wit, consider a more realistic variant of $C_{\text{trader}}$, where the stock trader is choosing uniformly up to $100,000$ shares. Due to the modular design of our prototype, the variant can be handled with ease in seconds. However, the Absynth tool is no longer able to compute a bound, while the prototype established by Wang et al. [2019] requires up to 30 seconds of runtime—additionally a significant amount of user guidance is required. Similar issues hold wrt. multiple nested loops (see Section 7). In contrast, we provide a novel modular and efficient automation, inspired by the continuation-passing style analysis method formalised in the ert-calculus, which however is free of the aforementioned scalability issues.

**An Intuitive Textbook Example.** 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 $C_{\text{coupons}}$ is depicted in Figure 1c. Here, coupons represents the number of unique coupons collected; draw samples 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 $\frac{1}{\text{coupons}} + 1$. In particular, the expected cost is finite.

The full pen-and-paper analysis of $C_{\text{coupons}}$ given by Kaminski et al. [2018]—which provides the optimal expected time bound $O(n \log n)$—spans several pages and requires non-trivial estimations. Thus, non surprisingly, automation poses significant challenges. So far, to the best of our knowledge, the Coupon Collector’s problem has been elusive to any automated analysis. As already mentioned, no existing tool can even *represent* the corresponding code $C_{\text{coupons}}$. To date support for *dynamic probabilistic branching* is lacking. Our prototype eco-$\text{imp}$ admits dynamic distribution and provides the bound $n + \frac{1}{2} \cdot n^2$ fully automatically in *a fraction of a second*. This automated analysis of the expected cost of $C_{\text{coupons}}$ is achieved via a constraint solver that allows for the iterative refinement of cost expressions; thus generic enough to handle the example.

If the distribution is set statically, e.g. to a uniform distribution of 10 coupons, the example becomes expressible by existing tools. Alas, only the Absynth tool can provide a (non-optimal) bound (see Section 7). Still the employed potential functions are not amenable to express the subtle dependency of the expected cost of $C_{\text{coupons}}$ on the (now) static distribution governing the draw. I.e. the generated constraint again grows linearly to the number of coupons. On the other hand, the constraints generated by our tool are succinct. Wrt. tool execution time this implies that our prototype eco-$\text{imp}$ handles a uniform distribution of up to 100 coupons in seconds. On the other
hand the Absynth tool cannot provide bounds for larger numbers, even after several minutes of runtime.

*Achieving Modularity of the Analysis.* The main novel aspect of our work lies in providing a modular (and thus scalable) analysis of probabilistic and non-deterministic programs. Of course, in a probabilistic setting this is not too straightforward, because running a command results in a distribution of possible end states, cf. Fioriti and Hermanns [2015].

We emphasise that composability of the analysis of the expected cost of sequential commands, as provided by the eut-calculus, does not induce composability of the synthesis of the corresponding bounding functions. We illustrate the difference, wrt. $C_{\text{trader}}$. To find closed-forms of the expected cost of loops, one searches for *upper invariants* (see Section 5.2). Concretely, as $C_{\text{trader}}$ features a nested loop, its expected cost wrt. the cost function $f(n, p, \text{min})$ of a continuation is driven by the following inter-dependent constraints (see Section 6).

$$
\begin{align*}
p > \text{min} &\geq 0 \implies O(n, p, \text{min}) \geq \frac{1}{44} \cdot \sum_{n=1}^{10} I(n, p+1, \text{min}) + \frac{3}{44} \cdot \sum_{n=1}^{10} I(n, p-1, \text{min}) \\
\neg(p > \text{min} \geq 0) &\implies O(n, p, \text{min}) \geq f(n, p, \text{min}) \\
\frac{n}{0} &\implies I(n, p, \text{min}) \geq p + I(n-1, p, \text{min}) \\
\neg(n \geq 0) &\implies I(n, p, \text{min}) \geq O(n, p, \text{min}).
\end{align*}
$$

Here, $O(n, p, \text{min})$ and $I(n, p, \text{min})$ stands for the cost before entering the inner and outer loop, respectively. Due to their inter-dependent nature, invariants $I$ and $O$ cannot be considered in isolation and their synthesis degenerates to a whole-program analysis. We refer to the composability of the synthesis of upper invariants as *modularity* of the (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.

The idea, underlying the gist of the methods achieving modularity for the analysis of deterministic programs can be summarised as follows, cf. [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 functions $f$ and $g$ measuring the costs of deterministic programs $C$ and $D$, respectively. These cost functions $f$, $g$ are dependent on the program state $\sigma$ before program execution, that is, $f$ and $g$ depend on the variable assignments at the beginning of $C$ and $D$, respectively. Then $f(\sigma) + g(\sigma')$ gives the resource usage of the sequential command $C;D$, whenever $\sigma'$ denotes the variable assignment after the execution of $C$. Estimating $\sigma'$ in terms of $C$ and $\sigma$, gives rise to a modular analysis, where runtime and size analysis is alternated. The observation, while conceptually simple, is nevertheless immensely useful. We suit it to probabilistic, non-deterministic programs. For that, we focus on expected cost and expected value analysis as substitution for the aforementioned runtime and value analysis. Similarly to the above sketch, we alternate between expected cost and value analysis. Here, we crucially exploit that *concave* cost functions, eg. multi-linear polynomials, distribute over expectations. In this setting, the above mentioned inter-dependent constraints become decomposable, as the synthesis of the bounding function for the inner loop ($I$) and the outer loop ($O$), respectively, can be performed independently. This gives rise to a modular methodology for the automated expected cost analysis of non-deterministic, probabilistic programs.

Our modular analysis is particularly effective in the analysis of nested loops, which has direct consequences for the scalability and speed of the analysis. For example, in comparison to the Absynth prototype—which can be considered among the most efficient implementations of such an analysis to date—we achieve a speed-up of up to three orders of magnitude on such examples. At the same time we retain the precision of the Absynth tool.
3 PROBABILISTIC REDUCTION SYSTEMS

In this section we quickly recap notions and notations of (weighted) probabilistic abstract 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. For brevity, we refer to the weighted variant again as probabilistic abstract reduction systems (PARSs). In the sequel, costs $\mathbb{C}$ are represented as non-negative real numbers extended with $\infty$, i.e. $\mathbb{C} \triangleq \mathbb{R}_+^\infty$, where $\mathbb{R}_+^\infty = \mathbb{R}_+ \cup \{\infty\}$. Note that costs are positive, but unbounded. Otherwise, the below defined expected cost function would be ill-defined.

A (discrete) subdistribution over $A$ is a function $\delta : A \rightarrow \mathbb{R}_+^\infty$ such that $\sum_{a \in A} \delta(a) \leq 1$, and a distribution if $\sum_{a \in A} \delta(a) = 1$. We may write subdistributions $\delta$ as $\langle \delta(a) : a \rangle_{a \in A}$. The set of all subdistributions over $A$ is denoted by $\mathcal{D}(A)$. We restrict ourselves to distributions over countable sets $A$. The expectation of a function $f : A \rightarrow \mathbb{R}_+^\infty$ wrt. a distribution $\delta$ 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 $a \rightarrow \delta$ 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 $a \rightarrow \delta$ (among possibly many) and then sampling the reduct from $\delta$. To allow for non-uniform costs, we endow each rule with a weight $w \in \mathbb{R}_+$, 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 $\rightarrow \cdot \subseteq A \times \mathbb{R}_+ \times \mathcal{D}(A)$, 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 $a \downarrow$. The program $\mathcal{C}_{\text{geo}}$ from Figure 1b, for instance, is modelled by the PARS $\xrightarrow{\text{geo}}$, defined by

$$\mathcal{C}_{\text{geo}}(x) \xrightarrow{1_{\text{geo}}} \langle \frac{1}{2} : 2 \cdot x, \frac{1}{2} : \text{geo}(2 \cdot x) \rangle$$

for all $x \in \mathbb{Z}$.

Following [Avanzini et al. 2019b; Avanzini and Yamada 2020], we define the dynamics of a PARS $\rightarrow$ in terms of a (weight-indexed) reduction relation $\rightarrow \cdot \subseteq \mathcal{M}(A) \times \mathbb{R}_+ \times \mathcal{M}(A)$ over multidistributions $\mathcal{M}(A)$. These are countable multisets $\langle p_i : a_i \rangle_{i \in I}$ 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 $\mu$, $\nu$, . . . The notion of expectation is extended to multidistributions in the natural way by $\mathbb{E}_\mu(f) \triangleq \sum_{(p,a) \in \mu} p \cdot f(a)$. Finally, the reduction relation is inductively find by

$$\frac{a \xrightarrow{w} \delta}{\mu \xrightarrow{w} \delta} \quad \frac{\mu \xrightarrow{[1 : a]} \delta}{\mu \xrightarrow{[1 : a]} \delta} \quad \frac{\mu \xrightarrow{\cup_{i \in I} p_i : a_i} \nu}{\cup_{i \in I} p_i \cdot a_i \xrightarrow{\cup_{i \in I} \delta} \nu}$$

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, $\cup$ refers to the usual notion of multiset union. Put differently, $\mu \xrightarrow{w} \nu$ indicates that some elements $a_i$ with associated probability $p_i$ are replaced by $\delta_i$, 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 $\sum_{i \in I} p_i \cdot w_i$, corresponding to the expected cost of the single reduction step $\mu \xrightarrow{w} \nu$. For instance, the PARS $\xrightarrow{\text{geo}}$ gives rise to the reduction

$$\langle 1 : \text{geo}(1) \rangle \xrightarrow{1_{\text{geo}}} \langle \frac{1}{2} : 2, \frac{1}{2} : \text{geo}(2) \rangle \xrightarrow{\frac{1}{2}_{\text{geo}}} \langle \frac{1}{2} : 2, \frac{1}{4} : 4, \frac{1}{4} : \text{geo}(4) \rangle \xrightarrow{\frac{1}{4}_{\text{geo}}} \cdots$$

We write $\mu \xrightarrow{w} \nu$ if $\mu \xrightarrow{w} \cdots \xrightarrow{w} \nu$ for $w = \sum_{i \in I} w_i$.

**Expected cost.** A reduction from $a \in A$ is a, generally infinite, sequence $\Lambda : \langle 1 : a \rangle \xrightarrow{w_1} \mu_1 \xrightarrow{w_2} \mu_2 \cdots$. The infinite sum $w = \sum_{i \in \mathbb{N}} w_i \in \mathbb{C}$ gives the expected cost of this specific reduction. 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 \( \text{ecost}[\rightarrow]: A \rightarrow \mathbb{C} \), 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
\[
\text{ecost}(\rightarrow)(a) = \sup \{ w : \{1 : a\} \xrightarrow{n} w \}.
\]
For instance, we have \( \text{ecost}[\rightarrow_{\text{geo}}](\text{geo}(x)) = \sup \{ \sum_{i=0}^n \lambda^i | n \in \mathbb{N} \} = \frac{\lambda^n}{1 - \lambda} = 2 \).

**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]).

**Proposition 3.1 ([Avanzini et al. 2019b; Avanzini and Yamada 2020]).** Suppose each rule in the PARS \( \rightarrow \) is attributed weight one. Then \( \text{ecost}[\rightarrow] \) 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 \( \text{ecost}[\rightarrow] \).

### 3.1 Expected Cost Transformers for PARSs

In this section we define an expected cost transformer \( \text{ect}[\rightarrow] \) for arbitrary PARSs \( \rightarrow \). 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 \( \omega \)-CPO, if every \( \omega \)-chain \( a_0 \sqsubseteq a_1 \sqsubseteq \cdots \) has a supremum \( \sup \{ a_n : n \in \mathbb{N} \} \in A \). A function \( f: A \rightarrow B \) between two \( \omega \)-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 \( \omega \)-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 \( f: A \rightarrow A \) is continuous and \( A \) features a least element \( \bot \), the least fixed-point \( \text{lfp}(f) \) of \( f \) is given \( \text{lfp}(f) \triangleq \sup_{n\in\mathbb{N}} f^n(\bot) \) for \( f^n \) the \( n \)-fold composition of \( f \). Let \( C^A \triangleq \{ f : f: A \rightarrow \mathbb{C} \} \) 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 \( f, g \in C^A \) etc. In particular, \( 0 = \lambda a.0 \) and \( \infty = \lambda a.\infty \). The proof of the following is standard.

**Proposition 3.2 (Cost Functions Form an \( \omega \)-CPO).** For any \( A, (C^A, \leq) \) is an \( \omega \)-CPO, with least and greatest element \( 0 \) and \( \infty \), respectively. The supremum of \( \omega \)-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 \( \rightarrow \). 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.
We consider an $\in \tau \models A$ Modular Cost Analysis for Probabilistic Programs 9

Here, probabilistic effects are written as halted with final store $\tau$ to be executed under the current store. Particularly, the incurred cost can depend on the programs state rather than being constant. The non-deterministic choice operator is a no-op otherwise. Here $\chi$ is given as follows.

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

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

**Lemma 3.4 (Central Properties of $\text{ect}[\rightarrow]$).**

(1) continuity: $\text{ect}[\rightarrow](\sup_{n \in \mathbb{N}} f_n) = \sup_{n \in \mathbb{N}} \text{ect}[\rightarrow](f_n)$ for all $\omega$-chains $(f_n)_{n \in \mathbb{N}}$.

(2) monotonicity: $f \leq g \implies \text{ect}[\rightarrow](f) \leq \text{ect}[\rightarrow](g)$.

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

**Theorem 3.5 (Expected Cost via Cost Transformer).**

$$\text{ecost}[\rightarrow] = \text{ect}[\rightarrow](0).$$

**4 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 $\ll$ and where the assignment statement is generalised to one that can sample from a distribution. A command $\text{consume}(e)$ signals the consumption of $e \geq 0$ resource units.

We fix a finite set of integer-valued variables $\text{Var}$. Stores are denoted by $\sigma \in \Sigma \triangleq \text{Var} \rightarrow \mathbb{N}$. With $\phi \in \text{BExp} \triangleq \Sigma \rightarrow \mathbb{B}$, $e \in \text{Exp} \triangleq \Sigma \rightarrow \mathbb{N}$, and $d \in \text{DExp} \triangleq \Sigma \rightarrow \mathcal{D}(\mathbb{Z})$ we denote, Boolean, Integer, and Integer-valued distribution expression over Var, respectively. The syntax of program commands $\text{Cmd}$ is given as follows.

$$\text{C, D ::= x := d | \text{skip} | \text{abort} | \text{consume}(e) | \text{C; D} | \text{if } (\phi) \{ \text{C} \} \{ \text{D} \} | \text{while } (\phi) \{ \text{C} \} | \{ \text{C} \} \ll \{ \text{D} \}.}$$

Commands are fairly standard. The assignment statement $x := d$ samples a value from $d$, i.e. an expression that evaluates to a distribution over integers. This command generalises the usual non-probabilistic assignment $x := e$. The command $\text{consume}(e)$ 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 $\{ \text{C} \} \ll \{ \text{D} \}$ executes either $\text{C}$ or $\text{D}$. 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 $\text{Conf} \triangleq (\text{Cmd} \times \Sigma) \cup \Sigma$. Elements $(\text{C}, \sigma) \in \text{Conf}$ are denoted by $\sigma \triangleright \text{C}$ and signal that the command $\text{C}$ is to be executed under the current store $\sigma$, whereas $\sigma \in \text{Conf}$ indicates that the computation has halted with final store $\sigma$. The (infinite) PARS is depicted in Figure 2. Rules $\sigma \triangleright \text{C} \xrightarrow{\text{DExp} \triangleright} \{1 : \gamma\}$ without probabilistic effect are written as $\sigma \triangleright \text{C} \xrightarrow{\text{Exp}} \gamma$ for brevity. For $\phi \in \text{BExp}$ and $\sigma \in \Sigma$ we denote by $\sigma \triangleright \phi$ that $\phi$ evaluates on $\sigma$ to true. Note that only in Rule (Consume) resources are consumed. Here, $\langle z \rangle \triangleq \max(0, z)$ denotes Macaulay brackets. As we had for PARSs, incurred costs are thus always non-negative.
\[
\sigma \triangleright x := d \Downarrow \{p_1 : \sigma[x/i] \mid p_1 = d(\sigma)(i) > 0\}_{i \in \mathbb{Z}} \quad \text{[ASSIGN]}
\]

\[
\sigma \triangleright \text{skip} \Downarrow \sigma \quad \text{[Skip]}
\]

\[
\sigma \triangleright \text{abort} \Downarrow \emptyset \quad \text{[ABORT]}
\]

\[
\sigma \triangleright \text{consume}(e) \Downarrow \varepsilon(\sigma) \quad \text{[CONSUME]}
\]

\[
\sigma \cdot \text{if } (\phi) \{C\} \{D\} \Downarrow \sigma \cdot C \quad \text{[IF]}
\]

\[
\sigma \cdot \text{if } (\phi) \{C\} \{D\} \Downarrow \sigma \cdot D \quad \text{[IF]}
\]

\[
\sigma \cdot \text{while } (\phi) \{C\} \Downarrow \sigma \cdot C \cdot \text{while } (\phi) \{C\} \quad \text{[WHILE]}
\]

\[
i \in \{1,2\} \quad \sigma \cdot C \cdot D \Downarrow \{p_i : \sigma_i \cdot C_i, D_i \}_{i \in I} \cup \{q_j : \sigma_j \cdot D_j \}_{j \in J} \quad \text{[COMPOSE]}
\]

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

\[
\begin{array}{lll}
\text{consume}(e) & \langle e \rangle + f & f \\
\text{skip} & f & f \\
\text{abort} & 0 & 0 \\
\end{array}
\]

\[
\begin{array}{ll}
x \triangleright d & \lambda v. \llbracket f[x/v] \rrbracket(\sigma) \\
C;D \cdot \text{ect}[C](\text{ect}[D](f)) & \lambda v. \llbracket f[x/v] \rrbracket(\sigma) \\
\text{if } (\phi) \{C\} \{D\} \cdot \llbracket \cdot \rrbracket \cdot \text{ect}[C](f) + \llbracket \cdot \rrbracket \cdot \text{ect}[D](f) & \llbracket \cdot \rrbracket \cdot \text{evalue}[C](f) + \llbracket \cdot \rrbracket \cdot \text{evalue}[D](f) \\
\text{while } (\phi) \{C\} \cdot \text{lfp}(\lambda F. \llbracket \cdot \rrbracket \cdot \text{ect}[C](F) + \llbracket \cdot \rrbracket \cdot f) & \text{lfp}(\lambda F. \llbracket \cdot \rrbracket \cdot \text{evalue}[C](F) + \llbracket \cdot \rrbracket \cdot f) \\
\{C\} \Leftarrow \{D\} & \text{max}(\text{ect}[C](f), \text{ect}[D](f)) \\
\end{array}
\]

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

5 EXPECTED COST AND EXPECTED VALUE TRANSFORMERS

We now suite the 
the-rt-transformer of Kaminski et al. [2016] to an expected cost transformer. To this end, for \( \phi \in \text{BExp} \), we lift Iverson brackets \([ \cdot ]\) to stores, resulting in the cost function \([\phi](\sigma) \triangleq 1\) if \( \sigma \models \phi \), and \([\phi](\sigma) \triangleq 0\) otherwise. In particular, \([\phi] \cdot f + [\neg \phi] \cdot g\) evaluates to \(f(\sigma)\) on stores \(\sigma \models \phi\), and to \(g(\sigma)\) if \(\sigma \not\models \neg \phi\). 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 \text{ect}[\cdot](\cdot): \text{Cmd} \rightarrow \mathbb{C}^\Sigma \rightarrow \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]. The cost \text{ect}[C](f) should be seen as the cost of evaluating \(C\) wrt. to a continuation of expected cost \(f\).

Definition 5.1 (Expected Cost Transformer). The expected cost transformer \text{ect}[\cdot](\cdot): \text{Cmd} \rightarrow \mathbb{C}^\Sigma \rightarrow \mathbb{C}^\Sigma\) for commands \(C\) is defined through the rules given in the second column in Figure 3. For \(C \in \text{Cmd}\), we set \(\text{ecost}[C] \triangleq \text{ect}[C](\emptyset)\) and call \(\text{ecost}[C]\) the expected cost of \(C\).

Remark. The earlier defined expected cost transformer \text{ect}[\rightarrow] wrt. \text{PARS} \rightarrow (see Section 3) abstracts the above definition of the expected cost transformer wrt. \text{programs} \(C\). As we see in equation (‡), 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 \text{consume}(e), a cost of \(e\) is
incurred (provided \(e(\sigma) \geq 0\)) in addition to the cost of the continuation. While skip is a no-op, consequently \(\text{ect}[\text{skip}](f) = f\), the command abort immediately aborts the computation and thus \(\text{ect}[\text{abort}](f) = 0\). Running an assignment \(x := d\) followed by a continuation amounts to first sampling a value \(\sigma\) for \(x\) according to \(d\), say with probability \(p_\sigma\), updating \(x\) in the given store \(\sigma\) to \(\phi\), and then running the continuation on the updated store. The overall expected resource consumption is thus given by the sum of expected costs \(f[x/\sigma]\) of all such runs, weighted by their probability \(p_\sigma\). If \(\text{Bernoulli}(p)\) denotes the Bernoulli distribution with parameter \(p\), (yielding 1 with probability \(p\) and 0 with probability \(p - 1\)), then
\[
\text{ect}[x := \text{Bernoulli}(1/3)](f) = 1/3 \cdot f[x/1] + 2/3 \cdot f[x/0].
\]
For sequential commands \(C;D\), \(\text{ect}[C;D]\) is given by the composition \(\text{ect}[C] \circ \text{ect}[D]\), transforming the cost \(f\) of a continuation to that of running \(C\), then \(D\) followed by the continuation. For conditionals if \((\phi)\{C\}\{D\}\), the transformer perform a case analysis on the guard. Considering loops while \((\phi)\{C\}\), the expected cost transformer is defined as the least fixed-point of the functional
\[
\lambda F. \phi \cdot \text{ect}[C](F) + [-\phi] \cdot f.
\]
We will see below that this fixed-point is always defined. The transformer \(\text{ect}[\text{while } (\phi) \{C\}]\) thus in particular enjoys
\[
\text{ect}[\text{while } (\phi) \{C\}](f) = [\phi] \cdot \text{ect}[C](\text{ect}[\text{while } (\phi) \{C\}](f)) + [-\phi] \cdot f
\]
that is, it evaluates to the cost of unfolding the loop when the loop’s guard \(\phi\) 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.

### 5.1 Well-definedness and Soundness

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

**Lemma 5.2 (Central Properties of \(\text{ect}[\text{while } (\phi) \{C\}]\)).**

1. Continuity: \(\text{ect}[C](\sup_{n \in \mathbb{N}} f_n) = \sup_{n \in \mathbb{N}} \text{ect}[C](f_n)\) for all \(\omega\)-chains \((f_n)_{n \in \mathbb{N}}\);
2. Monotonicity: \(f \leq g \Rightarrow \text{ect}[C](f) \leq \text{ect}[C](g)\).

In particular, continuity ensures that in the case for loop \(C = \text{while } (\phi) \{D\}\), the least fixed point underlying \(\text{ect}[C]\) is well-defined. Now that we have established that \(\text{ect}[C](f)\), and hence \(\text{ecost}[C]\), is well-defined, the remaining objective of this section is to prove that \(\text{ecost}[C](\sigma)\) indeed gives the expected cost of running \(C\) on store \(\sigma\). To this end, we show that \(\text{ect}[C](f)(\sigma)\) coincides with \(\text{ect}[\rightarrow](f)(\sigma \triangleright C)\) defined in terms of the underlying PARS in Section 3.

Let us denote by \(\text{ect}[\rightarrow](f)(\triangleright C)\) the function \(\lambda \sigma. \text{ect}[\rightarrow](f)(\sigma \triangleright C)\). The correspondence thus becomes
\[
\text{ect}[C](f) = \text{ect}[\rightarrow](f)(\triangleright C).
\]

Since the definition of \(\text{ect}[C](f)\) 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 5.3 (Composition and Loop Lemma).**

1. \(\text{ect}[\rightarrow](f)(\triangleright C;D) = \text{ect}[\rightarrow](\text{ect}[\rightarrow](f)(\triangleright D))\triangleright C)\);
2. \(\text{ect}[\rightarrow](f)(\triangleright \text{while } (\phi) \{C\}) = \text{lfp}\{\phi\} \cdot \text{ect}[\rightarrow](g)(\triangleright C) + [-\phi] \cdot f\).
Note that the right-hand side in (2) is well-defined by Lemma 3.4(1). Relying on this auxiliary lemma, our central soundness result follows by a standard induction on $C$.

Theorem 5.4 (Soundness & Completeness). For every command $C \in \text{Cmd}$, we have

1. $\text{ect}[^\rightarrow](f)(\bullet \circ C) = \text{ect}[C](f)$; and consequently
2. $\text{ecost}[^\rightarrow](\bullet \circ C) = \text{ecost}[C]$.

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 $C$.

### 5.2 Upper Invariants

To find closed-forms for the runtime of loops, Kaminski et al. [2016] propose to search for upper invariants, i.e. prefix points of the loops characteristic function. The following constitutes a straightforward generalisation of Kaminski et al. [2016, Theorem 3]. For a Boolean expression $\phi \in \text{BExp}$, and two cost functions $f, g \in \mathbb{C}_\leq$, let us denote by $\phi \equiv f \leq g$ that $f(\sigma) \leq g(\sigma)$ holds for all stores $\sigma$ with $\phi \neq \sigma$.

**Theorem 5.5 (Upper Invariant).** Let $I \in \mathbb{C}_\leq$. If $\phi \equiv \text{ect}[C](I) \leq I$ and $\neg \phi \equiv f \leq I$, then

$$\text{ect}[\text{while} (\phi) \{C\}](f) \leq I.$$ 

Observe that the two premises are equivalent to $[\phi] \cdot \text{ect}[C](I) + [\neg \phi] \cdot f \leq I$, that is, $I$ is a pre-fixed point of the functional $\lambda F. [\phi] \cdot \text{ect}[C](F) + [\neg \phi] \cdot f$. 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 theorem, the problem of computing $\text{ecost}[C]$ can be reduced to a set of inequalities whose solution gives an upper bound on the expected cost of $C$. Let us illustrate this on a simple example.

**Example 5.6.** Reconsider the program $C_{\text{geo}}$ from Figure 1b and let $D$ denote the body of the loop in $C_{\text{geo}}$. A cost expression $I$ is an upper invariant for this loop wrt. $f$ if (i) $b = 1 \equiv \text{ect}[D](I) \leq I$; (ii) $b \neq 1 \equiv f \leq I$ holds. Unfolding the definition of $\text{ect}[D](I)$ yields

$$\text{ect}[D](I) = 1 + (b/1, x/2 \cdot x) + (b/0, x/2 \cdot x)/2.$$ 

Define now $I \equiv [b = 1] \cdot 2 + f$, that is, $I$ evaluates to $2 + f(\sigma)$ on stores $\sigma$ with $b = 1$ and behaves like $f$ otherwise. In particular, $I[b/1, x/2 \cdot x] = 2 + f$ and $I[b/0, x/2 \cdot x] = f$. The upper invariant condition thus simplifies to $b = 1 \equiv 2 + f \leq [b = 1] \cdot 2 + f$ and $b \neq 1 \equiv f \leq [b = 1] \cdot 2 + f$. A case analysis on $b = 1$ then shows that left- and right-hand sides are actually equal. Since in $C_{\text{geo}}$, the variable $b$ and $x$ are initially set to one, we obtain $\text{ecost}[C_{\text{geo}}] = I[b/1, x/1] = 2$ as a consequence of Theorem 5.4 and Theorem 5.5.

Theorem 5.5 suggests the following two stage approach towards an automated cost analysis of a program $C$ via Theorem 5.4. (i) evaluate $\text{ecost}[C] = \text{ect}[C](0)$ 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 $\kappa(b_1, \ldots, b_k)$, 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 $\phi \equiv \text{ect}[C](\kappa_I(b_1, \ldots, b_k)) \leq \kappa_I(b_1, \ldots, b_k)$ and $\neg \phi \equiv \kappa_f(b_1, \ldots, b_k) \leq \kappa_f(b_1, \ldots, b_k)$.

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

6 ALTERNATING EXPECTED COST AND VALUE ANALYSIS

While the calculus developed in the previous section is indeed compositional, the cost analysis via Theorem 5.5, as described above, is not. This is most apparent in the case of nested loops, abstractly represented as follows:

$$\text{while} (\phi) \{ \text{while} (\psi) \{ C \} \}.$$  

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.

$$\phi \not\vdash I \leq O \land \neg \phi \not\vdash f \leq O \land \psi \vdash \text{ect}[C](I) \leq I \land \neg \psi \not\vdash O \leq I.$$  

These constraints cannot be considered in isolation. In particular, if we express $I$ and $O$ in terms of linear combinations $\kappa_I(b_1, \ldots, b_k)$ and $\kappa_O(b_1, \ldots, b_k)$ of base functions $b_i$, the undetermined combinator $\kappa_I$ and $\kappa_O$ cannot be synthesised separately. Thus, the analysis degenerates to a whole-program analysis.

As emphasised in Section 2, 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 $C$, run on an initial store $\sigma$, is conceivable as a subdistribution $\mu$ over stores (many, in the case $C$ is also non-deterministic). Then $\text{ect}[C](f)(\sigma)$ yields the expected cost of $C$ plus the expected value $\mathbb{E}_\mu(f)$ of the cost function $f$ on the distribution of states $\mu$. When $f$ coincides with the cost of a continuation, we re-obtain the initial intuition that $\text{ect}[C](f)$ yields the cost of running $C$ followed by the corresponding continuation. We make this correspondence precise. First, we define the expected value transformer $\text{evalue}[C]$ of a command $C$ in correspondence to $\text{ect}[C]$, while ignoring costs.

**Definition 6.1 (Expected Value Transformer).** The expected value transformer $\text{evalue}[C]: \mathcal{C}^\Sigma \to \mathcal{C}^\Sigma$ for commands $C$ is defined through the rules given in the third column in Figure 3.

Note that $\text{evalue}[C](f) = \text{ect}[	ext{costFree}(C)](f)$ holds, where the cost-free program costFree($C$) is obtained from $C$ by dropping all cost annotations consume($e$). Particularly, Theorem 5.5 remains intact if we substitute $\text{evalue}[-]$ for $\text{ect}[-]$. The next lemma establishes a separation of the expected cost and value computation, which underlies the expected cost transformer $\text{ect}[C]$.

**Lemma 6.2 (Separating Expected Cost and Value).**

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

If $C$ features non-determinism, the right-hand side may over-approximate $\text{ect}[C](f)$. Whereas in $\text{ect}[C](f)$ cost and expected values are kept in sync across branches, this property is lost when considering $\text{ecost}[C]$ and $\text{evalue}[C](f)$ independently. A similar result, albeit restricted to purely probabilistic programs, has been observed by Kaminski et al. [2018, Thm 8.1].

Consider a while loop, while ($\phi$) {$C$}. As above, we express the upper invariant $I$ as the combination $\kappa(b_1, \ldots, b_k)$ of base functions $b_i$. Reconsider the first premise of Theorem 5.5: $\phi \vdash \text{ect}[C](\kappa(b_1, \ldots, b_k)) \leq \kappa(b_1, \ldots, b_k)$. By Lemma 6.2, this constraint is entailed by

$$\phi \vdash \text{ecost}[C] + \text{evalue}[C](\kappa(b_1, \ldots, b_k)) \leq \kappa(b_1, \ldots, b_k),$$ (2)
Moreover, when the upper invariant \( \kappa \) is linear in all its arguments, it distributes over expectations and hence over the expected value transformer \( \text{evalue}[C] \). Consequently (2) becomes
\[
\phi \equiv \text{ecost}[C] + \kappa(\text{evalue}[C](b_1), \ldots, \text{evalue}[C](b_k)) \leq \kappa(b_1, \ldots, b_k).
\] (3)

First, this decomposes the analysis of the cost of the loop’s body \( C \) from the analysis of how \( \kappa(b_1, \ldots, b_k) \) changes in expectation during one iteration. Second, none of the calls to \( \text{evalue}[C] \) do reference the combinator \( \kappa \). These can be determined in independence of the analysis of \( \text{while} (\phi) \{ C \} \). We generalise this observation. As usual, we call \( \kappa \) concave if \( p \cdot \kappa(\bar{r}) + (1-p) \cdot \kappa(\bar{s}) \leq \kappa(p \cdot \bar{r} + (1-p) \cdot \bar{s}) \) (where \( 0 \leq p \leq 1 \)) and (weakly) monotone if \( \bar{r} \leq \bar{s} \) implies \( \kappa(\bar{r}) \leq \kappa(\bar{s}) \). Together, the two conditions yield the following sufficient criterion to distribute \( \kappa \) over expectations.

**Lemma 6.3 (Distribute over Expected Values).** Suppose \( \kappa : C^k \rightarrow C \) is monotone. Furthermore, suppose \( \kappa \) is concave if the command \( C \) is probabilistic. Then \( \text{evalue}[C](\kappa(b_1, \ldots, b_k)) \leq \kappa(\text{evalue}[C](b_1), \ldots, \text{evalue}[C](b_k)) \).

Note that concavity is only needed to distribute \( \kappa \) over the expected value given by probabilistic statements, that is, probabilistic assignments from non-dirac distributions. On the other hand, it is not difficult to design an example showing the necessity of the restriction to concavity in this case.

The above lemmas in conjunction with Theorem 5.5 yields the main result of this section.

**Theorem 6.4 (Modular Upper-Invariants).** Let \( C \) and \( \kappa \) be as in Lemma 6.3. Then
1. \( \phi \equiv \text{ecost}[C] + \kappa(\text{evalue}[C](b_1), \ldots, \text{evalue}[C](b_k)) \leq I \implies \text{cost}[\text{while} (\phi) \{ C \}] \leq I \);
2. \( \phi \equiv \kappa(\text{evalue}[C](b_1), \ldots, \text{evalue}[C](b_k)) \leq I \land \neg \phi \equiv f \leq I \implies \text{cost}[\text{while} (\phi) \{ C \}](f) \leq I \).

Note that in the first point, we discharge the pre-condition \( f = 0 \leq I \) in Theorem 5.5. Concerning the second point, we exploit \( \text{evalue}[\text{while} (\phi) \{ C \}](f) = \text{ecost}[\text{while} (\phi) \{ \text{cost}[\text{costFree}](C) \}](f) \) and \( \text{cost}[\text{costFree}(C)] = 0 \). The theorem gives a modular recursive procedure to infer upper bounds on costs of loops, as follows, cf. Section 7. (i) Estimate the cost of one iteration of the body \( C \). (ii) 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. (iii) Solve the recurrence, given by the implication, where \( \kappa \) is to be determined. Crucially the loop’s body \( C \) becomes treatable in complete independence of the loop itself. Via Lemma 6.2 and Lemma 6.3, sequential composition can be treated similarly to Theorem 6.4.

**Theorem 6.5 (Modular Sequential Analysis).** Let \( C \) and \( \kappa \) be as in Lemma 6.3. Then
1. \( \text{ecost}[D](f) \leq \kappa(b_1, \ldots, b_k) \implies \text{ecost}[C; D] \leq \text{ecost}[C] + \kappa(\text{evalue}[C](b_1), \ldots, \text{evalue}[C](b_k)); \)
2. \( \text{evalue}[D](f) \leq \kappa(b_1, \ldots, b_k) \implies \text{evalue}[C; D](f) \leq \kappa(\text{evalue}[C](b_1), \ldots, \text{evalue}[C](b_k)). \)

6.1 Analysis of Examples

We conclude the section, by discussing the analysis of the running examples of Section 2 using our methodology. Further, we briefly comment on 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 \( C_{\text{geo}} \) (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 \( C_{\text{geo}} \) via Theorem 6.4(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 \( \text{ecost}[D] \) is one, which follows by unfolding the definition. By Theorem 6.4(1), \( \kappa(\langle b \rangle) \) is an upper bound to the cost of the loop if
\[
b = 1 \equiv \text{ecost}[D] + \kappa(\text{evalue}[D](\langle b \rangle)) \leq \kappa(\langle b \rangle) \iff b = 1 \equiv 1 + \kappa(1/2) \leq \kappa(\langle b \rangle).
\]
A case analysis on \( b = 1 \) shows that the constraint is satisfied with \( \kappa(x) \triangleq 2 \cdot x \). Finally, since \( b := 1 \) before entering the loop we derive the optimal bound \( \kappa(1) = 2 \) for \( C_{\text{geo}} \).

**Trader.** Recall program \( C_{\text{Trader}} \) from Figure 1a. It is not difficult to see that the cost of the inner loop is given by \( \langle n \rangle \cdot \langle \text{price} \rangle \). This then yields

\[
\text{ecost}[D] = 1/44 \cdot \sum_{n=0}^{10} n \cdot \langle \text{price} + 1 \rangle + 3/44 \cdot \sum_{n=0}^{10} n \cdot \langle \text{price} - 1 \rangle,
\]

as bound the cost of the body \( D \) of the outer loop. \Wrt. the outer loop itself, consider the base functions \( b_1 \triangleq \langle \text{min} + 1 \rangle \cdot \langle \text{price} - \text{min} \rangle \) and \( b_2 \triangleq \langle \text{price} - \text{min} \rangle^2 \). Applying Theorem 6.4(1), the cost of \( C_{\text{Trader}} \) is given by \( \kappa(b_1, b_2) \) subject to the constraint

\[
0 \leq \text{min} \leq \text{price} - 1 \vdash \text{ecost}[D] + \kappa(\text{evaluate}[D](b_1), \text{evaluate}[D](b_2)) \leq \kappa(b_1, b_2).
\]

Note that the loop body \( D \) increments and decrements the price with probabilities \( 1/4 \) and \( 3/4 \), respectively, whereas \( \text{min} \) remains unchanged. Formally,

\[
\text{evaluate}[D](b_1) = 1/4 \cdot \langle \text{min} + 1 \rangle \cdot \langle \text{price} + 1 - \text{min} \rangle + 3/4 \cdot \langle \text{min} + 1 \rangle \cdot \langle \text{price} - 1 - \text{min} \rangle,
\]

\[
\text{evaluate}[D](b_2) = 1/4 \cdot \langle \text{price} + 1 - \text{min} \rangle^2 + 3/4 \cdot \langle \text{price} - 1 - \text{min} \rangle^2,
\]

which can be derived by unfolding evaluate\([D] \) and applying Theorem 6.4(2). In turn, the above constraint holds with \( \kappa(x, y) \triangleq 10 \cdot x + 5 \cdot y \), yielding an upper bound

\[
\kappa(b_1, b_2) = 10 \cdot \langle \text{min} + 1 \rangle \cdot \langle \text{price} - \text{min} \rangle + 5 \cdot \langle \text{price} - \text{min} \rangle^2,
\]

on the cost of \( C_{\text{Trader}} \). Note that this upper bound can be derived with our implementation in 25s.

**Coupon Collector.** Recall program \( C_{\text{Coupons}} \) of Figure 1c, and let \( D \) denote the main loop’s body. While \( \text{ecost}[D] = 1 \) is straightforward 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 \( b \), evaluate\([D](b) \) is given by

\[
[1 \leq n] \cdot \sum_{\text{draw}=1}^{n} ([\text{coupons} < \text{draw}] \cdot b[\text{coupons}/\text{coupons} + 1] + [\text{draw} \leq \text{coupons}] \cdot b)/n.
\]

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

\[
b_1' \triangleq \langle n - \text{coupons} \rangle \cdot \langle (n - \text{coupons} - 1)^2 + (n - \text{coupons}) \cdot \langle \text{coupons} \rangle \rangle/n,
\]

\[
b_2' \triangleq \langle n - \text{coupons} \rangle \cdot \langle (n - \text{coupons} - 1) \cdot \langle \text{coupons} + 2 \rangle + (\text{coupons} + 1) \cdot \langle \text{coupons} \rangle \rangle/n,
\]

for \( \text{ecost}[D](b_i) \). The constraint \( 0 \leq i \leq n - 1 \vdash 1 + \kappa(b'_1, b'_2) \leq \kappa(b_1, b_2) \), describing the cost of the loop, is then shown satisfiable by taking \( \kappa(x, y) \triangleq x + 1/2 \cdot y \). Substituting \( 0 \) for \( \text{coupons} \) in the bound \( \kappa(b_1, b_2) \) due to the initial assignments yields the overall estimate \( \langle n \rangle + 1/2 \cdot \langle n \rangle^2 \) for the expected cost of \( C_{\text{Coupons}} \). This analysis was performed in 195ms with our tool.

**Queuing Network.** Wang et al. [2019] represent fork-join queues (see Kim and Agrawala [1989]) for 2 servers as a probabilistic program with a unitary cost model, that measures progress of the sub-jobs in the respective queues. A fixed probability is employed to model the arrival of new jobs, while \( n \) jobs are processes. With probability \( 1/2 \) the job is handled by the first server and with probability \( 1/2 \) by the second server. In the default, the job is passed to both servers. Server 1 takes 3 units for completion, while Server 2 takes 2 time units. If the job is split on the servers, the Server 1 and 2 require 2 and 1 time units, respectively. The cost for the fork-join queue is given by the expected time of completion of each job. Our prototype computes the (asymptotically) optimal bound \( 113/44(n + 1) + 125/243 + 125/244 \) in 2.215s. Our bound is more precise than the bound obtained in [Wang et al. 2019] and the execution time of eco-imp is 35-times faster.
function \textsc{ecost}[C] \\
switch C do \\
\textbf{case} D; E where D contains a loop: \\
\hspace{1em} \kappa(\vec{b}) \leftarrow \textsc{ecost}[E] \\
\hspace{1em} \vec{b}' \leftarrow \text{foreach } b \in \vec{b}; \text{evaluate}[D](b) \\
\hspace{1em} \text{return } \textsc{ecost}[D] + \kappa(\vec{b}') \\
\textbf{case} while (\phi) \{D\}: \\
\hspace{1em} g \leftarrow \textsc{ecost}[D] \\
\hspace{1em} \vec{b} \leftarrow \textsc{selectbases}(\phi, g) \\
\hspace{1em} \vec{b}' \leftarrow \text{foreach } b \in \vec{b}; \text{evaluate}[D](b) \\
\hspace{1em} \kappa \leftarrow \textsc{lineartemplate}(\text{length}(\vec{b})) \\
\hspace{1em} \theta \leftarrow \text{solve}(\phi \equiv g + \kappa(\vec{b}') \leq \kappa(\vec{b})) \\
\hspace{1em} \text{return } \theta(\kappa)(\vec{b}) \\
\textbf{default}: \text{return } \textsc{symboliccost}(C) \\
\textbf{default: return } \textsc{symbolicvalue}(\phi, f) \\

7 IMPLEMENTATION

In this section, we give an overview of the implementation of our methodology within our tool \textsc{eco-imp}. The core of our tool is given by the algorithms \textsc{ecost}[C] and \textsc{evaluate}[C] outlined in Figure 4, computing for a given program C and cost function f, an upper bound to \textsc{ecost}[C] and \textsc{evaluate}[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 3.4(2)) soundness of the overall algorithm remains intact.

Sequential composition. For a command C = D; E, our implementation relies on Theorem 6.4, except when D does not contain a loop. In this case, the algorithms proceed by unfolding according to the rules in Figure 4 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 6.4, following to recipe given on page 14. The subprocedure \textsc{selectbases}, outlined below, selects a vector of base functions \vec{b} from the loops guard \phi, 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 \vec{b}' to its expectation \textsc{evaluate}[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 \textsc{solve} is applied to the constraint dictated by Theorem 6.4. Should this procedure establish a solution \theta, ie. an assignment to the undetermined coefficients q_i underlying \kappa that validates the supplied constraint, the instantiated cost function \theta(\kappa)(\vec{b}) = \sum_i \theta(q_i) \cdot \vec{b}_i is returned. The dedicated constraint solver underlying the procedure \textsc{solve} is detailed below in Section 7.1.

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 \phi, 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 \epsilon_1 \leq \epsilon_2 we take \(\epsilon_2 - \epsilon_1 + 1 \in \vec{b}_\phi.\)
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 \( d(\sigma) = \{p_i(\sigma) : e_i(\sigma)\}_{0 \leq i \leq k} \) for a fixed constant \( k \) and \( p_i = e_i / f_i \), such as \( \{\lfloor x + 2i \rfloor \}_{0 \leq i \leq 9} \). Noteworthy, probabilities of probabilistic branches are not necessarily constant. The expected cost of a probabilistic assignment \( x := d \) is given by

\[
\text{ect}[x := d](f)(\sigma) = \sum_{1 \leq i \leq k} p_i(\sigma) \cdot f[x/e_i(\sigma)] .
\]

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. Our implementation supports this way a variety of standard distributions, noteworthy sampling from uniform distributions with bounded support, Bernoulli, binomial and hypergeometric distributions. A distinct feature of our overall methodology is that 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 \( \text{Uniform}(e_1, e_2) \) for \( e_1 \leq e_2 \). This, for example, is crucial to represent the Coupon Collector’s problem properly. Note, that in this case

\[
\text{ect}[x := d](f)(\sigma) = [e_1 \leq e_2] \cdot (\sum_{i \leq e_2} f[x/i])/(e_2 - e_1 + 1) .
\]

To find a closed form for the bounded sum \( \sum_{i \leq e_2} f[x/i] \), our implementation seeks for an upper bound \( \kappa(\vec{b}) \) subject to the constraint \( e_1 \leq i \land i \leq e_2 \equiv f[x/i] + \kappa(\vec{b}[i/i + 1]) \leq \kappa(\vec{b}) \).

7.1 Constraint Solving Mechanism

In this section, we highlight the constraint solving mechanism as implemented in our prototype \text{eco-im}, 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 \text{Exp} and \text{cost expressions} \text{CExp} as follows.

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

In \text{Exp}, we use integer variables \( x \in \text{Var} \) to denote cost functions \( \lambda \sigma.\sigma(x) \). Cost expressions denote fractions over integer expressions, closed under guards, sum and maximum. Furthermore, we restrict \text{BExp} to Boolean formulas over atoms \( e \geq f \), in disjunctive normal form. This still permits the usual comparison operators over integer expressions, e.g., \( e > f \) can be represented as \( e \geq f + 1 \) or \( e = f \) as \( e \geq f \land f \geq e \). 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 \( \langle e \rangle \triangleq [0 \leq e] \cdot e \). Not every cost expression is an expectation in \( C^\Sigma \) though, as cost expression need not be well-formed.

We say \( c \in \text{CExp} \) is well-formed wrt. store \( \sigma \in \Sigma \) if (i) \( c = e/f \) such that \( e(\sigma) \geq 0 \) and \( f(\sigma) > 0 \); (ii) \( c = [\phi] \cdot d \) such that \( \sigma \models \phi \) and \( d \) is well-formed wrt. \( \sigma \); (iii) \( c = c_1 + c_2 \) or \( c = \text{max}(c_1, c_2) \) such that both \( c_1 \) and \( c_2 \) are well-formed wrt. \( \sigma \). Well-formedness of \( c \) under \( \sigma \) guarantees that \( c(\sigma) \) is defined and non-negative. We call \( c \) well-formed wrt. a Boolean expression \( \phi \), if \( c \) is well-formed.
for all stores \( \sigma \) such that \( \sigma \models \phi \). 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 build 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 \( c \in CExp \) and \( e \in CExp \), 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:

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

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 \( \sigma \) where the fraction is negative. However, for well-formed cost expression \( e_1/e_2 \), the equality holds.

**Constraint simplification.** In Figure 5, we present an inference system over constraints \( \phi \models c \leq d \), where \( c, f \in CExp \) and \( \phi, \psi \in BExp \). 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 \( \phi \models c \leq d \) is well-formed, if \( c, d \) are well-formed under \( \phi \). It is valid if for all stores \( \sigma \in \Sigma \) with \( \sigma \models \phi \), 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 7.1.** Let \( \Delta \) be a derivation of a well-formed constraint \( \phi \models c \leq d \). Then this constraint is valid if and only if all premises in \( \Delta \) are.

This statement is proven by induction on the derivation \( \Delta \). The rules concerning division, rule \( (\text{divL}) \) and \( (\text{divR}) \), are sound because the divisor \( f \) is positive on all stores \( \sigma \) satisfying \( \phi \), due to the well-formedness assumption.

**Example 7.2.** Recall that the analysis of program \( C_{geo} \) in Section 6.1 required solving the constraint \( b = 1 + \lfloor 1 + k/(1/2) \rfloor \leq \lfloor (b) \rfloor \). Fix \( k(x) \triangleq q \cdot x \) for undetermined \( q \in \mathbb{R}_{\geq 0} \) and note that \( (b) = [0 \leq b] \cdot b \). By unfolding \( k \), the cost of the loop in \( C_{geo} \) is given by \( k((b)) \) if \( b = 1 \equiv 1 + q/2 \leq [0 \leq b] \cdot q \cdot b \), is valid for some \( q \in \mathbb{R}_{\geq 0} \). Eliminating guards yield several constraints, but only the constraint \( b = 1 \land 0 \leq b = 1 + q/2 \leq [0 \leq b] \cdot q \cdot b \), is not subject to rule \( (\text{triv}) \) or rule \( (\text{contr}) \). Rule \( (\text{divL}) \) then yields \( b = 1 \land 0 \leq b = 2 + q \leq 2 \cdot q \cdot b \). By Theorem 7.1, 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 \( \phi \models c \leq d \), the simplification rules of Figure 5 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
We restrict to the multi-indices $W_{\text{ang}}$ et al. 2019.\textsuperscript{19} which, subject to the side-condition $\text{eco-imp}$ $q$, the solution $\nu$ now translates this solution to a solution $\lambda$ 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 $\text{min}$ to 100.

constraints are of the form

$$\bigwedge_i c_i \geq 0 \iff e \geq 0,$$

by employing additionally the identity $f \geq g \iff f - g \geq 0$. Any expression $e$ can be understood as a polynomial $p(\bar{x})$ in the program variables $\bar{x}$. 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].

**Proposition 7.3.** Let $I = (i_1, \ldots, i_m) \in \mathbb{N}^m$ denote a multi-index such that $p, p_{1 \leq i \leq m} : \mathbb{Z}^n \to \mathbb{R}$. Further, let $p_I(\bar{x}) = p_{1}^{i_1}(\bar{x}) \cdots p_{m}^{i_m}(\bar{x})$. If $p(\bar{x}) = \sum_{I \in \mathbb{N}^m} c_I \cdot p_I(\bar{x})$ for some $c_I \in \mathbb{R}_{\geq 0}$, then $\bigwedge_{I=1}^m p_I(\bar{x}) \geq 0$ implies $p(\bar{x}) \geq 0$.

Incidentally, Proposition 7.3 expresses a (gross) simplification of Handelman’s theorem [Handelman 1988], which turns out to be quite effective in our context. Recall that overall, we are concerned with synthesising concrete values for the undetermined $q_i \in \mathbb{R}_{\geq 0}$ stemming from templates $\kappa(b) \doteq \sum q_i \cdot b_i$, 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 7.3 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. \textsuperscript{20}

**Example 7.4 (Example 7.2 continued).** The final remaining constraint from the running example simplifies to

$$b - 1 \geq 0 \land 1 - b \geq 0 \land b \geq 0 \implies 2 \cdot q \cdot b - q - 2 \geq 0.$$

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 $b$, representing the conjuncts in the assumption. Proposition 7.3 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 $q \geq 0$, holds with $q = 2, c_{(1,0,0)} = 4$ and $c_{(0,1,0)} = c_{(0,0,1)} = 0$. Theorem 7.1 now translates this solution to a solution $\kappa(x) \doteq 2 \cdot x$ for the initial constraint in Example 7.2. The constraint solving mechanism established this way thus derives the (optimal) bound given by hand in Section 6.1.

### 7.2 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
Avanzini, Moser and Schaper.

Fig. 7. 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.

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 we list the program C_coupons from Figure 1c as coupons-n and also instantiate n to constants 10, 50 and 100. Similar, the trader example C_trader 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 of Ngo et al. [2018] (column B) and Wang et al. [2019] (column C). On some of the examples, the tool Absynth does not provide an upper bound in a reasonable amount of time; as timeout we have use 5 minutes; corresponding examples have been marked with —. Wrt. to tool by Wang et al. [2019], we were able to slightly extend the results given in their work. However, the tool requires extensive manual invariant annotations. Conclusively, various examples were not amendable to an experimental evaluation and corresponding rows have been left blank. Execution times are obtained on an Intel i7-7600U CPU with 2.8GHz and 16GB RAM.

Precision. The upper 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, inferred by the prototype of Wang et al. [2019]. In particular, the bounds for all but one example are on the same order of magnitude. In example prnes, eco-imp infers a quadratic rather than a linear bound. Vice versa, for example C4B_t30 our tool infers an asymptotic optimal linear bound, whereas Absynth yields a quadratic one. 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 due to Kaminski et al. [2016], where eco-imp’s precision equals the measured expected cost, cf. Figure 6.

Speed. Across the benchmark, eco-imp outperforms the other tools. The only exception to this assessment are the examples queueing-network and prnes. While our tool achieves already significant speedups in the linear benchmark from [Ngo et al. 2018] (annotated in parenthesis besides execution times), the gains are most pronounced in examples with nested loops and non-linear bounds. This assessment remains intact when comparing to the implementation of Wang et al. on the subset of examples treated by Wang et al. [2019].

Scalability. In Figure 7, 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, comparing eco-imp and the tool Absynth. Similarly, the
figure on the right depicts the effects on extending the number of shares in $C_{\text{trader}}$, comparing all tools. Our results are competitive and highlight sharply the prime benefit of our implementation, viz, its modularity.

8 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 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. 2019a; 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 $C_{\text{coupons}}$. Our expected cost transformer constitutes a generalisation of the $ert$-calculus. In particular, note that value $C$ 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 7).

9 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 prowess 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. [2019b]; 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 via eco-imp (columns A), the Absynth prototype from [Ngo et al. 2018] (columns B) and the prototype of [Wang et al. 2019] (columns C). Integer factor annotated besides execution time give the relative speedup of our implementation.

<table>
<thead>
<tr>
<th>Program</th>
<th>A</th>
<th>Inferred Bound</th>
<th>C</th>
<th>Execution times in secs. (factor)</th>
</tr>
</thead>
<tbody>
<tr>
<td>linear benchmark from [Ngo et al. 2018]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2drwalk</td>
<td>2(n−d+1)</td>
<td>2(n−d+1)</td>
<td></td>
<td>0.026s 0.286s (11)</td>
</tr>
<tr>
<td>bayesian_network</td>
<td>5(n)</td>
<td>5(n)</td>
<td></td>
<td>0.002s 0.127s (63)</td>
</tr>
<tr>
<td>ber</td>
<td>2(n−x)</td>
<td>2(n−x)</td>
<td></td>
<td>0.001s 0.014s (14)</td>
</tr>
<tr>
<td>bin</td>
<td>0.2(9+n)</td>
<td>0.91n−0.91x−0.91</td>
<td></td>
<td>6.030s (3151)</td>
</tr>
<tr>
<td>C4B_t09</td>
<td>41(x)</td>
<td>8.27(x)</td>
<td></td>
<td>0.006s 0.079s (13)</td>
</tr>
<tr>
<td>C4B_t13</td>
<td>1.25(x)+y</td>
<td>1.25x+y−1.25</td>
<td></td>
<td>0.005s 0.025s (5)</td>
</tr>
<tr>
<td>C4B_t15</td>
<td>(x−1)+(x)</td>
<td>(x−1)+(x)</td>
<td></td>
<td>0.006s 0.026s (4)</td>
</tr>
<tr>
<td>C4B_t19</td>
<td>2(i−100)+(i+k+51)</td>
<td>2(i)+(i+k+51)</td>
<td></td>
<td>0.005s 0.022s (4)</td>
</tr>
<tr>
<td>C4B_t38</td>
<td>(y+2)+(x)</td>
<td>0.33(x+2)+0.167(2+y)+(2+y)</td>
<td></td>
<td>0.003s 0.240s (80)</td>
</tr>
<tr>
<td>C4B_t61</td>
<td>7+10/3/(l−7)</td>
<td>0.06(l−1)+(l)</td>
<td></td>
<td>0.005s 0.036s (7)</td>
</tr>
<tr>
<td>condend</td>
<td>2(m)</td>
<td>0.42(5+st)+(mt−st)</td>
<td></td>
<td>0.015s 0.083s (6)</td>
</tr>
<tr>
<td>cooling</td>
<td></td>
<td></td>
<td></td>
<td>0.003s 0.045s (15)</td>
</tr>
<tr>
<td>coupon</td>
<td>25</td>
<td>6/5</td>
<td></td>
<td>0.001s 0.039s (33)</td>
</tr>
<tr>
<td>cowboy_duel</td>
<td>7/2</td>
<td>1.2</td>
<td></td>
<td>0.004s 0.149s (37)</td>
</tr>
<tr>
<td>cowboy_duel_3way</td>
<td>8/24</td>
<td>2.083</td>
<td></td>
<td>0.006s 0.230s (38)</td>
</tr>
<tr>
<td>fcall</td>
<td>2(n−x)</td>
<td>2(n−x)</td>
<td></td>
<td>0.001s 0.021s (21)</td>
</tr>
<tr>
<td>filling_vol</td>
<td>3/5(1+oTF)+3/5(oTF)</td>
<td>0.33(10+oTF)+0.33(oTF)</td>
<td></td>
<td>0.006s 0.230s (38)</td>
</tr>
<tr>
<td>geo</td>
<td>5</td>
<td>5</td>
<td></td>
<td>0.001s 0.018s (18)</td>
</tr>
<tr>
<td>hyper</td>
<td>16/28(5−n−1)</td>
<td>5(n−x)</td>
<td></td>
<td>0.002s 0.034s (17)</td>
</tr>
<tr>
<td>linear01</td>
<td>0.6(x)</td>
<td>0.6x</td>
<td></td>
<td>0.001s 0.011s (12)</td>
</tr>
<tr>
<td>no_loop</td>
<td>5</td>
<td>5</td>
<td></td>
<td>0.001s 0.010s (10)</td>
</tr>
<tr>
<td>prdwalk</td>
<td>3/5(−n−x)</td>
<td>1.14(4+n−n−x)</td>
<td></td>
<td>0.002s 0.027s (13)</td>
</tr>
<tr>
<td>prnes</td>
<td>3/5(y−99)+(−n)+3/5(n−99)</td>
<td>68.48(−n)+0.05(y)</td>
<td></td>
<td>0.126s 0.034s (6)</td>
</tr>
<tr>
<td>presq</td>
<td>6(x−y−2)+3(y−9)</td>
<td>1.65(x−y)+0.15(y)</td>
<td></td>
<td>0.010s 0.030s (3)</td>
</tr>
<tr>
<td>proseq_bin</td>
<td>5/6(x−y−2)+3(y−9)</td>
<td>1.65(1+x−y)+0.15(y)</td>
<td></td>
<td>0.016s 0.038s (2)</td>
</tr>
<tr>
<td>prospeed</td>
<td>5/6(n−x−2)+(2+m−y)</td>
<td>0.067(n−x)+2(m−y)</td>
<td></td>
<td>0.013s 0.033s (3)</td>
</tr>
<tr>
<td>race</td>
<td>11/6(t−h+3)</td>
<td>0.67(t−h+3)</td>
<td></td>
<td>0.010s 0.048s (5)</td>
</tr>
<tr>
<td>rdsseq</td>
<td>3/5(x)+y</td>
<td>2.25(x)+y</td>
<td></td>
<td>0.007s 0.032s (5)</td>
</tr>
<tr>
<td>rdspeed</td>
<td>5/6(n−x−2)+(2+m−y)</td>
<td>0.67(n−x)+2(m−y)</td>
<td></td>
<td>0.012s 0.096s (8)</td>
</tr>
<tr>
<td>rdwalk</td>
<td>2(n+x+1)</td>
<td>2(n−x−y+1)</td>
<td></td>
<td>0.003s 0.058s (19)</td>
</tr>
<tr>
<td>rejection_sampling</td>
<td>3/5(10.011−n)</td>
<td>4.933(10.010−n)</td>
<td></td>
<td>0.049s 0.139s (3)</td>
</tr>
<tr>
<td>rfind_lv</td>
<td>2</td>
<td>2</td>
<td></td>
<td>0.001s 0.015s (15)</td>
</tr>
<tr>
<td>rfind_mc</td>
<td>2</td>
<td>2</td>
<td></td>
<td>0.002s 0.019s (9)</td>
</tr>
<tr>
<td>robot</td>
<td>3/5(6+n)</td>
<td>0.38(6+n)</td>
<td></td>
<td>0.006s 1.546s (28)</td>
</tr>
<tr>
<td>roulette</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Table 2. Automatically derived bounds on the expected cost via eco-imp (columns A), the Absynth prototype from [Ngo et al. 2018] (columns B) and the prototype of [Wang et al. 2019] (columns C). Integer factor annotated besides execution time give the relative speedup of our implementation.

<table>
<thead>
<tr>
<th>Program</th>
<th>Inferred Bound</th>
<th>Execution times in secs. (factor)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>A</strong></td>
<td><strong>B</strong></td>
<td><strong>C</strong></td>
</tr>
<tr>
<td>simple_game</td>
<td>2(x + 1)</td>
<td>2(x + 1)</td>
</tr>
<tr>
<td>sprdwalk</td>
<td>2(n - x)</td>
<td>2(n - x)</td>
</tr>
<tr>
<td>trapped_miner</td>
<td>15/n</td>
<td>7.5(n)</td>
</tr>
<tr>
<td><strong>Non-linear benchmark from [Ngo et al. 2018]</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>complex</td>
<td>(w) + 18(M·N + N)θ(N) + 3(y)</td>
<td>(w) + 18(M·θ(N)) + 9(N) + 3(y)</td>
</tr>
<tr>
<td>multirace</td>
<td>(m + 1)(n)</td>
<td>2(m)(n) + 4(n)</td>
</tr>
<tr>
<td>pol04</td>
<td>13/2(x) + 3/2(x)²</td>
<td>7.5(x) + 4.5(x)²</td>
</tr>
<tr>
<td>pol05</td>
<td>3/2(x) + 3/2(x)²</td>
<td>(x) + (x)²</td>
</tr>
<tr>
<td>pol06</td>
<td>1(m)/(1+p) + (p-m)/(1+p)</td>
<td>0.5(p-m) + (p-m)(p) + (p-m)²</td>
</tr>
<tr>
<td>rdbub</td>
<td>3/(n-1)²</td>
<td>1.5(n-1)(n-2)</td>
</tr>
<tr>
<td>trader (-20)</td>
<td>20(1+m)(p-m) + 10(p-m)²</td>
<td>20(m)(p-m) + 10(p-m) + 10(p-m)²</td>
</tr>
<tr>
<td><strong>additional examples from [Wang et al. 2019]</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>zrobot</td>
<td>1/(1+a-b) (a-b)</td>
<td>—</td>
</tr>
<tr>
<td>queueing-network</td>
<td>1.132/48 + 0.904(n+1) + 12/40 + 12/244</td>
<td>0.05(n+1) + 0.014(n+1)²</td>
</tr>
<tr>
<td><strong>examples from this work</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>bridge</td>
<td>(x-a)(b-x)</td>
<td>—</td>
</tr>
<tr>
<td>nest-1</td>
<td>4(n)</td>
<td>2(n + 1)</td>
</tr>
<tr>
<td>nest-2</td>
<td>4(n) + 16(n)²</td>
<td>2(1+n) + 4(1+n)²</td>
</tr>
<tr>
<td>nest-3</td>
<td>2(2+n) + 16(n)³</td>
<td>2(1+n) + 4(1+n)² + 8(1+n)³</td>
</tr>
<tr>
<td>nest-4</td>
<td>144(2+n)²(n) + 10(2+n) + 254n⁴</td>
<td>—</td>
</tr>
<tr>
<td>trader-10</td>
<td>10(1+m)(p-m) + 5(p-m)²</td>
<td>10((m)(p-m) + (p-m)²) + 5(p-m)</td>
</tr>
<tr>
<td>trader-100</td>
<td>100(1+m)(p-m) + 50(p-m)²</td>
<td>—</td>
</tr>
<tr>
<td>trader-1000</td>
<td>1,000(1+m)(p-m) + 500(p-m)²</td>
<td>—</td>
</tr>
<tr>
<td>trader-10000</td>
<td>10,000(1+m)(p-m) + 5,000(p-m)²</td>
<td>—</td>
</tr>
<tr>
<td>trader-100000</td>
<td>100,000(1+m)(p-m) + 50,000(p-m)²</td>
<td>—</td>
</tr>
<tr>
<td>coupons-10</td>
<td>110</td>
<td>1550</td>
</tr>
<tr>
<td>coupons-50</td>
<td>2250</td>
<td>—</td>
</tr>
<tr>
<td>coupons-100</td>
<td>10100</td>
<td>—</td>
</tr>
<tr>
<td>coupons-n</td>
<td>(n) + 1/n²</td>
<td>not supported</td>
</tr>
</tbody>
</table>
REFERENCES


A Modular Cost Analysis for Probabilistic Programs


A MATHEMATICAL BACKGROUND

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

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

\[ \text{lfp}(\chi) = \sup \chi^n(\bot). \]

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

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

B MISSING PROOFS

Proofs of Section 3

Proposition 3.2 (Cost functions form an $\omega$-CPO). For any $A$, $(\mathbb{C} \times A, \leq)$ is an $\omega$-CPO, with least and greatest element $0$ and $\infty$, respectively. The supremum of $\omega$-chains $(f_n)_{n \in \mathbb{N}}$ is given point-wise:

\[ \sup_{n \in \mathbb{N}} f_n = \lambda a. \sup_{n \in \mathbb{N}} f_n(a). \]

Proof. Since $(\mathbb{C}, \leq)$ form an $\omega$-CPO with least element $0$ and greatest element $\infty$, the proposition follows by Proposition A.1. □

In the remainder of this section, we prove Lemma 3.4 and Theorem 3.5. Recall that

\[ \chi_f(g) = \lambda a. \begin{cases} f(a) & \text{if } a \downarrow, \\ \sup \{w + E_{\mu}(g) \mid a \xrightarrow{w} \mu\} & \text{else}. \end{cases} \]

Lemma B.1 (Well-definedness of $\text{ect}[\to]$). The stepping function $\chi_f(g)$ is continuous in $f$ and $g$, that is,

(1) $\chi_{\sup_{n \in \mathbb{N}} f_n}(g) = \sup_{n \in \mathbb{N}} \chi_f(g_n)$ for all $\omega$-chains $(f_n)_{n \in \mathbb{N}}$; and

(2) $\chi_f(\sup_{n \in \mathbb{N}} g) = \sup_{n \in \mathbb{N}} \chi_f(g_n)$ for all $\omega$-chains $(g_n)_{n \in \mathbb{N}}$.

Proof. Point 1 follows by a simple case analysis on arguments $a$. Concerning Point 2 we have to prove that

\[ \chi_f(\sup_{n \in \mathbb{N}} g_n)(a) = \sup_{n \in \mathbb{N}} \chi_f(g_n)(a), \]

for every chain $g_1 \leq g_2 \leq \cdots$ and all $a \in A$. If $a \downarrow$ the claim is trivial. As a consequence of Lemma A.3, the cost transformer

\[ h_{\mu, w} = \lambda f : w + E_{\mu}(f), \]

is continuous. Now observe that for any doubly indexed sequence $(r_{i,j})_{i \in I, j \in J}$ of extended reals,

\[ \sup_{i \in I} \sup_{j \in J} r_{i,j} = \sup_{j \in J} \sup_{i \in I} r_{i,j}. \]

, Vol. 1, No. 1, Article . Publication date: August 2020.
We thus obtain

\[
\chi_f(\sup_{n \in \mathbb{N}} g_n)(a) = \sup_{a \xrightarrow{w,\mu} \mathbb{N}} h_{\mu,w}(\sup_{n \in \mathbb{N}} g_n) \quad \text{(by definition)}
\]

\[
= \sup_{a \xrightarrow{w,\mu} \mathbb{N}} h_{\mu,w}(g_n) \quad \text{(continuity of } h_{\mu,w})
\]

\[
= \sup_{n \in \mathbb{N}} \sup_{a \xrightarrow{w,\mu} \mathbb{N}} h_{\mu,w}(g_n) \quad \text{(observation)}
\]

\[
= \sup_{n \in \mathbb{N}} \chi_{g_n}(a) \quad \text{(by definition)}.
\]

\[\square\]

**Lemma B.2 (Continuity of \emph{ec} \rightarrow\).** The expected cost transformer \emph{ect}[\rightarrow\] is continuous, ie.

\[
\text{ect}[\rightarrow\](\sup_{n \in \mathbb{N}} f_n) = \sup_{n \in \mathbb{N}} \text{ect}[\rightarrow\](f_n),
\]

for all \(\omega\)-chains \((f_n)_{n \in \mathbb{N}}\).

**Proof.** It is well known that the least fixed point operator is continuous on continues functions \cite[Chapter 8]{Winskel1993}, particularly, by Lemma B.1(2) this implies

\[
\text{lfp}(\sup_{n \in \mathbb{N}} \chi_{f_n}) = \sup_{n \in \mathbb{N}} \text{lfp}(\chi_{f_n}).
\]

Thus

\[
\text{ect}[\rightarrow\](\sup_{n \in \mathbb{N}} f_n) = \text{lfp}(\sup_{n \in \mathbb{N}} \chi_{f_n}) \quad \text{(by definition)}
\]

\[
= \text{lfp}(\sup_{n \in \mathbb{N}} \chi_{f_n}) \quad \text{(Lemma B.1(1))}
\]

\[
= \sup_{n \in \mathbb{N}} \text{lfp}(\chi_{f_n}) \quad \text{(above observation)}
\]

\[
= \sup_{n \in \mathbb{N}} \text{ect}[\rightarrow\](f_n).
\]

\[\square\]

**Theorem 3.5 (Expected Cost via Cost Transformer).**

\[
\text{ecost}[\rightarrow\] = \text{ect}[\rightarrow\](0).
\]

**Proof.** A standard induction reveals that \(\chi_f^n(0) = \sup\{ w \mid a \xrightarrow{w}^n \mu \}. \) Consequently, by Kleene’s Fixed-Point Theorem

\[
\text{ect}[\rightarrow\](f) = \sup_{n \in \mathbb{N}} \chi_f^n(0) \quad \text{(above observation)}
\]

\[
= \sup_{n \in \mathbb{N}} \sup\{ w \mid a \xrightarrow{w}^n \mu \}
\]

\[
\{ w \mid a \xrightarrow{w}^* \mu \} = \bigcup_{n \in \mathbb{N}} \{ w \mid a \xrightarrow{w}^n \mu \}
\]

\[
= \sup\{ w \mid a \xrightarrow{w}^* \mu \} \quad \text{(by definition)}
\]

\[
= \text{ect}[\rightarrow\](0)
\]

\[\square\]
Proofs of Section 5

**Lemma B.3 (Continuity of Expectation Transformers).** For every \( \omega \)-chain \((f_n)_{n \in \mathbb{N}} \),

\[
\text{ect}[C](\sup_{n \in \mathbb{N}} f_n) = \sup_{n \in \mathbb{N}} \text{ect}[C](f_n).
\]

**Proof.** The proof is by induction on the command \( C \).

- **Case consume\((e)\).** We have

\[
\text{ect}[\text{consume}(e)](\sup_{n \in \mathbb{N}} f_n) = \langle e \rangle + \sup_{n \in \mathbb{N}} f_n \quad \text{(by definition)}
\]

\[
= \sup_{n \in \mathbb{N}} (\langle e \rangle + f_n) \quad \text{((e) is constant wrt. } n \in \mathbb{N})
\]

\[
= \sup_{n \in \mathbb{N}} \text{ect}[\text{consume}(e)](f_n) \quad \text{(by definition)}.
\]

- **Case skip.** As above, we have

\[
\text{ect}[\text{skip}](\sup_{n \in \mathbb{N}} f_n) = \sup_{n \in \mathbb{N}} f_n = \sup_{n \in \mathbb{N}} \text{ect}[\text{skip}](f_n).
\]

- **Case abort.** By definition,

\[
\text{ect}[\text{abort}](\sup_{n \in \mathbb{N}} f_n) = 0 = \sup_{n \in \mathbb{N}} \text{ect}[\text{abort}](f_n).
\]

- **Case \( x := d \).** As a consequence of Lemma A.3, we have

\[
\mathbb{E}_\delta(\sup_{n \in \mathbb{N}} g_n) = \sup_{n \in \mathbb{N}} \mathbb{E}_\delta(g_n),
\]

for all distribution \( \delta \in \mathcal{D}(\mathbb{Z}) \) and \( g_i : \mathbb{Z} \to \mathbb{R}_{\geq 0} \), where \( \sup_{n \in \mathbb{N}} g_n = \lambda i. \sup_{n \in \mathbb{N}} g_n(i) \). Thus we have

\[
\text{ect}[x := d](\sup_{n \in \mathbb{N}} f_n)
\]

\[
= \lambda \sigma. \mathbb{E}_d(\sigma)(\lambda i. (\sup_{n \in \mathbb{N}} f_n)(\sigma[x/i])) \quad \text{(definition)}
\]

\[
= \lambda \sigma. \mathbb{E}_d(\sigma)(\sup_{n \in \mathbb{N}} \lambda i. f_n(\sigma[x/i])) \quad \text{(def. sup)}
\]

\[
= \lambda \sigma. \sup_{n \in \mathbb{N}} \mathbb{E}_d(\sigma)(\lambda i. f_n(\sigma[x/i])) \quad \text{(MCT)}
\]

\[
= \sup_{n \in \mathbb{N}} \lambda \sigma. \mathbb{E}_d(\sigma)(\lambda i. f_n(\sigma[x/i])) \quad \text{(def. sup)}
\]

\[
= \sup_{n \in \mathbb{N}} \text{ect}[x := d](f_n) \quad \text{(definition)}.
\]

- **Case \( C; D \).** We have

\[
\text{ect}[C; D](\sup_{n \in \mathbb{N}} f_n)
\]

\[
= \text{ect}[C](\text{ect}[D](\sup_{n \in \mathbb{N}} f_n)) \quad \text{(definition)}
\]

\[
= \text{ect}[C](\sup_{n \in \mathbb{N}} \text{ect}[D](f_n)) \quad \text{(IH on D)}
\]

\[
= \sup_{n \in \mathbb{N}} \text{ect}[C](\text{ect}[D](f_n)) \quad \text{(IH on C)}
\]

\[
= \sup_{n \in \mathbb{N}} \text{ect}[C; D](f_n) \quad \text{(definition)}.
\]
- Case if $\{\phi\} \{D\}$. We have

$$\text{ect}[\text{if } (\phi) \{C\} \{D\}] = (\sup_n f_n)$$

$$(\text{definition})$$

$$= [\phi] \cdot \text{ect}[C] + [-\phi] \cdot \text{ect}[D]$$

$$(IH \text{ on } C \text{ and } D)$$

$$= [\phi] \cdot (\sup_n (f_n)) + [-\phi] \cdot (\sup_n (f_n))$$

$$= \sup_n [\phi] \cdot \text{ect}[C] + [-\phi] \cdot \text{ect}[D]$$

$$(\text{definition})$$

- Case while $\{\phi\} \{C\}$. Reconsider the characteristic function of while $\{\phi\} \{C\}$ wrt. expectation $f$:

$$\chi_f^{(\phi, C)} = \lambda F.[\phi] \cdot \text{ect}[C](F) + [-\phi] \cdot f.$$ 

Observe that (i) $\chi_{\sup_{n \in \mathbb{N}} f_n}^{(\phi, C)} = \sup_{n \in \mathbb{N}} \chi_f^{(\phi, C)}$, and moreover, from induction hypothesis it follows that all $\chi_f^{(\phi, C)}$ are continuous. The least fixed-point operator is continuous on continuous functions, particularly (ii) $\text{lfp}(\sup_{n \in \mathbb{N}} \chi_f^{(\phi, C)}) = \sup_{n \in \mathbb{N}} \text{lfp}(\chi_f^{(\phi, C)})$. Thus we get

$$\text{ect}[\text{while } (\phi) \{C\}] = (\sup_n f_n)$$

$$(\text{definition})$$

$$= \text{lfp}(\chi_f^{(\phi, C)})$$

$$(\text{observation (i)})$$

$$= \sup_{n \in \mathbb{N}} \text{lfp}(\chi_f^{(\phi, C)})$$

$$(\text{observation (ii)})$$

$$= \sup_{n \in \mathbb{N}} \text{ect}[\text{while } (\phi) \{C\}](f_n)$$

$$(\text{definition})$$

- Case $\{C\} \not\supseteq \{D\}$. For the case of non-deterministic choice, observe that for expectations $g_n$ and $h_n$ ($n \in \mathbb{N}$) we have

$$\max(\sup_n g_n, \sup_n h_n) = \sup_n \max(g_n, h_n).$$

Consequently,

$$\text{ect}[\{C\} \not\supseteq \{D\}] = (\sup_n f_n)$$

$$(\text{definition})$$

$$= \text{max}(\text{ect}[C](f_n), \text{ect}[D](f_n))$$

$$(IH \text{ on } C \text{ and } D)$$

$$= \text{max}(\sup_{n \in \mathbb{N}} \text{ect}[C](f_n), \sup_{n \in \mathbb{N}} \text{ect}[D](f_n))$$

$$(\text{observation})$$

$$= \sup_{n \in \mathbb{N}} \text{ect}[\{C\} \not\supseteq \{D\}](f_n)$$

$$(\text{definition})$$
Lemma B.4 (Monotonicity of Transformers).
\[ f \leq g \implies \text{ect}[C](f) \leq \text{ect}[C](g). \]

Proof. This is an immediate consequence of Lemma B.3.

Lemma 5.2 (Central Properties of $\text{ect}[\text{while}(\phi)\{C\}]$).

1. Continuity: $\text{ect}[C](\sup_{n \in \mathbb{N}} f_n) = \sup_{n \in \mathbb{N}} \text{ect}[C](f_n)$ for all $\omega$-chains $(f_n)_{n \in \mathbb{N}}$;
2. Monotonicity: $f \leq g \implies \text{ect}[C](f) \leq \text{ect}[C](g)$.


Proofs of Section 5.2

To ease readability of the proofs that follow, let us define recursively

\[ \text{ect}[\rightarrow](0)(f)(\gamma) \triangleq 0 \]
\[ \text{ect}[\rightarrow](n+1)(f)(\sigma) \triangleq f(\sigma) \]
\[ \text{ect}[\rightarrow](n+1)(f)(\sigma \triangleright C) \triangleq \sup\{w + E_\mu(\text{ect}[\rightarrow](n)(f)) \mid \sigma \triangleright C w \xrightarrow{\nu} \mu\}. \]

Then $\text{ect}[\rightarrow](n)$ give us finite approximations of $\text{ect}[\rightarrow](f)$ in the following sense.

Lemma B.5.
\[ \text{ect}[\rightarrow](f) = \sup_{n \in \mathbb{N}} \text{ect}[\rightarrow](n)(f). \]

Proof. Recall that $\text{ect}[\rightarrow](f) = \text{lfp}(\chi_f) = \chi_f^n(\emptyset)$, since terminal objects are precisely the stores $\sigma$ we get

\[ \chi_f(g) = \lambda \gamma: \begin{cases} f(\sigma) & \text{if } \gamma = \sigma \in \Sigma, \\ \sup\{w + E_\mu(g) \mid \gamma \xrightarrow{\nu} \mu\} & \text{else.} \end{cases} \]

It is thus sufficient to show that $\text{ect}[\rightarrow](n)(f) = \chi_f^n(\emptyset)$. This follows by a standard induction on $n$.

Observe that $\text{ect}[\rightarrow](n)(f)$ is monotone in $f$ and $m$, ie. $\text{ect}[\rightarrow](n)(f) \leq \text{ect}[\rightarrow](m)(g)$ if $f \leq g$ and $n \leq m$.

Lemma B.6.
\[ \text{ect}[\rightarrow](n)(f)(\bullet \triangleright C; D) \leq \text{ect}[\rightarrow](n)(\text{ect}[\rightarrow](n)(f)(\bullet \triangleright D))(\bullet \triangleright C). \]

Proof. The proof is by induction on $n$. It is sufficient to consider the inductive step. Fix $\sigma \in \Sigma$, and suppose $\sigma \triangleright C; D \xrightarrow{\nu} \mu$, hence $\sigma \triangleright C \xrightarrow{\nu} \nu$ where

\[ \mu = \big\{p_i : \sigma_i \triangleright D\big\}_{i \in I} \uplus \big\{q_j : \tau_j \triangleright C_j; D\big\}_{j \in J} \text{ for,} \]
\[ \nu = \big\{p_i : \sigma_i\big\}_{i \in I} \uplus \big\{q_j : \tau_j \triangleright C_j\big\}_{j \in J}. \]
Abbreviate \( g = \text{ect}[\rightarrow](n)(f)(\bullet \triangleright D) \) and observe that
\[
\mathbb{E}_\mu(\text{ect}[\rightarrow](n)(f)) \\
= \sum_{i \in I} p_i \cdot \text{ect}[\rightarrow](n)(f)(\sigma_i \triangleright D) \\
+ \sum_{j \in J} q_j \cdot \text{ect}[\rightarrow](n)(f)(\tau_j \triangleright C_j ; D)
\]
(induction hypothesis)
\[
\leq \sum_{i \in I} p_i \cdot \text{ect}[\rightarrow](n)(f)(\sigma_i \triangleright D) \\
+ \sum_{j \in J} q_j \cdot \text{ect}[\rightarrow](n)(g)(\tau_j \triangleright C_j)
\]
(since \( \text{ect}[\rightarrow](n)(f)(\sigma_i \triangleright D) = \text{ect}[\rightarrow](n)(g)(\sigma_i) \))
\[
= \mathbb{E}_\nu(\text{ect}[\rightarrow](n)(g))
\]
In conclusion
\[
\text{ect}[\rightarrow](n+1)(f)(\sigma \triangleright C ; D) \\
\text{(definition of } \text{ect}[\rightarrow](n+1)(f)\text{)} \\
= \sup\{w + \mathbb{E}_\mu(\text{ect}[\rightarrow](n)(f)) | \sigma \triangleright C; D \xrightarrow{\nu} \mu\} \\
\text{(above observation)} \\
\leq \sup\{w + \mathbb{E}_\nu(\text{ect}[\rightarrow](n)(g)(\sigma \triangleright C)) | \sigma \triangleright C \xrightarrow{\nu} \nu\} \\
\text{(definition of } \text{ect}[\rightarrow](n)(g)\text{)} \\
= \text{ect}[\rightarrow](n+1)(g)(\sigma \triangleright C) \\
\text{(since } \text{ect}[\rightarrow](n)(f)(\bullet \triangleright D) \text{ and monoton. of } \text{ect}[\rightarrow](n+1)\text{)} \\
\leq \text{ect}[\rightarrow](n+1)(\text{ect}[\rightarrow](n+1)(f)(\bullet \triangleright D))(\sigma \triangleright C). \quad \square
\]

**Lemma B.7 (Composition).**
\[
\text{ect}[\rightarrow](f)(\bullet \triangleright C ; D) = \text{ect}[\rightarrow](\text{ect}[\rightarrow](f)(\bullet \triangleright D))(\bullet \triangleright C).
\]

**Proof.** Note that \( \text{lhs} \leq \text{rhs} \) is an immediate consequence of Lemma B.5 and Lemma B.6. For the inverse direction, let us abbreviate \( g = \text{ect}[\rightarrow](f)(\bullet \triangleright D) \). By Lemma B.5, it is sufficient to show
\[
\text{ect}[\rightarrow](n)(g)(\bullet \triangleright C) \leq \text{ect}[\rightarrow](f)(\bullet \triangleright C ; D)
\]
for all \( n \in \mathbb{N} \). The proof is by induction on \( n \), we consider the inductive step. Suppose \( \sigma \triangleright C \xrightarrow{\nu} \mu \), hence \( \sigma \triangleright C; D \xrightarrow{\nu} \nu \) where
\[
\mu = \{ p_i : \sigma_i \} \cup \{ q_j : \tau_j \triangleright C_j \} \quad \text{and} \\
\nu = \{ p_i : \sigma_i \triangleright D \} \cup \{ q_j : \tau_j \triangleright C_j ; D \}.
\]

\[\text{Vol. 1, No. 1, Article . Publication date: August 2020.}\]
Observe

\[ E_\mu(\text{ect}[\rightarrow]^{(n)}(g)) \]

(definition of \( \text{ect}[\rightarrow]^{(n)}(g) \))

\[ = \sum_{i \in I} p_i \cdot g(\sigma_i) + \sum_{j \in J} q_j \cdot \text{ect}[\rightarrow]^{(n)}(g)(\tau_j \triangleright C_j) \]

(induction hypothesis)

\[ \leq \sum_{i \in I} p_i \cdot g(\sigma_i) + \sum_{j \in J} q_j \cdot \text{ect}[\rightarrow]^{(n+1)}(g)(\tau_j \triangleright C_j; D) \]

(since \( g(\sigma_i) = \text{ect}[\rightarrow](f)(\sigma_i \triangleright D) \))

\[ = E_\nu(\text{ect}[\rightarrow](f)). \]

Conclusively,

\[ \text{ect}[\rightarrow]^{(n+1)}(g)(\sigma \triangleright C) \]

(definition of \( \text{ect}[\rightarrow]^{(n+1)}(g) \))

\[ = \sup \{ w + E_\mu(\text{ect}[\rightarrow]^{(n)}(g)) \mid \sigma \triangleright C \xrightarrow{w} \mu \} \]

(above observation)

\[ \leq \sup \{ w + E_\nu(\text{ect}[\rightarrow](f)) \mid \sigma \triangleright C; D \xrightarrow{w} \nu \} \]

\[ = \text{ect}[\rightarrow](f)(\sigma \triangleright C; D). \quad \Box \]

**Lemma B.8 (Loops).**

\[ \text{ect}[\rightarrow](f)(\bullet \triangleright \text{while}(\phi) \{C\}) = \text{lfp}(g, [\phi] \cdot \text{ect}[\rightarrow](g)(\bullet \triangleright C) + [\neg \phi] \cdot f). \]

**Proof.** Define

\[ \Gamma_f^{(0)} \triangleq 0 \quad \Gamma_f^{(n+1)} \triangleq [\phi] \cdot \text{ect}[\rightarrow](\Gamma_f^{(n)})(\bullet \triangleright C) + [\neg \phi] \cdot f. \]

Consequently, the right-hand side \( rhs \) is equal to \( \sup_{n \in \mathbb{N}} \Gamma_f^{(n)} \). To prove \( lhs \leq rhs \), we show

\[ \text{ect}[\rightarrow]^{(n)}(f)(\bullet \triangleright \text{while}(\phi) \{C\}) \leq \Gamma_f^{(n)}. \]
by induction on \( n \). It is sufficient to consider the inductive step, where we have

\[
\text{ect}[\rightarrow]^{(n+1)}(f)(\bullet \trianglerightwhile (\phi) \{C\})
\]

\((\text{definition of } \text{ect}[\rightarrow]^{(n+1)}(f); \text{case analysis on } \sigma \models \phi)\)

\[
= [\phi] \cdot \text{ect}[\rightarrow]^{(n)}(f)(\bullet \triangleright C; \text{while } (\phi) \{C\}) + [-\phi] \cdot f
\]

\((\text{Lemma B.6})\)

\[
\leq [\phi] \cdot \text{ect}[\rightarrow]^{(n)}(\text{ect}[\rightarrow]^{(n)}(\bullet \triangleright while (\phi) \{C\})(\bullet \triangleright C)) + [-\phi] \cdot f
\]

\((\text{IH & monotonicity of } \text{ect}[\rightarrow]^{(n)}(\cdot)(\bullet \triangleright C))\)

\[
\leq [\phi] \cdot \text{ect}[\rightarrow]^{(n)}(\Gamma_f^{(n)})(\bullet \triangleright C) + [-\phi] \cdot f
\]

\((\text{definition of } \Gamma_f^{(n+1)})\)

\[
= \Gamma_f^{(n)}.
\]

Hence, by Lemma B.5,

\[
\text{lhs} = \sup_{n \in \mathbb{N}} \text{ect}[\rightarrow]^{(n)}(f)(\bullet \triangleright while (\phi) \{C\})
\]

\[
\leq \sup_{n \in \mathbb{N}} \Gamma_f^{(n)} = \text{rhs}.
\]

Finally, to prove \( \text{rhs} \leq \text{lhs} \), we show

\[
\Gamma_f^{(n)} \leq \text{ect}[\rightarrow](f)(\bullet \triangleright while (\phi) \{C\}),
\]

by induction on \( n \). Then

\[
\Gamma_f^{(n+1)}
\]

\((\text{definition of } \Gamma_f^{(n+1)})\)

\[
= [\phi] \cdot \text{ect}[\rightarrow](\Gamma_f^{(n)})(\bullet \triangleright C) + [-\phi] \cdot f
\]

\((\text{induction hypothesis})\)

\[
= [\phi] \cdot \text{ect}[\rightarrow](\text{ect}[\rightarrow](f)(\bullet \triangleright while (\phi) \{C\})(\bullet \triangleright C)) + [-\phi] \cdot f
\]

\((\text{Lemma B.6})\)

\[
= [\phi] \cdot \text{ect}[\rightarrow](f)(\bullet \triangleright while (\phi) \{C\}) + [-\phi] \cdot f
\]

\((\text{definition of } \text{ect}[\rightarrow](f); \text{case analysis on } \sigma \models \phi)\)

\[
= \text{ect}[\rightarrow](f)(\bullet \triangleright while (\phi) \{C\}).
\]

\[\Box\]

**Lemma 5.3 (Composition and Loop Lemma).**

1. \( \text{ect}[\rightarrow](f)(\bullet \triangleright C; D) = \text{ect}[\rightarrow](\text{ect}[\rightarrow](f)(\bullet \triangleright D)(\bullet \triangleright C)) \)
2. \( \text{ect}[\rightarrow](f)(\bullet \triangleright \text{while } (\phi) \{C\}) = \text{lfp}(\ell \cdot [\phi] \cdot \text{ect}[\rightarrow](\bullet \triangleright C) + [-\phi] \cdot f) \).

**Proof.** The two properties have been proven in Lemma B.7 and Lemma B.8, respectively. \[\Box\]

**Theorem 5.4 (Soundness & Completeness).** For every command \( C \in \text{Cmd} \), we have

1. \( \text{ect}[\rightarrow](f)(\bullet \triangleright C) = \text{ect}[C](f) \); and consequently
2. \( \text{ecost}[\rightarrow](\bullet \triangleright C) = \text{ecost}[C] \).

\[\text{Vol. 1, No. 1, Article } \text{. Publication date: August 2020.}\]
PROOF. The proof of this statement is by induction on the $C$. The only interesting cases are those of composition and loops, which follow by Lemma B.7 and Lemma B.8, respectively. □

PROOFS OF SECTION 6

**Lemma 6.2 (Separating Expected Cost and Value).**

\[
\text{ect}[C](f) \leq \text{ecost}[C] + \text{evalue}[C](f).
\]

**Proof.** One proofs the stronger property

\[
\text{ect}[C](f + g) \leq \text{ect}[C](g) + \text{evalue}[C](f)
\]

by a standard induction on $C$. 

\[
\text{ect}[\text{consume}(e)](f + g) = \langle e \rangle + (g + f) \\
= ((\langle e \rangle + g) + f \\
= \text{ect}[\text{consume}(e)](g) + \text{evalue}[\text{consume}(e)](f)
\]

\[
\text{ect}[\text{skip}](f + g) = f + g = \text{ect}[\text{skip}](g) + \text{evalue}[\text{skip}](f)
\]

\[
\text{ect}[\text{abort}](f + g) = 0 = \text{ect}[\text{abort}](g) + \text{evalue}[\text{abort}](f)
\]

\[
\text{ect}[C;D](f + g) = \text{ect}[C](\text{ect}[D](f + g)) \\
\leq \text{ect}[C](\text{ect}[D](g) + \text{evalue}[D](f)) \\
\leq \text{ect}[C]((\text{ect}[D](g)) + \text{evalue}[C](\text{evalue}[D](f)) \\
= \text{ect}[C;D](g) + \text{evalue}[C;D](f)
\]
while

and consequently, the inequality witnesses that $\mathfrak{m}_2$.

Theorem 6.5 (Modular Sequential Analysis). Let $\mathcal{C}$ and $\kappa$ be as in Lemma 6.3. Then

1. $\phi \models \text{ecost}[\mathcal{C}] + \kappa(\text{value}[\mathcal{C}](b_1), \ldots, \text{value}[\mathcal{C}](b_k)) \leq I \Rightarrow \text{ecost}[\text{while } (\phi) \{ \mathcal{C} \}] \leq I$;
2. $\phi \models \kappa(\text{value}[\mathcal{C}](b_1), \ldots, \text{value}[\mathcal{C}](b_k)) \leq I \land \neg \phi \Vdash f \leq I \Rightarrow \text{value}[\text{while } (\phi) \{ \mathcal{C} \}](f) \leq I$.

Proof. Concerning the first inequality, Lemma 6.2 and Lemma 6.3 yield $\lambda_f(\kappa(\bar{b})) \leq \lambda_f(\phi, \mathcal{C})(\kappa(\bar{b}))$, and consequently, the inequality witnesses that $\kappa(\bar{b})$ is an upper invariant (Theorem 5.5) for while $\{ \phi \}$ wrt $f$. Relying only on Lemma 6.3, the second inequality witnesses that $\kappa(\bar{b})$ is an upper invariant for the expected cost of

$$\text{ect}[\text{while } (\phi) \{ \text{costFree}(\mathcal{C}) \}](f)$$
$$= \text{ect}[\text{costFree}(\text{while } (\phi) \{ \mathcal{C} \})](f)$$
$$= \text{value}[\text{while } (\phi) \{ \mathcal{C} \}] .$$

Theorem 6.5 (Modular Sequential Analysis). Let $\mathcal{C}$ and $\kappa$ be as in Lemma 6.3. Then

1. $\text{ecost}[D](f) \leq \kappa(b_1, \ldots, b_k) \Rightarrow \text{ecost}[\mathcal{C}; D] \leq \text{ecost}[\mathcal{C}] + \kappa(\text{value}[\mathcal{C}](b_1), \ldots, \text{value}[\mathcal{C}](b_k))$;
2. $\text{value}[D](f) \leq \kappa(b_1, \ldots, b_k) \Rightarrow \text{value}[\mathcal{C}; D](f) \leq \kappa(\text{value}[\mathcal{C}](b_1), \ldots, \text{value}[\mathcal{C}](b_k))$. 
**Proof.** The statement follows from the definition of $\text{ect}[C]$, Lemma 6.2 and $\kappa$ being concave.

\[
\text{ect}[C; D](f) \\
= \text{ect}[C](\text{ect}[D](f)) \\
\leq \text{ect}[C](\kappa(\vec{b})) \\
= \text{ecost}[C] + \text{evaluate}[C](\kappa(\vec{b})) \\
= \text{ecost}[C] + \kappa(\text{evaluate}[C](\vec{b})).
\]

\[\square\]