# On Continuation-Passing Transformations and Expected Cost Analysis

MARTIN AVANZINI, Inria Sophia Antipolis – Méditerranée, France
GILLES BARTHE, Max Planck Institute for Security and Privacy, Germany
UGO DAL LAGO, University of Bologna, Italy and Inria Sophia Antipolis – Méditerranée, France

We define a continuation-passing style (CPS) translation for a typed $\lambda$-calculus with probabilistic choice, unbounded recursion, and a tick operator — for modeling cost. The target language is a (non-probabilistic) $\lambda$-calculus, enriched with a type of extended positive reals and a fixpoint operator. We then show that applying the CPS transform of an expression $M$ to the continuation $\lambda v.0$ yields the expected cost of $M$. We also introduce a formal system for higher-order logic, called EHOL, prove it sound, and show it can derive tight upper bounds on the expected cost of classic examples, including Coupon Collector and Random Walk. Moreover, we relate our translation to Kaminski et al.'s ert-calculus, showing that the latter can be recovered by applying our CPS translation to (a generalization of) the classic embedding of imperative programs into $\lambda$-calculus. Finally, we prove that the CPS transform of an expression can also be used to compute pre-expectations and to reason about almost sure termination.

CCS Concepts: • **Theory of computation → Program analysis**; • **Software and its engineering → Functional languages**.

Additional Key Words and Phrases: cost analysis, probabilistic programming, CPS transformation

**ACM Reference Format:**
Martin Avanzini, Gilles Barthe, and Ugo Dal Lago. 2021. On Continuation-Passing Transformations and Expected Cost Analysis. *Proc. ACM Program. Lang.* 5, ICFP, Article 87 (August 2021), 30 pages. https://doi.org/10.1145/3473592

## 1 INTRODUCTION

Randomized computation has been one of the most fruitful extensions of the standard, deterministic, computational model, since the birth of computer science [De Leeuw et al. 1956; III 1974; Rabin 1963; Santos 1969]. While randomization has been pervasive, and sometimes essential, in the design of, e.g., efficient algorithms [Motwani and Raghavan 1995] and cryptographic primitives [Goldwasser and Micali 1984], the development of a proper theory of randomized programming languages has been (starting from the pioneering works by Saheb-Jaromi [Saheb-Djahromi 1978] and Kozen [Kozen 1981]) much slower, and is still a very active research area (see, e.g., [Batz et al. 2021; Ehrhard et al. 2018]).

Among the program properties of interest, one certainly finds functional properties, like program correctness, but also nonfunctional, more intentional ones. An example of non-functional properties of particular interest are those related to the *execution cost*, a key property of programs and the main focus of cost analysis. In general, the intended result of cost analysis are *estimates* about the

Authors' addresses: Martin Avanzini, martin.avanzini@inria.fr, Inria Sophia Antipolis – Méditerranée, France; Gilles Barthe, gjbarthe@gmail.com, Max Planck Institute for Security and Privacy, Germany; Ugo Dal Lago, ugo.dallago@unibo.it, University of Bologna, Italy and Inria Sophia Antipolis – Méditerranée, France.


2475-1421/2021/8-ART87
https://doi.org/10.1145/3473592

execution cost, which can have various levels of accuracy. A wide range of approaches for cost analyses exist, based on, e.g., abstract interpretation [Albert et al. 2012], type systems [Jost et al. 2010] and program logics [Atkey 2011; Danielsson 2008; Nielson 1987; Radicek et al. 2018]. The latter are generally more expressive, and are well suited for examples that require either value sensitivity or functional verification.

This paper is concerned about cost analysis of randomized *higher-order* programs, i.e. programs which can sample values from chosen distributions during execution, and at the same time are capable of treating functions as first-class citizens. The evaluation of such a program results in a *distribution* of values, and has an *expected cost*, namely the average cost the program experiences along its execution. In this setting, the intended result of cost analysis is an *upper bound* for the *expected cost*. Some of the approaches used in the deterministic setting extend to the probabilistic setting, namely type systems and program logics [Kaminski and Katoen 2017; Kaminski et al. 2016; Olmedo et al. 2016]. In a higher-order setting, the only technique which has been studied is the one based on types, which have indeed been proved to be applicable to higher-order randomized languages [Avanzini et al. 2019a; Dal Lago and Grellois 2017; Wang et al. 2020]. The main advantage of adopting type systems is the inherent compositionality of the obtained methodology. On the other hand, type systems for randomized languages are generally designed around abstractions which prevent them to prove arbitrary bounds.

A different approach to analyse a given program is driven by *program transformations*: the program at hand is transformed into another one, possibly written in a simpler language for which some analysis methodologies have already been provided. This, of course, requires a proof that the result of the analysis of the transformed program can somehow be *lifted back* to the source program. Examples of such an approach are found in the realm of deterministic programs (e.g., [Avanzini et al. 2015; Cutler et al. 2020; Wegbreit 1975]). However, the design of program transformations for (higher-order) randomized programs is a largely unexplored research area. In particular, reducing expected cost analysis to another, better analysed, and conceptually simpler kind of analysis seems natural, but to the best of the authors knowledge, has not been studied so far. This is precisely what we do in this paper. Specifically, we make the following contributions:

- We provide a continuation-passing style (*CPS*) transformation for a probabilistic variation of Plotkin's PCF. The target of the transformation is given by the pure fragment of the source language, extended with a type of extended positive reals. We define a denotational semantics of the target language, by giving meaning to recursive definitions in terms of a non-standard interpretation of positive real numbers. Under this interpretation, the expected *cost* of the source program is the *denotation* of the target program applied to the continuation $\lambda v.0$. This way, the expected cost of the source program can be analysed, thanks to the program transformation, by looking at the extensional behavior of the pure program, thus taking advantage of any methodology for reasoning about the latter.
- As one example of such a methodology, we introduce a form of unary higher-order logic [Aguirre et al. 2017] in which statements about terms of the target language can be proved to hold in a sound way, even in the presence of recursive definitions. This crucially relies on a restriction on the kind of predicates one employs when dealing with functions defined as fixed points. Despite ruling out many logical formulas, this class includes the kind of predicates one is interested at while proving upper bounds on the expected cost.
- We illustrate the overall methodology via the analysis of two classic examples, viz a random walk example and an implementation of the coupon collectors problem.
- We show that the ert-calculus of Kaminski et al. [2018] — a weakest pre-expectation calculus for reasoning about the expected runtime of randomized algorithms written in an imperative probabilistic language — is recovered through an application of our CPS transformation on a

```
coupons : List(C) → List(C)
let coupons cs =
  letrec collect os =
    if cs ⊆ os
    then cs
    else collect (draw(cs)^✓ :: os)
  in collect [ ]
```

(a) The coupons function.

```
couponsCPS : List(C) → (List(C) → Real^+) → Real^+
let couponsCPS cs k =
  letrec collectCPS os k =
    if cs ⊆ os
    then k os
    else 1 + ∑_{c∈cs} 1/|cs| ∗ collectCPS (c :: os) k
  in collectCPS [ ] k
```

(b) The CPSed function.

Fig. 1. A functional implementation of the coupon-collector problem (a) and the result of the costed CPS transformation (b).

standard embedding of (probabilistic) imperative programs within our source language. Our methodology thus strictly extends upon the strength of the ert-calculus, a calculus that is not only sound but also *complete* for imperative, probabilistic programs.

- We show that our methodology is not limited to expected cost analysis, but can also be used to reason about pre-expectations and almost sure termination.

*Outline.* We start with an informal explanation in Section 2. The target language is defined in Section 3, and the transformation — as well as the target language — is formalised and proved correct in Section 4. In Section 5 we introduce the aforementioned higher-order logic. Section 6 shows the embedding of the ert-calculus in our setting. In Section 7 we briefly discuss some further implication of our results. Finally, we draw pointers to related work in Section 8 and conclude in Section 9.

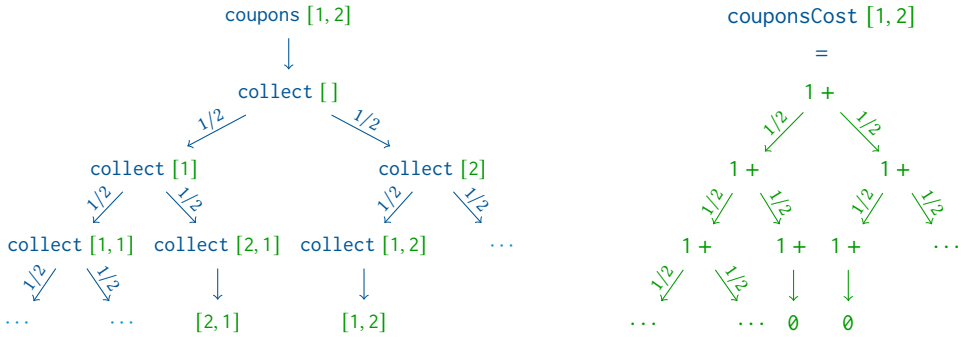
## 2 RANDOMIZED PROGRAMS AND CPS: A BIRD'S EYE VIEW

In this section, we illustrate how our continuation-passing transformation works on some concrete examples of randomized programs, and in particular on the so-called *coupon-collector* example. Consider the piece of code in Fig. 1a, written in a functional language. The idea is that someone wants to collect all the coupons in a list $cs$, and that she does so by keeping track of the coupons she already has in another, initially empty, list $os$. Coupons are collected by iteratively drawing coupons from $cs$ and adding them to $os$, until all the desired coupons in $cs$ are also part of $os$.

The program at hand can be seen as a randomized program, due to the presence of the primitive $\texttt{draw} : \texttt{List(C)} \to \texttt{C}$, which samples an element uniformly at random from the argument list. The program also produces another kind of effect, namely the one raised by the $(\cdot)^{\checkmark}$ primitive. This is meant to be a way of modeling cost from within the program: whenever a draw is made, the cost of the underlying computation is increased by 1. As an example, calling coupons with the argument list $[1, 2]$ gives rise to the recursion tree in Figure 2a. Note that, as indicated on the arrows, each recursive call happens with probability $1/2$ — the probability of drawing a coupon uniformly from the supplied list of coupons $[1, 2]$.

The main question now is: what is the *expected cost* of this program? What is the average amount of times the $(\cdot)^{\checkmark}$ primitive is executed? We could be very lucky, and incur a cost equal to the cardinality of $cs$, but oftentimes we get *the same coupon* more than once, so the number of draws required to get the whole collection $cs$ would be larger. We could even be very unlucky and indefinitely continue to get the same coupon, that is, the program diverges.

One way to tackle *cost analysis*, going back to the seminal work of Rosendahl [1989], lies in turning the program at hand into a second one, which is structurally quite similar but computes




(a) The probabilistic recursion tree of coupons [1, 2].

(b) The result of couponsCost [1, 2].

Fig. 2. The probabilistic recursion tree of coupons when supplied the initial list of coupons 1, 2 and the expected cost computed via the function couponsCost.

the cost of execution *in addition*. Such a *program transformation* thus turns an *intensional* property, namely the cost of execution, into an *extensional* one. Standard methods, such as *recurrence relations* employed by Rosendahl or *abstract interpretations* [Cousot and Cousot 1977] can establish now a cost analysis of the original, *source program*. While Rosendahl studied only a pure, first-order fragment of `Lisp`, more recently conceptually similar ideas have been successfully applied to the cost-analysis of higher-order, functional programs [Cutler et al. 2020; Danner et al. 2015]. None of these works, though, deal with randomized programs and expected cost analysis.

Coming back to the coupon collector example, translating this idea directly suggests that we are searching for a function of type $\texttt{List(C)} \to \texttt{List(C)} \times \texttt{Nat}$, which, given an initial list of coupons returns besides the list of collected coupons also the overall cost, as a natural number. Surely, such a term can be found, by simply threading through the computation a counter that is incremented whenever the $(\cdot)^{\checkmark}$ primitive is encountered. However, this is only of little help to our concerns: the resulting program would still be probabilistic. How would we arrive at an expected cost analysis, where the randomized behavior has been resolved? One way to overcome this problem — in the context of expected runtime analysis of imperative programs — has been given by Kaminski et al. [2018] in the form of a *pre-expectation calculus*. The expected cost *of a continuation* is gradually turned into that *of an overall computation*. In our setting, this would imply that the expected cost of a term $P : \sigma$ is indeed given in terms of a functional term $M : (\sigma \to \texttt{Real}^+) \to \texttt{Real}^+$. When supplied with an argument $\kappa\colon \sigma \to \texttt{Real}^+$ expressing the (non-negative) cost of the underlying continuation, this term is supposed to return the overall expected cost of evaluating $P$ followed by the continuation supplied with the result of $P$.

Concerning the term $P = \texttt{draw}(cs)$, for instance, such a cost transformer $M$ is given $\lambda k.\sum_{c \in cs} {}^1\!/_{|cs|} * k\ c$ for $|cs|$ the cardinality of $cs$.[1] Notice how this term simply gives the expected value of $k$ on the uniform distribution on coupons $cs$. How do we lift this idea to arbitrary programs? As it turns out, on the non-probabilistic fragment of the language a standard *continuation-passing style* (*CPS*) *transformation* is sufficient. In the case of $(\cdot)^{\checkmark}$, we simply add one to the output, and in the case of sampling primitives we proceed as above. Thereby, we have turned a CPS transformation into an *expected cost transformer*, much in the spirit of the expected runtime transformer of Kaminski et al. [2018]. Indeed, the overall approach is quite reminiscent of the way Ramsey and Pfeffer [2002]

[1] Avoiding syntactic sugar, in a standard functional language this term could be defined as $\lambda k.\texttt{sum}\ (\texttt{map}\ ((*\ {}^1\!/_{|cs|}) \circ k)\ cs)$.

give semantics to a stochastic lambda calculus within Haskell, with the main difference that we're focusing on costs and directly compute expectations.

When applied to the function coupons, the so suited CPS transformation results in the program given in Fig. 1b. In effect, couponsCPS computes the expected cost of coupons *together with* the one of the argument continuation when applied to the result of coupons. The sum of the two is what couponsCPS finally produces in output. Please note how the continuation $k$ is passed around from the current call to collectCPS to the next one.

But what if we are interested in the cost of coupons *alone*? Well, it suffices to apply couponsCPS to the continuation $\lambda v.\mathtt{0}$, obtaining something like the following program:

$$\mathtt{couponsCost} : \mathtt{List(C)} \to \mathtt{Real}^+$$

$$\begin{array}{l}\mathtt{let\ couponsCost}\ cs = \\ \quad \mathtt{letrec\ collectCost}\ os = \\ \quad\quad \mathtt{if}\ cs \subseteq os\ \mathtt{then\ 0\ else\ 1} + \sum_{c \in cs} {}^{1}\!/_{|cs|} * \mathtt{collectCost}\ (c :: os) \\ \quad \mathtt{in\ collectCost}\ [\,]\end{array}$$

At least, now, we have obtained a program which *looks like* it can then be used to analyse the expected cost of coupons, e.g. via standard type-theoretical tools. A careful look at couponsCost reveals that this is not really the case, at least if couponsCost is interpreted as a deterministic functional program. Indeed, if $cs$ is *not* the empty list, then executing couponsCost $cs$ produces nothing less than an infinite tree of recursive calls to collectCost. For example, the recursive calls induced by couponsCost [1, 2] forms a tree isomorphic to the one in Figure 2a. In which sense, then, does couponsCost $cs$ compute the expected cost of coupons $cs$? The answer is that it does so only *up to approximations*. As an example, couponsCost [1, 2] can be seen as computing a numerical expression, namely that in Figure 2b which, being infinitary, only denotes a real number (or infinity) when the type of extended positive real numbers $\mathtt{Real}^+$ (and related operators) is interpreted in a slightly non-standard way. In other words, the fact the tree is infinite *does not* result in undefinedness, but rather in an approximation process.

Some questions, then, remain unanswered. In which sense is the transformation from coupons to couponsCPS correct? How could we analyse the behavior of any program in the target language of our CPS transformation? Clearly, these questions cannot be answered by the same kinds of techniques employed classically, and this is precisely the bulk of our technical contribution. To prove correctness of our CPS transformation, with the operational semantics of randomized programming languages being inherently infinitary, one has to reason up to approximations. As the source language of randomized programs is infinitary, so is the target language. To reason about programs in the target language, we introduce a higher-order logic, dubbed *EHOL*. Very briefly, in this logic judgments have the form

$$\Gamma \mid \Phi \vdash M : \sigma \mid \phi\ ,$$

that can be seen as an extension of the usual typing-judgment $\Gamma \vdash M : \sigma$ with assertions $\Phi$ acting as assumptions and $\phi$ an assertion talking about the term $M$ through a distinguished variable $\mathbf{r}$.

The novel aspect of EHOL lies in the fact that it can be used to reason about infinitary computations such as the one underlying couponsCost. This is achieved, not through constraining the logical system and therefore inherently limiting its power, but rather by considering well-behaved assertions $\phi$ whenever dealing with recursive definitions. In short, such predicates have to be continuous in $\mathbf{r}$, thereby enabling reasoning on infinite computations, such as the one illustrated in Figure 2b, via their finite approximations. Such continuous assertions are syntactically captured in our notion of $\leq$-positive assertions (see Section 5). Crucially, these encompass assertions such as $\mathbf{r} \leq N$ that represent upper-bounds.

**Typing rules, pure fragment**

$$\frac{\Gamma(x) = \sigma}{\Gamma \vdash x : \sigma} \qquad \frac{\Gamma; x : \sigma \vdash P : \tau}{\Gamma \vdash \lambda x.P : \sigma \to \tau} \qquad \frac{\Gamma \vdash P : \sigma \to \tau \quad \Gamma \vdash Q : \sigma}{\Gamma \vdash P \cdot Q : \tau} \qquad \frac{\Gamma; f : \tau \to \sigma; x : \tau \vdash P : \sigma}{\Gamma \vdash \mathtt{letrec}\ f\ x = P : \tau \to \sigma}$$

$$\frac{\mathsf{s} \in \mathcal{C} \cup \mathcal{F} \quad \mathsf{s} : \mathsf{B}_1 \times \cdots \times \mathsf{B}_n \to \mathsf{B} \quad \Gamma \vdash P_1 : \mathsf{B}_1 \quad \cdots \quad \Gamma \vdash P_n : \mathsf{B}_n}{\Gamma \vdash \mathsf{s}(P_1, \ldots, P_n) : \mathsf{B}}$$

$$\frac{\Gamma \vdash P : \mathsf{B} \quad \mathsf{c} : \mathsf{B}_1 \times \cdots \times \mathsf{B}_k \to \mathsf{B} \quad \Gamma; x_1 : \mathsf{B}_1; \cdots x_k : \mathsf{B}_k \vdash Q : \sigma \quad \Gamma; y : \mathsf{B} \vdash R : \sigma}{\Gamma \vdash \mathsf{case}\ P\ \mathsf{of}\ \{\mathsf{c}(\vec{x}) \mapsto Q \mid y \mapsto R\} : \sigma}$$

**Typing rules, non-pure fragment**

$$\frac{\Gamma \vdash P : \sigma}{\Gamma \vdash P^{\checkmark} : \sigma} \qquad \frac{\mathsf{p} : \mathsf{B}_1 \times \cdots \times \mathsf{B}_n \to \mathsf{B} \quad \Gamma \vdash P_1 : \mathsf{B}_1 \quad \cdots \quad \Gamma \vdash P_n : \mathsf{B}_n}{\Gamma \vdash \mathsf{p}(P_1, \ldots, P_n) : \mathsf{B}}$$

Fig. 3. Simple Typing Rules.

**Reduction rules, pure fragment**

$$(\lambda x.P) \cdot V \mapsto_\beta P[V/x]$$
$$(\mathtt{letrec}\ f\ x = P) \cdot V \mapsto_\mu P[\mathtt{letrec}\ f\ x = P/f][V/x]$$
$$\mathsf{case}\ \mathsf{c}(\vec{V})\ \mathsf{of}\ \{\mathsf{c}(\vec{x}) \mapsto P \mid y \mapsto Q\} \mapsto_\iota P[\vec{V}/\vec{x}]$$
$$\mathsf{case}\ \mathsf{d}(\vec{V})\ \mathsf{of}\ \{\mathsf{c}(\vec{x}) \mapsto P \mid y \mapsto Q\} \mapsto_\iota Q[\mathsf{d}(\vec{V})/y] \qquad (\mathsf{c} \neq \mathsf{d})$$
$$\mathsf{f}(V_1, \ldots, V_n) \mapsto_\delta \mathsf{f}_{\mathcal{I}}(V_1, \ldots, V_n)$$

**Reduction rules, non-pure fragment**

$$P^{\checkmark} \overset{1}{\mapsto}_\epsilon P \qquad \mathsf{p}(V_1, \ldots, V_n) \mapsto_\sigma \mathsf{p}_{\mathcal{I}}(V_1, \ldots, V_n)$$

Fig. 4. Single Step Reduction Relations of the Pure and non-Pure Fragments.

Returning to our example, within EHOL we can derive validity of the judgment

$$\cdot \mid \cdot \vdash \mathtt{couponsCost} : \mathtt{List(C)} \to \mathtt{Real}^+ \mid \forall cs : \mathtt{List(C)}.\ \mathbf{r} \leq |cs| \cdot \textstyle\sum_{n=1}^{|cs|} 1/n\ ,$$

yielding the optimal bound for couponsCost in terms of the provided list of coupons $cs$, which then translates to a bound on the cost of coupons.

Summing up, what we then have obtained is a novel methodology for reasoning about the expected cost of probabilistic programs. Our approach distinguishes itself through its expressiveness and — as we demonstrate on a non-trivial example — allows a form of compositional reasoning that scales well also to the presence of higher-order combinators.

## 3 A PROBABILISTIC HIGHER-ORDER LANGUAGE

In this section we define our vehicle calculus. Its pure fragment, defined first, is a mild extension of Plotkin's PCF with built-in operations and algebraic data types such as lists or trees. The impure fragment then extends upon this language with primitives for sampling and cost.

### 3.1 The Pure Fragment

We consider a simply-typed language, with *base types* drawn from a set $\mathcal{B}$, such as Bool for Booleans, Int for integer numbers or List($\sigma$) for *lists* over elements of type $\sigma \in \mathcal{B}$. Simple types are given

by the following grammar:

$$\sigma, \tau ::= \mathsf{B} \mid \sigma \to \tau \qquad \textit{(Simple Types)}$$

As usual, $\to$ binds to the right.

To define terms, we consider three disjoint sets $\mathcal{X}, \mathcal{C}$ and $\mathcal{F}$, of *variables*, *constructor (symbols)* and *function (symbols)*, respectively. All three sets may be infinite but, if not mentioned otherwise, are countable. Each symbol $\mathsf{s} \in \mathcal{C} \cup \mathcal{F}$ is associated with a type schema, in notation $\mathsf{s} : \mathsf{B}_1 \times \cdots \times \mathsf{B}_n \to \mathsf{B}$ or simply $\mathsf{s} : \mathsf{B}$ when $n = 0$. While constructors serve to form ground values, any function symbol computes a function on ground values. For instance, we may use constructors $\mathtt{i} : \mathtt{Int} \in \mathcal{C}$ to embed integer numbers $i \in \mathbb{Z}$ within the language, and introduce function symbols such as $(\mathtt{+}) : \mathtt{Int} \times \mathtt{Int} \to \mathtt{Int}$ that implement arithmetic operations on such values. Without loss of generality, we also impose that primitive operations are fully applied, so for instance $\mathtt{+}\, x$ is not a valid term. *Values* and *terms* with free variables $\mathcal{X}$ are formed from the following grammars, where $\mathsf{c} \in \mathcal{C}$, $\mathsf{s} \in \mathcal{C} \cup \mathcal{F}$ and $f, x, \vec{x}$ and $y$ range over variables in $\mathcal{X}$.

$$V, W ::= x \mid \lambda x.P \mid \mathtt{letrec}\ f\ x = P \mid \mathsf{c}(V_1, \ldots, V_n) \qquad \textit{(values)}$$

$$P, Q, R ::= V \mid \mathsf{s}(P_1, \ldots, P_n) \mid P \cdot Q \mid \mathtt{case}\ P\ \mathtt{of}\ \{\mathsf{c}(\vec{x}) \mapsto Q \mid y \mapsto R\} \quad \textit{(terms).}$$

Application, which we indicate as $(\cdot)$, binds to the left.

In examples, and as we have already done before, we may use several standard shortcuts, such as $\mathtt{if}\ G\ \mathtt{then}\ P\ \mathtt{else}\ Q$ as an abbreviation for $\mathtt{case}\ G\ \mathtt{of}\ \{\mathtt{true} \mapsto P \mid \_ \mapsto Q\}$ and $\mathtt{let}\ x = P\ \mathtt{in}\ Q$ for $(\lambda x.Q) \cdot P$. Throughout the following, we consider only well-typed terms, that is we tacitly assume that any term $P$ is such that $\Gamma \vdash P : \sigma$ holds according to the rules from Figure 3 for some *typing context* $\Gamma$ (a mapping from variables to simple types) and type $\sigma$. We denote by $\Lambda^\sigma(\Gamma)$ the set of all such terms, we use $\Lambda^\sigma \triangleq \Lambda^\sigma(\varnothing)$ to denote the set of ground terms of type $\sigma$, and we let $\Lambda \triangleq \bigcup_\sigma \Lambda^\sigma$ denote the set of all well-typed terms. In correspondence, the sets $\mathcal{V}^\sigma(\Gamma)$, $\mathcal{V}^\sigma$ and $\mathcal{V}$ denote the well-typed (ground) values (of type $\sigma$).

We now turn to the semantics of expressions. To each function symbol $\mathsf{f} : \mathsf{B}_1 \times \cdots \times \mathsf{B}_n \to \mathsf{B}$ we associate a *primitive* $\mathsf{f}_\mathcal{I} : \mathcal{V}^{\mathsf{B}_1} \times \cdots \times \mathcal{V}^{\mathsf{B}_n} \to \mathcal{V}^{\mathsf{B}}$ which is meant to capture the intended semantics of $\mathsf{f}$. *Evaluation contexts* are formed from the following grammar, where $\mathsf{s} \in \mathcal{C} \cup \mathcal{F}$.

$$\mathbb{C}, \mathbb{D} ::= \Box \mid P \cdot \mathbb{C} \mid \mathbb{C} \cdot V \mid \mathsf{s}(\vec{P}, \mathbb{C}, \vec{V}) \mid \mathtt{case}\ \mathbb{C}\ \mathtt{of}\ \{\mathsf{c}(\vec{x}) \mapsto P \mid x \mapsto Q\} \quad \textit{(evaluation contexts).}$$

With $\mathbb{C}[P]$ we denote the term obtained by replacing the hole $\Box$ in $\mathbb{C}$ by $P$. We denote by $\longrightarrow_l \subseteq \Lambda \times \Lambda$ ($l \in \{\beta, \mu, \iota, \delta\}$) the contextual closure of the reduction rules $\mapsto_l$ depicted in Figure 4; with $\longrightarrow$ we denote the union of these three relations. Notice that evaluation is right-to-left.

Up to now, the introduced language can be seen as nothing more than a standard extension of (call-by-value) PCF to a given set of base types which includes, but which is not necessarily limited to, the natural numbers. The next step consists of rendering our vehicle calculus effectful.

### 3.2 The Non-pure Fragment

The randomized higher-order language $\Lambda_p$ that we are considering is an extension of the pure language $\Lambda$ with primitives for sampling and cost. For the sake of simplicity, we stick to a language in which sampling can be done from *discrete* distributions. However, we believe all our results could be lifted to a setting in which measure theory plays a key role, see e.g. [Ramsey and Pfeffer 2002].

*Preliminaries and Syntax.* Let $\mathrm{D}(A)$ denote the set of *discrete (sub)distributions* over $A$, namely the set of total functions $d$ from $A$ to $\mathbb{R}_{[0,1]}$ whose sum is itself in $\mathbb{R}_{[0,1]}$ and whose support $\mathrm{supp}(d) = \{a \in A \mid d(a) \neq 0\}$ is countable. Moreover, let $\mathbb{R}^+$ denote the non-negative reals, and $\mathbb{R}^{+\infty}$ its extension by $\infty$. We denote by $\{a_i^{p_i}\}_{i \in I}$ the distribution in $\mathrm{D}(A)$ assigning probability $p_i$

to $a_i \in A$ for every $i \in I$, or simply by $\{a_1^{p_1}, \ldots, a_n^{p_n}\}$ when the support $\{a_1, \ldots, a_n\}$ is finite. The expected value of a function $f : A \to \mathbb{R}^+$ on $d \in \mathsf{D}(A)$ is denoted by $\mathbb{E}_d(f)$. Since we restrict ourselves to discrete distributions, this is given by $\sum_{a \in \mathrm{supp}(d)} d(a) \cdot f(a)$. Finally, we denote by $\sum_{i \in I} p_i \cdot d_i$ the *convex combination of distributions* $d_i$. Note that probabilities $p_i$ are expected to sum up to $\leq 1$, thereby the convex combination always results in a distribution. For instance, we have $1/2 \cdot \{a^1\} + 1/2 \cdot \{b^{1/3}, c^{2/3}\} = \{a^{1/2}, b^{1/6}, c^{1/3}\}$.

To incorporate probabilistic sampling, we assume a designated set of *distribution symbols* $\mathcal{D}$. As for constructors and functions, each symbol $\mathsf{p} \in \mathcal{D}$ is given a type schema $\mathsf{p} : \mathsf{B}_1 \times \cdots \times \mathsf{B}_n \to \mathsf{B}$. As expected, each such $\mathsf{p}$ is associated with a *primitive* $\mathsf{p}_{\mathcal{I}} : \mathcal{V}^{\mathsf{B}_1} \times \cdots \times \mathcal{V}^{\mathsf{B}_n} \to \mathsf{D}(\mathcal{V}^{\mathsf{B}})$. For instance, we may consider $\mathtt{ber} : \mathtt{Ratio} \to \mathtt{Bool} \in \mathcal{D}$ with $\mathtt{ber}_{\mathcal{I}}(\mathsf{q}) \triangleq \{\mathtt{true}^q, \mathtt{false}^{1-q}\}$ for any rational $0 \leq q \leq 1$. Note that $\mathsf{p}_{\mathcal{I}}(V_1, \ldots, V_n)$ can be a proper sub-distribution, in which case the missing probability signals abnormal termination. Semantically, this will be interpreted identical to non-termination. For instance, we may set $\mathtt{ber}_{\mathcal{I}}(\mathsf{q}) \triangleq \varnothing$ when $q$ is not in the interval $[0, 1]$.

To endow programs with a cost model, as mentioned before, we make use of an operator $(\cdot)^{\checkmark}$ that incurs a cost of one to the cost of evaluating its arguments. In summary, impure terms are obtained by extending the pure terms as follows:

$$P, Q, R ::= \ldots \mid \mathsf{p}(P_1 \ldots, P_n) \mid P^{\checkmark} \qquad \textit{(impure terms).}$$

With $\Lambda_p^{\sigma}(\Gamma)$ we denote the set of impure terms typeable under typing context $\Gamma$ as $\sigma$ (see Fig. 3) and, identical to before, $\Lambda_p$ ($\Lambda_p^{\sigma}$) denotes the set of well-typed terms (of type $\sigma$).

*Example 3.1 (Random Walks).* For illustration, consider the following recursively defined function:

$$\mathtt{walk} : (\mathtt{Int} \to \mathtt{Bool}) \to (\mathtt{Int} \to \mathtt{Int}) \to \mathtt{Int} \to \mathtt{Int}$$

$$\mathtt{walk} \triangleq (\mathtt{letrec}\ w\ p\ f\ n = \mathtt{if}\ p \cdot n\ \mathtt{then}\ n\ \mathtt{else}\ (w \cdot p \cdot f \cdot (f \cdot n))^{\checkmark})$$

This function iterates as long as the provided predicate $p : \mathtt{Int} \to \mathtt{Bool}$ holds, with the next recursion parameter $n : \mathtt{Int}$ computed by a stepping function $f : \mathtt{Int} \to \mathtt{Int}$. The tick in the else-branch signifies that we are interested in a cost corresponding to the number of recursive calls. While $\mathtt{walk}$ is itself non-probabilistic, we can apply it to several interesting probabilistic argument functions. As an example, consider

$$\mathtt{geo} \triangleq \mathtt{walk} \cdot (\lambda\_.\mathtt{ber}(1/2)) \cdot (\lambda n.n + 1)\ .$$

Then, for instance, the output of $\mathtt{geo} \cdot \mathtt{0}$ follows the *geometric distribution* $\{\mathtt{0}^{1/2}, \mathtt{1}^{1/4}, \mathtt{2}^{1/8}, \ldots\}$. As we will see, the expected number of iterations, i.e., the expected cost of $\mathtt{geo}\ \mathtt{0}$ is constant. As another example, consider the term

$$\mathtt{randomWalk}_{\mathsf{p}} \triangleq \mathtt{walk} \cdot (\lambda n.n \leq \mathtt{0}) \cdot (\lambda n.\mathtt{if}\ \mathtt{ber}(\mathsf{p})\ \mathtt{then}\ n - \mathtt{1}\ \mathtt{else}\ n + \mathtt{1})\ ,$$

parameterised by a rational constant $\mathsf{p}$, which performs a biased random walk over positive integers. It is folklore that such a random walk over non-negative integers is bounded only when $\mathsf{p} > 1/2$, i.e., when the walk is more likely to go down than up.

*Operational Semantics.* In order to capture the two effects, probabilistic sampling and cost, we express the operational semantics of the language $\Lambda_p$ by a *(probabilistic) reduction relation* $\longrightarrow \subseteq \mathsf{D}(\Lambda_p) \times \mathbb{R}^{+\infty} \times \mathsf{D}(\Lambda_p)$, defined through a (weighted) probabilistic abstract reduction system [Avanzini et al. 2020; Bournez and Garnier 2005] over terms $\Lambda_p$. Triples $(d, r, e) \in \longrightarrow$, written as $d \xrightarrow{r} e$, signify that the term distribution $d$ evolves in one-step to a reduct-distribution $e$, producing an expected cost equal to $r$.

$$\frac{}{\{V^1\} \xrightarrow{0} \{V^1\}}\,[\textsc{Val}] \qquad \frac{P \overset{r}{\mapsto}_l \{Q_i^{p_i}\}_{i\in I} \quad l \in \{\beta,\mu,\iota,\delta,\epsilon,\sigma\}}{\{\mathbb{C}[P]^1\} \xrightarrow{r} \{\mathbb{C}[Q_i]^{p_i}\}_{i\in I}}\,[\textsc{Step}]$$

$$\frac{\forall i \in I. d_i \xrightarrow{r_i} e_i \quad \sum_{i\in I} p_i \le 1 \quad r = \sum_{i\in I} p_i \cdot r_i}{\sum_{i\in I} p_i \cdot d_i \xrightarrow{r} \sum_{i\in I} p_i \cdot e_i}\,[\textsc{Conv}]$$

Fig. 5. Probabilistic Reduction Relation on Term-Distributions.

To define this relation formally, let us first extend evaluation contexts to the impure fragment by setting

$$\mathbb{C}, \mathbb{D} ::= \cdots \mid \mathsf{p}(\vec{P}, \mathbb{C}, \vec{V}) \qquad \textit{(impure evaluation contexts)}\,.$$

The reduction rules for the two non-pure constructs are given in the second part of Figure 4. The operator $(\cdot)^{\checkmark}$ produces a cost of one without evaluating its argument, thereby attributing functions such as the ticked $\Omega$-term $\mathtt{letrec}\ f\,x = (f \cdot x)^{\checkmark}$ an infinite cost, rather than a cost of zero. Saying it another way, we are *call-by-value*, but *cost-by-name*.

To avoid notational overhead, we may drop $r$ in $d \xrightarrow{r} e$ if $r = 0$ and identify Dirac distributions $\{P^1\}$ with terms $P$. Thereby, all reduction rules $\mapsto_l$ from Figure 4, even the pure ones, can be seen as ternary relations $\mapsto_l \subseteq \Lambda_p \times \mathbb{R}^{+\infty} \times \mathrm{D}(\Lambda_p)$. Based on these, in Figure 5 we define the probabilistic reduction relation on distributions. Informally, $d \xrightarrow{r} e$ if $e$ is obtained by replacing in $d$ all reducible terms by corresponding reduct-distributions, preserving values. The cost $r$ is given by the average cost of all involved reduction steps. Formally, it is most convenient to define this relation through the three inference rules from Figure 5. Let us look at probabilistic rewriting defined this way on an example.

*Example 3.2 (Example 3.1 continued).* Reconsider the term $\mathtt{geo}$ and let us abbreviate $(\lambda\_.\mathtt{ber}(1/2))$ and $(\lambda n.n + 1)$ by $\mathtt{p}$ and $\mathtt{f}$, respectively. Thus $\mathtt{geo} = \mathtt{walk}\ \mathtt{p}\ \mathtt{f}$ which, when supplied the integer $\mathtt{n} \in C$, gives rise to the probabilistic reduction sequence

$$\begin{aligned}
\{\underline{\mathtt{walk}\cdot\mathtt{p}\cdot\mathtt{f}\cdot\mathtt{n}}^1\} &\xrightarrow{0} \{\mathtt{if}\ \underline{\mathtt{p}\cdot\mathtt{n}}\ \mathtt{then}\ \mathtt{n}\ \mathtt{else}\ (\mathtt{walk}\cdot\mathtt{p}\cdot\mathtt{f}\cdot(\mathtt{f}\cdot\mathtt{n}))^{\checkmark 1}\} \\
&\xrightarrow{0} \{\mathtt{if}\ \underline{\mathtt{ber}(1/2)}\ \mathtt{then}\ \mathtt{n}\ \mathtt{else}\ (\mathtt{walk}\cdot\mathtt{p}\cdot\mathtt{f}\cdot(\mathtt{f}\cdot\mathtt{n}))^{\checkmark 1}\} \\
&\xrightarrow{0} \{\underline{\mathtt{if}\ \mathtt{true}\ \mathtt{then}\ \mathtt{n}\ \mathtt{else}\ (\mathtt{walk}\cdot\mathtt{p}\cdot\mathtt{f}\cdot(\mathtt{f}\cdot\mathtt{n}))^{\checkmark 1/2}}, \\
&\qquad \underline{\mathtt{if}\ \mathtt{false}\ \mathtt{then}\ \mathtt{n}\ \mathtt{else}\ (\mathtt{walk}\cdot\mathtt{p}\cdot\mathtt{f}\cdot(\mathtt{f}\cdot\mathtt{n}))^{\checkmark 1/2}}\} \\
&\xrightarrow{0} \{\mathtt{n}^{1/2}; \underline{(\mathtt{walk}\cdot\mathtt{p}\cdot\mathtt{f}\cdot(\mathtt{f}\cdot\mathtt{n}))^{\checkmark 1/2}}\} \\
&\xrightarrow{1/2} \{\mathtt{n}^{1/2}; \mathtt{walk}\cdot\mathtt{p}\cdot\mathtt{f}\cdot(\underline{\mathtt{f}\cdot\mathtt{n}})^{1/2}\} \xrightarrow{0} \cdots\,,
\end{aligned}$$

where redexes are underlined. Reducing the redex $\mathtt{ber}(1/2)$ in the third step effectively forks the reduction into two probabilistic branches, each weighted with probability $1/2$. Continuing the reduction on the branch where $\mathtt{ber}(1/2)$ reduces to $\mathtt{true}$ yields the constant $\mathtt{n}$. This constant now persists throughout the overall reduction. On the other branch, we eventually reach the recursive call to $\mathtt{walk}\cdot\mathtt{p}\cdot\mathtt{f}\cdot(\mathtt{f}\cdot\mathtt{n})$. Here, the cost incurred by the tick, weighted with the corresponding probability $1/2$, is reflected in the reduction step. This process now repeats, *ad libitum*, halving however the probability of performing a recursive call at each iteration, and thereby also the

imposed costs. Writing $d \xrightarrow{r}{}^{*} e$ for a finite sequence $d = d_0 \xrightarrow{r_1} \dots \xrightarrow{r_n} d e_n = e$ with $r = \sum_{i=1}^{n} r_i$, a reduction of $\mathtt{geo} \cdot \mathtt{0}$ has overall the shape

$$\{(\underline{\mathtt{walk} \cdot \mathtt{p} \cdot \mathtt{f} \cdot \mathtt{0}})^{1}\} \xrightarrow{1/2}{}^{*} \{\mathtt{0}^{1/2}, (\underline{\mathtt{walk} \cdot \mathtt{p} \cdot \mathtt{f} \cdot \mathtt{1}})^{1/2}\} \xrightarrow{1/4}{}^{*} \{\mathtt{0}^{1/2}, \mathtt{1}^{1/4}, (\underline{\mathtt{walk} \cdot \mathtt{p} \cdot \mathtt{f} \cdot \mathtt{2}})^{1/4}\} \xrightarrow{1/8}{}^{*} \cdots .$$

This infinite sequence gradually approaches the geometric distribution $\{\mathtt{0}^{1/2}, \mathtt{1}^{1/4}, \mathtt{2}^{1/8}, \dots\}$, with an overall cost of $1/2 + 1/4 + 1/8 + \cdots = 1$.

As hinted by this example, the reduction relation $\longrightarrow$ is deterministic, i.e., for any term $P$, there is precisely one maximal (and infinite) reduction sequence

$$\{P^1\} = d_0 \xrightarrow{r_0} d_1 \xrightarrow{r_1} d_2 \xrightarrow{r_2} \cdots .$$

As we just saw, such a reduction sequence can be seen as gradually approaching a *normal-form distribution* $\mathsf{nf}(P) \in \mathsf{D}(\mathcal{V})$ over values, giving the distribution of values to which $P$ evaluates to. From such a sequence we can also compute an element $\mathsf{ecost}(P) \in \mathbb{R}^{+\infty}$ which stands for the cost of the *whole* computation starting from $P$. Formally, the *expected cost* $\mathsf{ecost}(P) \in \mathbb{R}^{+\infty}$ of this evaluation is given by the sum of all the $r_i$. Equivalently, $\mathsf{ecost} : \Lambda_p \to \mathbb{R}^{+\infty}$ can be defined as the least function, ordered point-wise, satisfying the following equations:

$$\mathsf{ecost}(V) = 0 \qquad \text{and} \qquad \mathsf{ecost}(P) = r + \sum_{i \in I} p_i \cdot \mathsf{ecost}(Q_i) \quad \text{if } P \xrightarrow{r} \{Q_i^{p_i}\}_{i \in I}.$$

Note that the equations are exhaustive, since every typeable term is either a value or reducible.[2] Via a telescoping-sum argument, it can be shown that $\mathsf{ecost}(P)$ coincides with the mean cost emitted along all probabilistic reduction paths, compare e.g. [Avanzini et al. 2020], thereby matching its intended meaning. In a similar spirit, $\mathsf{nf} : \Lambda_p \to \mathsf{D}(\mathcal{V})$ can be defined as the least function such that:

$$\mathsf{nf}(V) = \{V^1\} \qquad \text{and} \qquad \mathsf{nf}(P) = \sum_{i \in I} p_i \cdot \mathsf{nf}(Q_i) \quad \text{if } P \xrightarrow{r} \{Q_i^{p_i}\}_{i \in I}.$$

Here, distributions are ordered point-wise. The fact that $\mathsf{nf}(P)$ is well-defined for every $P \in \Lambda_p$ comes from the fact that distributions form an cpo with respect to the point-wise ordering.

### 3.3 A Semantic Expected Cost Transformer

The expected cost transformer outlined in Section 2 lifts the expected cost $f : \mathcal{V} \to \mathbb{R}^{+\infty}$ of a continuation to that of evaluating a term $P$ followed by the continuation. This was done syntactically in the form of a program transformation. As we will see in the next sections, this will enable us to reason compositionally. Here, we give an alternative definition of the same function based on the operational semantics, against which we will then show correctness of the proper transformer, to be defined formally in the next section.

*Definition 3.3 (Semantic Expected Cost Transformer).* The function

$$\mathsf{Ect}[\cdot]\{\cdot\} : \Lambda_p \to (\mathcal{V} \to \mathbb{R}^{+\infty}) \to \mathbb{R}^{+\infty} ,$$

is defined as the least function satisfying

$$\mathsf{Ect}[V]\{f\} \triangleq f(V) \qquad \text{and} \qquad \mathsf{Ect}[P]\{f\} = r + \sum_{i \in I} p_i \cdot \mathsf{Ect}[Q_i]\{f\} \quad \text{if } P \xrightarrow{r} \{Q_i^{p_i}\}_{i \in I}.$$

[2] To be precise, ecost is defined as the least-fixed point of the functional $\xi : (\Lambda_p \to \mathbb{R}^{+\infty}) \to (\Lambda_p \to \mathbb{R}^{+\infty})$ defined by $\xi(G) \triangleq P \mapsto$ if $P \in \mathcal{V}$ then 0 else $r + \sum_{i \in I} p_i \cdot G(Q_i)$ where $P \xrightarrow{r} \{Q_i^{p_i}\}_{i \in I}$ in the else-branch. This fixed-point always exists (see [Winskel 1993]), and assigns a cost $c \in \mathbb{R}^{+\infty}$ even to non-terminating programs.

It can be shown that this cost-transformer is continuous and monotone, hence well-defined, see also [Avanzini et al. 2020] where this operator is defined in a more general setting.

*Example 3.4 (Example 3.2 continued).* Reconsider the reduction drawn in Example 3.2. Applying Definition 3.3 for an arbitrary $f : \mathcal{V} \to \mathbb{R}^{+\infty}$ we have

$$\begin{aligned}\mathsf{Ect}[\mathtt{walk}\cdot\mathtt{p}\cdot\mathtt{f}\cdot\mathtt{n}]\{f\} &= \cdots = \mathsf{Ect}\Big[\mathtt{if}\ \mathtt{ber}(1/2)\ \mathtt{then}\ \mathtt{n}\ \mathtt{else}\ (\mathtt{walk}\cdot\mathtt{p}\cdot\mathtt{f}\cdot(\mathtt{f}\cdot\mathtt{n}))^{\checkmark}\Big]\{f\}\\ &= \cdots = 1/2\cdot\mathsf{Ect}[\mathtt{n}]\{f\} + 1/2\cdot\mathsf{Ect}\Big[(\mathtt{walk}\cdot\mathtt{p}\cdot\mathtt{f}\cdot(\mathtt{f}\cdot\mathtt{n}))^{\checkmark}\Big]\{f\}\\ &= \cdots = 1/2\cdot f(\mathtt{n}) + 1/2\cdot(1+\mathsf{Ect}[\mathtt{walk}\cdot\mathtt{p}\cdot\mathtt{f}\cdot(\mathtt{n}+1)]\{f\})\ ,\end{aligned}$$

thus particularly,

$$\mathsf{Ect}[\mathtt{walk}\cdot\mathtt{p}\cdot\mathtt{f}\cdot\mathtt{0}]\{f\} = 1/2\cdot f(\mathtt{0}) + 1/2\cdot(1+1/2\cdot f(1)+1/2\cdot(1+1/2\cdot f(2)+1/2\cdot(\ldots)))\ .$$

Notice how, by letting $f$ be the constant zero function, this term converges to the expected cost of $\mathtt{walk}\cdot\mathtt{p}\cdot\mathtt{f}\cdot\mathtt{0}$. Dual, by unticking the recursive calls, thereby eliminating all subexpression 1+, we obtain the expected value of $f$ on the distribution of its normal forms.

The last statement can be generalized, which is almost immediate to see by contrasting Definition 3.3 with the definition of ecost and nf. To this end, let us call $P \in \Lambda_p$ *cost-free* when $\mathsf{ecost}(P) = 0$, e.g., when $P$ does not contain any occurrence of $(\cdot)^{\checkmark}$. Then $\mathsf{Ect}[P]\{\cdot\}$ can be brought in correspondence to the expected cost and value on normal form distributions in the following way.

Lemma 3.5. *Let $P \in \Lambda_p$. The following equalities hold:*

(1) $\mathsf{ecost}(P) = \mathsf{Ect}[P]\{V \mapsto 0\}$*; and*
(2) $\mathbb{E}_{\mathsf{nf}(P)}(f) = \mathsf{Ect}[P]\{f\}$ *when $P$ is cost-free.*

Finally, the following structural properties will be exploited in the inductively defined cost transformer which will be introduced in the next section. Notable, these properties tell us how the expected cost of a term relate to that of its subterms, guiding the inductive definition of the transformer given in the next section.

Lemma 3.6 (Structural Properties).

(1) $\mathsf{Ect}\Big[\mathsf{s}(\vec{P}, Q, \vec{W})\Big]\{f\} = \mathsf{Ect}[Q]\Big\{V \mapsto \mathsf{Ect}\Big[\mathsf{s}(\vec{P}, V, \vec{W})\Big]\{f\}\Big\}$, *for every* $\mathsf{s} \in \mathcal{C} \cup \mathcal{F} \cup \mathcal{D}$*;*
(2) $\mathsf{Ect}[P \cdot Q]\{f\} = \mathsf{Ect}[Q]\{W \mapsto \mathsf{Ect}[P]\{V \mapsto \mathsf{Ect}[V \cdot W]\{f\}\}\}$*; and*
(3) $\mathsf{Ect}[\mathsf{case}\ P\ \mathsf{of}\ \{\mathsf{c}(\vec{x}) \mapsto Q \mid y \mapsto R\}]\{f\} = \mathsf{Ect}[P]\Big\{V \mapsto \begin{cases}\mathsf{Ect}\Big[Q[\vec{W}/\vec{x}]\Big]\{f\} & \text{if } V = \mathsf{c}(\vec{W}),\\ \mathsf{Ect}[R[V/y]]\{f\} & \text{if } V \neq \mathsf{c}(\vec{W}).\end{cases}\Big\}$.

## 4 COMPUTING EXPECTED COSTS

This section is devoted to defining a syntactic variant of the expected cost transformer, which differs from the semantic version from Definition 3.3 in two ways. First, it is defined by induction on the structure of terms, thereby enabling compositional reasoning. Second, it is indeed an (efficiently computable) program transformation, resulting in a (pure) PCF-term rather than a cost function, thereby enabling classical program analysis.

The first step we need to make consists in extending the pure fragment of our language, so as to encompass (real-valued) cost functions. This is necessary because costs are measured by real numbers, and the latter have to be treated differently from ordinary data, as will be apparent from the underlying denotational semantics.

### 4.1 The Target Language and Its Semantics

The target language $\Lambda_{\mathbb{R}}$ is given by a fragment of the pure language introduced before, endowed with a dedicated base type $\mathtt{Real}^+$ to express real numbers $r \in \mathbb{R}^{+\infty}$. We allow constants $\mathsf{r} \in \mathcal{C}$ for all $r \in \mathbb{R}^{+\infty}$ as well as *continuous* (particularly, *non-decreasing*) functions on $\mathbb{R}^{+\infty}$ — such as multiplication or addition — as primitive functions in $\mathcal{D}$. Terms in the target language, as the image of a CPS translation, will have the restricted form

$$E, F ::= x \mid \lambda x.M \mid \mathtt{letrec}\ f\ x = M \mid \mathsf{s}(E_1, \dots, E_n)$$
$$M, N, O ::= E \mid M \cdot E \mid \mathtt{case}\ E\ \mathtt{of}\ \{\mathsf{c}(\vec{x}) \mapsto N \mid y \mapsto O\}\ ,$$

where $\mathsf{s} \in \mathcal{C} \cup \mathcal{F}$. The most notable changes, in comparison to the full probabilistic language, lies in the exclusion of effectful operations — $(\cdot)^{\checkmark}$ and sampling primitives $\mathsf{p} \in \mathcal{D}$ — and that $\beta$-redexes are confined to head position. We furthermore restrict recursion to computations on $\mathbb{R}^{+\infty}$, particularly, all recursive definitions are constrained to $\mathtt{letrec}\ f\ x = \lambda\vec{y}.M$ where $M$ is an expression of type $\mathtt{Real}^+$. In other words, we subject terms to the typing rules of Figure 3, where the rule governing `letrec` is replaced by the typing rule

$$\frac{\Gamma; f : \sigma_1 \to \cdots \to \sigma_k \to \mathtt{Real}^+; x_1 : \sigma_1, \dots; x_k : \sigma_k \vdash M : \mathtt{Real}^+}{\Gamma \vdash \mathtt{letrec}\ f\ x_1 \cdots x_k = M : \mathtt{Real}^+}\ .$$

Here, we use the notation $\mathtt{letrec}\ f\ x_1\ \cdots x_k = M$ for $\mathtt{letrec}\ f\ x_1 = \lambda x_2 \cdots x_k.M$. Below, we use $\Lambda^{\sigma}_{\mathbb{R}}(\Gamma)$ to denote the terms within this target language, typeable under context $\Gamma$ with type $\sigma$.

*Denotational Semantics.* The target language, being pure, would allow for a standard domain-theoretic denotational semantics. The non-standard behavior of the real-number type, however, makes it necessary to slightly diverge from the usual path. More specifically, the type $\mathtt{Real}^+$ is interpreted as $\mathbb{R}^{+\infty}$ endowed with the vertical ordering, and not by way of the discrete order, as one would possibly expect. An immediate implication is that recursive functions such as $f$ above, *converge to* rather than *compute* their result. Well-definedness is in essence given in by the Monotone Convergence Theorem. For instance, the operator $\sum^{\infty}_{(\cdot)} : \mathtt{Nat} \to (\mathtt{Nat} \to \mathtt{Real}^+) \to \mathtt{Real}^+$ defined as $\mathtt{letrec}\ \sum^{\infty}_{n} f = f \cdot n + \sum^{\infty}_{\mathsf{s}(n)} f$, will have the usual convergence properties of infinite sums (over monotone $f$), and would not return $\bot$ as in the usual domain theoretic model of PCF.

Before delving into the details of the model, let us recall some basic notions and results on domain theory, using terminology from Winskel [1993, Section 8]. A partial order $(D, \sqsubseteq)$ is called a *complete partial order* (*cpo*) (sometimes referred to as predomain) if any *$\omega$-chain* $d_0 \sqsubseteq d_1 \sqsubseteq \cdots$ has a least upper bound $\bigsqcup_{n\in\mathbb{N}} d_n$ in $D$. We do not require in general that $D$ is equipped with a least element, but if so, the least element is denoted by $\bot_D$ or $\bot$ when $D$ is clear from context. Relevant examples of cpo's are *discrete cpos* where $D$ is equipped with the identity relation, $\mathbb{R}^{+\infty}$ *ordered vertically* (i.e., $r_1 \sqsubseteq r_2$ iff $r_1 \leq_{\mathbb{R}} r_2$ or $r_2 = \infty$), and the function space $[D \longrightarrow E]$ of *continuous functions* between cpos $D$ and $E$, ordered point-wise, i.e. $f \sqsubseteq g$ iff for all $d \in D$, $f(d) \sqsubseteq g(d)$. Here a function $f : D \to E$ is continuous if it is *monotone* ($d \sqsubseteq e$ implies $f(d) \sqsubseteq f(e)$) and $\bigsqcup_{n\in\mathbb{N}} f(d_n) = f(\bigsqcup_{n\in\mathbb{N}} d_n)$ for all $\omega$-chains $d_0 \sqsubseteq d_1 \sqsubseteq \cdots$. All functions $f : D \to E$ on discrete domains $E$ are continuous, composition $(\circ) : [E \longrightarrow F] \to [D \longrightarrow E] \to [D \longrightarrow F]$ and application $\mathsf{apply} : [D \longrightarrow E] \times D \to E$ are continuous, and so is $\mathsf{lfp} : [D \longrightarrow D] \to D$ defined by $\mathsf{lfp}(f) \triangleq \bigsqcup_{n\in\mathbb{N}} f^n(\bot_D)$ when $D$ has bottom element $\bot_D$. Here, $f^n$ denotes the $n$-fold composition of $f$. All this is nothing more than the basic machinery one needs to give a domain-theoretic denotational semantics to languages like PCF.

Finally, we can give the actual interpretation of the target language's *types*, each of them being put in correspondence with a cpo. More precisely, $\mathtt{Real}^+$ is interpreted as $\mathbb{R}^{+\infty}$ ordered vertically, while $\mathsf{B} \in \mathcal{B}$ where $\mathsf{B} \neq \mathtt{Real}^+$ will be put in correspondence with the discrete cpo on terms values

$$
\begin{aligned}
[\![x]\!]_\rho &\triangleq \rho(x)\\
[\![\lambda x.M]\!]_\rho &\triangleq v \mapsto [\![M]\!]_{\rho;x\mapsto v}\\
[\![\texttt{letrec}\ f\,\vec{x} = M]\!]_\rho &\triangleq \mathsf{lfp}(F \mapsto (\vec{v} \mapsto [\![M]\!]_{\rho;f\mapsto F;\vec{x}\mapsto\vec{v}}))\\
[\![\mathsf{s}(E_1,\dots,E_n)]\!]_\rho &\triangleq \mathsf{apply}([\![\mathsf{s}]\!], ([\![E_1]\!]_\rho,\dots,[\![E_n]\!]_\rho))\\
[\![M\cdot E]\!]_\rho &\triangleq \mathsf{apply}([\![M]\!]_\rho, [\![E]\!]_\rho)\\
[\![\mathsf{case}\ E\ \mathsf{of}\ \{\mathsf{c}(\vec{x})\mapsto M \mid y\mapsto N\}]\!]_\rho &\triangleq \mathsf{match}_\mathsf{c}([\![E]\!]_\rho, \vec{v}\mapsto [\![M]\!]_{\rho;\vec{x}\mapsto\vec{v}}, v\mapsto [\![N]\!]_{\rho;y\mapsto v})
\end{aligned}
$$

Fig. 6. Denotational Semantics of the Target Language.

$\mathcal{V}^\mathsf{B}$. Functional types are interpreted by continuous functions ordered point-wise. In other words:

$$[\![\mathtt{Real}^+]\!] \triangleq \mathbb{R}^{+\infty} \qquad [\![\mathsf{B}]\!] \triangleq \mathcal{V}^\mathsf{B} \qquad [\![\sigma\to\tau]\!] \triangleq [[\![\sigma]\!] \longrightarrow [\![\tau]\!]]\ .$$

Notice that $[\![\sigma]\!]$ is always a cpo, but is not guaranteed to be endowed with a least element. In particular, any base type other than $\mathtt{Real}^+$ and any function returning one of such type is interpreted as a cpo which is *not*, however, pointed. In other words, since we just confined recursion to functions producing a $\mathtt{Real}^+$ in the target language, we are permitted to interpret the remaining functions as total ones. This has significant practical implications. Later on, when we study the target language, we will not have to reason about non-termination of such functions.

Attributing a meaning to terms requires first giving a semantics $[\![\mathsf{s}]\!] : [\![\mathsf{B}_1]\!]\times\cdots\times[\![\mathsf{B}_n]\!]\to[\![\mathsf{B}]\!]$ to any symbols $\mathsf{s}:\mathsf{B}_1\times\cdots\times\mathsf{B}_n\to\mathsf{B}\in\mathcal{C}\cup\mathcal{F}$. For constructor $\mathsf{c}:\mathsf{B}_1\times\cdots\times\mathsf{B}_n\to\mathsf{B}\in\mathcal{C}$, $[\![\mathsf{c}]\!]$ is simply the constructor itself, i.e. $[\![\mathsf{c}]\!]\,(V_1,\dots,V_n)\triangleq\mathsf{c}(V_1,\dots,V_n)$. To each such constructor $\mathsf{c}$, we furthermore associate the continuous operator

$$\mathsf{match}_\mathsf{c} : \mathcal{V}^\mathsf{B}\times[\mathcal{V}^{\mathsf{B}_1}\times\cdots\times\mathcal{V}^{\mathsf{B}_n}\longrightarrow D]\times[\mathcal{V}^{\mathsf{B}_n}\longrightarrow D]\to D$$

$$\mathsf{match}_\mathsf{c}(t,f,g) = \begin{cases}\mathsf{apply}(f,(V_1,\dots,V_n)) & \text{if } t=\mathsf{c}(V_1,\dots,V_n),\\ \mathsf{apply}(f,t) & \text{otherwise.}\end{cases}$$

Matching the operational semantics of the source language, primitives $\mathsf{f}:\mathsf{B}_1\times\cdots\times\mathsf{B}_n\to\mathsf{B}\in\mathcal{F}$ are interpreted by $[\![\mathsf{f}]\!]\,(V_1,\dots,V_n)\triangleq\mathsf{f}_{\mathcal{I}}(V_1,\dots,V_n)$. We require all such functions to be continuous in all argument coordinates. Thereby, in particular, all primitives have to be non-decreasing in their real-valued argument positions. Note that this requirement is vacuously satisfied by the (non-real-valued) primitives of the source language. For instance, addition and multiplication $(+),(*):\mathtt{Real}^+\times\mathtt{Real}^+\to\mathtt{Real}^+$ as well as, e.g., Integer fractions $(/):\mathtt{Int}\times\mathtt{Int}\to\mathtt{Real}^+$ are continuous, whereas subtraction $(-):\mathtt{Real}^+\times\mathtt{Real}^+\to\mathtt{Real}^+$ and real fractions $(/):\mathtt{Real}^+\times\mathtt{Real}^+\to\mathtt{Real}^+$ are not.

A term $\Gamma\vdash M:\sigma$ can now be interpreted as a function $[[\![\Gamma]\!]\longrightarrow[\![\sigma]\!]]$, with $[\![\cdot]\!]$ naturally extended to any typing context $\Gamma$. Semantics is formalised in Figure 6, where we write $[\![M]\!]_\rho$ for $[\![M]\!]\,(\rho)$.

Remark. *Well-definedness of $[\![M]\!]_\rho$ follows by construction, as all operators are continuous. By the typing restrictions on* $\texttt{letrec}\ f\,\vec{x}=M$, *the functional $F\vec{v}\mapsto[\![M]\!]_{\rho;f\mapsto F;\vec{x}\mapsto\vec{v}}$ underlying its semantics lives in the space $[D\longrightarrow D]$ where $D=[\![\sigma_1]\!]\to\cdots\to[\![\sigma_n]\!]\to\mathbb{R}^{+\infty}$, whose bottom element is the constant zero function and top element the constant $\infty$ function. Fixed-points of a functional, while always well-defined, are in general not reached within a finite number of unfoldings, as exemplified by the operator $\sum^\infty_{(\cdot)}:\mathtt{Nat}\to(\mathtt{Nat}\to\mathtt{Real}^+)\to\mathtt{Real}^+$ above.*

For $P$ and $Q$ of type $\sigma$, we write $P\equiv Q$ when $P$ and $Q$ are semantically equal, that is, when $[\![P]\!]=[\![Q]\!]$. By slight abuse of notation, let us also write $P\equiv d$ or $d\equiv P$ if $[\![P]\!]=d$. Similar, we write

**Mapping of Types**

$$\mathsf{B}^\dagger \triangleq \mathsf{B} \qquad (\sigma \to \tau)^\dagger \triangleq \sigma^\dagger \to (\tau^\dagger \to \mathtt{Real}^+) \to \mathtt{Real}^+$$

**Mapping of Values**

$$x^\dagger \triangleq x \qquad (\lambda x.P)^\dagger \triangleq \lambda x k.\mathrm{ect}[P]\{k\}$$

$$\mathsf{c}(V_1,\dots,V_k)^\dagger \triangleq \mathsf{c}(V_1^\dagger,\dots,V_k^\dagger) \qquad (\mathtt{letrec}\ f\ x = P)^\dagger \triangleq \mathtt{letrec}\ f\ x\ k = \mathrm{ect}[P]\{k\}$$

**Expected Cost Transformer**

$$\mathrm{ect}[V]\{\kappa\} \triangleq \kappa \cdot V^\dagger$$

$$\mathrm{ect}[P \cdot Q]\{\kappa\} \triangleq \mathrm{ect}[Q]\{\lambda z.\mathrm{ect}[P]\{\lambda y.y \cdot z \cdot \kappa\}\}$$

$$\mathrm{ect}\left[\begin{array}{ll}\mathsf{case}\ P\ \mathsf{of}\ \{ & \\ \quad \mathsf{c}(\vec{x}) \mapsto Q & \\ \quad \mid \quad y \mapsto R\ \} & \end{array}\right]\{\kappa\} \triangleq \mathrm{ect}[P]\left\{\begin{array}{ll}\lambda v.\mathsf{case}\ v\ \mathsf{of}\ \{ & \\ \quad \mathsf{c}(\vec{x}) \mapsto \mathrm{ect}[Q]\{\kappa\} & \\ \quad \mid \quad y \mapsto \mathrm{ect}[R]\{\kappa\}\ \} & \end{array}\right\}$$

$$\mathrm{ect}\left[P^{\checkmark}\right]\{\kappa\} \triangleq 1 + \mathrm{ect}[P]\{\kappa\}$$

$$\mathrm{ect}[\mathsf{s}(P_1,\dots,P_k)]\{\kappa\} \triangleq \begin{cases}\overline{\mathrm{ect}}[P_k,\dots,P_1]\{\lambda z_k\dots z_1.\kappa \cdot \mathsf{s}(z_1,\dots,z_k)\} & \text{if } \mathsf{s} \in \mathcal{C} \cup \mathcal{F},\\ \overline{\mathrm{ect}}[P_k,\dots,P_1]\left\{\lambda z_k\dots z_1.\mathsf{E}_{\mathsf{s}(z_1,\dots,z_k)}(\kappa)\right\} & \text{if } \mathsf{s} \in \mathcal{D}.\end{cases}$$

$$\overline{\mathrm{ect}}[\,]\{\kappa\} \triangleq \kappa$$

$$\overline{\mathrm{ect}}[P_1,P_2,\dots,P_k]\{\lambda z_1 z_2\dots z_k.\kappa\} \triangleq \mathrm{ect}[P_1]\left\{\lambda z_1.\overline{\mathrm{ect}}[P_2,\dots,P_k]\{\lambda z_2\dots z_k.\kappa\}\right\}$$

Fig. 7. Expected Cost Transformer.

$P \leqq Q$ if $[\![P]\!] \le [\![Q]\!]$, and extend this notation to $d \in [\![\sigma]\!]$ as with equality. It can be shown that the denotational agree with the operational semantics in the following way.

Lemma 4.1. $P \longrightarrow Q \implies P \equiv Q$.

Despite the language being pure, the operational semantics does not *precisely* match the denotational one, even at base types.

*Example 4.2.* Consider the term $\mathtt{f} = (\mathtt{letrec}\ f\ x = {}^1\!/_2 * x + {}^1\!/_2 * (f \cdot x))$. As it can be easily verified, $[\![\mathtt{f} \cdot \mathtt{r}]\!] = r$ for every real number $r$. On the other hand, reducing the term $\mathtt{f} \cdot \mathtt{r}$ through $\longrightarrow$ never reaches a normal form, but only approximates it:

$$\mathtt{f}\ \mathtt{r} \to^* {}^{\mathtt{r}}\!/_2 + {}^1\!/_2 * (\mathtt{f}\ \mathtt{r}) \to^* {}^{\mathtt{r}}\!/_2 + {}^1\!/_2 * ({}^{\mathtt{r}}\!/_2 + {}^1\!/_2 * (\mathtt{f}\ \mathtt{r})) \to^* \cdots .$$

### 4.2 The Cost Transformer

Our *expected cost transformers*, defined next, can be seen as an adaptation of the first-order, one-pass CPS transformation by Danvy and Nielsen [2003], with answer type restricted to $\mathtt{Real}^+$. As such, the transformer makes the evaluation order explicit. However, in the translation of a term $P : \sigma$, rather than receiving a continuation as argument, the cost transformer receives the expected cost of the continuation, as a term $\kappa : \sigma \to \mathtt{Real}^+$. This approach requires that for any primitive sampling operation, the expectation wrt. $\kappa : \mathsf{B} \to \mathtt{Real}^+$ is expressible in the target language. To this end, we assume for every $\mathsf{p} : \mathsf{B}_1 \times \cdots \times \mathsf{B}_k \to \mathsf{B} \in \mathcal{D}$ that a term

$$\vdash \mathsf{E}_{\mathsf{p}(\cdot,\dots,\cdot)}(\cdot) : \mathsf{B}_1 \to \cdots \to \mathsf{B}_k \to (\mathsf{B} \to \mathtt{Real}^+) \to \mathtt{Real}^+ ,$$

exists, satisfying $\left[\!\left[\mathsf{E}_{\mathsf{p}(V_1,\dots,V_m)}(\kappa)\right]\!\right] = \mathbb{E}_{\mathsf{p}_I(V_1,\dots,V_m)}\left([\![\kappa]\!]\right)$ for all $V_i \in \mathcal{V}^{\mathsf{B}_i}$ $(1 \le i \le m)$.

*Definition 4.3 (Expected Cost Transformer).* The *expected cost transformer*

$$\mathsf{ect}[\cdot]\{\cdot\} : \forall\sigma\,\Gamma.\Lambda_p^{\sigma}(\Gamma) \to \Lambda_{\mathbb{R}}^{\sigma^\dagger\to\mathtt{Real}^+}(\Gamma^\dagger) \to \Lambda_{\mathbb{R}}^{\mathtt{Real}^+}(\Gamma^\dagger)$$

is given in Figure 7. As usual for CPS transformations, the transformer is defined by mutual recursion with a function $(\cdot)^\dagger : \forall\sigma\,\Gamma.\mathcal{V}^{\sigma}(\Gamma) \to \mathcal{V}_{\mathbb{R}}^{\sigma^\dagger}(\Gamma)$ on values. The corresponding translation on types, in notation $(\cdot)^\dagger$, is also given in Figure 7. It is extended component-wise to typing contexts.

The following auxiliary lemma confirms well-definedness of the transformer.

Lemma 4.4. *If* $\Gamma \vdash P : \sigma$ *and* $\Gamma^\dagger \vdash \kappa : \sigma^\dagger \to \mathtt{Real}^+$ *then* $\Gamma^\dagger \vdash \mathsf{ect}[P]\{\kappa\} : \mathtt{Real}^+$.

As usual for CPS transformations, the expected cost transformer may introduce *administrative redexes*, which do not find counterparts in the source program. For instance, $\mathsf{ect}[V \cdot W]\{\kappa\} = (\lambda z.((\lambda y.y \cdot z \cdot \kappa) \cdot V^\dagger) \cdot W^\dagger$ introduces two such redexes, unlike the specialized and semantically equivalent translation $V^\dagger \cdot W^\dagger \cdot \kappa$. Administrative redexes can often be optimized away thanks to Lemma 4.1. Similar, but maybe less obvious, $n$-ary functions such as $\lambda x\ y.P : \sigma_1 \to \sigma_2 \to \tau$ can be translated to $\lambda x\ y\ k.\mathsf{ect}[P]\{k\} : \sigma_1^\dagger \to \sigma_2^\dagger \to (\tau^\dagger \to \mathtt{Real}^+) \to \mathtt{Real}^+$ without changing semantics, provided that they are not partially applied. Note that partial applications can always be resolved by $\eta$-expansion. For the sake of brevity, we will apply such simplifications frequently in examples.

Before we continue with a proof of correctness for this program transformation, let us illustrate it on the running examples. To this end, let us first look on how expectations on some sampling primitives can be computed within the target language.

*Example 4.5 (Computing Expectations in the Target Language).*

- *Bernoulli Distributions*: For Bernoulli distributions $\mathtt{ber}(\mathtt{p})$ that we have already used before, we define $\mathsf{E}_{\mathtt{ber}(\mathtt{p})}(\kappa)$ as the term $\mathtt{p} * (\kappa \cdot \mathtt{true}) + (1 - \mathtt{p}) \cdot (\kappa \cdot \mathtt{false})$, assuming $0 \le p \le 1$. For the cases when ber is supplied with an unexpected argument value, i.e., $p < 0$ or $1 < p$, we set $\mathsf{E}_{\mathtt{ber}(\mathtt{p})}(\kappa) \triangleq 0$, in agreement with the operational semantics where a term $\mathtt{ber}(\mathtt{p})$ abnormally terminates whenever $p$ is not in the interval $[0, 1]$.
- *Uniform Distributions*: For distributions $\mathtt{unif}(\mathtt{n}, \mathtt{m})$, sampling an integer in the interval $[\mathtt{n}, \mathtt{m}]$, we can define $\mathsf{E}_{\mathtt{unif}(\mathtt{n},\mathtt{m})}(\kappa)$ as $\mathtt{E} \cdot \mathtt{n} \cdot \mathtt{m} \cdot \kappa$ where

  $$\mathtt{E} = (\mathtt{letrec}\ e\ l\ u\ k = \mathtt{if}\ l > u\ \mathtt{then}\ \mathtt{0}\ \mathtt{else}\ (1/u - l + 1) * \kappa \cdot l + e \cdot (l + 1) \cdot u \cdot k)\ ,$$

  assuming $\mathtt{unif}_{\mathcal{I}}(\mathtt{n}, \mathtt{m}) = \varnothing$ for $n > m$.
- *Enumerable Distributions*: Generalizing the previous example, consider a sampling primitive $\mathtt{p} : \mathtt{B}_1 \to \cdots \to \mathtt{B}_k \to \mathtt{B}$, with $\mathtt{p}_{\mathcal{I}}(V_1, \ldots, V_k) = \{W_n^{p_n}\}_{n\in\mathbb{N}}$, such that we can define an enumerator $\mathtt{enum}_{\mathtt{p}} : \mathtt{B}_1 \to \cdots \to \mathtt{B}_k \to \mathtt{Nat} \to \mathtt{Real}^+ \times \mathtt{B}$ (e.g., as a primitive function) with $\mathtt{enum}_{\mathtt{p}}(V_1, \ldots, V_k, n)$ returning $(\mathtt{p}_n, W_n)$ for every $n \in \mathbb{N}$. We can then define $\mathsf{E}_{\mathtt{p}(\vec{x})}(\kappa) = \mathtt{E}\cdot\vec{x}\cdot\mathtt{0}\cdot\kappa$ where

  $$\mathtt{E} = (\mathtt{letrec}\ e\ \vec{x}\ n\ k = \mathtt{let}\ (p_n, w_n) = \mathtt{enum}_{\mathtt{p}}(\vec{x}, n)\ \mathtt{in}\ p_n * (k \cdot w_n) + e \cdot \vec{x} \cdot (n + 1) \cdot k)\ .$$

*Example 4.6 (Coupon Collector Problem, formally).* Let us define expectations of a continuation $\kappa$ for sampling from $\mathtt{draw}(cs)$ in correspondence to that for uniform distributions given in Example 4.5. Applying the value transformation on the coupon collector function from Fig. 1a and removing administrative redexes results in the function depicted in Fig. 1b, i.e., $\mathtt{coupons}^\dagger \equiv \mathtt{couponsCPS}$. Consequently, for any list cs of coupons, we have

$$\mathsf{ect}[\mathtt{coupons} \cdot \mathtt{cs}]\{\kappa\} = \mathsf{ect}[\mathtt{cs}]\{\lambda z.\mathsf{ect}[\mathtt{coupons}]\{\lambda y.y \cdot z \cdot \kappa\}\} \equiv \mathtt{couponsCPS} \cdot \mathtt{cs} \cdot \kappa\ .$$

*Example 4.7 (Example 3.1 continued).* Slightly simplifying, the three functions from Example 3.1 get translated into:

$$\begin{aligned}
&\mathtt{walkCPS} : (\mathtt{Int} \to (\mathtt{Bool} \to \mathtt{Real}^+) \to \mathtt{Real}^+) \\
&\qquad \to (\mathtt{Int} \to (\mathtt{Int} \to \mathtt{Real}^+) \to \mathtt{Real}^+) \\
&\qquad \to \mathtt{Int} \\
&\qquad \to (\mathtt{Int} \to \mathtt{Real}^+) \to \mathtt{Real}^+ \\
&\mathtt{walkCPS} = (\mathtt{letrec}\ w\ p\ f\ n\ k = p \cdot n \cdot (\lambda b.\mathtt{if}\ b\ \mathtt{then}\ k \cdot n\ \mathtt{else}\ 1 + f \cdot n \cdot (\lambda m.w \cdot p \cdot f \cdot m \cdot k)))
\end{aligned}$$

$$\begin{aligned}
&\mathtt{geoCPS} : \mathtt{Int} \to (\mathtt{Int} \to \mathtt{Real}^+) \to \mathtt{Real}^+ \\
&\mathtt{geoCPS} = \mathtt{walkCPS} \cdot (\lambda_.{}^1\!/_2 * (k \cdot \mathtt{true}) + {}^1\!/_2 * (k \cdot \mathtt{false})) \cdot (\lambda nk.k \cdot (n+1))
\end{aligned}$$

$$\begin{aligned}
&\mathtt{randomWalkCPS_p} : \mathtt{Int} \to (\mathtt{Int} \to \mathtt{Real}^+) \to \mathtt{Real}^+ \\
&\mathtt{randomWalkCPS_p} = \mathtt{walkCPS} \cdot (\lambda k.k \cdot (n \leq \mathtt{0})) \cdot (\lambda nk.\mathtt{p} * (k \cdot (n-1)) + (1-\mathtt{p}) * (k \cdot (n+1)))\ .
\end{aligned}$$

In the remainder of this section, we prove *correctness* of the expected cost transformer. This amounts to proving that the semantic transformer introduced here matches the semantic one introduced in Section 3.3. Specifically, on terms $\vdash P : \mathtt{B}$ of base types, this correspondence amounts to the equality:

$$\mathrm{ect}[P]\{\kappa\} \equiv \mathrm{Ect}[P]\{[\![\kappa]\!]\}\ , \tag{1}$$

for every $\vdash \kappa : \mathtt{B} \to \mathtt{Real}^+$. By Lemma 3.5, this correctness result allows us to instantiate the transformer to one for reasoning about expected costs and values. To prove the equation, we show that $\mathrm{ect}[P]\{\kappa\}$ is bounded by $\mathrm{Ect}[P]\{[\![\kappa]\!]\}$ from *above* and *below*. This is the subject of the next two sections.

*4.2.1 Upper Bound.* Proving that the syntactic CPS transform is an upper bound to the semantic one cannot be proved directly, unless the same statement is extended to terms $P$ of arbitrary type. While the left-hand-side of Equation (1) stays in the form $\mathrm{ect}[P]\{\kappa\}$, the right-hand-side becomes $\mathrm{Ect}[P]\{V \mapsto [\![\kappa \cdot V^\dagger]\!]\}$. This is because $\kappa$ expect a result in continuation passing form $V^\dagger$, whereas in Equation (1) we implicitly employed $V^\dagger = V$ for any value $V$ of base-type. The proof essentially relies on the following auxiliary lemma, which basically states that the syntactic CPS transformation is consistent with the denotational semantics, taking costs into account.

Lemma 4.8.

$$P \xrightarrow{k}_l \{Q_i^{p_i}\}_{i\in I} \implies \mathrm{ect}[P]\{\kappa\} \equiv k + \sum_{i\in I} p_i \cdot \mathrm{ect}[Q_i]\{\kappa\}\ .$$

With this lemma in mind, it is then not difficult to see that $\mathrm{ect}[\cdot]\{\kappa\}$ defines a fixed-point of the equation underlying $\mathrm{Ect}[P]\{V \mapsto [\![\kappa \cdot V^\dagger]\!]\}$. Until now, however, we have not shown that it is the least such fixed-point. Conclusively, we only get the following upper bound result.

Lemma 4.9.

$$\mathrm{Ect}[P]\{V \mapsto [\![\kappa \cdot V^\dagger]\!]\} \leqq \mathrm{ect}[P]\{\kappa\}\ .$$

Specializing $\kappa$ to the constant zero function, we thus get $\mathrm{ecost}(P) \leqq \mathrm{ect}[P]\{\lambda\_.\mathtt{0}\}$ via Lemma 3.5.

*4.2.2 Lower Bound.* To prove the lower-bound result, we introduce *finite approximations* of the expected cost transformer. The construction is similar, but carries an additional counter of type `Nat`, with elements of this type built from constructors $\mathtt{0} : \mathtt{Nat}$ and $\mathtt{s} : \mathtt{Nat} \to \mathtt{Nat}$. The counter is decremented at each function application, once it reaches zero the computation is aborted.

**Mapping of Types**

$$\mathsf{B}^{\ddagger} \triangleq \mathsf{B}$$
$$(\sigma \to \tau)^{\ddagger} \triangleq \mathtt{Nat} \to \sigma^{\ddagger} \to (\mathtt{Nat} \to \tau^{\ddagger} \to \mathtt{Real}^{+}) \to \mathtt{Real}^{+}$$

**Mapping of Values**

$$x^{\ddagger} \triangleq x$$
$$\mathsf{c}(V_1, \ldots, V_k)^{\ddagger} \triangleq \mathsf{c}(V_1^{\ddagger}, \ldots, V_k^{\ddagger})$$
$$(\lambda x.P)^{\ddagger} \triangleq \lambda n x k.[n > 0]\Big(\mathrm{ect}[P]^{n-1}\{k\}\Big)$$
$$(\mathtt{letrec}\ f\ x = P)^{\ddagger} \triangleq \lambda n.[n > 0]\big(\mathtt{letrec}\ f\ x\ k = \mathrm{ect}[P]^{n-1}\{k\}\big)$$

**Step-Indexed Expected Cost Transformer**

$$\mathrm{ect}[V]^n\{\kappa\} \triangleq \kappa \cdot n \cdot V^{\ddagger}$$
$$\mathrm{ect}[P \cdot Q]^n\{\kappa\} \triangleq \mathrm{ect}[Q]^n\{\lambda m z.\mathrm{ect}[P]^m\{\lambda o y.y \cdot o \cdot z \cdot \kappa\}\}$$
$$\mathrm{ect}\begin{bmatrix}\mathtt{case}\ P\ \mathtt{of}\ \{ & \\ \quad \mathsf{c}(\vec{x}) \mapsto Q & \\ \mid \quad y \mapsto R\ \}\end{bmatrix}^n\{\kappa\} \triangleq \mathrm{ect}[P]^n\left\{\lambda m z.[m > 0]\left(\begin{array}{l}\mathtt{case}\ z\ \mathtt{of}\ \{ \\ \quad \mathsf{c}(\vec{x}) \mapsto \mathrm{ect}[Q]^{m-1}\{\kappa\} \\ \mid \quad y \mapsto \mathrm{ect}[R]^{m-1}\{\kappa\}\ \}\end{array}\right)\right\}$$
$$\mathrm{ect}\Big[P^{\checkmark}\Big]^n\{\kappa\} \triangleq 1 + \mathrm{ect}[P]^n\{\kappa\}$$
$$\mathrm{ect}[\mathsf{s}(P_1, \ldots, P_k)]^n\{\kappa\} \triangleq \begin{cases}\overline{\mathrm{ect}}[P_k, \ldots, P_1]^n\{\lambda m z_k \ldots z_1.\kappa \cdot m \cdot \mathsf{s}(z_1, \ldots, z_k)\} \\ \qquad \text{if } \mathsf{s} \in \mathcal{C} \cup \mathcal{F}, \\ \overline{\mathrm{ect}}[P_k, \ldots, P_1]^n\big\{\lambda m z_k \ldots z_1.\mathsf{E}_{\mathsf{s}(z_1,\ldots,z_k)}(\kappa \cdot m)\big\} \\ \qquad \text{if } \mathsf{s} \in \mathcal{D}.\end{cases}$$
$$\overline{\mathrm{ect}}[\,]^n\{\lambda m.\kappa\} \triangleq (\lambda m.\kappa) \cdot n$$
$$\overline{\mathrm{ect}}[P_1, P_2, \ldots, P_k]^n\{\lambda m z_1 z_2 \ldots z_k.\kappa\} \triangleq \mathrm{ect}[P_1]^n\big\{\lambda m z_1.\overline{\mathrm{ect}}[P_2, \ldots, P_k]^m\{\lambda m z_2 \ldots z_k.\kappa\}\big\}$$

Fig. 8. Finitely Approximated Expected Cost Transformer.

Otherwise, the transformation behaves similar to the expected cost transformer. To this end, let us use $[V > 0](M)$ as an abbreviation for the term $\mathtt{case}\ V\ \mathtt{of}\ \{\mathsf{s}(\_) \mapsto M \mid \_ \mapsto \mathtt{0}\}$.

*Definition 4.10 (Step-Indexed Expected Cost Transformer).* The *step-indexed expected cost transformer*

$$\mathrm{ect}[\cdot]^{(\cdot)}\{\cdot\} : \forall \sigma\, \Gamma.\Lambda_p^{\sigma}(\Gamma) \to \Lambda_{\mathbb{R}}(\mathtt{Nat}) \to \Lambda_{\mathbb{R}}^{\mathtt{Nat} \to \sigma^{\ddagger} \to \mathtt{Real}^{+}}(\Gamma^{\ddagger}) \to \Lambda_{\mathbb{R}}^{\mathtt{Real}^{+}}(\Gamma^{\ddagger})$$

is given in Figure 8. As before, it is defined mutual recursively with a one place function $(\cdot)^{\ddagger} : \forall \sigma\, \Gamma.\mathcal{V}^{\sigma}(\Gamma) \to \mathcal{V}_{\mathbb{R}}^{\sigma^{\ddagger}}(\Gamma^{\ddagger})$ on values, and $(\cdot)^{\ddagger}$ is also used to transform types.

In what follows we prove two properties. First, we show that the step indexed version indeed approximates the expected cost transformer, in the sense that by increasing the step-counter the result of the transformer can be brought arbitrary close to the original definition. In the limit, the two transformers thus coincide. The correspondence is made precise in the following simulation relation.

*Definition 4.11 (Simulation Relation).* We mutually define two type-indexed, binary relations

$$\cong_\sigma \subseteq \Lambda_{\mathbb{R}}^{\sigma^\dagger} \times \Lambda_{\mathbb{R}}^{\sigma^\ddagger} \quad \text{and} \quad \simeq_\sigma \subseteq \Lambda_{\mathbb{R}}^{\sigma^\dagger \to \mathtt{Real}^+} \times \Lambda_{\mathbb{R}}^{\mathtt{Nat} \to \sigma^\ddagger \to \mathtt{Real}^+}$$

as follows:

- $M \cong_{\mathsf{B}} \hat{M}$ if there exists, $\mathsf{s} : \mathsf{B}_1 \times \cdots \times \mathsf{B}_k \to \mathsf{B} \in C \cup \mathcal{F}$, such that (i) $M \equiv \mathsf{s}(M_1, \ldots, M_k)$, (ii) $\hat{M} \equiv \mathsf{s}(\hat{M}_1, \ldots, \hat{M}_k)$, and (iii) $M_i \cong_{\mathsf{B}_i} \hat{M}_i$ for all $1 \le i \le k$.
- $M \cong_{\sigma \to \tau} \hat{M}$ if $M \cdot V \cdot \kappa \equiv \sup_{n \in \mathbb{N}} (\hat{M} \cdot \mathsf{n} \cdot \hat{V} \cdot \hat{\kappa})$ for all $V \cong_\sigma \hat{V}$ and $\kappa \simeq_\tau \hat{\kappa}$.
- $\kappa \simeq_\sigma \hat{\kappa}$ if $\kappa \cdot V \equiv \sup_{n \in \mathbb{N}} (\hat{\kappa} \cdot \mathsf{n} \cdot \hat{V})$ for all $V \cong_\sigma \hat{V}$.

In turn, this relation allows us to prove the following approximation lemma by structural induction, much in the style of logical relations.

Lemma 4.12. *For all $P \in \Lambda_p^\sigma$:*

(1) *if $P \in \mathcal{V}^\sigma$, then $P^\dagger \cong_\sigma P^\ddagger$; and*
(2) $\mathsf{ect}[P]\{\kappa\} \equiv \sup_n \mathsf{ect}[P]^{\mathsf{n}}\{\hat{\kappa}\}$ *for all $\kappa \simeq_\sigma \hat{\kappa}$.*

Second, with the following lemma we prove that independently of the stepping counter, the approximated cost transformer is bounded by the semantic counterpart:

Lemma 4.13. *For all $\hat{\kappa}$ such that $[\![\hat{\kappa}]\!]$ is non-decreasing in its first argument:*

$$\mathsf{ect}[P]^{\mathsf{n}}\{\hat{\kappa}\} \leqq \mathsf{Ect}[P]\{V \mapsto [\![\hat{\kappa} \cdot \mathsf{n} \cdot V^\ddagger]\!]\} \ .$$

The lower bound result is now almost an immediate consequence of Lemma 4.12(2) and Lemma 4.13. In the two considered cases, it is not difficult to construct $\hat{\kappa}$ from $\kappa$ with $\kappa \simeq_\sigma \hat{\kappa}$.

Lemma 4.14. *If $P \in \Lambda_p^\sigma$ and let $\kappa : \sigma \to \mathtt{Real}^+$ where either $\sigma = \mathsf{B}$ or $\kappa$ is constant. Then*

$$\mathsf{ect}[P]\{\kappa\} \leqq \mathsf{Ect}[P]\{[\![\kappa]\!]\} \ .$$

Combining this lower-bound result with the upper-bound proven in Lemma 4.9 yields now the final result of this section.

Theorem 4.15. *For any $P \in \Lambda_p^\sigma$, $\mathsf{ecost}(P) \equiv \mathsf{ect}[P]\{\lambda v.\mathtt{0}\}$.*

## 5 ON HIGHER-ORDER LOGIC AND EXPECTATIONS

Higher-Order Logic can be seen as a program logic for higher-order programs [Aguirre et al. 2017]. In this section, we will show that our target language can be reasoned about by way of tools of the same kind, namely by the *expectation higher-order logic* (*EHOL*), a unary higher-order logic specifically tailored for our target language. In particular, statements deriving upper bounds on the expected cost of the program at hand can be derived in EHOL and, thanks to Theorem 4.15, can be lifted back to the operational semantics. Any upper bound on $\mathsf{ect}[P]\{\lambda v.\mathtt{0}\}$ is also an upper bound of the expected cost of $P$.

Very informally, unary higher-order logic is a form of refinement type system where the assertions are kept separate from simple types. *Assertions* are derived by way of the following grammar:

$$\phi ::= P(M_1, \ldots, M_n) \mid \phi \Rightarrow \phi \mid \phi \wedge \phi \mid \phi \vee \phi \mid \forall x : \sigma.\phi \mid \exists x : \sigma.\phi$$

where $P$ ranges over a set of base predicates, including a binary equality symbol, each having a denotational semantics $[\![P]\!]$. Each assertion $\phi$ can have free *term* variables, and we write $\Gamma \vdash \phi$ when $\phi$ is well-typed by the typing context $\Gamma$. If this holds, we can define $[\![\phi]\!]_\rho$ as the naturally defined truth value, see Fig. 9. Judgments of the logic are of two forms:

$$\begin{aligned}
[\![P(M_1,\ldots,M_n)]\!]_\rho &\Leftrightarrow ([\![M_1]\!]_\rho,\ldots,[\![M_n]\!]_\rho) \in [\![P]\!] & [\![\neg\phi]\!]_\rho &\Leftrightarrow \text{ not } [\![\phi]\!]_\rho \\
[\![\exists x:\sigma.\phi]\!]_\rho &\Leftrightarrow \text{ for some } v \in [\![\sigma]\!],\ [\![\phi]\!]_{\rho;x\mapsto v} & [\![\phi\vee\psi]\!]_\rho &\Leftrightarrow [\![\phi]\!]_\rho \text{ or } [\![\psi]\!]_\rho \\
[\![\forall x:\sigma.\phi]\!]_\rho &\Leftrightarrow \text{ for every } v \in [\![\sigma]\!],\ [\![\phi]\!]_{\rho;x\mapsto v} & [\![\phi\wedge\psi]\!]_\rho &\Leftrightarrow [\![\phi]\!]_\rho \text{ and } [\![\psi]\!]_\rho
\end{aligned}$$

Fig. 9. Semantical Interpretation of Formulas in EHOL.

- The first kind of judgments derive assertions, are said to be *logical judgments*, and have the shape
$$\Gamma \mid \Phi \vdash \phi\ ,$$
where $\Gamma$ is a typing context, $\Phi$ is a finite set of assertions, and $\phi$ is an assertion. When we write $\Gamma \mid \Phi \vdash \phi$, then $\Phi$ and $\phi$ are always assumed to be well-typed under $\Gamma$. Apart from well-typedness, the assertions $\Phi$ and $\phi$ can be any assertions of the language, and as such talks about specific terms *only through* the base predicates. The judgement $\Gamma \mid \Phi \vdash \phi$ is said to be *valid* if for every $\rho \in [\![\Gamma]\!]$, if $[\![\Phi]\!]_\rho$ then $[\![\phi]\!]_\rho$. Note that validity is well-defined, due to the well-typedness condition on $\Phi$ and $\phi$.
- The second kind of judgments derive assertions about a specific program $M$, and are called *program judgments*. They have the following form:
$$\Gamma \mid \Phi \vdash M : \sigma \mid \phi\ .$$
Here, as above, $\Gamma$ is a typing context and $\Phi$ is a set of assertions. Now however, a term $M$ is put in evidence such that $M$ has type $\sigma$ in the context $\Gamma$ (i.e., $\Gamma \vdash M : \sigma$ holds) and $\phi$ talks about $M$ through a distinguished variable $\mathbf{r}$ (i.e., $\Gamma, \mathbf{r} : \sigma \vdash \phi$). For brevity, as with logical judgements we tacitly assume that all logical assertions occurring in program judgements are well-typed. Such a program judgement is *valid* if for every $\rho \in [\![\Gamma]\!]$, if $[\![\Phi]\!]_\rho$ then $[\![\phi]\!]_{\rho;\mathbf{r}\mapsto[\![M]\!]_\rho}$.

One can give formal rules for the the logical judgments in a standard way, once axioms for the basic predicates are identified [Aguirre et al. 2017]. The soundness of the obtained formal system with respect to the aforementioned set-theoretic semantics, i.e. that derivability implies validity, can be proved by induction on derivations.

But how about program judgments? Following the literature [Aguirre et al. 2017], some syntax-directed rules can indeed be given which remain valid under our interpretation. However, some care is required for recursive definitions. The complete system is given in Fig. 11. Here, we employ the usual convention that typing environments are juxtaposed only when they do not share any name, e.g., $\Gamma; x : \sigma$ is defined only when $x$ does not occur in $\Gamma$. As mentioned above, all judgements occurring in the rules are supposed to be well-typed.

To reason in an inductive style about recursive functions that converge to, rather than compute a result, we require that the assertion attached to this function through EHOL is *Scott-admissible*. Briefly, Scott-admissiblity simply states that an assertion holds for the bottom element of the underlying domain, and that it is closed under upper limits. A way to enforce Scott-admissibility consists of restricting to the class of assertions $\phi$ such that $\Gamma \vdash pos(\phi)$ is derivable by the rules in Figure 10. These assertions are called $\leq$-positive. Examples of $\leq$-positive assertions include $\mathbf{r} \leq N$, stating that the result of a computation is bounded from above by $N$, for $N$ an expression in which $\mathbf{r}$ does not occur free.

We now establish the soundness of the logic w.r.t. the denotational semantics of the language.

Theorem 5.1 (Soundness of EHOL). *If* $\Gamma \mid \Phi \vdash M : \tau \mid \phi$ *then for every* $\rho \in [\![\Gamma]\!]$,
$$[\![\Phi]\!]_\rho \qquad \textit{implies} \qquad [\![\phi]\!]_{\rho;\mathbf{r}\mapsto[\![M]\!]_\rho}\ .$$

$$\frac{\Gamma \vdash M : \mathtt{Real}^+}{\Gamma \vdash pos(\mathbf{r} \leq M)}\,[\textsc{Leq}] \qquad \frac{\Gamma \vdash \phi \quad \Gamma \vdash pos(\psi)}{\Gamma \vdash pos(\phi \Rightarrow \psi)}\,[\textsc{Arr}] \qquad \frac{\Gamma \vdash pos(\phi) \quad \Gamma \vdash pos(\psi)}{\Gamma \vdash pos(\phi \wedge \psi)}\,[\textsc{Conj}]$$

$$\frac{\Gamma \vdash pos(\phi) \quad \Gamma \vdash pos(\psi)}{\Gamma \vdash pos(\phi \vee \psi)}\,[\textsc{Disj}] \qquad \frac{\Gamma, x : \sigma \vdash pos(\phi)}{\Gamma \vdash pos(\forall x : \sigma.\phi)}\,[\textsc{Forall}]$$

Fig. 10. Inference System to Derive $\leq$-Positiveness of Assertions.

$$\frac{\Gamma \vdash x : \sigma \qquad \Gamma \mid \Phi \vdash \phi[x/\mathbf{r}]}{\Gamma \mid \Phi \vdash x : \sigma \mid \phi}\,[\textsc{Var}] \qquad \frac{\Gamma \mid \Phi \vdash M : \sigma \mid \phi \quad \Gamma; x : \sigma \mid \Phi \vdash \phi[x/\mathbf{r}] \Rightarrow \psi[x/\mathbf{r}]}{\Gamma \mid \Phi \vdash M : \sigma \mid \psi}\,[\textsc{Sub}]$$

$$\frac{\Gamma; x : \sigma \mid \Phi, \phi \vdash M : \tau \mid \psi}{\Gamma \mid \Phi \vdash \lambda x.M : \sigma \to \tau \mid \forall x : \sigma.\ \phi \Rightarrow \psi[\mathbf{r}\ x/\mathbf{r}]}\,[\textsc{Abs}]$$

$$\frac{\Gamma \mid \Phi \vdash M : \sigma \to \tau \mid \forall x : \sigma.\ \phi[x/\mathbf{r}] \Rightarrow \psi[\mathbf{r}\ x/\mathbf{r}] \quad \Gamma \mid \Phi \vdash V : \sigma \mid \phi}{\Gamma \mid \Phi \vdash M \cdot V : \tau \mid \psi[V/x]}\,[\textsc{App}]$$

$$\frac{\begin{array}{c}\Gamma; \vec{x} : \vec{\sigma} \vdash pos(\psi) \\ \Gamma; f : \vec{\sigma} \to \mathtt{Real}^+; \vec{x} : \vec{\sigma} \mid \Phi, \forall \vec{z} : \vec{\sigma}.\ \phi[\vec{z}/\vec{x}] \Rightarrow \psi[\vec{z}/\vec{x}][f\ \vec{z}/\mathbf{r}], \phi \vdash M : \mathtt{Real}^+ \mid \psi\end{array}}{\Gamma \mid \Phi \vdash \mathtt{letrec}\ f\ \vec{x} = M : \vec{\sigma} \to \mathtt{Real}^+ \mid \forall \vec{x} : \vec{\sigma}.\ \phi \Rightarrow \psi[\mathbf{r}\ \vec{x}/\mathbf{r}]}\,[\textsc{Letrec}]$$

$$\frac{\begin{array}{c}\mathsf{s} : \mathsf{B}_1 \times \cdots \times \mathsf{B}_k \to \mathsf{B} \in \mathcal{C} \cup \mathcal{F} \qquad \Gamma \mid \Phi \vdash M_1 : \mathsf{B}_1 \mid \psi_1 \qquad \cdots \qquad \Gamma \mid \Phi \vdash M_k : \mathsf{B}_k \mid \psi_k \\ \Gamma \mid \Phi \vdash \forall x_1 \cdots x_k.\psi_1[x_1/\mathbf{r}] \wedge \cdots \wedge \psi_k[x_k/\mathbf{r}] \Rightarrow \psi[\mathsf{s}(x_1, \ldots, x_k)/x]\end{array}}{\Gamma \mid \Phi \vdash \mathsf{s}(M_1, \ldots, M_k) : \mathsf{B} \mid \psi[\mathbf{r}/x]}\,[\textsc{Fun}]$$

$$\frac{\begin{array}{c}\mathsf{c} : \mathsf{B}_1 \times \cdots \times \mathsf{B}_k \to \mathsf{B} \in C \qquad \Gamma \mid \Phi \vdash M : \mathsf{B} \mid \phi \\ \Gamma; x_1 : \mathsf{B}_1; \cdots; x_k : \mathsf{B}_k \mid \Phi; \phi[\mathsf{c}(x_1, \ldots, x_k)/\mathbf{r}] \vdash N : \sigma \mid \psi \\ \Gamma; y : \mathsf{B} \mid \Phi; \forall x_1 \cdots x_k.y \neq \mathsf{c}(x_1, \ldots, x_k) \wedge \phi[y/\mathbf{r}] \vdash O : \sigma \mid \psi\end{array}}{\Gamma \mid \Phi \vdash \mathsf{case}\ M\ \mathsf{of}\ \{\mathsf{c}(\vec{x}) \mapsto N \mid y \mapsto O\} : \sigma \mid \psi}\,[\textsc{Case}]$$

Fig. 11. Higher-order Logic EHOL.

For an informal explanation and intuitions about EHOL, we now conclude this section with an analysis of the running examples.

### 5.1 Examples

*Coupon collector.* Let us first return to the coupon collector function, whose translated version is given in Figure 1b. For brevity, we consider the slightly simplified variant `couponsCost` that elides the continuation, which, removing some syntactic sugar, reads as follows:

$$\mathtt{couponsCost} : \mathtt{List(C)} \to \mathtt{Real}^+$$
$$\mathtt{couponsCost} \triangleq \lambda cs.(\mathtt{letrec}\ \mathtt{collectCost}\ os = \mathtt{if}\ cs \subseteq os\ \mathtt{then}\ \mathtt{0}\ \mathtt{else}\ \mathtt{1} + \textstyle\sum_{c \in cs} {}^{1}\!/_{|cs|} * \mathtt{collectCost}\ (c :: os)) \cdot [\,]$$

The most important step in the analysis lies in the treatment of the auxiliary recursive function $\mathtt{collectCost} : \mathtt{List(C)} \to \mathtt{Real}^+$, which we resolve as follows. Here, we will exploit that $os$ — the list of coupons drawn so far — consists only of coupons from the overall list $cs$ of coupons. We

express this as a predicate $os \subseteq cs$, conceiving $os$ and $cs$ as sets. What we would now like to prove is that for any such $os$, the result of $\texttt{collectCost} \cdot os$ is bounded by a function in $os$. Let us denote this function by $Q(os)$; the concrete definition of $Q$ is deferred to the end of this paragraph. In other words, we would like to verify:

$$cs : \texttt{List(C)} \mid \cdot \vdash \texttt{letrec}\ \texttt{collectCost}\ os = P : \texttt{List(C)} \to \texttt{Real}^+ \\ \mid \forall os : \texttt{List(C)}.\ os \subseteq cs \Rightarrow \mathbf{r} \cdot os \leq Q(os)\ ,$$

for $P$ the body of $\texttt{collectCost}$. For brevity, let us omit types in universal quantification. Observe that $\mathbf{r} \cdot os \leq Q(os)$ is $\leq$-positive, and so by Rule (Letrec) this judgment boils down to

$$\Gamma \mid \forall os'.\ os' \subseteq cs \Rightarrow \texttt{collectCost} \cdot os' \leq Q(os;); os \subseteq cs \\ \vdash \texttt{if}\ cs \subseteq os\ \texttt{then}\ \texttt{0}\ \texttt{else}\ \texttt{1} + \textstyle\sum_{c \in cs} {}^{\texttt{1}}/_{|cs|} * \texttt{collectCost}\ (c :: os) : \texttt{Real}^+ \mid \mathbf{r} \leq Q(os)\ , \quad (\star)$$

where $\Gamma \triangleq cs : \texttt{List(C)}; os : \texttt{List(C)}$. In effect, we should thus prove that evaluating (the body of) $\texttt{collectCost}$ on $cs$ yields indeed a result bounded by $Q(os)$, where we may assume that this property holds for any recursive call on $os \subseteq cs$. Let us collect in $\Phi$ the two logical premises of this judgment, and proceed with Rule (Case) on the desugared-conditional. Here, first one verifies that, for the guard,

$$\Gamma \mid \Phi \vdash cs \subseteq os : \texttt{Bool} \mid \mathbf{r} = cs \subseteq os\ ,$$

holds. In the treatment of the then- and else-branches, we can now assume $\texttt{true} = cs \subseteq os$ and $\texttt{false} = cs \subseteq os$, respectively. Let us denote these two facts by $cs \subseteq os$ and $cs \not\subseteq os$ for brevity. Concerning the two branches, it is standard to check

$$\Gamma \mid \Phi; cs \subseteq os \vdash \texttt{0} : \texttt{Real}^+ \mid \mathbf{r} \leq \texttt{0}$$
$$\Gamma \mid \Phi; cs \not\subseteq os \vdash \texttt{1} + \textstyle\sum_{c \in cs} {}^{\texttt{1}}/_{|cs|} * \texttt{collectCost}\ (c :: os) : \texttt{Real}^+ \mid \mathbf{r} \leq \texttt{1} + \sum_{c \in cs} {}^{\texttt{1}}/_{|cs|} * Q(c :: os)\ .$$

To finalise the proof, though, we will have to show $\mathbf{r} \leq Q(os)$ for both branches, as demanded by Rule (Case) on the judgement ($\star$). To this end, we can use Rule (Sub) in the two branches, which effectively constraints $Q(os)$ to satisfy

$$\Gamma \mid \Phi; cs \subseteq os \vdash \texttt{0} \leq Q(os) \quad \text{and} \quad \Gamma \mid \Phi; cs \not\subseteq os \vdash \texttt{1} + \textstyle\sum_{c \in cs} {}^{\texttt{1}}/_{|cs|} * Q(c :: os) \leq Q(os)\ .$$

Coming to a concrete bound, let us define $Q(os) \triangleq |cs| * H(|cs \setminus os|)$, where $H(n) = \sum_{i=1}^{n} {}^{1}/_{i}$ denotes the $n$-th harmonic number, with the convention that $H(0) = 0$. While the first constraint can be immediately discharged, note that the latter simplifies to

$$\Gamma \mid \Phi; cs \not\subseteq os \vdash 1 + \textstyle\sum_{c \in cs} H(|cs \setminus c :: os|) \leq |cs| * H(|cs \setminus os|)\ .$$

Now observe that, when $c \in cs$ is already in $os$, then $H(|cs \setminus c :: os|)$ is simply $H(|cs \setminus os|)$. On the other hand, when $c$ is a newly collected coupon, exploiting that the set of collected coupons $os$ is a proper subset of all coupons $cs$ under the given assumptions, then $H(|cs \setminus c :: os|) = H(|cs \setminus os| - 1)$ with $|cs \setminus os| > 0$. It is thus the case that the above constraint is equivalent to

$$\Gamma \mid \Phi; cs \not\subseteq os \vdash 1 + |os| * H(|cs \setminus os|) + |cs \setminus os| * H(|cs \setminus os| - 1) \leq |cs| * H(|cs \setminus os|)\ .$$

Using the identity

$$H(|cs \setminus os|) = H(|cs \setminus os| - 1) + \frac{1}{|cs \setminus os|}$$

this last constraint is now not difficult to discharge.

We have now established, within EHOL, the concrete bound $Q(os) \triangleq |cs| * H(|cs \setminus os|)$ for the auxiliary function $\texttt{collectCost}$. From here, we finally see that

$$\cdot \mid \cdot \vdash \texttt{couponsCost} : \texttt{List(C)} \to \texttt{Real}^+ \mid \forall cs.\ \mathbf{r} \leq |cs| * H(|cs|)\ .$$

Observe that couponsCost is nothing else than couponsCPS with the continuation $k$ specialized to the constant-zero function. With Theorem 4.15, the so derived bound now transfers to the probabilistic source program coupons from Figure 1a. Notice that this bound, which lies in $O(|cs| * \log(|cs|))$, is precise.

*Higher-order random walks.* Let us now turn our attention to the higher-order combinator function walk from Example 3.1, whose CPSed version is given in Example 4.7. As usual for many higher-order combinators and not unique to probabilistic programs, its (expected) cost — here the number of iterations — is impossible to judge by looking at walk alone: the cost depends crucially on the supplied predicate $\mathtt{p} : \mathtt{Int} \to \mathtt{Bool}$ used to terminate the walk, and the function $\mathtt{f} : \mathtt{Int} \to \mathtt{Int}$ picking the next position in terms of the current one. Indeed, if we instantiate walk for instance to geo, as we have done in Example 3.1, then the expected cost becomes constant, while if we instantiate walk as in $\mathtt{randomWalk}_{\mathtt{p}}$ it can even become unbounded.

Is it then even conceivable to reason about this function compositionally? With EHOL, we can answer this question in the affirmative. The logic is expressive enough to articulate the interactions between functions and its arguments in the necessary granularity on the CPSed terms, even at higher-order types. Let us illustrate this with a single EHOL proof on the CPSed term walkCPS of walk given in Example 4.7. As a second step, we then show how the derived bound can be instantiated to reason about the expected cost of geo and randomWalk, despite that their complexity vastly differs.

For concrete parameters p and f, let us express the expected cost of walk as a function $G(n)$ in terms of the current integer position. Let us first assume that f and p are non-probabilistic, and let us denote by an assertion $\psi(n)$ the Boolean value of $\mathtt{p} \cdot n$ within EHOL. Then $G(n)$ should, in the case the else-branch is executed (i.e., $\neg\psi(n)$), bind the cost $G(\mathtt{f} \cdot n) + 1$ of the recursive call plus one for the current iteration. As the cost transformer translates $\mathtt{f} \cdot n$ to a CPSed term $\mathtt{fCPS} \cdot n \cdot \kappa$, with $\kappa$ the cost of its continuation — in our case that of the recursive call — this amounts to saying that fCPS should satisfy

$$\forall n : \mathtt{Int}.\ \neg\psi(n) \Rightarrow \forall r : \mathtt{Int} \to \mathtt{Real}^+.(\forall m : \mathtt{Int}.\ \top \Rightarrow r\ m \leq G(m)) \Rightarrow \mathtt{fCPS}\ n\ r + 1 \leq G(n)\ .$$

The universally quantified variable $r$ refers to the cost of the recursive call and is therefore by assumption bounded by $G$.

Let us now consider the general case. So far, in our argumentation we have neglected that the then-branch of the conditional incurs a cost depending on the supplied argument $k : \mathtt{Int} \to \mathtt{Real}^+$. Second, the guard p may be probabilistic, hence the two branches of the conditional are potentially only executed with a certain probability and, conclusively, their expected cost could be much higher than the expected cost $G(n)$, i.e., the value of $\mathtt{walkCPS}\ f\ p\ n\ k$. Therefore, let us introduce two functions $T(n)$ and $E(n)$ to denote bounds for the then- and else-branch, respectively. First, we set up $T$ to indeed bind $k$, via

$$\phi(k) \triangleq \forall n : \mathtt{Int}.\psi(n) \Rightarrow k \cdot n \leq T(n)\ .$$

Second, the above constrain on the stepping-function turns into

$$\phi(f) \triangleq \forall n : \mathtt{Int}.\ \neg\psi(n) \Rightarrow \forall r : \mathtt{Int} \to \mathtt{Real}^+.(\forall m : \mathtt{Int}.\ \top \Rightarrow r \cdot m \leq G(m)) \Rightarrow f \cdot n \cdot r + 1 \leq E(n)\ .$$

Let us denote by $[\cdot]$ the indicator function on assertions. Once the two above bounds are in place, a bound for the conditional

$$\mathtt{if}\ b\ \mathtt{then}\ k \cdot n\ \mathtt{else}\ 1 + f \cdot n \cdot \kappa_r$$

is given with $[\psi(n)] * T(n) + [\neg\psi(n)] * E(n)$ when $b = \psi(n)$, for any $\kappa_r : \mathtt{Int} \to \mathtt{Real}^+$ bounded by $G$. What we now require for the CPSed version $p : \mathtt{Int} \to (\mathtt{Bool} \to \mathtt{Real}^+) \to \mathtt{Real}^+$ of the guard

— the entry point of the recursion — is the following:

$$\phi(p) \triangleq \forall n : \mathtt{Int}.\ \top \Rightarrow \forall cond : \mathtt{Bool} \to \mathtt{Real}^+.$$
$$(\forall b : \mathtt{Bool}.\ b = \psi(n) \Rightarrow cond \cdot b \leq [\psi(n)] * T(n) + [\neg\psi(n)] * E(n)) \Rightarrow p \cdot n \cdot cond \leq G(n)$$

This formula ties the assertion $\psi(n)$ to the possibly sampled Boolean outcome within the conditional continuation, whose bound is assumed to be of the shape as just derived. Finally, this assertion also relates the outcome of $\mathtt{walkCPS}$'s body to the bounding function $G(n)$.

It is now standard to check that the assertions on the arguments yield that $\mathtt{walkCPS} \cdot p \cdot f \cdot n \cdot k$ is bounded by $G(n)$, i.e.,

$$\cdot \mid \cdot \vdash \mathtt{walkCPS} : \sigma \mid \forall p.\ \phi(p) \Rightarrow \forall f.\phi(f) \Rightarrow \forall n.\top \Rightarrow \forall k.\ \phi(k) \Rightarrow \mathbf{r} \cdot f \cdot n \cdot k \leq G(n)\ ,$$

for $\sigma$ the type of $\mathtt{walkCPS}$, is valid.

Let us now look at the instances $\mathtt{geoCPS}$ and $\mathtt{randomWalkCPS_p}$ also given in Example 4.7, yielding precisely the expected cost of functions $\mathtt{geo}$ and $\mathtt{randomWalk}$ given in Example 3.1. Assume these two functions are executed in an empty context. Correspondingly, in both cases we restrict our attention to $k = \lambda n.\mathtt{0}$ and set $T(n) \triangleq 0$. Recall that in the case of $\mathtt{geoCPS}$, functions

$$\mathtt{p} \triangleq (\lambda\_k.{}^1\!/\!_2 * (k \cdot \mathtt{true}) + {}^1\!/\!_2 * (k \cdot \mathtt{false})) \quad \text{and} \quad \mathtt{f} \triangleq (\lambda nk.k \cdot (n + 1))$$

are supplied as arguments $p$ and $f$ to $\mathtt{walkCPS}$, respectively. Fix now $\psi(n) \triangleq b$, and take $G(n) \triangleq 1$ and $E(n) \triangleq 2$. The term $[\psi(n)] * T(n) + [\neg\psi(n)] * E(n)$ simplifies to $[\neg b] * 2$. As we can derive validity of the judgments

$$\cdot \mid \cdot \vdash \mathtt{p} : \mathtt{Int} \to (\mathtt{Bool} \to \mathtt{Real}^+) \to \mathtt{Real}^+ \mid \forall n.\ \top \Rightarrow \forall cond.\ (\forall b.\ b = b \Rightarrow cond \cdot b \leq [\neg b] * 2) \Rightarrow p \cdot n \cdot cond \leq 1\ ,$$

and

$$\cdot \mid \cdot \vdash \mathtt{f} : \mathtt{Int} \to (\mathtt{Int} \to \mathtt{Real}^+) \to \mathtt{Real}^+ \mid \forall n.\ \neg b \Rightarrow \forall r.(\forall m.\ r \cdot m \leq 1) \Rightarrow f \cdot n \cdot r + 1 \leq 2\ ,$$

thereby witnessing that $\phi(\mathtt{p})$ and $\phi(\mathtt{f})$ hold, we conclude that

$$n : \mathtt{Int} \mid \cdot \vdash \mathtt{geoCPS} \cdot \mathtt{p} \cdot \mathtt{f} \cdot n \cdot (\lambda n.\mathtt{0}) : \mathtt{Real}^+ \mid \mathbf{r} \leq 1\ .$$

This shows that, independent on $n$, $\mathtt{geo}$'s expected cost is bounded by one.

The function $\mathtt{randomWalkCPS_p}$ is treated in a similar fashion, using the parameters $\psi(n) \triangleq (n \leq 0)$, $T(n) = 0$ and $E(n) = G(n) \triangleq {}^1\!/\!_{2p-1} * \mathrm{abs}(n)$ for $\mathrm{abs}(n)$ the absolute value of the Integer $n$. While checking $\phi(\mathtt{p})$ for $\mathtt{p} = \lambda k.k \cdot (n \leq \mathtt{0})$ poses no problems, verifying $\phi(\mathtt{f})$ for

$$\mathtt{f} = \lambda nk.\mathtt{p} * (k \cdot (n - 1)) + (1 - \mathtt{p}) * (k \cdot (n + 1))$$

leaves us, slightly simplifying, with the constraint

$$n : \mathtt{Int} \mid n > \mathtt{0} \Rightarrow p * {}^1\!/\!_{2p-1} * \mathrm{abs}(n-1) + (1-p) * {}^1\!/\!_{2p-1} * \mathrm{abs}(n+1) \leq {}^1\!/\!_{2p-1} * \mathrm{abs}(n)\ .$$

As this constraint is valid whenever $p > {}^1\!/\!_2$, we then ultimately conclude that the expected cost of $\mathtt{randomWalk_p} \cdot \mathtt{n}$ is bounded by ${}^1\!/\!_{2p-1} * \mathrm{abs}(n)$ for such $p$.

## 6 EMBEDDING OF THE ERT CALCULUS

In Section 2, we have motivated the use of a CPS translation for reasoning about expected costs by coupling Rosendahl's program transformation approach with ideas from the expected runtime transformer of Kaminski et al. [2018], itself presented in the form of a weakest pre-expectation calculus for reason about the expected runtime of randomized algorithms written in an imperative probabilistic language. In this section, we are going to show how the ert-calculus can be embedded

| $\mathtt{C}$ | $(\!\|\mathtt{C}\|\!) : \mathtt{Val}_n \to \mathtt{Val}_n$ | $(\!\|\mathtt{C}\|\!)^\dagger : \mathtt{Val}_n \to (\mathtt{Val}_n \to \mathtt{Real}^+) \to \mathtt{Real}^+$ |
|---|---|---|
| `empty` | $\lambda\sigma.\sigma$ | $\lambda\sigma k.k \cdot \sigma$ |
| `skip` | $\lambda\sigma.\sigma^\checkmark$ | $\lambda\sigma k.1 + k \cdot \sigma$ |
| `halt` | $\Omega$ | $\lambda\sigma k.\mathtt{0}$ |
| $x_i :\approx \mu$ | $\lambda\sigma.\mathtt{upd}_i(\sigma, \mathtt{p}_\mu(\sigma))^\checkmark$ | $\lambda\sigma k.1 + \mathtt{E}_{\mu(\sigma)}(\lambda v.k \cdot \mathtt{upd}_i(\sigma, v))$ |
| $\mathtt{C}_1;\mathtt{C}_2$ | $\lambda\sigma.(\!\|\mathtt{C}_2\|\!) \cdot ((\!\|\mathtt{C}_1\|\!) \cdot \sigma)$ | $\lambda\sigma k.(\!\|\mathtt{C}_1\|\!)^\dagger \cdot \sigma \cdot (\lambda\sigma'.(\!\|\mathtt{C}_2\|\!)^\dagger \cdot \sigma' \cdot k)$ |
| `if` $(\zeta)$ $\{\mathtt{C}_1\}$ $\{\mathtt{C}_2\}$ | $\lambda\sigma.(\mathtt{if}\ \mathtt{p}_\zeta(\sigma)$ `then` $(\!\|\mathtt{C}_1\|\!) \cdot \sigma$ `else` $(\!\|\mathtt{C}_2\|\!) \cdot \sigma)^\checkmark$ | $\lambda\sigma k.1 + \mathtt{E}_{\zeta(\sigma)}(\lambda b.\mathtt{if}\ b$ `then` $(\!\|\mathtt{C}_1\|\!)^\dagger \cdot \sigma \cdot k$ `else` $(\!\|\mathtt{C}_2\|\!)^\dagger \cdot \sigma \cdot k)$ |
| `while` $(\zeta)$ $\{\mathtt{D}\}$ | `letrec` $loop\ \sigma =$ $(\mathtt{if}\ \mathtt{p}_\zeta(\sigma)$ `then` $loop \cdot ((\!\|\mathtt{D}\|\!) \cdot \sigma)$ `else` $\sigma)^\checkmark$ | `letrec` $loop\ \sigma\ k =$ $1 + \mathtt{E}_{\zeta(\sigma)}(\lambda b.\mathtt{if}\ b$ `then` $(\!\|\mathtt{D}\|\!)^\dagger \cdot \sigma \cdot (\lambda\sigma'.loop \cdot \sigma' \cdot k)$ `else` $k \cdot \sigma)$ |

Fig. 12. Embedding of the ert-Calculus within our Language and Resulting Expectation Transformer.

into our formalism by way of a state-passing monad, thus witnessing that our approach strictly extends upon this sound and complete methodology.

The imperative language under consideration is based on Dijkstra's *Guarded Command Language*, additionally equipped with sampling operations. More specifically, the underlying language of programs (in the style of IMP, see e.g. [Winskel 1993]) includes assignments, sequencing, conditionals, and loops. Variable assignments and guards are however allowed to be probabilistic. A full list of *program commands* $\mathtt{C}$ can be seen in the leftmost column of Figure 12, where $\mu$ stands for a probabilistic expression and $\zeta$ for a probabilistic guard. All variables are global in IMP, and without any loss of generality we can assume that any *program value* $v : \mathtt{Val}$ is drawn from a discrete but otherwise arbitrary domain, while *program states* $\sigma : \mathtt{Val}_n$ map the $n$ program variables $x_1, \ldots, x_n$ to their content $v_i : \mathtt{Val}$.

The second column of Figure 12 gives a functional interpretation of imperative commands $\mathtt{C}$ within our source language, as a probabilistic term $\vdash (\!|\mathtt{C}|\!) : \mathtt{Val}_n \to \mathtt{Val}_n$ mapping initial states to final ones. From an operational perspective, `empty` and `skip` are no-ops. Whereas `empty` incurs no cost, `skip` incurs a runtime cost of one by definition. The command `halt` signals an abnormal termination and is interpreted as the non-terminating function $\Omega = \mathtt{letrec}\ f\ x = (f \cdot x)$. Probabilistic assignment $x_i :\approx \mu$ assigns to $x_i$ a value sampled from the (discrete) distribution expression $\mu$. Within the function interpretation of such assignments, $\mu$ is represented as a sampling primitive $\mathtt{p}_\mu : \mathtt{Val}_n \to \mathtt{Val} \in \mathcal{D}$, a primitive function $\mathtt{upd}_i : \mathtt{Val}_n \times \mathtt{Val} \to \mathtt{Val}_n$ is used to update the content of $x_i$ with the value sampled value, and a runtime cost of one is incurred. The interpretation of command composition, conditionals and while-loops is standard. Since guards $\zeta$ can be probabilistic, such expressions are represented via probabilistic primitives $\mathtt{p}_\zeta : \mathtt{Val}_n \to \mathtt{Bool} \in \mathcal{D}$, as in the case of assignments.

In the rightmost column of Figure 12 we draw the value translation on the functional interpretation of commands $\mathtt{C}$. We have simplified the presentation by removing administrative redexes, and exploiting that $(\!|\mathtt{halt}|\!)^\dagger = \Omega^\dagger$ is semantically equivalent to the constant zero function.

*Correctness of the Embedding.* The just described embedding can be proved correct by relating it to the so-called *expected runtime transformer* as defined by Kaminski et al. Let $\mathbb{T}$ denote the set of *runtimes*, i.e., the set of functions from program states to $\mathbb{R}^{+\infty}$. The expected runtime transformer

expresses the expected runtime of any program $\mathtt{C}$ by means of a transformer $\mathrm{ert}[\mathtt{C}] : \mathbb{T} \to \mathbb{T}$ on runtimes. In particular, $\mathrm{ert}[\mathtt{C}]$ applied to the constant zero function yields the expected runtime of $\mathtt{C}$. The following captures the fact that what we do in Figure 12 is not different from $\mathrm{ert}[\mathtt{C}]$, thus correct.

Proposition 6.1. *For all imperative programs* $\mathtt{C}$ *and* $\vdash \kappa : \mathtt{Val}_{\mathsf{n}} \to \mathtt{Real}^{+}$,

$$\mathrm{ert}[\mathtt{C}](\llbracket \kappa \rrbracket) = \left(\sigma \mapsto \llbracket \mathrm{ect}\left[\langle\!|\mathtt{C}|\!\rangle \cdot \sigma\right]\{\kappa\}\rrbracket\right) .$$

## 7 DISCUSSION

*Expectations and AST.* We have showed that the expected cost of any expression $P$ can be obtained by applying the denotational interpretation of its CPS transform to the continuation $\lambda v.0$. However, the usefulness of the CPS translation is not limited to deriving expected costs. In particular, it can be used for computing the (pre-)expectation of any expression, as captured by the following result.

Theorem 7.1. $\mathbb{E}_{\mathrm{nf}(P)}\left(\llbracket \kappa \rrbracket\right) \equiv \mathrm{ect}[P]\{\kappa\}$ *for any cost-free term* $P \in \Lambda_{p}^{\mathsf{B}}$ *and* $\kappa \in \Lambda_{\mathbb{R}}^{\mathsf{B}\to\mathtt{Real}^{+}}$.

As an example, the CPS translation can be used to reason about the probability of termination. Specifically, Theorem 7.1 instantiated with $\kappa$ being set to $\lambda v.1$ boils down to saying that the semantics of $\mathrm{ect}[P]\{\kappa\}$ precisely gives the probability of convergence of $P$. Giving an estimation to the probability of termination and to the expected cost are well known to be highly undecidable problems having incomparable recursion-theoretic statuses, the former being $\Pi_0^2$-complete and the second being $\Sigma_0^2$-complete [Kaminski and Katoen 2015].

*Type-based Verification.* The formal system of higher-order logic we introduced in Section 5 is a sound methodology to derive upper bounds on the expected cost of higher-order randomised programs *via* a CPS translation, as witnessed by Theorem 4.15 and Theorem 5.1. In EHOL, types and assertions are separated, and play different roles: the former plays the usual role of guaranteeing the absence of type errors, without providing any guarantee on the actual I/O behavior of terms, while the latter can be employed to specify such constraints. A different approach is that of refinement types, in which specifications are embedded into types. Since refinement types are well-known to be interpretable in HOL [Aguirre et al. 2017], the obtained type systems would arguably not add anything to HOL in terms of expressive power, although taking the form of a proper type system.

*Continuous Distributions.* The CPS transformation we consider in this paper takes as input terms of a probabilistic $\lambda$-calculus in which sampling can be performed on *discrete* distributions, only. Generalizing what we do to transformation in which, on the other hand, we can also perform sampling from *continuous* distributions, would be extremely interesting as a first step towards moving towards Bayesian programming languages. The main obstacle, in this sense, consists in endowing the target language with an operator for integration, itself necessary for interpreting the expectation of programs which sample from continuous distributions.

*Applicability.* While we have shown that the expected cost of the source program is precisely reflected within the target program, and thereby our CPS translation constitutes not only a sound but also complete methodology for reasoning about expected costs, our program logic EHOL has been proven only sound for deriving upper-bounds. On the other hand, we conjecture that EHOL is also complete relative to the underlying programming logic, thereby further witnessing the strength of our overall approach. A formal proof, though, is beyond the scope of this work.

Orthogonal, it would also be interesting to implement EHOL in the style of an interactive theorem prover, and couple this implementation with the CPS translation. Such an implementation would greatly improve upon the applicability of our approach. While this is clearly feasible, a further step

would be to incorporate some form of automatism for deriving upper-bounds. At least for programs whose cost analysis is not relying on too complicated functional properties, it seems feasible to incorporate template based approach resting on SMT-solvers — the pre-dominant approach underlying automated tools (see e.g. [Avanzini et al. 2015; Wang et al. 2020]). Ideally, one would arrive at a semi-automated analysis tool, where simple proof steps are automatically discharged and the user can focus on the non-trivial aspects of the cost analysis.

## 8 RELATED WORK

This is definitely not the first paper about randomised higher-order computation. Starting from the pioneering work by Saheb-Djaromi [Saheb-Djahromi 1978], the field has developed at a relatively slow pace, mainly due to the intrinsic difficulties one faces when giving denotational semantics to such programs [Jung and Tix 1998]. Recently, satisfactory solutions for the problem have been devised [Ehrhard et al. 2014; Goubault-Larrecq 2019; Heunen et al. 2017], and also other challenges like termination and complexity analysis or program equivalence have been tackled.

The growing interest in Bayesian programming idioms such as Anglican [Tolpin et al. 2016], Church [Goodman et al. 2008], Hakaru [Narayanan et al. 2016], themselves functional languages, has also served as a motivation for the study of higher-order probabilistic programming languages. In these languages, programs are meant to embody probabilistic models rather than algorithms, and evaluating a program consists of applying inference algorithms to the underlying model. For all this to be useful, programs are required not only to sample from (various forms of) distributions but also to condition the result of sampling to the observed data. The vehicle lambda-calculus we consider here does not have any primitive for conditioning. Noticeably, program transformations, some of them reminiscent to ours [Ramsey and Pfeffer 2002], have been used to improve the performances of inference algorithms.

Continuation-passing style transformations [Danvy and Filinski 1992; Plotkin 1975] are a pervasive tool in program transformation and compiler construction [Appel 1991]. They are, in particular, very effective as a way to lift various forms of effects away from the underlying program [Filinski 1994], this way restoring purity, and facilitating program analysis and optimizations. The kind of CPS transformation we consider here is peculiar, as it is focused on probabilistic and cost effects. CPS transformations specifically tailored for randomised $\lambda$-calculi have already been considered [Dal Lago and Zorzi 2012], the underlying goal being the one of targeting a calculus in which the underlying reduction strategy does not matter. CPS transformations have already found applications in the context of Anglican and other Bayesian programming languages, where they serve as a way to implement Bayesian inference (see also [Ścibior et al. 2018]).

Over the recent years, the literature on (almost-sure, bounded) termination and, as a refinement, resource analysis of probabilistic, specifically imperative programs has significantly grown. Solutions have taken the form of abstract interpretations [Chakarov and Sankaranarayanan 2014; Monniaux 2001]; martingales, e.g., 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 [Avanzini et al. 2019b; Bournez and Garnier 2005]; model checking [Katoen 2016]; program logics [Avanzini et al. 2020; Kaminski et al. 2018; McIver et al. 2018; Ngo et al. 2018; Wang et al. 2018]; proof assistants [Barthe et al. 2009; Tassarotti and Harper 2018]; recurrence relations [Sedgewick and Flajolet 1996]; methods based on program analysis [Celiku and McIver 2005; Katoen et al. 2010; Kozen 1985]; and finally, symbolic inference [Gehr et al. 2016]. Notable, among them is the ert-calculus that we relate to in Section 6.

When, on the other hand, the underlying program has higher-order functions, the range of techniques currently available is much narrower and is essentially reduced to types. Although the

kind of verification technique presented in this paper is substantially different, being based on program transformation, a comparison with type-based techniques is in order.

The recent works on expected cost analysis by way of linear dependent types [Avanzini et al. 2019a] and intersection types [Dal Lago et al. 2021] aim at giving very expressive type systems in which bounds on the expected cost of the typed programs can be derived. In the first case, we are talking about a system obtained by generalizing Dal Lago and Gaboardi's ideas [Dal Lago and Gaboardi 2011] to a probabilistic $\lambda$-calculus, obtaining a very expressive, although not relatively complete, methodology. The feasibility of type inference is not considered. In the second case, we are faced with a type system which is complete, but unsuitable for verification: as usual in intersection-type systems [Bucciarelli et al. 2003], the expressiveness in terms of the *functions* (as opposed to the *computations*) that can be captured is very low. In other words, typing a functional program requires, in general, giving a distinct type derivation for every possible input.

Notable also, in [Wang et al. 2020], the amortized cost analysis underlying *resource aware ML* [Hoffmann et al. 2012] has been suited to probabilistic programs. This work sets itself apart by having a fully automated inference machinery, differently from what we do here. The two particular instances of `walk` given in Example 3.1 lie within the scope of [Wang et al. 2020]'s methodology, *when specialized* to two first-order programs. However, depending on its functional arguments, the expected cost of `walk` can vary between constant, to linear, or even be infinite. To the best of our knowledge, the higher-order function `walk` itself is beyond the scope of any automated technique, so far. Within EHOL, as shown in Section 5, we are able to derive a *parametric* bounding function for the higher-order combinator, encompassing these two concrete instances. This makes our treatment of this example subtly different and more general than the one in [Wang et al. 2020].

## 9 CONCLUSION

In this paper, we studied how CPS program transformations can be employed in the setting of randomised computation, particularly as a technique for reducing the problem of estimating the expected cost of a randomised program to the one of analyzing the input-output behavior of its CPS counterpart. Along the way, we proved the introduced program transformation correct, and we showed how a formal system of unary higher-order logic soundly derives upper bounds to the output-value of deterministic programs obtained via our CPS transformation. This, in turn, can be lifted back to an upper-bound on the expected cost of the initial randomized program.

We see the idea of channeling the expected cost through the underlying continuation as a natural one, which could be useful also for other calculi. As an example, extending what we have done here to calculi with continuous distributions looks feasible. Another related but orthogonal direction consists of capturing a form of (soft) conditioning primitives akin to `score` or `observe` by way of CPS, this way going towards the inference machinery of languages like Anglican. Finally, how about generalizing all this to algebraic effects? This would rely on a generic notion of expectation, which is unfortunately lacking.

## ACKNOWLEDGMENTS

This work is partially supported by the Agence Nationale de la Recherge (ANR) under the Grant ANR-19-CE48-0014 (https://www.irif.fr/anrpps) and the European Research Council under the ERC CoG DIAPASoN GA 818616 (https://site.unibo.it/diapason/en).

## REFERENCES

S. Agrawal, K. Chatterjee, and P. Novotný. 2018. Lexicographic Ranking Supermartingales: An Efficient Approach to Termination of Probabilistic Programs. *PACMPL* 2, POPL (2018), 34:1–34:32. https://doi.org/10.1145/3158122

A. Aguirre, G. Barthe, M. Gaboardi, D. Garg, and P.-Y. Strub. 2017. A Relational Logic for Higher-order Programs. *PACMPL* 1, ICFP (2017), 21:1–21:29. https://doi.org/10.1145/3110265

Elvira Albert, Puri Arenas, Samir Genaim, German Puebla, and Damiano Zanardini. 2012. Cost analysis of object-oriented bytecode programs. *Theor. Comput. Sci.* 413, 1 (2012), 142–159. https://doi.org/10.1016/j.tcs.2011.07.009

Andrew W. Appel. 1991. *Compiling with Continuations.* Cambridge University Press. https://doi.org/10.1017/CBO9780511609619

R. Atkey. 2011. Amortised Resource Analysis with Separation Logic. *Logical Methods in Computer Science* 7, 2 (2011). https://doi.org/10.2168/LMCS-7(2:17)2011

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

M. Avanzini, U. Dal Lago, and G. Moser. 2015. Analysing the Complexity of Functional Programs: Higher-Order Meets First-Order. In *Proc. of 20$^{th}$ ICFP*. ACM, 152–164. https://doi.org/10.1145/2784731.2784753

M. Avanzini, U. Dal Lago, and A. Yamada. 2019b. On Probabilistic Term Rewriting. *SCP* 185 (2019), 102338.

M. Avanzini, G. Moser, and M. Schaper. 2020. A Modular Cost Analysis for Probabilistic Programs. *PACMPL* 4, OOPSLA (2020), 172:1–172:30. https://doi.org/10.1145/3428240

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

K. Batz, B. L. Kaminski, J.-P. Katoen, and C Matheja. 2021. Relatively Complete Verification of Probabilistic Programs: An Expressive Language for Expectation-based Reasoning. *PACMPL* 5, POPL (2021), 1–30. https://doi.org/10.1145/3434320

O. Bournez and F. Garnier. 2005. Proving Positive Almost-Sure Termination. In *Proc. of 16$^{th}$ RTA (LNCS, Vol. 3467)*. Springer, 323–337. https://doi.org/10.1007/978-3-540-32033-3_24

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

A. Bucciarelli, A. Piperno, and I. Salvo. 2003. Intersection Types and lambda-Definability. *MSCS* 13, 1 (2003), 15–53. https://doi.org/10.1017/S0960129502003833

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

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

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

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

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

K. Chatterjee, P. Novotný, and D. Zikelic. 2017b. Stochastic Invariants for Probabilistic Termination. In *Proc. of 44$^{th}$ POPL*. ACM, 145–160. https://doi.org/10.1145/3009837.3009873

P. Cousot and R. Cousot. 1977. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proc. of 4$^{th}$ POPL*. ACM, 238–252. https://doi.org/10.1145/512950.512973

J. W. Cutler, D. R. Licata, and N. Danner. 2020. Denotational Recurrence Extraction for Amortized Analysis. *PACMPL* 4, ICFP (2020), 97:1–97:29. https://doi.org/10.1145/3408979

U. Dal Lago, C. Faggian, and S. Ronchi Della Rocca. 2021. Intersection Types and (Positive) Almost-sure Termination. *PACMPL* 5, POPL (2021), 1–32. https://doi.org/10.1145/3434313

Ugo Dal Lago and Marco Gaboardi. 2011. Linear Dependent Types and Relative Completeness. *Log. Methods Comput. Sci.* 8, 4 (2011).

U. Dal Lago and C. Grellois. 2017. Probabilistic Termination by Monadic Affine Sized Typing. In *Proc. of 26$^{th}$ ESOP (LNCS)*. Springer, 393–419. https://doi.org/10.1145/3293605

U. Dal Lago and M. Zorzi. 2012. Probabilistic Operational Semantics for the Lambda Calculus. *RAIRO - TIA* 46, 3 (2012), 413–450. https://doi.org/10.1051/ita/2012012

N. A. Danielsson. 2008. Lightweight Semiformal Time Complexity Analysis for Purely Functional Data Structures. In *Proc. of 35$^{th}$ POPL*. ACM, 133–144. https://doi.org/10.1145/1328438.1328457

N. Danner, D. R. Licata, and Ramyaa. 2015. Denotational Cost Semantics for Functional Languages with Inductive Types. In *Proc. of 20$^{th}$ ICFP*. ACM, 140–151. https://doi.org/10.1145/2784731.2784749

Oliver Danvy and Andrzex Filinski. 1992. Representing Control: a Study of the CPS Transformation. *Mathematical Structures in Computer Science* 2, 4 (1992), 361–391. https://doi.org/10.1017/S0960129500001535

O. Danvy and L. R. Nielsen. 2003. A First-Order One-pass CPS Transformation. *TCS* 308, 1-3 (2003), 239–257. https://doi.org/10.1016/S0304-3975(02)00733-8

K. De Leeuw, E. Moore, C. Shannon, and N. Shapiro. 1956. Computability by Probabilistic Machines. *Automata Studies* 34 (1956), 183–198.

T. Ehrhard, M. Pagani, and C. Tasson. 2014. Probabilistic Coherence Spaces are Fully Abstract for Probabilistic PCF. In *POPL*. ACM. https://doi.org/10.1145/2535838.2535865

T. Ehrhard, M. Pagani, and C. Tasson. 2018. Full Abstraction for Probabilistic PCF. *J. ACM* 65, 4 (2018), 23:1–23:44. https://doi.org/10.1145/3164540

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

Andrzej Filinski. 1994. Representing Monads. In *Conference Record of POPL'94: 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Portland, Oregon, USA, January 17-21, 1994*. ACM Press, 446–457.

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

S. Goldwasser and S. Micali. 1984. Probabilistic Encryption. *JCSS* 28, 2 (1984), 270–299. https://doi.org/10.1016/0022-0000(84)90070-9

N. D. Goodman, V. K. Mansinghka, D. M. Roy, K. Bonawitz, and J. B. Tenenbaum. 2008. Church: A Language for Generative Models. In *UAI*. 220–229.

J. Goubault-Larrecq. 2019. A Probabilistic and Non-Deterministic Call-by-Push-Value Language. In *Proc. of 34th LICS*. IEEE, 1–13. https://doi.org/10.1109/LICS.2019.8785809

C. Heunen, O. Kammar, S. Staton, and H. Yang. 2017. A Convenient Category for Higher-order Probability Theory. In *Proc. of 32nd LICS*. IEEE Computer Society, 1–12. https://doi.org/10.1109/LICS.2017.8005137

J. Hoffmann, K. Aehlig, and M. Hofmann. 2012. Resource Aware ML. In *Proc. of 24th CAV (LNCS, Vol. 7358)*. Springer, Heidelberg, DE, 781–786. https://doi.org/10.1007/978-3-642-31424-7_64

J. T. Gill III. 1974. Computational Complexity of Probabilistic Turing Machines. In *Proceedings of STOC 1974*. ACM, 91–95. https://doi.org/10.1145/800119.803889

S. Jost, K. Hammond, H.-W. Loidl, and M. Hofmann. 2010. Static Determination of Quantitative Resource Usage for Higher-order Programs. In *Proc. of 37th POPL*. ACM, 223–236. https://doi.org/10.1145/1706299.1706327

A. Jung and R. Tix. 1998. The Troublesome Probabilistic Powerdomain. *ENTCS* 13 (1998), 70–91. https://doi.org/10.1016/S1571-0661(05)80216-6

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

B. Lucien Kaminski, J.-P. Katoen, C. Matheja, and F. Olmedo. 2016. Weakest Precondition Reasoning for Expected Run-Times of Probabilistic Programs. In *Proc. of 25th ESOP (LNCS, Vol. 9632)*. Springer, 364–389. https://doi.org/10.1145/3208102

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

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

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

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

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

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

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

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

Rajeev Motwani and Prabhakar Raghavan. 1995. *Randomized Algorithms*. Cambridge University Press.

P. Narayanan, J. Carette, W. Romano, C. Shan, and R. Zinkov. 2016. Probabilistic Inference by Program Transformation in Hakaru (System Description). In *Proc. of 13rd FLOPS (LNCS, Vol. 9613)*. Springer, 62–79. https://doi.org/10.1007/978-3-319-29604-3_5

N. C. Ngo, Q. Carbonneaux, and J. Hoffmann. 2018. Bounded Expectations: Resource Analysis for Probabilistic Programs. In *Proc. of 39th PLDI*. ACM, 496–512. https://doi.org/10.1145/3192366.3192394

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

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

G.D. Plotkin. 1975. Call-by-name, Call-by-value and the $\lambda$-calculus. *TCS* 1, 2 (1975), 125–159. https://doi.org/10.1016/0304-3975(75)90017-1

M. O. Rabin. 1963. Probabilistic Automata. *Information and Control* 6, 3 (1963), 230–245.

I. Radicek, G. Barthe, M. Gaboardi, D. Garg, and F. Zuleger. 2018. Monadic Refinements for Relational Cost Analysis. *PACMPL* 2, POPL (2018), 36:1–36:32. https://doi.org/10.1145/3158124

N. Ramsey and A. Pfeffer. 2002. Stochastic Lambda Calculus and Monads of Probability Distributions. In *Proc. of 29$^{th}$ POPL*. ACM, 154–165. https://doi.org/10.1145/503272.503288

M. Rosendahl. 1989. Automatic Complexity Analysis. In *FPCA*. 144–156. https://doi.org/10.1145/99370.99381

N. Saheb-Djahromi. 1978. Probabilistic LCF. In *MFCS*. 442–451. https://doi.org/10.1007/3-540-08921-7_92

E. S. Santos. 1969. Probabilistic Turing Machines and Computability. *Proc. of AMS* 22, 3 (1969), 704–710.

A. Ścibior, O. Kammar, M. Vákár, S. Staton, H. Yang, Y. Cai, K. Ostermann, S. K. Moss, C. Heunen, and Z. Ghahramani. 2018. Denotational Validation of Higher-order Bayesian Inference. *PACMPL* 2, POPL (2018), 60:1–60:29. https://doi.org/10.1145/3158148

R. Sedgewick and P. Flajolet. 1996. *An Introduction to the Analysis of Algorithms*. Addison-Wesley-Longman.

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

J. Tassarotti and R. Harper. 2018. Verified Tail Bounds for Randomized Programs. In *Proc. of 9$^{th}$ ITP (LNCS, Vol. 10895)*. Springer, 560–578. https://doi.org/10.1007/978-3-030-01090-4_28

D. Tolpin, J.-W. van de Meent, H. Yang, and F. D. Wood. 2016. Design and Implementation of Probabilistic Programming Language Anglican. In *Proc. of 28$^{th}$ IFL*. ACM, 6:1–6:12. https://doi.org/10.1145/3064899.3064910

D. Wang, J. Hoffmann, and T. W. Reps. 2018. PMAF: an algebraic framework for static analysis of probabilistic programs. In *Proc. of 39$^{th}$ PLDI*. 513–528. https://doi.org/10.1145/3192366.3192408

D. Wang, D. M. Kahn, and J. Hoffmann. 2020. Raising Expectations: Automating Expected Cost Analysis with Types. *PACMPL* 4, ICFP (2020), 110:1–110:31. https://doi.org/10.1145/3408992

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

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

G. Winskel. 1993. *The Formal Semantics of Programming Languages*. MIT Press.