Compositional Expected Cost Analysis of Functional Probabilistic Programs*

Pedro H. Azevedo de Amorim
pedro.azevedo.de.amorim@cs.ox.ac.uk
University of Oxford
Oxford, UK

ABSTRACT

Reasoning about the cost of executing programs is one of the fundamental questions in computer science. In the context of programming with probabilities, however, the notion of cost is not so direct to define since the execution of a probabilistic program gives rise to a distribution over costs.

The expected cost is seen as an important metric for reasoning about probabilistic cost. In this work we define denotational semantics for reasoning about expected cost for a functional recursive language. We justify its validity by presenting case-studies ranging from randomized algorithms to stochastic processes and show how our semantics capture their intended expected cost.

1 INTRODUCTION

In a field where efficiency is almost as important as correctness, reasoning about the runtime cost of algorithms is of upmost importance. As such, algorithm designers and theoreticians have developed mathematical tools to aid them in reasoning about the cost of programs. A notable example is the use of recurrence relations to express the recursive cost structure of recursive programs. Computer scientists very early in their education learn, for instance, that merge sort's cost is given by the follwing recurrence relation

$$T(0) = 0$$
$$T(n) = 2T(n/2) + n$$

Which gives $O(n \log n)$ complexity. How can we formalize this claim? For simple pseudocode we can use our intuitions to correctly reason about its cost. But, when the language is more expressive either because it has computational effects, higher-order functions or inductive types, it becomes less clear how to assign costs to programs.

These issues have been explored by Danner et al [7–9], where the authors, in a span of several papers, have developed semantic techniques that lay on firm ground the cost analysis of functional programs with inductive types. This line of work has culminated in [20], where the authors use a Call-By-Push-Value (CBPV) metalanguage and the writer monad to define a recurrence extraction mechanism for a functional language with recursion and list datatypes.

Though their results are impressive, the only effect their technique can handle is recursion. This is limiting because many problems can be solved more efficiently by, for instance, having access to probabilistic primitives, so-called randomized algorithms [26].

A notable example is a probabilistic variant of the quicksort algorithm. In the worst case, i.e. when the list is in reverse order, quicksort requires $O(n^2)$ comparisons, a far-cry from its promised complexity $O(n \log(n))$. By being able to uniformly sample elements from the input list you can average out this worst case and recover an average cost of $O(n \log(n))$.

Due to the applicability of probabilistic algorithms, a lot of theoretical work has gone in developing the mathematical definitions and tools used for reasoning about them. In particular, for these algorithms, the familiar notion of cost is replaced by expected cost. Instead of considering the cost of a single execution, you average it over every probabilistic sampling done throughout the execution of the program. Familiar tools from deterministic cost analysis such as recurrence relations, have also been successfully developed in the probabilistic setting [17].

Unfortunately, much like in the deterministic case before the work of Danner et al, there has not been a systematic semantic study of how to bring these cost analysis concepts to expressive functional languages.

Our Work: Compositional Methods for Probabilistic Cost. In this work we give semantic foundations for reasoning about expected cost in the context of recursive probabilistic functional programs with lists. We start by defining **cert**: a CBPV metalanguage with operations for sampling from uniform distributions and for incrementing the cost of programs. This metalanguage is expressive enough to represent the cost structure of recursive probabilistic algorithms and stochastic processes.

Besides equipping our metalanguage with an equational theory, we make its semantics concrete by providing two denotational semantics for reasoning about the cost of probabilistic programs. The first one, which we call the *cost semantics*, uses the familiar writer monad transformer to combine the cost monad with a subprobability monad. The second semantics, which we call the *expected cost semantics*, encapsulates the compositional structure of the expected cost as a monad, allowing us to give a denotational semantics that keeps track of the expected cost of programs.

Then, in order to justify the mutual validity of these distinct semantics, we show that the expected cost semantics is a sound approximation to the cost semantics and to the equational theory. In the absence of recursion, we show that this approximation is an equality while it is an inequality in the presence of unbounded recursion. In order to achieve this soundness result we had to define a novel, and more expressive, logical relations argument for reasoning about the effect simulation problem [19] in a CBPV setting.

As applications, we showcase the capabilities of our semantics by using it to reason about the expected cost of the probabilistic algorithms quicksort and quickselect and stochastic processes such as a symmetric random walk. As a guiding example, throughout the paper we will use geometric distributions to illustrate different aspects of **cert** and its semantics.

$$\overline{\tau} \coloneqq F\tau \mid \tau \to \overline{\tau}$$
$$\tau \coloneqq U\overline{\tau} \mid 1 \mid \mathbb{N} \mid \tau \times \tau$$

$$t, u \coloneqq \lambda x.\ t \mid t\ V \mid \mathsf{ifZero}\ V\ \mathsf{then}\ t\ \mathsf{else}\ u \mid \mathsf{force}\ V \mid (x \leftarrow t); u$$
$$\mid \mathsf{produce}\ V \mid \mathsf{let}\ x\ \mathsf{be}\ V\ \mathsf{in}\ t \mid \mathsf{succ}\ t \mid \mathsf{pred}\ t$$
$$\mid \mathsf{let}\ (x, y) = V\ \mathsf{in}\ t$$
$$V \coloneqq x \mid () \mid n \in \mathbb{N} \mid \mathsf{thunk}\ t \mid (V_1, V_2)$$

Figure 1: Types and Terms of CBPV

Our approach contrasts with other work done on expected cost analysis [1–3, 5, 16, 21, 22, 31, 33] where the language analyzed was either imperative or first-order, or the cost structure was given by non-compositional methods. In spirit, the closest to what we have done is [20], which does denotational cost analysis in a deterministic recursive setting.

Our contributions. In summary, our contributions are the following:

- We define **cert**, a CBPV variant with primitives for increasing cost (charge_c) and for sampling from uniform distributions (rand) (§2)
- We define an equational theory for reasoning about the expected cost of programs (§2.3)
- We define a cost semantics, an expected cost semantics and show that the latter is sound with respect to the former by a novel logical relations argyument (§3.2)
- We prove that both semantics are sound with respect to the equational theory (§3.4.2)
- We justify the validity of the expected cost semantics by reasoning about two stochastic processes and two randomized algorithms (§4)

2 cert : A PROBABILISTIC COST-AWARE METALANGUAGE

Call-By-Push-Value is a metalanguage that provides fine-grained control over the execution of programs. This is achieved by distinguishing, at the type level, between values and computations, and by adding syntax for suspending and resuming the execution of programs [23]. By appropriately using these forcing and suspension primitives, it is possible to embed both Call-By-Name (CBN) and Call-By-Value (CBV) calculi into CBPV. Since we are interested in a uniform treatment of expected cost without restricting ourselves to one particular evaluation strategy, we use CBPV as our base language.

Figure 1 depicts the CBPV syntax, note that the base types 1 and $\mathbb{N}$ are value types and arrow types are computation types that receive as input a value type and output a computation type. At the center of the CBPV formalism are the type constructors F and U which allows types to move between value types and computation types. The constructor F plays a similar role to the monadic type constructor T from the monadic λ-calculus [25], while U is used to represent suspended — or thunked — computations.

This two-level approach also manisfests itself at the type judgement level, where the judgement $\Gamma \vdash_v V : \tau$ is only defined for value types while $\Gamma \vdash_c t : \overline{\tau}$ is defined for computation types, as shown in Figure 2. Both contexts only bind values, which justifies the arrow type having a value type in its domain, so that lambda abstractions will only introduce values to the context. The if-then-else operation checks if the value V is 0, in which case it returns the first branch, and otherwise it returns the second branch. The operations pred and succ are the predecessor and successor functions, respectively. The product introduction rule pairs two values while its elimination rule unpairs a product and uses them in a computation. Lambda abstraction binds a new value to the context while application applies a function to a value.

The less familiar rules are those for the type constructors F and U. The introduction rule for computations is produce V, which is the computation that does not incur any effect and just outputs the value V, while the introduction rule for U, thunk t suspends the computation t. Its elimination rule, force V resumes the suspended computation V. The last rule, $x \leftarrow t; u$ is what makes it possible to chain effectful computations together, since it receives a computation of type $F\tau$ as input, runs it and binds the result to the continuation u, which eventually will output a computation of type $\overline{\tau}$. This is a generalization of the monadic let rule where the output type does not have to be of type $F\tau$.

Though the syntax differs a bit from the monadic semantics of effects, every strong monad over a Cartesian closed category can interpret the CBPV calculus, as we describe in Appendix A.

Though this language is effective as a core calculus, by itself it cannot do much, since it has no "native" effect operations, meaning that there are no programs with non-trivial side-effects. In this section we will extend it CBPV so that it can program with three different effects: cost, probability and unbounded recursion. We call this extension **cert**, for **c**alculus for **e**xpected **r**un **t**ime, and we conclude the section by presenting its equational theory.

2.1 Cost and Probabilistic Effects

As it is common in denotational approaches to cost semantics, it is assumed that there is a cost monoid $\mathbb{C}$ — usually interpreted by $\mathbb{N}$ and addition — which acts on programs by operations charge_c that increases the current cost of the computation by c units, for every $c : \mathbb{C}$. The value types are extended with a type $\mathbb{C}$ and constants $\cdot \vdash_v c : \mathbb{C}$. Furthermore, since we also want to program with probabilities and unbounded recursion, we extend the language with a sampling primitive, as well as recursive definitions:

$$\frac{\Gamma \vdash_v V : \mathbb{C}}{\Gamma \vdash_c \mathsf{charge}_V : F1} \qquad \frac{\Gamma \vdash_v V : \mathbb{N}}{\Gamma \vdash_c \mathsf{rand}\, V : F\mathbb{N}} \qquad \frac{\Gamma, x : U\overline{\tau} \vdash_c t : \overline{\tau}}{\Gamma \vdash_c \mathsf{fix}\, x.\, t : \overline{\tau}}$$

The operation rand V uniformly samples a natural number in the interval $[0, V]$ and fix is the familiar fixed-point operator used for defining recursive programs. In interest of reducing visual polution, $\mathsf{charge}_V; t$ desugars to $(x \leftarrow \mathsf{charge}_V); t$, when x is not used in the body of t.

Example 2.1 (Geometric distribution)**.** With these primitives we can already program non-trivial distributions. For instance, the geometric distribution can be expressed as the program

$$\cdot \vdash_c \mathsf{fix}\, x.\, (\mathsf{produce}\ 0) \oplus ((y \leftarrow \mathsf{force}\ x); \mathsf{produce}\ (1 + y)) : F\mathbb{N},$$

$$\frac{}{\Gamma_1, x : \tau, \Gamma_2 \vdash_v x : \tau} \qquad \frac{n \in \mathbb{N}}{\Gamma \vdash_v n : \mathbb{N}} \qquad \frac{}{\Gamma \vdash_v () : 1} \qquad \frac{\Gamma \vdash_v V : \mathbb{N} \qquad \Gamma \vdash_c t : \overline{\tau} \qquad \Gamma \vdash_c u : \overline{\tau}}{\Gamma \vdash^c \mathsf{ifZero}\, V \,\mathsf{then}\, t \,\mathsf{else}\, u : \overline{\tau}} \qquad \frac{\Gamma \vdash_v V_1 : \tau_1 \qquad \Gamma \vdash_v V_2 : \tau_2}{\Gamma \vdash_v (V_1, V_2) : \tau_1 \times \tau_2}$$

$$\frac{\Gamma \vdash_c t : F\mathbb{N}}{\Gamma \vdash_c \mathsf{pred}\, t : F\mathbb{N}} \qquad \frac{\Gamma \vdash_c t : F\mathbb{N}}{\Gamma \vdash_c \mathsf{succ}\, t : F\mathbb{N}} \qquad \frac{\Gamma, x : \tau \vdash_c t : \overline{\tau}}{\Gamma \vdash_c \lambda x\,.\, t : \tau \to \overline{\tau}} \qquad \frac{\Gamma \vdash_v V : \tau \qquad \Gamma \vdash_c t : \tau \to \overline{\tau}}{\Gamma \vdash_c t\ V : \overline{\tau}}$$

$$\frac{\Gamma \vdash_v V : \tau}{\Gamma \vdash_c \mathsf{produce}\, V : F\tau} \qquad \frac{\Gamma \vdash_c t : \overline{\tau}}{\Gamma \vdash_v \mathsf{thunk}\, t : U\overline{\tau}} \qquad \frac{\Gamma \vdash_v V : U\overline{\tau}}{\Gamma \vdash_c \mathsf{force}\, V : \overline{\tau}} \qquad \frac{\Gamma \vdash_c t : F\tau' \qquad \Gamma, x : \tau' \vdash_c u : \overline{\tau}}{\Gamma \vdash_c (x \leftarrow t); u : \overline{\tau}}$$

$$\frac{\Gamma \vdash_v V : \tau_1 \times \tau_2 \qquad \Gamma, x : \tau_1, y : \tau_2 \vdash_c t : \overline{\tau}}{\Gamma \vdash_c \mathsf{let}\, (x, y) = V \,\mathsf{in}\, t : \overline{\tau}}$$

Figure 2: CBPV typing rules

where $t \oplus u$ is syntactic sugar for $(x \leftarrow \mathsf{rand}\ 1); \mathsf{ifZero}\, x \,\mathsf{then}\, t \,\mathsf{else}\, u$, i.e. it uniformly chooses between the left and right branch. Operationally, the program flips a fair coin, if the output is 0, it outputs 0, otherwise it recurses on x, binds the value to y and outputs $1 + y$.

By having fine-grained control over which operations have a cost, it is possible to decide which costs are reasoned about. For instance, if we want to keep track of how many coins were tossed when running the geometric distribution, we can modify it as such

$$\mathsf{fix}\, x.\, \mathsf{charge}_1; (\mathsf{produce}\ 0) \oplus (y \leftarrow \mathsf{force}\ x; \mathsf{produce}\ (1 + y)) : F\mathbb{N}$$

Example 2.2 (Deterministic Programs)**.** The charge operation can also be used to keep track of the number of recursive calls in your program. For instance, a recursive program that computes the factorial function can be instrumented to count the number of recursive calls as follows:

$$\mathsf{fix}\, f\,.\, \lambda n\,.\ \mathsf{ifZero}\, n \,\mathsf{then}\, (\mathsf{produce}\ 0) \,\mathsf{else}\, (\mathsf{charge}_1; n * f(n-1))$$

Whenever the if-guard is false, the cost is incremented by 1 and the function is recursively called.

2.2 Lists

Frequently, cost analysis are defined for algorithms defined over inductive data types, such as lists. As such, we will also extend our language with lists over value types.

$$\tau ::= \cdots \mid \mathsf{list}(\tau)$$

$$\frac{}{\Gamma \vdash^v \mathsf{nil} : \mathsf{list}(\tau)} \qquad \frac{\Gamma \vdash^v V_1 : \tau \qquad \Gamma \vdash^v V_2 : \mathsf{list}(\tau)}{\Gamma \vdash^v \mathsf{cons}\, V_1\, V_2 : \mathsf{list}(\tau)}$$

$$\frac{\Gamma \vdash_v V : \mathsf{list}(\tau) \qquad \Gamma \vdash^c t : \overline{\tau} \qquad \Gamma, x : \tau, xs : \mathsf{list}(\tau) \vdash^c u : \overline{\tau}}{\Gamma \vdash^c \mathsf{case}\, V \,\mathsf{of}\, \mathsf{nil} \Rightarrow t \mid (x, xs) \Rightarrow u : \overline{\tau}}$$

The primitive nil is the empty list, cons appends a value to the front of a list and case is for pattern-matching on lists and, in the presence of fix , can be used for defining non-structurally recursive functions over lists.

Example 2.3. The function that computes the length of a list can be defined as

$$\begin{array}{l}
\mu f : \mathsf{list}(\tau) \to F\mathbb{N}.\, \lambda l : \mathsf{list}(\tau). \\
\mathsf{case}\, l\, \mathsf{of} \\
\mid \mathsf{nil} \Rightarrow \\
\quad \mathsf{produce}\ 0 \\
\mid (\mathsf{hd}, \mathsf{tl}) \Rightarrow \\
\quad n \leftarrow (\mathsf{force}\ f)\ tl \\
\quad \mathsf{produce}\ (1 + n)
\end{array}$$

Example 2.4. A binary version of the familiar filter function that outputs two lists, one for the true elements and the other for the false elements can be written as

$$\begin{array}{l}
\mu f : \mathsf{list}(\tau) \to F(\mathsf{list}(\tau) \times \mathsf{list}(\tau)). \\
\lambda l : \mathsf{list}(\tau)\,. \\
\lambda p : \tau \to F\mathbb{N}. \\
\mathsf{case}\, l\, \mathsf{of} \\
\mid \mathsf{nil} \Rightarrow \\
\quad \mathsf{produce}\ (\mathsf{nil}, \mathsf{nil}) \\
\mid (\mathsf{hd}, \mathsf{tl}) \Rightarrow \\
\quad n \leftarrow p\ hd \\
\quad (l_1, l_2) \leftarrow (\mathsf{force}\ f)\ p\ tl \\
\quad \mathsf{if}\, n\, \mathsf{then} \\
\qquad \mathsf{produce}\ (\mathsf{cons}\, hd\, l_1, l_2) \\
\quad \mathsf{else} \\
\qquad \mathsf{produce}\ (l_1, \mathsf{cons}\, hd\, l_2)
\end{array}$$

Since we have adopted a $\mathbb{N}$-valued if-statement, the predicate p above outputs a natural number.

Example 2.5. We can use the functions above and write a randomized version of the quicksort algorithm that counts the number of

$$x \oplus_0 y = x$$
$$x \oplus_p y = y \oplus_{1-p} x$$
$$x \oplus_p x = x$$
$$x \oplus_p (y \oplus_q z) = (x \oplus_{pq} y) \oplus_{\frac{p(1-q)}{1-pq}} z$$

Figure 3: Barycentric Algebra Axioms

comparisons done as follows:

$\mu f : \mathsf{list}(\mathbb{N}) \to F(\mathsf{list}(\mathbb{N}))$.
$\lambda l : \mathsf{list}(\mathbb{N})$.
case l of
| nil $\Rightarrow$
  produce nil
| (hd, tl) $\Rightarrow$
  $len \leftarrow length\, l$
  $r \leftarrow \mathsf{rand}\ len$
  $pivot \leftarrow l[r]$
  $(l_1, l_2) \leftarrow biFilter\ (\lambda n.\ \mathsf{charge}_1; n \leq pivot)\ l$
  $less \leftarrow \mathsf{force}\ f\ l_1$
  $greater \leftarrow \mathsf{force}\ f\ l_2$
  produce $(less +\!\!+ pivot :: greater)$

In the program above we are accessing the r-th element of a list l using the familiar syntax $l[r]$. The algorithm is very similar to the non-randomized quicksort with the exception of uniformly choosing an element from the input list as the pivot.

We conclude this section by mentioning that there are many other sensible extensions, such as recursive and sum types. For our purposes, they are not necessary and so, in order to keep the language simple, we omit them. That being said, from a semantic point of view, these extensions are well-understood and straightforward to be accommodated by the denotational semantics we present in Section 3.

2.3 Equational Theory

We want to define a syntactic sound approximation to the expected cost of programs. We do this by extending the usual equational theory of CBPV with rules for the monoid structure of the charge operation. We present some of the equational theory in Figure 4, with the other rules shown in Appendix A. The first two rules are the familiar β and η rules for the arrow type, the next two are the monoid equations for the charge operation, the next one says that forcing a thunked computation is the same thing as running the computation, the next two explain how if-statements interact with natural numbers and the last one is the fixed point equation that unfolds one recursive call of the recursive computation t.

An alternative axiomatization of probabilistic effects is given by barycentric algebras, which are sets equipped with a collection of binary operations $\{\oplus_p\}_{p\in[0,1]}$ satisfying the equations depicted in Figure 3, where the intuition behind them being that the binary operation $\oplus_p$ is a convex combination with weight p. From an operational point of view, these operations capture a weighted binary choice between two computations.

Under this axiomatization and by making the cost monoid "continuous", it is also reasonable to add the equation:

$$(\mathsf{charge}_{c_1}; t) \oplus_p (\mathsf{charge}_{c_2}; u) = \mathsf{charge}_{pc_2+(1-p)c_1}; (t \oplus_p u)$$

This equation can be quite useful for simplifying branching processes. For instance, for a modified geometric distribution where you only keep track of the number of non-zero coin flips we get the following rewrites

$$(\mathsf{produce}\ 0) \oplus (\mathsf{charge}_1; y \leftarrow \mathsf{force}\ x; \mathsf{produce}\ (1+y)) =$$
$$(\mathsf{charge}_0; \mathsf{produce}\ 0) \oplus (\mathsf{charge}_1; y \leftarrow \mathsf{force}\ x; \mathsf{produce}\ (1+y)) =$$
$$\mathsf{charge}_{0.5}; (\mathsf{produce}\ 0) \oplus (y \leftarrow \mathsf{force}\ x; \mathsf{produce}\ (1+y))$$

Which allows us to get quantitative bounds on how this variant relates to the original geometric distribution. Indeed, it can be calculated that the original distribution has expected value 2 while this one has expected cost 1.

In order to keep **cert** as simple as possible, only rand will be a part of its surface syntax, but in its full equational definition in Appendix A, we present it with the barycentric algebra operations and equations. Though this seems like a simple equatoinal theory, as we will see in Section 3.4, the interaction of cost, probability and recursion has unexpected consequences.

3 DENOTATIONAL SEMANTICS

This section presents two concrete denotational semantics to our language. Since **cert** contains higher-order functions, unbounded recursion and probabilistic primitives, we are somewhat limited in our choice of semantic domain. Work by Vakar et al. [32] on probabilistic semantics has defined the category of ω-quasi Borel spaces $\omega\mathbf{Qbs}$ which satisfies all of our requirements and we choose it as our base category. This category was first defined as a variant of the quasi Borel space category [12] that preserves many of its good categorical properties, such as being Cartesian closed and having commutative probability monads, with the exception that it can interpret unbounded recursion, which is of significant importance for programming language semantics.

The category $\omega\mathbf{Qbs}$ admits commutative probabilistic powerdomains that can interpret probabilistic primitives such as rand. In particular, $\omega\mathbf{Qbs}$ admits a probabilistic powerdomain of subprobability distributions $P_{\leq 1}$ and, by using the writer monad transformer $P_{\leq 1}(\mathbb{C}\times -)$, it can also accommodate cost operations, as we explain in Section 3.1. With this new monad, it is possible to use the CBPV algebraic semantics to reason about the expected cost of programs by computing the marginal distribution over $\mathbb{C}$ and computing its expected value.

Unfortunately, this approach is non-compositional. In order to compute the expected cost of a program of type $F\tau$, we must first compute its (compositional) semantics which can then be used to obtain a distribution over the cost and then apply the expectation formula to it. We work around this issue by constructing a novel expected cost monad in Section 3.3 that makes the expected cost a part of the semantics and, as such, it is compositionally computed.

$$\frac{\Gamma, x : \tau \vdash_c t : \overline{\tau} \qquad \Gamma \vdash_v V : \tau}{\Gamma \vdash t\{V/x\} = (\lambda x.\ t)\ V : \overline{\tau}} \qquad \frac{\Gamma \vdash_c t : \tau \to \overline{\tau}}{\Gamma \vdash_c (\lambda x.\ t\ x) = t : \tau \to \overline{\tau}}$$

$$\frac{\Gamma \vdash_c t : \overline{\tau}}{\Gamma \vdash (\mathsf{charge}_0; t) = t : \overline{\tau}} \qquad \frac{}{\Gamma \vdash \mathsf{charge}_c; \mathsf{charge}_d = \mathsf{charge}_{c+d} : F1} \qquad \frac{\Gamma \vdash_c t : \overline{\tau}}{\Gamma \vdash \mathsf{force}\ (\mathsf{thunk}\ (t)) = t : \overline{\tau}}$$

$$\frac{\Gamma \vdash_c t : \overline{\tau} \quad \Gamma \vdash_c u : \overline{\tau}}{\Gamma \vdash \mathsf{ifZero}\ 0\ \mathsf{then}\ t\ \mathsf{else}\ u = t : \overline{\tau}} \qquad \frac{\Gamma \vdash_c t : \overline{\tau} \quad \Gamma \vdash_c u : \overline{\tau}}{\Gamma \vdash \mathsf{ifZero}\ (n+1)\ \mathsf{then}\ t\ \mathsf{else}\ u = u : \overline{\tau}} \qquad \frac{\Gamma, x : U\overline{\tau} \vdash_c t : \overline{\tau}}{\Gamma \vdash \mathsf{fix}\, x.\, t = t\{x/\mathsf{thunk}\ (\mathsf{fix}\, x.\, t)\} : \overline{\tau}}$$

Figure 4: Equational Theory (Selected Rules)

We start this section by going over the important constructions for ω**Qbs**, we define the cost semantics, followed by the expected cost semantics and, in order to justify their soundness, we show that the expected cost semantics is a sound approximation of the cost semantics by a logical relations argument. We conclude by showing that they are also sound with respect to the equational theory.

3.1 ω-quasi Borel spaces

A significant limitation, for the purposes of semantics, of the traditional category of measurable spaces and measurable functions **Meas** is that it is not Cartesian closed, meaning that it cannot interpret higher-order functions.

This led to an arduous search for a Cartesian closed category that conservatively extends **Meas**, resulting in the definition of quasi Borel spaces [12]. Though this category is very interesting in its own right, it is not fully adequate as a semantic basis for probabilistic programming languages, since it cannot handle unbounded recursion. This limitation has led to the discovery of ω-quasi Borel spaces, which we now define.

Definition 3.1 ([32])**.** An ω-quasi Borel space is a triple $(X, \leq, M_X)$ such that, $(X, \leq)$ is a ω-complete partial order (ωCPO), i.e. it is a partial order closed under suprema of ascending sequences, and $M_X \subseteq \mathbb{R} \to X$ is the set of *random elements* with the following properties:

- All constant functions are in M_X
- If $f : \mathbb{R} \to \mathbb{R}$ is a measurable function and $p \in M_X$, then $p \circ f \in M_X$
- If $\mathbb{R} = \bigcup_{n\in\mathbb{N}} U_n$, where for every n, U_n are pairwise-disjoint and Borel-measurable, and $\alpha_n \in M_X$ then the function $\alpha(x) = \alpha_n(x)$ if, and only if, $x \in U_n$ is also an element of M_X.
- For every ascending chain $\{f_n\}_n \subseteq M_X$, i.e. for every $x \in \mathbb{R}$, $f_n(x) \leq f_{n+1}(x)$, the pointwise supremum $\bigsqcup_n f_n$ is in M_X.

Note that, in the definition above, ωCPOs do not assume the existence of a least element, e.g. for every set X, the discrete poset $(X, =)$ is an ωCPO.

Definition 3.2. A measurable function between ω-quasi Borel spaces is a Scott continuous function $f : X \to Y$ — i.e. preserves suprema of ascending chains — such that for every $p \in M_X$, $f \circ p \in M_Y$.

Definition 3.3. The category ω**Qbs** has ω-quasi Borel spaces as objects and measurable functions as morphisms.

Theorem 3.4 ([32])**.** *The category* ω**Qbs** *is Cartesian closed.*

Furthermore, there is a full and faithful functor **Meas** $\to \omega$**Qbs**. More concretely, if you interpret a program that has as inputs and output measurable spaces, its denotation in ω**Qbs** will be a measurable function, even if the program uses higher-order functions, and any measurable function could potentially be the denotation of the program.

Inductive types. As shown in previous work [32], ω**Qbs** can also soundly accommodate full recursive types. In particular, it can give semantics to lists over A by solving the domain equation $\mathsf{list}(A) \cong 1 + A \times \mathsf{list}(A)$.

It is convenient that in ω**Qbs** the set of lists over A with appropriate random elements and partial order is a solution to the domain equation and it is the smallest one, i.e. it is an initial algebra. This means that when reasoning about lists expressed in **cert**, you may assume that they are just the set of lists over sets.

Probability and Partiality Monads. It is possible to construct probabilistic powerdomains in ω**Qbs**, making it possible to use this category as a semantic basis for languages with probabilistic primitives. Furthermore, the ωCPO structure can also be used to construct a partiality monad, making it possible to give semantics to programs with unbounded recursion

Definition 3.5. Let $\mathbf{C}$ be a category, a monad is a triple $(T, \eta^T, (-)^\#_T)$ where $T : \mathbf{C} \to \mathbf{C}$ is an endofunctor, $\eta^T : id \to T$ and $(-)^\#_T : \mathbf{C}(-, T=) \to \mathbf{C}(T-, T=)$ are natural transformations such that

$$f^\#_T \circ \eta = f$$
$$\eta^\#_T = id$$
$$(f^\#_T \circ g)^\#_T = f^\#_T \circ g^\#_T$$

The monad is said to be *strong* if there is a natural transformation $st_{A,B} : A \times TB \to T(A \times B)$ making certain diagrams commute [25]. When it is clear from the context, we will simply write η and $(-)^\#$, without the sub and superscript, respectively.

Lemma 3.6 ([32])**.** *The category* ω**Qbs** *admits strong commutative monads P and $P_{\leq 1}$ of probability and sub-probability distributions, respectively.*

At the categorical level, $P_{\leq 1}$ is defined as a submonad of the continuation monad $(- \to [0, \infty]) \to [0, \infty]$ and its monad structure

is similar to the one from probability monads in **Meas**, i.e. the unit at a point $a : A$ is given by the point mass distribution δ_a and $f^\#(\mu)$ is given by integrating f over the input distribution μ.

Furthermore, by construction, $\omega\mathbf{Qbs}$ admits a morphism $\int_A : (P_{\leq 1}A)\times(A \to \{0,1\}) \to [0,1]$ that maps a subprobability distribution and a "measurable set" of A into its measure. For example, if A is a measurable space, for every measurable set $X : A \to \{0,1\}$, and for every subprobability distribution $\mu : P_{\leq 1}A$, the map $(\mu, X) \mapsto \mu(X)$ is an $\omega\mathbf{Qbs}$ morphism and is equal to $\int_A$.

As we have mentioned above, it is also possible to define a lifting monad in $\omega\mathbf{Qbs}$ that adds a least element $\bot$ to a space, making them *pointed* ωCPOs. This monad, combined with the ωCPO strucure, is used to guarantee the existence of fixed points of endomorphisms between pointed ωCPOs.

Definition 3.7 ([32])**.** The lifting monad in $\omega\mathbf{Qbs}$ $X_\bot$ adds a fresh element $\bot$ to X, makes it the least element and the random elements $M_{X_\bot}$ are the functions $f : \mathbb{R} \to X_\bot$ such that there is a Borel measurable set $\mathcal{B}$ and a map $\alpha : \mathbb{R} \to X$ in M_X such that $f(x) = \alpha(x)$ for $x \in \mathcal{B}$ and $\bot$ otherwise.

The machinery we have defined so far is expressive enough to interpret **cert**, with exception of its cost operations. In non-effectful languages, the writer monad $(\mathbb{C}\times -)$ can be used to give semantics to cost operations such as charge_c.

Definition 3.8. If $(\mathbb{C}, 0, +)$ is a monoid, then $\mathbb{C}\times -$ is a monad — the *writer* monad — where the unit at a point a is $(0, a)$ and given a morphism $f : A \to \mathbb{C}\times B$, $f^\#(c, a) = (c + (\pi_1 \circ f)(a), (\pi_2 \circ f)(a))$.

What follows is how to combine the non-probabilistic cost monad $(\mathbb{C}\times -)$ with $P_{\leq 1}$ in order to define a probabilistic cost semantics.

3.2 A probabilistic cost semantics

As opposed to the deterministic case, the cost of a probabilistic computation is not a single value; instead, it is a distribution over the costs. For instance, consider the program:

$$\cdot \vdash_c (\mathsf{charge}_1; \mathsf{produce}\ 0) \oplus (\mathsf{produce}\ 2) : F\mathbb{N}$$

it either returns 2 without costing anything, or it returns 0 with a cost of 1. Denotationally, this program should be the distribution $\frac{1}{2}(\delta_{(1,0)} + \delta_{(0,2)})$. With equal probability, the program will either cost 1 and output 0 or cost 0 and output 2.

In the deterministic case, it is possible to encode the cost at the semantic-level by using the *writer* monad $\mathbb{C}\times -$. For probabilistic cost-analysis we can use the writer monad transformer.

Lemma 3.9. *If $T : \mathbf{C} \to \mathbf{C}$ is a strong monad then $T(\mathbb{C}\times -)$ is a strong monad.*

Proof. The strength of a monad is a natural transformation $A\times TB \to T(A\times B)$. When instantiating A to be $\mathbb{C}$, we can conclude that there is a distributive law between the writer monad and T, which allows us to conclude that $T(\mathbb{C}\times -)$ is a monad. Its strength is defined as $st^T; T(st^{\mathbb{C}\times -}) : A \times T(\mathbb{C}\times B) \to T(A\times(\mathbb{C}\times B)) \to T(\mathbb{C}\times(A\times B))$. □

When instantiating T to be the subprobability monad $P_{\leq 1}$, we get a monad for probabilistic cost, which justifies the denotation of the

$$[\![\mathsf{charge}_c]\!]_{CS} = \delta_{(c,())} \qquad [\![\mathsf{rand}\ V]\!]_{CS} = \sum_{i=0}^{[\![V]\!]_{CS}} \frac{1}{[\![V]\!]_{CS}} \delta_{(0,i)}$$

$$[\![\mathsf{fix}\, x.\, t]\!]_{CS} = \bigsqcup_n [\![t]\!]^n_{CS}\,(\bot)$$

Figure 5: Cost semantics of operations

program $(\mathsf{charge}_1; \mathsf{produce}\ 0) \oplus (\mathsf{produce}\ 2)$ being a distribution of a pair of a cost and natural number.

By using the monadic semantics of CBPV, we get a cost-aware probabilistic semantics, where most of its definitions follow the standard monadic CBPV semantics shown in Appendix A — denoted as $[\![\cdot]\!]^v_{CS}$ for values and $[\![\cdot]\!]^c_{CS}$ for computations. The noteworthy interpretations are for the one for the cost monoid, which is interpreted as $\mathbb{N}$, and for the effectful operations, whose semantics are depicted in Figure 5.

With this semantics we now define the expected cost of a distribution:

Definition 3.10. Let $\mu : P_{\leq 1}[0,\infty]$, its expected value is $\mathbb{E}(\mu) = \int_{[0,\infty]} x\, \mathrm{d}\mu$.

In the definition above we have chosen the most general domain for $\mathbb{E}$, but for every subset $X \subseteq [0,\infty]$ the expected distribution formula can be restricted to distributions over X.

Example 3.11 (Geometric Distribution)**.** This semantics makes it possible to reason about the geometric distribution defined as the program [1] $\cdot \vdash \mathsf{fix}\, x.0 \oplus (1 + x) : F\mathbb{N}$. It is possible to show that this program indeed denotes the geometric distribution by unfolding the semantics and obtaining the fixed point equation $\mu = \frac{1}{2}(\delta_0 + P_{\leq 1}(\lambda x.1 + x)(\mu))$, for which the geometric distribution is a solution.

Since we are interested in reasoning about the cost of programs, it is possible to reason about the expected amount of coins flipped during its execution by adding the charge_1 operation as follows $\mathsf{fix}\, x.\, \mathsf{charge}_1; (0 \oplus (1+x)) : F\mathbb{N}$, i.e. whenever a new coin is flipped, as modeled by the $\oplus$ operation, the cost increases by one. By construction, the cost distribution will also follow a geometric distribution.

If we want to compute the actual expected value we must compute $\sum_{n:\mathbb{N}} \frac{n}{2^n}$. This particular infinite sum can be calculated by using a standard trick. As we will in the next section, it is possible to encode this trick in the semantics itself, which simplifies significantly computing the expected value.

Expected cost and non-termination. When designing metalanguages for reasoning about cost of programs, it is expected that the cost of a non-terminating computation is infinite. Therefore, since subprobability distributions of mass less than 1 model possibly non-terminating computation, their cost should be ∞, This can be achieved this by modifying the expected cost function to

[1] For the sake of simplicity we have elided the some of the bureacracy of CBPV, such as produce and force .

$\infty \cdot (1 - \mu(\mathbb{C})) + \int_{\mathbb{C}} x \, \mathrm{d}\mu$ and defining $\infty \cdot 0 = 0$, meaning that every computation with non-zero chance of termination would, by definition, have infinite expected cost.

There are a few problems with this solution. Since the soundness proof of Section 3.4 requires the expected cost to be a morphism $P_{\leq 1}(\mathbb{C}) \to [0, \infty]$ in $\omega\mathbf{Qbs}$, we have to equip the extended positive real line with a set of random elements and a partial order. Traditionally, the bottom element of $[0, \infty]$ would be interpreted as ∞, in line with the slogan that "no information" equals infinite cost — this is the approach taken by [20]. However, as we will see in Section 4, when we have recursively defined programs of type $F\tau$, their expected costs are given by the supremum of increasing sequences of real numbers, starting from 0. Suggesting that 0 needs to be the bottom element while ∞ is the top element.

This alternative is also problematic, unfortunately, since it is not Scott-continuous. As an example consider the geometric distribution, where its cost distribution is given by the supremum of the sequence $\{\sum_i^n 2^{-i}\delta_i\}_n$. Therefore, $\mathbb{E}(\sum_i^n 2^{-i}\delta_i) = \infty$, for every $n \in \mathbb{N}$, resulting in

$$\sup_n \mathbb{E}\left(\sum_{i=1}^{n} 2^{-i}\delta_i\right) = \infty \neq \mathbb{E}\left(\sup_n \sum_{i=1}^{n} 2^{-i}\delta_i\right) = \mathbb{E}(\mathsf{geom}) = 2,$$

Where geom is the geometric distribution. Other approaches have not dealt with these difficulties because, unlike other effects, probabilities present genuinely interesting recursive programs of type $F\mathbb{N}$, as illustrated by the geometric distribution itself. In contrast, in [20], by a monotonicity argument, every recursive program of type $F\mathbb{N}$ either diverges or outputs a constant. Therefore, assigning infinite cost to these programs, though not very realistic from a modeling point of view, is not too damaging to the semantics.

With this in mind we argue that Definition 3.10 is a sensible choice. In Section 3.4, we go over the consequences of this definition insofar as the cost semantics interacts with the expected cost semantics.

3.3 A semantics for expected-cost

The semantics defined in the previous section can be used to compositionally compute the cost distribution for computations of type $F\tau$. In turn, by using Definition 3.10, its expected cost can be calculated.

Though the approach above is sound, it is not compositional. Indeed, after computing the distribution, we must compute the expected cost of an arbitrarily complex distribution. Instead, if we could compute the expected cost as we compute the semantics, we would avoid this lack of compositionality altogether.

We achieve this by making the expected value a part of the semantics by constructing a new monad that keeps track of the average cost. More concretely, computations of type $F\tau$ will denote a pair of an extended positive real number and a subprobability distribution. Intuitively, a computation will output its expected cost and the output subprobability distribution. We can show that this pair can be equipped with a monad structure, where the functorial action on morphisms is $f \mapsto id_{[0,\infty]} \times P_{\leq 1} f$, the unit at a point $a : A$ is the pair $(0, \delta_a)$ and the bind operation $(-)^{\#}$ adds the expected cost of the input with the average of the expected cost of the output, given the input distribution. Formally, given an $\omega\mathbf{Qbs}$ morphism

$$[\![\mathsf{charge}_c]\!]_{EC} = (c, \delta_{()}) \qquad [\![\mathsf{rand}\, V]\!]_{EC} = \left(0, \frac{\sum_{i=0}^{[\![V]\!]_{EC}} \delta_i}{[\![V]\!]_{EC} + 1}\right)$$

$$[\![\mathsf{fix}\, x.\, t]\!] = \bigsqcup_n [\![t]\!]^n_{EC}(\bot)$$

Figure 6: Expected cost semantics of operations

$f : X \to [0, \infty] \times P_{\leq 1} Y$, its bind is the function $f^{\#}(r, \mu) = (r + \int (\pi_1 \circ f)\, \mathrm{d}\mu, (\pi_2 \circ f)^{\#}_{P_{\leq 1}}(\mu))$.

Theorem 3.12. *The triple* $([0, \infty] \times P_{\leq 1}, \eta, (-)^{\#})$ *is a strong monad.*

Proof. Since $P_{\leq 1}$ is a monad, and the second component of the monad operations of $[0, \infty] \times P_{\leq 1} -$ are identical to the ones of $P_{\leq 1}$, we only need to prove the monad laws for the first component. The unit laws follow from:

$$\pi_1(\eta^{\#}(r, \mu)) = r + 0 = r$$
$$\pi_1((f^{\#} \circ \eta)(x)) = \pi_1(f^{\#}(0, \delta_x)) = 0 + \pi_1(f(x))$$

While the last law requires a bit more work:

$$\pi_1((f^{\#} \circ g^{\#})(r, \mu)) =$$
$$\pi_1(f^{\#}(r + \int (\pi_1 \circ g)\, \mathrm{d}\mu, (\pi_2 \circ g)^{\#}_{P_{\leq 1}}(\mu))) =$$
$$r + \int (\pi_1 \circ g)\, \mathrm{d}\mu + \int (\pi_1 \circ f)\, \mathrm{d}((\pi_2 \circ g)^{\#}_{P_{\leq 1}}(\mu)) =$$
$$\pi_1((f^{\#} \circ g)^{\#}(r, \mu))$$

The last equation follows from the monad laws of $P_{\leq 1}$. □

With this monad it is possible to define a new semantics to **cert** that interprets the effectful operations a bit differently from the cost semantics, as we depict in Figure 6, where $[\![\cdot]\!]^c_{EC}$ is the computation semantics while $[\![\cdot]\!]^v_{CS}$ is the value semantics; the cost monoid is still interpreted as $\mathbb{N}$.

Example 3.13 (Revisiting the geometric distribution)**.** Now that we have this new semantics, we can revisit a slightly generalized variant of the geometric distribution example where the fair coin $\oplus$ is replaced by its biased version $\oplus_p$, for $p \in [0, 1]$.

By unfolding the semantics we obtain the fixed point equation $E = 1 + (1 - p)E$, i.e. $E = \frac{1}{p}$. Note that if $p = 0$ the equation above does not have a solution. Concretely, in this case the program would never terminate.

As we have noted in the previous section, the cost semantics can be used to reason about the expected cost by using Definition 3.10. Something which will play an important role in our soundness proof is the fact that this definition interacts well with the monadic structure of $P_{\leq 1}$.

Lemma 3.14. *Let* $\mu : P_{\leq 1} A$ *and* $f : A \to P_{\leq 1}([0, \infty])$, $\mathbb{E}(f^{\#}(\mu)) = \int_A \mathbb{E}(f(a))\mu(\mathrm{d}a)$.

Proof. This can be proved by unfolding the definitions

$$E(f^{\#}(\mu)) = \int_{[0,\infty]} x f^{\#}(\mu)(\mathrm{d}x) = \int_{[0,\infty]} x \left(\int_A f(a)\mu(\mathrm{d}a) \right)(\mathrm{d}x) =$$

$$\int_A \int_{[0,\infty]} x f(a)(\mathrm{d}x)\mu(\mathrm{d}a) = \int_A \mathbb{E}(f(a))\mu(\mathrm{d}a)$$

Note that in the third equation we had to reorder the integrals, which is valid because $P_{\leq 1}$ is commutative. □

With this lemma in mind, we may state some basic definitions that allows us to describe precisely how the cost and expected cost semantics relate.

Definition 3.15. A monad morphism is a natural transformation $\gamma : T \to S$, where $(T, \eta^T, (-)^{\#}_T)$ and $(S, \eta^S, (-)^{\#}_S)$ are monads over the same category, such that

$$\gamma \circ \eta^T = \eta^S$$

$$(\gamma \circ g)^{\#}_S \circ \gamma = \gamma \circ g^{\#}_T, \text{ for every } g : A \to TB$$

Theorem 3.16. *There is a monad morphism $E : P(\mathbb{N} \times -) \to [0,\infty] \times P$.*

Proof. We define the morphism

$$E_A(\mu) = (\mathbb{E}(P(\pi_1)(\mu)), P(\pi_2)(\mu))$$

The first monad morphism equation follows by inspection and the second one follows mainly from Lemma 3.14, when restricting it to the probabilistic distributions, i.e. total mass equal to 1. □

Lemma 3.17. *The natural transformation E, when extend to subprobability distributions, is not a monad morphism.*

Proof. Let $\frac{1}{2}(\delta_{(0,1)} + \delta_{(1,2)})$ be a distribution over $\mathbb{C} \times \mathbb{N}$ and $f(0) = \frac{1}{2}\delta_0$, $f(n+1) = 0$ be a subprobability kernel. It follows by inspection that $((E \circ f)^{\#}_{[0,\infty] \times P_{\leq 1}} \circ E)(\mu) \neq (E \circ f^{\#}_{P_{\leq 1}(\mathbb{N} \times -)})(\mu)$ □

Theorem 3.16 says that the different cost semantics interact well in the probabilistic case. In the subprobabilistic case this is not true, as illustrated by Lemma 3.17. This formalizes the intuitions behind the subtleties in the interaction of expected cost and nontermination explained in the previous section.

3.4 Soundness Theorems

We have three ways for reasoning about expected cost: by using the equational theory or by using either of the denotational semantics. Now, we want to understand how they relate to one another. When we restrict the language and semantics to the probabilistic case, i.e. without unbounded recursion, we can prove strong guarantees about the different semantics. For instance, the expected cost semantics gives the same value as the expected cost of the cost distribution in the cost semantics. Furthermore, both of these semantics satisfy the same equations.

Unfortunately, in the presence of unbounded recursion, i.e. subprobability distributions, the two semantics are not "the same" anymore. What we show in this section is that in the presence of subprobability distributions, the expected cost semantics is an upper bound on the expected cost of the cost distribution. Furthermore, even though both of these semantics validate the base equational theory, many useful extensions are not sound in the cost semantics and some unusual equations do hold in it, as we explain in Section 3.4.2. This leads us to believe that the expected cost semantics provides the right level of abstraction for reasoning about the cost of recursive probabilistic programs.

3.4.1 Denotational Soundness.

Denotational Soundness: Probabilistic Case. As we have seen, by using Theorem 3.12 it is possible to compositionally reason about the expected cost of running programs. The problem is that it is unclear if this new semantics agrees with the cost semantics or if it is reasoning about a different property altogether. In this section we will restrict the monad to its *total* submonad of probability distributions, i.e. use the monads $P(\mathbb{C} \times -)$ and $[0,\infty] \times P$ and remove the recursion operation.

In order to solve this problem we must show that the expected cost semantics is sound with respect to the cost semantics. Intuitively, we want to show that if we have a program $\vdash_c t : F\mathbb{N}$, then its denotations under both $[\![\cdot]\!]_{CS}$ and $[\![\cdot]\!]_{EC}$ agree in the sense that the expected value for the second marginal of $[\![t]\!]_{CS}$ is equal to $\pi_1([\![t]\!]_{EC})$, and the distribution of the second marginal of $[\![t]\!]_{CS}$ is equal to $\pi_2([\![t]\!]_{EC})$.

In the literature, this property has been called the *effect simulation problem* and many semantic techniques have been developed for solving it [19]. One of the most general ones uses a construction called $\top\top$-lifting [18], which consists of extending ideas from the categorical logical relations literature to the monadic setting.

A simplified version of the main theorem in [19] is the following:

Theorem 3.18 ([19]). *Let $\mathbf{C}$ be a category, $T : \mathbf{C} \to \mathbf{C}$ and $S : \mathbf{C} \to \mathbf{C}$ be monads such that there is a monad morphism $\gamma : T \to S$, then for every program $\cdot \vdash_c t : F\mathbb{N}$, $\gamma_{\mathbb{N}}([\![t]\!]^c_{CS}) = [\![t]\!]^c_{EC}$.*

We can see how this theorem is relevant to what we want to prove: there are two monads $P(\mathbb{C} \times -)$ and $[0,\infty] \times P$, a monad morphism between them and we want to prove that their output distributions are the same and the expected cost of both semantics are equal — precisely how the monad morphism E is defined.

The problem with the theorem above is that, even in its most general form proved by [19], it can only handle base types and it gives no guarantees for programs of type, say $\mathrm{list}(\mathbb{N}) \to F(\mathrm{list}(\mathbb{N}))$, which is a significant limitation to the case studies we study in Section 4.

We circumvent these issues by defining a two-level version of the $\top\top$-lifting construction that works for our particular case, though we do not have a general categorical construction yet. Concretely, we construct a two-level logical relations argument, mimicking the two-level structure present in CBPV. At the base of our argument is the relational-lifting construction, defined as follows:

Definition 3.19. Let $\mathcal{R} \subseteq A \times B$ be a complete binary relation, i.e. it is closed under suprema of ascending sequences, its lifting $\mathcal{R}^{\#} \subseteq P_{\leq 1}(A) \times P_{\leq 1}(B)$ is defined as $\mu \, \mathcal{R}^{\#} \, \nu$ if there is a distribution $\theta : P_{\leq 1}(\mathcal{R})$ such that its first and second marginals are, respectively, μ and ν.

This definition interacts well with the monadic structure of $P_{\leq 1}$. Concretely, it is stable with respect to the unit and bind of $P_{\leq 1}$. The same definition can be restricted to P. We can now define two families of relations, one for value types and one for computation

types:

$$\mathcal{V}_\tau \subseteq [\![\tau]\!]^v_{CS} \times [\![\tau]\!]^v_{EC}$$
$$\mathcal{V}_{\mathbb{N}} = \{(n,n) \mid n \in \mathbb{N}\}$$
$$\mathcal{V}_{U\overline{\tau}} = C_{\overline{\tau}}$$

$$C_{\overline{\tau}} \subseteq [\![\overline{\tau}]\!]^c_{CS} \times [\![\overline{\tau}]\!]^c_{EC}$$
$$C_{F\tau} = \{((r,\nu),\mu) \mid \mathbb{E}(\mu_1) = r \wedge \nu \mathcal{V}_\tau^\# \mu_2\}$$
$$C_{\tau\to\overline{\tau}} = \{(f_1,f_2) \mid \forall x_1, x_2, x_1 \mathcal{V}_\tau x_2 \Rightarrow f_1(x_1) C_{\overline{\tau}} f_2(x_2)\}$$

We can prove by induction the following lemma:

Lemma 3.20. *For every τ (resp. $\overline{\tau}$), the relation $\mathcal{V}_\tau$ (resp. $C_{\overline{\tau}}$) is an ωCPO, where the partial order structure is the same as the one from $[\![\tau]\!]^v_{CS} \times [\![\tau]\!]^v_{EC}$ (resp. $[\![\overline{\tau}]\!]^c_{CS} \times [\![\overline{\tau}]\!]^c_{EC}$). Furthermore, the computation relations have a least element.*

Lemma 3.21. *For every type τ (resp. $\overline{\tau}$), there is a set M and partial order $\leq$ such that the triple $(\mathcal{V}_\tau, M, \leq)$ (resp. $(C_{\overline{\tau}}, M, \leq)$) is an ω-quasi Borel space with the such that the injection function is a morphism in ω**Qbs**.*

Proof. We only make explicit the proof for value types, since the case of computation types is basically the same. We define the order $\leq$ to be the same as the one in $[\![\tau]\!]^v_{CS} \times [\![\tau]\!]^v_{EC}$ and M to be the *restricted* random elements $\{f \in M_{\tau_1} \mid f(\mathbb{R}) \subseteq \mathcal{V}_\tau\}$.

Since by the lemma above the logical relations are ωCPOs, M is closed under suprema of ascending chains. and $(\mathcal{V}_\tau, M, \leq)$ is an ω-quasi Borel space. The injection into $[\![\tau]\!]^v_{CS} \times [\![\tau]\!]^v_{EC}$ being a morphism follows by construction. □

We can now state the denotational soundness theorem:

Theorem 3.22. *For every $\Gamma = x_1 : \tau_1, \ldots, x_n : \tau_n$, $\Gamma \vdash_v V : \tau$, $\Gamma \vdash_c t : \overline{\tau}$ and if for every $1 \leq i \leq n$, $\cdot \vdash_v V_i : \tau_i$ and $[\![V_i]\!]_{CS}\ \mathcal{V}_{\tau_i}\ [\![V_i]\!]_{EC}$, then*

$$\left[\!\left[\mathsf{let}\ \overline{x_i} = \overline{V_i}\ \mathsf{in}\ t\right]\!\right]_{CS} C^c_{\overline{\tau}} \left[\!\left[\mathsf{let}\ \overline{x_i} = \overline{V_i}\ \mathsf{in}\ t\right]\!\right]^c_{EC} \text{ and}$$
$$\left[\!\left[\mathsf{let}\ \overline{x_i} = \overline{V_i}\ \mathsf{in}\ V\right]\!\right]^v_{CS} \mathcal{V}_\tau \left[\!\left[\mathsf{let}\ \overline{x_i} = \overline{V_i}\ \mathsf{in}\ V\right]\!\right]^v_{EC},$$

where the notation $\overline{x_i} = \overline{V_i}$ means a list of n let-bindings or, in the case of values, a list of substitutions.

Proof. The proof follows by induction and is shown in Appendix B. □

We now have a very precise sense in which the expected-cost semantics is related to the cost semantics:

Corollary 3.23. *The expected-cost semantics is sound with respect to the cost semantics, i.e. for every program $\cdot \vdash_c t : F\tau$, the expected cost of the second marginal of $[\![t]\!]^c_{CS}$ is equal to $\pi_1([\![t]\!]^c_{EC})$.*

That being said, the theorem above is only valid for probability distributions. The key lemma in Appendix B relies on the fact that the total mass of distributions is always 1.

Denotational Soundness: Subprobabilistic Case. Though both cost semantics are "equivalent" in the sense explained above, programming without recursion is somewhat limiting when it comes to probabilities. Indeed, without an infinitely supported base distribution, every definable distribution has finite support. In particular, it would not be possible to define the geometric distribution.

Therefore, it seems reasonable to prove the soundness theorem above for $P_{\leq 1}$. Unfortunately, it does not hold in the subprobabilistic case, as alluded by Lemma 3.17.

Example 3.24. Consider the programs

$$t = \mathsf{charge}_2; \mathsf{produce}\ 0$$
$$u = \lambda x.\ \mathsf{ifZero}\ x\ \mathsf{then}\ (\bot \oplus (\mathsf{charge}_4; \mathsf{produce}\ 0))\ \mathsf{else}\ \bot$$

We can show

$$[\![x \leftarrow t; u\ x]\!]^c_{CS} = (3, \frac{1}{2}\delta_0) \neq (4, \frac{1}{2}\delta_0) = [\![x \leftarrow t; u\ x]\!]^c_{EC}$$

The counter-example above leaves us in a predicament: which semantics is the "true" semantics? We argue that the expected cost semantics is better suited for reasoning about programs. Consider the program $\mathsf{charge}_c; \bot$. From a programming point of view, the programs charges c units of costs and then loops forever. The cost semantics of the program above will be the zero distribution over $\mathbb{C}$ and the terminal object 1. Therefore, using Definition 3.10, its cost is 0. The expected cost semantics, however, gives a much more sensible semantics, it is the pair $(c, 0)$, where 0 is the zero distribution over 1.

That being said, these two semantics are not completely unrelated and we can still prove a weaker version of Corollary 3.23. This is achieved by defining basically the same logical relations as before, with the exception of $C_{F\tau}$, which now becomes

$$C_{F\tau} = \{((r,\nu),\mu) \mid \mathbb{E}(\mu_1) \leq r \wedge \nu \mathcal{V}_\tau^\# \mu_2\}$$

Corollary 3.25. *The expected-cost semantics is sound with respect to the cost semantics, i.e. for every program $\cdot \vdash_c t : F\tau$, the expected cost of the second marginal of $[\![t]\!]^c_{CS}$ is at most $\pi_1([\![t]\!]^c_{EC})$.*

Proof. The proof can also be found in Appendix B. □

3.4.2 Equational Soundness. The last section has argued that in the presence of unbounded recursion, the cost semantics validates some unrealistic equations that are not validated by the expected cost semantics. That being said, when it comes to the base equational theory of Figure 4, both semantics are sound with respect to it.

Theorem 3.26. *If $\Gamma \vdash_c t = u : \overline{\tau}$ then $[\![t]\!]^c_{EC} = [\![u]\!]^c_{EC}$ and $[\![t]\!]^c_{CS} = [\![u]\!]^c_{CS}$.*

Proof. The proof follows by induction on the equality rules, where the inductive cases follow directly from the inductive hypothesis while the base cases follow by inspection. For instance, the equation $\mathsf{charge}_0; t = t$ is true because $\mathbb{N}$ is a monoid and 0 is its unit. □

It is interesting to understand to what extent these equational theories differ. For instance, the cost semantics validates the equation $\bot; t = \bot = t; \bot$. As we have argued before, this equation is too extreme for the purposes of expected cost, since it says that $\mathsf{charge}_c; \bot = \bot$. An even more egregious equation that it satisfies is

$\mathrm{fix}\, x.\, \mathsf{charge}_1; x = \bot$. That equation says that the cost of infinity is the same as no cost at all, as long as the program does not terminate.

These equations are connected to the commutativity equation:

$$\frac{\Gamma \vdash_c x : F\tau_1 \qquad \Gamma \vdash_c u : F\tau_2 \qquad \Gamma, x : \tau_1, y : \tau_2 \vdash_c t' : \overline{\tau}}{\Gamma \vdash_c (x \leftarrow t; y \leftarrow u; t') = (y \leftarrow u; x \leftarrow t; t') : \overline{\tau}}$$

Theorem 3.27. *The cost semantics validates the commutativity equation.*

Proof. The proof follows basically by commutativity of $P_{\leq 1}$:

$$[\![x \leftarrow t; y \leftarrow u; t']\!]_{CS} =$$
$$\int_{\mathbb{N}\times A}\int_{\mathbb{N}\times B} P_{\leq 1}(f)([\![t']\!]_{CS}(a,b))\,[\![u]\!]_{CS}(\mathrm{d}n_1\,\mathrm{d}a)\,[\![t]\!]_{CS}(\mathrm{d}n_2\,\mathrm{d}b) =$$
$$\int_{\mathbb{N}\times B}\int_{\mathbb{N}\times A} P_{\leq 1}(f)([\![t']\!]_{CS}(a,b))\,[\![t]\!]_{CS}(\mathrm{d}n_2\,\mathrm{d}b)\,[\![u]\!]_{CS}(\mathrm{d}n_1\,\mathrm{d}a) =$$
$$[\![y \leftarrow u; x \leftarrow t; t']\!]_{CS}, \text{ where } f(n,c) = (n + n_1 + n_2, c) \qquad \square$$

This equation is usually useful for reasoning about probabilistic programs. However, it is too strong for the purposes of reasoning about expected cost. Indeed, consider the programs

$$t = \bot; \mathsf{charge}_c; \mathsf{produce}\ ()$$
$$u = \mathsf{charge}_c; \bot; \mathsf{produce}\ ()$$

From an operational point of view, the first program will run a non-terminating program and never reach the charge_c operation, while the second one starts by increasing the cost by c and then loops forever — closer to how cost works in the real world, e.g. if the charge operation is modeling a monetary cost, such as a call to an API, only the second program will cost something.

Lemma 3.28. *The monad $[0,\infty] \times P_{\leq 1}$ is not commutative.*

Proof. The two terms above are a counter example when $c > 0$:

$$[\![t]\!]_{EC} = (0,0) \neq (c,0) = [\![u]\!]_{EC} \qquad \square$$

Inequational theory. In the context of cost analysis, it can also be useful to reason about upper/lower bounds on the cost. The different versions of geometric distributions have already demonstrated this. Though reasoning semantically about these bounds is immediate, this is not the case at the syntactic level. One way of addressing this is by defining *inequational* theories where you can reason about programs being less than or equal to other programs.

For the expected cost, however, one must be careful in terms of which rules to include. While the rule $\mathsf{charge}_c \leq \mathsf{charge}_d$, whenever $c \leq d$ is reasonable whenever the cost monoid comes equipped with a partial order, in the presence of probability it is not so easy to syntactically reason about these properties. A simple non-trivial example would be comparing the two variants of quicksort.

Since in this work we were mainly interested on the denotational aspects of expected cost, we leave a more thorough investigation of the inequational properties of expected cost to future work.

4 EXAMPLES

In this section we will show how the expected cost semantics can be used to reason about the expected cost of probabilistic programs. We present four examples, two randomized algorithms and two recursive stochastic processes, illustrating the versability of **cert**.

4.1 Expected coin tosses

A classic problem in basic probability theory is computing the expected number of coin flips necessary in order to obtain n heads in a row. We can model this stochastic process as the following recursive probabilistic program:

$$\begin{array}{l}\mu f : \mathbb{N} \to F1.\, \lambda n : \mathbb{N}.\\ \mathsf{if}\ n\ \mathsf{then}\\ \quad \mathsf{produce}\ ()\\ \mathsf{else}\\ \quad (\mathsf{force}\ f)\ (n-1);\\ \quad \mathsf{charge}_1;\\ \quad (\mathsf{produce}\ () \oplus (\mathsf{force}\ f)\ n)\end{array}$$

For every n, the program above simulates the probabilistic structure of flipping coins until obtaining n heads in a row. When its input is 0, it outputs () without flipping any coins. If the input is greater than 0, in order to flip n heads in a row it must first flip $n-1$ heads in a row — hence the call to $f(n-1)$ — flip a new coin while increasing the current counter by 1 and, if it is heads, you have obtained n heads in a row and may output (), otherwise you must recursively start the process again from n: the left and right branches of $\oplus$, respectively.

By unfolding the semantics, we obtain that the expected number of coin tosses is given by the following recurrence relation:

$$T(0) = 0$$
$$T(n+1) = 1 + T(n) + \frac{1}{2}T(n+1)$$

Which has the closed-form solution $T(n) = 2(2^n - 1)$.

4.2 Randomized Quicksort

Quicksort is one of the main sorting algorithms and, as the name suggests, it is very fast. The problem with it is that it has an obvious worst-case scenario: if the input list is in reverse order, quicksort requires $O(n^2)$ comparisons. However, if you randomly choose an element of the list as your pivot, you can prove that it only requires $O(n \log(n))$ comparisons.

In Example 2.5 we present a program that implements the randomized algorithm. If we simply interpret the expected cost of this program denotationally, it will be a function mapping lists to real numbers. This is not how such an analysis is done in practice, where people reason about how the cost increases as the length of the list increases, regardless of which elements it contains.

In our semantics, the denotation of the program is hiding the fact that its cost only depends on the length of its argument. We make this precise by defining a measurable function using the program in figure 7 $qck_{\mathbb{N}} : \mathbb{N} \to \mathbb{R} \times P_{\leq 1}\mathbb{N}$ that corresponds to the quicksort structure assuming that the input is a natural number. Let $len : \mathrm{list}(\mathbb{N}) \to \mathbb{N}$ be the function that outputs the length of its input.

```
μf : ℕ → Fℕ. λn : ℕ.
if n then
  produce 0
else
  charge_{n−1};
  x ← rand n;
  (force f) x;
  (force f) (n − x);
  produce n
```

Figure 7: Quicksort over natural numbers

Lemma 4.1. *The following diagram commutes:*

$$\begin{array}{ccc} \mathsf{list}(\mathbb{N}) & \xrightarrow{len} & \mathbb{N} \\ {\scriptstyle qck_{\mathsf{list}(\mathbb{N})}}\downarrow & & \downarrow{\scriptstyle qck_{\mathbb{N}}} \\ F\mathsf{list}(\mathbb{N}) & \xrightarrow[F(len)]{} & F\mathbb{N} \end{array}$$

Proof. This can be proved by strong induction on the length of the input list. If the list is empty, then its length is 0 and the diagram commutes. If, however, the list not not empty, there will be a bijection between the recursive calls to lists of size n' to sampling n' in $qck_{\mathbb{N}}$. By using the strong inductive hypothesis to these lists of size n', we can conclude. □

We can now conclude our analysis, since by the soundness theorem and the commutative diagram above, the expected cost of quickSort is given by $\pi_1 \circ qck_{\mathbb{N}} \circ len$. Since the expression $\pi_1 \circ qck_{\mathbb{N}}$ satisfies the recursive definition:

$$T(0) = 0$$
$$T(n) = n - 1 + \frac{2}{n}\sum_{i=0}^{i-1} T(i)$$

This allows us to conclude that quickSort has an expected cost of $O(n \log(n))$.

4.3 Quickselect

Consider the Quickselect problem, which receives as input an unordered list l and a natural number n and outputs the the n-th largest element of l. This algorithm is very similar to quicksort: you choose a pivot, split the list into elements larger and smaller than it, then recurse on the appropriate list.

```
μf : list(ℕ) → ℕ → F(ℕ).
λl : list(ℕ).
λn : ℕ.
case l of
| nil ⇒
  produce nil
| (hd, tl) ⇒
  len ← length l
  r ← rand len
  pivot ← len[r]
  (l1, l2) ← biFilter (λn. charge_1; n ≤ pivot) l
  lgth ← len l1
  if lgth < n − 1 then
    (force f) l1 n
  elseif lgth == n − 1 then
    produce pivot
  else
    (force f) l2 (n − lgth)
```

In the best case, this algorithm runs in linear time and in the worst case, quadratic time. If we choose the pivot uniformly random, we get linear time, which is given by the following recurrence relation.

$$T(0) = 0$$
$$T(n) = n - 1 + \frac{1}{n}\sum_{i} T(i)$$

4.4 Random Walks

Random walks are classic examples of probabilistic processes. For this example we are interested in the symmetric random walk over the natural numbers. At every point n the probability of moving to $n-1$ or $n+1$ is $\frac{1}{2}$. Furthermore, we are assuming the variant where at 0 you move to 1 with probability 1. We can write a program that simulates such a random walk with a point of departure $i : \mathbb{N}$ and a point of arrival $j : \mathbb{N}$:

```
randomWalk = μf : ℕ → ℕ → F1. λi : ℕ j : ℕ.
if i = j then
  produce ()
else
  charge_1;
  if i then
    (force f) 1 j
  else
    ((force f) (i − 1) j) ⊕ ((force f) (i + 1) j)
```

The program receives the starting and end points, i and j, respectively, as arguments, and if they are equal, you stop the random walk. Otherwise, you take one step of the random walk, i.e. you

take step to either $i-1$ or $i+1$ with equal probability, with the exception of when $i = 0$, in which case you go to 1. This iterative behaviour can be straightforwardly captured with recursion, as illustrated by the program above.

It is now possible to compute the expected value on the number of rounds that are necessary in order to reach your target. This is given by the following two-argument recursive relation.

$$
\begin{aligned}
T(i,i) &= 0 \\
T(0,j) &= 1 + T(1,j) \\
T(i,j) &= 1 + \frac{1}{2}(T(i-1,j) + T(i+1,j))
\end{aligned}
$$

This recurrence relation is well-known in the theory of Markov chains — see [29] for an introduction. Something interesting about it is that when $i > j$, this stochastic process reduces to the symmetric random walk without an absorbing state, which is known to have ∞ expected cost.

5 RELATED WORK

There has been much work done on logic and language techniques for reasoning about the cost of programs.

Type Theories for Cost Analysis. Recent work [10, 28] have developed (in)equational theories for reasoning about costs of programs inside a modal depedently-typed logic. Their framework can reason about monadic effects by using the Writer monad transformer, similarly to what we have done, but, due to being inside a total dependent type theory, they cannot represent fixed-point combinators. Furthermore, they can only handle discrete probabilities, whereas we can easily accommodate both arbitrary recursion as well as continuous distributions.

Other work has focused in designing type theories for doing relational reasoning of programs [6, 30]. Even though these approaches can reason about functional programs as well, they are limited to deterministic programs.

In work by Avanzini et al [2], the authors define a graded, substructural type system for reasoning about expected cost of functional programs, even using a randomized quicksort as an example. One of the main limitations of their system with respect to **cert** is that, due to the substructural invariants of their type system, it can only type check a limited subset of the programs that **cert** can. For instance, it cannot type check the functional programming staples of fold and map functions over lists. Furthermore, they have not addressed how feasible type checking in their system is or if it is even decidable.

In other work by Avanzini et al [1], the authors describe a continuation passing style (CPS) transformation into a metalanguage for reasoning about expected cost of programs. Compared to **cert**, both metalanguages can both handle functional programming, however, due to its CPS semantics, reasoning about recursive programs becomes less tractable, especially in the presence of higher-order functions. Indeed, while in the older substructural type system it is possible to reason about a randomized quicksort algorithm [2], this is not the case anymore in the CPS case, forcing them to restrict their analysis to an imperative language that they embed into their metalanguage.

Automatic Resource Analysis. One fruitful research direction has been the automatic amortized resource analysis (AARA) [13, 14, 27] which uses a type system to annotate programs with their cost and automatically infer the cost of the program. By now, these techniques have been extended to reason about recursive types [11], probabilistic programs [33] and programs [24] with local state.

Something quite appealing about their approach is that it is completely automatic, whereas our approach requires solving a, possibly hard, recurrence relation by hand. That being said, their system can only accommodate polynomial bounds, meaning that they cannot infer the $n\log(n)$ bound for the probabilistic quicksort like we do. Recently, AARA has been extended to accommodate exponential bounds [15] in deterministic programs, though it is still unclear if the same technique can be extended to the probabilistic setting, meaning that they cannot analyse the behaviour of exponentially slow programs such as the expected coin tosses one.

There have been other type-based approach to automatically reasoning about cost of programs, such as the language TiML [34]. This language allow users to annotate type signatures with cost-bounds and the type checking algorithm and infer and check these bounds. The main limitation of TiML in comparison to our work is that it cannot handle probabilistic programs. There has also been work done on automated reasoning about cost for first-order probabilistic programs by Avanzini et al. [3]. The main limitation of this work when compared to **cert** is that it can only handle first-order imperative programs.

It is an interesting line of future work understanding to what extent solving recurrence relations can be automated in the context of **cert**.

Recurrence for Expected Cost. There has been some work done in exploring languages for expressing recurrence relations for expected cost. For example, [31] provides a language for representing probabilistic recurrence relations and a tool for analysing their tail-bounds. The main drawback of these approaches is that the languages are not very expressive. In particular they do not have higher-order functions.

[22] define a first-order probabilistic functional language for manipulating data structures and automatically infer bounds on the expected cost of programs. The main limitation of their approach compared to ours is that their language is first-order.

Reasoning about expected cost has also been explored for imperative languages. For instance, in [4] the authors fevelop a weakest pre-condition calculus for reasoning about the expected cost of programs. Again, they can only reason about first-order imperative programs.

6 CONCLUSION AND FUTURE WORK

In this work we have presented **cert**, a metalanguage for reasoning about expected cost of recursive probabilistic programs. It extends the existing work of [20] to the probabilistic setting. We have proposed two different kinds of extensions, one based on the writer monad transformer while the second one uses a novel *expected cost* monad. Furthermore, we have showed that in the absence of unbounded recursion, these two semantics coincide, while when programming with subprobability distributions we have proved

that the expected cost semantics is an upper bound to the cost semantics.

We have justified the versatility of our expected cost semantics by presenting a few case-studies. In particular, the expected cost semantics obtains, compositionally, the familiar recurrence cost relations for non-trivial programs. In particular, for the randomized quicksort algorithm, the semantic recurrence relation recovers the $O(n \log n)$ bound.

We conjecture that the techniques presented in this paper can be used in various other tantalizing directions. By slightly modifying the expected cost monad we suspect that we can also reason about higher-moments of probabilistic programs. In particular, the variance of costs is a useful measure to be able to reason about.

Going beyond probability, by adopting a more abstract approach on the expected cost monad, it might be possible to reason about other kinds of effects. For instance, when reasoning about non-deterministic programs it is useful to give bounds on the worse/best case scenarios.

More generally, we are also interested in laying on firm categorical grounds our logical relations technique. Furthermore, the lack of a monad morphism in the subprobabilistic case suggests that there might extensions of this techniques based on 2-category theory, where the monad morphism laws do not hold exactly, only up to a 2-cell, or inequality in this particular case.

REFERENCES

[1] Martin Avanzini, Gilles Barthe, and Ugo Dal Lago. 2021. On continuation-passing transformations and expected cost analysis. In *International Conference on Functional Programming (ICFP)*.

[2] Martin Avanzini, Ugo Dal Lago, and Alexis Ghyselen. 2019. Type-based complexity analysis of probabilistic functional programs. In *Logic in Computer Science (LICS)*.

[3] Martin Avanzini, Georg Moser, and Michael Schaper. 2020. A modular cost analysis for probabilistic programs. In *Object-oriented Programming, Systems, Languages, and Applications (OOPSLA)*.

[4] Kevin Batz, Benjamin Lucien Kaminski, Joost-Pieter Katoen, Christoph Matheja, and Lena Verscht. 2023. A calculus for amortized expected runtimes.

[5] Krishnendu Chatterjee, Hongfei Fu, Petr Novotnỳ, and Rouzbeh Hasheminezhad. 2016. Algorithmic analysis of qualitative and quantitative termination problems for affine probabilistic programs. In *Principles of Programming Languages (POPL)*.

[6] Ezgi Çiçek, Gilles Barthe, Marco Gaboardi, Deepak Garg, and Jan Hoffmann. 2017. Relational cost analysis. In *Principles of Programming Languages (POPL)*.

[7] Joseph W Cutler, Daniel R Licata, and Norman Danner. 2020. Denotational recurrence extraction for amortized analysis.

[8] Norman Danner, Daniel R Licata, and Ramyaa Ramyaa. 2015. Denotational cost semantics for functional languages with inductive types. In *International Conference on Functional Programming (ICFP)*.

[9] Norman Danner, Jennifer Paykin, and James S Royer. 2013. A static cost analysis for a higher-order language. In *7th workshop on Programming languages meets program verification*.

[10] Harrison Grodin, Jonathan Sterling, Yue Niu, and Robert Harper. 2023. Decalf: A Directed, Effectful Cost-Aware Logical Framework. *arXiv preprint arXiv:2307.05938* (2023).

[11] J. Grosen, D. M. Kahn, and J. Hoffmann. 2023. Automatic Amortized Resource Analysis with Regular Recursive Types. In *Logic in Computer Science (LICS)*.

[12] Chris Heunen, Ohad Kammar, Sam Staton, and Hongseok Yang. 2017. A convenient category for higher-order probability theory. In *Logic in Computer Science (LICS)*.

[13] Jan Hoffmann, Ankush Das, and Shu-Chun Weng. 2017. Towards automatic resource bound analysis for OCaml. In *Symposium on Principles of Programming Languages (POPL)*.

[14] Jan Hoffmann and Zhong Shao. 2015. Automatic static cost analysis for parallel programs. In *European Symposium on Programming (ESOP)*.

[15] David M Kahn and Jan Hoffmann. 2020. Exponential automatic amortized resource analysis. In *Foundations of Software Science and Computation Structures (FoSSaCS)*.

[16] Benjamin Lucien Kaminski, Joost-Pieter Katoen, Christoph Matheja, and Federico Olmedo. 2016. Weakest precondition reasoning for expected run–times of probabilistic programs. In *European Symposium on Programming (ESOP)*.

[17] Richard M Karp. 1994. Probabilistic recurrence relations. *Journal of the ACM (JACM)* (1994).

[18] Shin-ya Katsumata. 2005. A semantic formulation of ⊤⊤-lifting and logical predicates for computational metalanguage. In *International Workshop on Computer Science Logic*. Springer, 87–102.

[19] Shin-ya Katsumata. 2013. Relating computational effects by ⊤⊤-lifting. *Information and Computation* (2013).

[20] GA Kavvos, Edward Morehouse, Daniel R Licata, and Norman Danner. 2019. Recurrence extraction for functional programs through call-by-push-value. In *Principles of Programming Languages (POPL)*.

[21] Satoshi Kura, Natsuki Urabe, and Ichiro Hasuo. 2019. Tail probabilities for randomized program runtimes via martingales for higher moments. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*.

[22] Lorenz Leutgeb, Georg Moser, and Florian Zuleger. 2022. Automated expected amortised cost analysis of probabilistic data structures. In *Computer Aided Verification (CAV)*.

[23] Paul Blain Levy. 2001. *Call-by-push-value*. Ph. D. Dissertation.

[24] Benjamin Lichtman and Jan Hoffmann. 2017. Arrays and references in resource aware ML. In *Formal Structures for Computation and Deduction (FSCD 2017)*.

[25] E Moggi. 1989. Computational lambda-calculus and monads. In *Logic in Computer Science (LICS)*.

[26] Rajeev Motwani and Prabhakar Raghavan. 1995. *Randomized algorithms*. Cambridge university press.

[27] Van Chan Ngo, Quentin Carbonneaux, and Jan Hoffmann. 2018. Bounded expectations: resource analysis for probabilistic programs. In *Programming Language Design and Implementation (PLDI)*.

[28] Yue Niu, Jonathan Sterling, Harrison Grodin, and Robert Harper. 2022. A cost-aware logical framework. In *Principles of Programming Languages (POPL)*.

[29] James R Norris. 1998. *Markov chains*. Number 2. Cambridge university press.

[30] Vineet Rajani, Marco Gaboardi, Deepak Garg, and Jan Hoffmann. 2021. A unifying type-theory for higher-order (amortized) cost analysis. In *Symposium on Principles of Programming Languages (POPL)*.

[31] Yican Sun, Hongfei Fu, Krishnendu Chatterjee, and Amir Kafshdar Goharshady. 2023. Automated Tail Bound Analysis for Probabilistic Recurrence Relations. In *Computer Aided Verification (CAV)*.

[32] Matthijs Vákár, Ohad Kammar, and Sam Staton. 2019. A domain theory for statistical probabilistic programming. In *Principles of Programming Languages (POPL)*.

[33] Di Wang, David M Kahn, and Jan Hoffmann. 2020. Raising expectations: automating expected cost analysis with types. *International Conference on Functional Programming (ICFP)*.

[34] Peng Wang, Di Wang, and Adam Chlipala. 2017. TiML: a functional language for practical complexity analysis with invariants. In *Object-oriented Programming, Systems, Languages, and Applications (OOPSLA)*.

A MONADIC SEMANTICS OF CBPV

Let $\mathbf{C}$ be a Cartesian closed category and $T : \mathbf{C} \to \mathbf{C}$ a strong monad over it. An alternative definition of monads is it being a triple (T, η, μ), where T and η are natural transformations as before, but $\mu : T^2 \to T$, the multiplication, replaces the bind natural transformation. The monad laws under this definition become:

$$\begin{array}{ccc} T^3 & \xrightarrow{T\mu} & T^2 \\ {\scriptstyle \mu_T}\downarrow & & \downarrow{\scriptstyle \mu} \\ T^2 & \xrightarrow[\mu]{} & T \end{array} \qquad \begin{array}{ccccc} T & \xrightarrow{T\eta} & T^2 & \xleftarrow{\eta_T} & T \\ & \searrow^{1} & \downarrow{\scriptstyle \mu} & \swarrow_{1} & \\ & & T & & \end{array}$$

It is possible to show that these definitions are equivalent: given bind $(-)^{\#}$, the multiplication can be defined as $\mu = id^{\#}_{TA}$. Conversely, given a multiplication, the bind is defined as $f^{\#} = Tf; \mu$. This alternative definition is a bit better suited for the original purposes of monads, where it was used as a unifying way of representing concepts from universal algebra.

This alternative presentation lend itself quite well to the semantics of CBPV-based calculi where, given a monad T, computation types denote T-algebras:

Definition A.1. A T-algebra is a pair (A, α), where A is a $\mathbf{C}$ object and $\alpha : TA \to A$ is a morphism, such that

$$\begin{array}{ccc} A & \xrightarrow{\eta_A} & TA \\ & \searrow^{id_A} & \downarrow{\scriptstyle \alpha} \\ & & A \end{array} \qquad \begin{array}{ccc} T^2A & \xrightarrow{\mu_A} & TA \\ {\scriptstyle T\alpha}\downarrow & & \downarrow{\scriptstyle \alpha} \\ TA & \xrightarrow[\alpha]{} & A \end{array}$$

Given a T-algebra (A, α) we denote by $A_\bullet$ the object of the T-algebra.

Example A.2. Given an object A, the pair (TA, μ_A) is a T-algebra.

Example A.3. Given a T-algebra (A, α) and an objects B, we can equip $B \to B$ with the T-algebra structure $\alpha_{B\to A} = \varepsilon; B \Rightarrow (st; T(ev; \alpha))$, where $\varepsilon_A : A \to (B \Rightarrow (B \times A))$ is the unit of the Cartesian closed adjunction.

Algebras and their morphisms can be organized as a category, frequently denoted by $\mathbf{C}^T$. However, for the purposes of CBPV a different category is used:

Definition A.4. The category $\widetilde{\mathbf{C}}^T$ is the full subcategory of $\mathbf{C}$ that contain T-algebras as objects. This category is also called the category of algeras and plain maps.

The idea is that values are interpreted as objects in $\mathbf{C}$ while computation types are T-algebras. Assuming the only the base type in the calculus to be $\mathbb{N}$ and an object $\mathbb{N}$ in the base category, The interpretation of values and computations are as follows:

$$\begin{aligned} [\![\mathbb{N}]\!]^v &= \mathbb{N} \\ [\![U\overline{\tau}]\!]^v &= [\![\overline{\tau}]\!]^c_\bullet \\ [\![\tau_1 \times \tau_2]\!]^v &= [\![\tau_1]\!]^v \times [\![\tau_2]\!]^v \end{aligned}$$

$$\begin{aligned} [\![F\tau]\!]^c &= (T\,[\![\tau]\!]^v, \mu_{[\![\tau]\!]^v}) \\ [\![\tau \to \overline{\tau}]\!]^c &= ([\![\tau]\!]^v \Rightarrow [\![\overline{\tau}]\!]^c, \alpha_{[\![\tau]\!]^v \to [\![\overline{\tau}]\!]^c}) \end{aligned}$$

It is also possible to give semantics to the terms of the language as depicted in Figure 8. The semantics of if-statements use the fact that $\mathbb{N} \cong 1 + \mathbb{N}$, so you can define its semantics by using the universal property of coproducts $[t, (!_{\mathbb{N}}; u)]$, where $!_A : A \to 1$ is the unique arrow into the terminal object. The abstraction and application rule use the adjoint structure (Λ, ev), of Cartesian closed categories, where Λ and ev are the unit and counit of the adjunction, respectively. The produce rule uses the unit of the monad while the bind rule is the sequential composition of a free algebra with a non-free algebra and, therefore, requires applying the functor T and using the algebra structure of the output — when the output map is a free algebra, this operation is equal to the bind of the monad. Thunk and force are basically no-ops in this semantics, while the rules let and unpair are sequential compositions. Pair is the universal property of products.

A.1 Equational presentation of cert

For the sake of simplicity of the equational theory, we will assume the barycentric operations $\oplus_p$ with the syntactic sugar

$$\begin{aligned} rnd(0) &\rightsquigarrow \text{produce } 0 \\ rnd(n+1) &\rightsquigarrow T(n) \oplus_{\frac{1}{n+2}} \text{produce } (n+1) \end{aligned}$$

Lemma A.5. *For every $n : \mathbb{N}$, $[\![rnd(n)]\!] = [\![\text{rand } n]\!]$.*

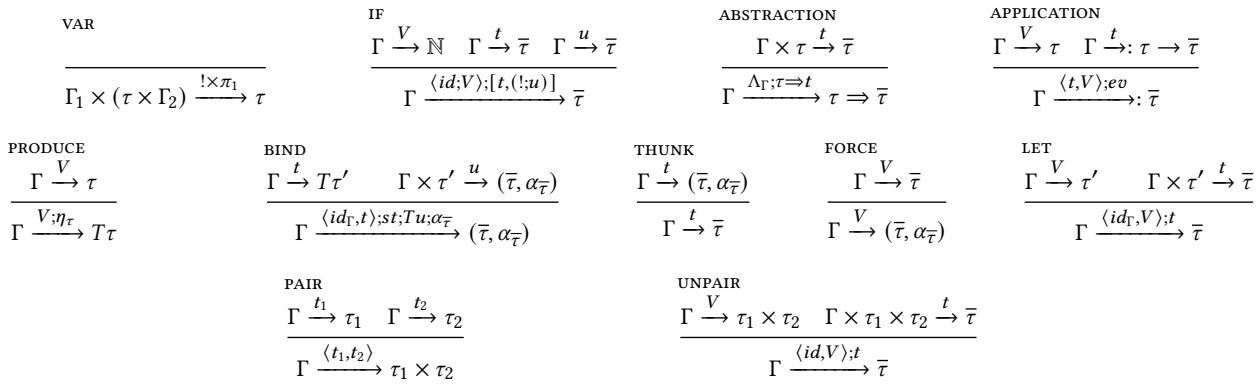


Figure 8: CBPV monadic semantics

Proof. The proof follows by induction. □

Below we present the non-structural equations of **cert**. The first block is present in every CBPV calculus with natural numbers. The second block is the recursion equation, the third block are the barycentric algebra equations and the final three are the monoid equations.

$$\mathsf{ifZero}\ 0\ \mathsf{then}\ t\ \mathsf{else}\ u \equiv t$$
$$\mathsf{ifZero}\ (n+1)\ \mathsf{then}\ t\ \mathsf{else}\ u \equiv u$$
$$t \equiv \mathsf{ifZero}\ x\ \mathsf{then}\ t\ \mathsf{else}\ t$$
$$(\lambda x.\ t)\ V \equiv t\{V/x\}$$
$$\mathsf{let}\ x\ \mathsf{be}\ V\ \mathsf{in}\ t \equiv t\{V/x\}$$
$$t \equiv \lambda x.\ t\ x$$
$$\mathsf{force}\ (\mathsf{thunk}\ t) \equiv t$$
$$\mathsf{thunk}\ (\mathsf{force}\ V) \equiv V$$
$$x \leftarrow (\mathsf{produce}\ V); t \equiv t\{V/x\}$$
$$x \leftarrow t; \mathsf{produce}\ x \equiv t$$

$$\mathsf{fix}\, x.\, t = t\{(\mathsf{thunk}\ (\mathsf{fix}\, x.\, t))/x\}$$

$$t \oplus_0 u \equiv t$$
$$t \oplus_p u \equiv u \oplus_{1-p} t$$
$$t \oplus_p t \equiv t$$
$$t \oplus_p (u \oplus_q t') \equiv (t \oplus_{pq} u) \oplus_{\frac{p(1-q)}{1-pq}} t'$$

$$\mathsf{charge}_n; \mathsf{charge}_m \equiv \mathsf{charge}_{n+m}$$
$$\mathsf{charge}_0; t \equiv t$$
$$t; \mathsf{charge}_0 \equiv t$$

B DENOTATIONAL SOUNDNESS PROOF

Lemma B.1. *If $\mu_1\ C_{F\tau}\ \mu_2$ then for every pair of functions $f_1 : A_1 \to \mathbb{R}$ and $f_2 : A_2 \to \mathbb{R}$, such that for every $a_1\ C_A\ a_2$, $f_1(a_1) = f_2(a_2)$, $\int f_1 \,\mathrm{d}\mu_1 = \int f_2 \,\mathrm{d}\mu_2$*

Proof. Since by assumption $\mu_1\ C_{F\tau}\ \mu_2$, there is a coupling ν over the support of $\mathcal{V}_\tau$, which allows us to conclude:

$$\int f_1 \,\mathrm{d}\mu_1 = \int \frac{1}{2}(f_1 + f_2)\,\mathrm{d}\nu = \int f_2 \,\mathrm{d}\mu_2$$

The equalities above hold because in the support of ν, $f_1(a_1) = f_(a_2)$, making $f_1 = \frac{1}{2}(f_1 + f_2) = f_2$ and since ν is a joint distribution with marginals μ_1 and μ_2, we have the equality of integrals above. □

The following lemma is the most technical aspect of the soundness proof and, intuitively, is saying that the logical relations for computation types can be equipped with "algebra" structures. Furthermore, since we are proving two similar looking theorems for the probabilistic and subprobabilistic cases, and the soundness proof in both cases is basically the same, we will only present the proof to the subprobabilistic case, and highlight in the proof what would differ for the probabilistic case.

Lemma B.2. *Let τ_1, τ_2 and $\overline{\tau}$ be types and $f_1 : [\![\tau_1]\!]^v_{CS} \times [\![\tau_2]\!]^v_{CS} \to [\![\overline{\tau}]\!]^c_{CS}$, and $f_2 : [\![\tau_1]\!]^v_{EC} \times [\![\tau_2]\!]^v_{EC} \to [\![\overline{\tau}]\!]^c_{EC}$ be ω**Qbs** morphisms such that $f_1 \times f_2$, when the input is restricted to $\mathcal{V}_{\tau_1} \times \mathcal{V}_{\tau_2}$, the output is restricts to $C_{\overline{\tau}}$. It is true that $(st; T_1(f_1); \alpha_{\overline{\tau}}) \times (st; T_2(f_2); \alpha_{\overline{\tau}})$, when its input is restricted to $\mathcal{V}_{\tau_1} \times C_{F\tau_2}$, the output is still restricted to $C_{\overline{\tau}}$.*

Proof. This can be proved by induction on the computation type $\overline{\tau}$:

$F\tau$: In order to prove $f_1^\#(r, v)\ C_{F\tau'}\ f_2^\#(\mu)$ we have to prove that their expected costs are related by the inequality given by the definition of $C_{F\tau}$ and show that there is a coupling over ν and μ_2, where μ_2 is the second marginal of μ, such that it factors through the inclusion $P_{\leq 1}(\mathcal{V}_{\tau'}) \hookrightarrow P_{\leq 1}([\![\tau']\!]^v_{CS} \times [\![\tau']\!]^v_{EC})$.

By unfolding the definitions, we get

$$\pi_1(f_1^\#(r, v)) = r + \int (\pi_1 \circ f_1)\,\mathrm{d}\nu$$
$$\mathbb{E}(f_2^\#(\mu)) = \int\int n\|f_2(a)\|\mu(\mathrm{d}n, \mathrm{d}a) + \int n\,\mathrm{d}(f_2^\#(\mu))$$

In the second expression the left hand side term being add corresponds to the expected cost of the input while the second one corresponds to the cost of the continuation. As such, it is sensible that in order to reason about their difference we should reason individually about $r - \int\int n\|f_2(a)\|$ and $\int (\pi_1 \circ f_1)\,\mathrm{d}\nu - \int n\,\mathrm{d}(f_2^\#(\mu))$.

$$\int\int n\|f_2(a)\|\,\mathrm{d}\mu \leq \int\int n\,\mathrm{d}\mu \leq r$$

For the second expression, assuming $\forall a'\ \mathcal{V}_{\tau'}\ a, \mathbb{E}(f_2(a)) \leq (\pi_1 \circ f_1)(a')$,

$$\int \mathbb{E}(f_2(a))\gamma(\mathrm{d}a, \mathrm{d}a') \leq \int (\pi_1 \circ f_1)(a')\gamma(\mathrm{d}a, \mathrm{d}a') = \int (\pi_1 \circ f_1)(a')\nu(\mathrm{d}a')$$
$$\int n\,\mathrm{d}(f_2^\#(\mu)) = \int \mathbb{E}(f_2(a))\mu(\mathrm{d}a) = \int \mathbb{E}(f_2(a))\gamma(\mathrm{d}a, \mathrm{d}a')$$

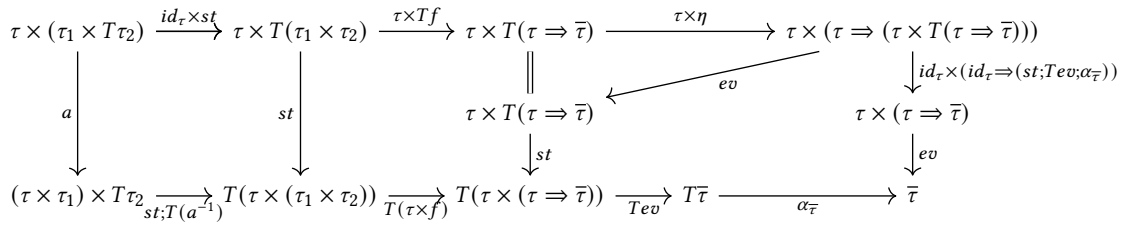
By adding these two inequalities we obtain exactly the first condition of the relation $C_{F\tau}$. In the probabilistic case every inequality is an equality, since $\|f(a)\| = 1$ and the definition of the $C_{F\tau}$ would be equalities as well. The second condition follows from observing that when restricting the domain of $f_1 \times f_2$ to $\mathcal{V}_\tau$, we can extract from it a function $g : \mathcal{V}_\tau \to P_{\leq 1}(\mathcal{V}_{\tau'})$ such that, given inputs (v_1, v_2), the marginals of $g(v_1, v_2)$ are equal to $\pi_2(f_1(v_1))$ and $f_2(v_2)_2$ since, by assumption, $f_1(v_1)\ C_{F\tau'}\ f_2(v_2)$.

Given this function, we define the coupling $g^\#(\mu')$, where μ' is the coupling given by $(r, v)\ C_{F\tau}\ \mu$. Showing that it has the right marginals follows from linearity of the marginal function, concluding the proof. This part of the proof remains the same in the probabilistic case

$\tau \to \overline{\tau}$: This case relies more on notation and, therefore, in order to simplify the presentation, we will rely on the symmetry of f_1 and f_2 and work on the generic expression $st; T(f); \alpha_{\tau\to\overline{\tau}}$ that can be instatiated to both f_1 and f_2.

By definition of $C_{\tau\to\overline{\tau}}$, in order to define a morphism $\mathcal{V}_{\tau_1} \times \mathcal{V}_{\tau_2} \to C_{\tau\to\overline{\tau}}$, it suffices to defines its transpose $\mathcal{V}_\tau \times (\mathcal{V}_{\tau_1} \times \mathcal{V}_{\tau_2}) \to C_{\overline{\tau}}$. Since the algebra structure of $\alpha_{\tau\to\overline{\tau}}$ is defined as $\eta; id_\tau \Rightarrow (st; T(ev); \alpha_{\overline{\tau}})$, we want to show that the the map $id_\tau \times (st; Tf; \eta; id_\tau \Rightarrow (st; T(ev); \alpha_{\overline{\tau}})); ev$, i.e. can be rewritten in the format $st; T(f'); \alpha_{\overline{\tau}}$, so that we can apply the inductive hypothesis. This equation

holds, up to isomorphism, by the following commutative diagram:



From left to right, the first diagram commutes by definition of strong monad, the second commutes from naturality of the strength of T, the triangular diagram commutes by the Cartesian closed adjunction and the final diagram commutes by naturality of ev.

□

Note that the same argument also holds for the parametric case where the domain has shape $[\![\Gamma]\!]^v \times [\![\tau]\!]^v$ and you must use the strength $st_{\Gamma,\tau}$ of the monad in order to only Klesli-lift the second input of f_i.

We can now prove the full statement.

Proof. The proof follows from mutual induction on $\Gamma \vdash^v V : \tau$ and $\Gamma \vdash^c t : \overline{\tau}$. Many of the cases follow by just applying the induction hypothesis or by assumptions of theorem. We go over the most interesting cases:

Comp This case follows from Lemma B.2.

Fix This theorem follows from the induction hypothesis and from the fact that the relations $C^c_{\overline{\tau}}$ are closed under suprema of ascending chains.

Produce First apply the induction hypothesis to V and assume that $[\![V]\!]_1 = v_1$ and $[\![V]\!]_2 = v_2$. By construction, $\eta^{T_1}(v_1)\ C^v_{F\tau}\ \eta^{T_2}(v_2)$, since they both have the same expected value and the coupling is $\delta_{(v_1,v_2)}$.

Case By applying the inductive hypothesis to V we may do case analysis on it and if it is the empty list, we use the inductive hypothesis on t and, otherwise, we use the inductive hypothesis on u.

□