Denotational Foundations for Expected Cost Analysis[*]

ANONYMOUS AUTHOR(S)

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 stops being deterministic, since it depends on the probabilistic samples made throughout the execution of the program. This interaction is further complicated by the non-trivial interaction between cost, recursion and evaluation strategy.

In this work we introduce **cert**: a Call-By-Push-Value (CBPV) metalanguage for reasoning about probabilistic cost. We equip **cert** with an operational cost semantics and define two denotational semantics — a cost semantics and an expected-cost semantics. We prove operational soundness and adequacy for the denotational cost semantics and a cost adequacy theorem for the expected-cost semantics.

We formally relate both denotational semantics by stating and proving a novel *effect simulation* property for CBPV. We also prove a canonicity property of the expected-cost semantics as the minimal semantics for expected cost and probability by building on recent advances on monadic probabilistic semantics.

Finally, we illustrate the expressivity of **cert** and the expected-cost semantics by presenting case-studies ranging from randomized algorithms to stochastic processes and show how our semantics capture their intended expected cost.

1 INTRODUCTION

This paper addresses the following question: what is the semantic essence of expected cost analysis? Since randomized algorithms [33] and reasoning about resource consumption in programs are central pillars of computer science, it is important to lay these analyses on solid theoretical grounds.

In the case of deterministic cost analysis, the line of work initiated by Danner et al. [10–12, 26] consolidated our denotational understanding. These ideas were later elegantly refined in the context of modal dependently-typed theories [35]. However, in the case of expected cost analysis, while much work has been done in their theory [1, 17, 23] and practice [3, 28, 41], there is no denotational semantics satisfying the following desiderata:

- Compositional. When reasoning about the expected cost of programs, it should suffice to reason about the cost of its components.
- Close to probability theory, allowing for useful lemmas to be used when reasoning about programs.
- Validates interesting and non-trivial program equations. For instance, in the context of probabilistic programming, being able to reorder disjoint program fragments is natural and useful for justifying program optimizations.

In this work we propose the first semantics that validates all of these desiderata. Furthermore, we establish formal connections and soundness properties between this novel semantics and other denotational approaches to expected cost analysis [1, 17] by proving a generalization of Filinski's *effect simulation* property [14].

[*] Title note

XXXX-XXXX/2024/10-ART $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

Our Work: Denotational Methods for Expected Cost. In this work we shed some light on the semantic foundations of expected cost analysis in the context of recursive probabilistic functional programs. We start by defining **cert**: a call-by-push-value (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. It is the first cost-aware metalanguage that can accommodate continuous distributionsm, recursion and different evaluation strategies.

Besides equipping our metalanguage with an equational theory and an operational cost semantics, we provide 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 a 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 directly tracks 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 upper bound in the presence of unbounded recursion. In order to achieve this soundness result, we state and prove a generalization of the effect simulation problem [25] beyond base types. This generalization interacts well with the CBPV type structure and provides a novel semantic technique for doing relational reasoning of effectful programs.

Importantly, we compare our semantics to the influential pre-expectation program transformations introduced by Kaminski et al. [23]. We argue that our expected cost semantics provides better abstractions for expected-cost reasoning, since many useful properties that are proved by delicate syntactic arguments in the pre-expectation semantics, hold automatically and unconditionally in our semantics. Furthermore, we show that our expected cost model satisfies useful commutativity equations that are not validated by the pre-expectation one. Finally, we prove an unexpected connection between these two monads by showing that the expected cost monad can be obtained as the minimal submonad of the pre-expectation monad that can accommodate probability and expected cost, showing that besides providing a compositional account to expected cost, the expected cost monad is canonical.

As applications, we showcase the capabilities of our semantics by using it to reason about the expected cost of probabilistic algorithms and stochastic processes. We highlight our analysis of a stochastic variant of the convex hull algorithm [15] that was outside of reach of other logic/PL techniques for expected cost analysis due to its combination of continuous probability, modular interaction of cost and behaviour, and deep probabilistic reasoning. As a guiding example, throughout the paper we will use geometric distributions to illustrate different aspects of **cert** and its semantics.

Our approach contrasts with other work done on expected cost analysis where the language analyzed was either imperative [3, 23, 27] or first-order [28, 39], or the cost structure was given by non-compositional methods [2, 7, 41]. In spirit, the closest to what we have done is [26], which does denotational cost analysis in a deterministic recursive setting, and [1], which uses a continuation-passing style transformation to reason about the expected cost of a call-by-value language.

Our contributions. The main contributions of this paper are the following

- The metalanguage **cert**, a CBPV variant with primitives for recursion, increasing cost (charge c) and for sampling from uniform distributions (uniform). (§2)
- A novel expected cost semantics based on an expected cost monad that accommodates familiar and useful reasoning principles. (§3.2)

$$
\begin{aligned}
\overline{\tau} &\coloneqq F\tau \mid \tau \to \overline{\tau} \\
\tau &\coloneqq U\overline{\tau} \mid 1 \mid \mathbb{N} \mid \mathbb{R} \mid \tau \times \tau
\end{aligned}
$$

$$
\begin{aligned}
t, u &\coloneqq \lambda x\,.\,t \mid t\ V \mid \mathsf{ifZero}\ V\ \mathsf{then}\ t\ \mathsf{else}\ u \mid \mathsf{force}\ \ V \mid (x \leftarrow t); u \\
&\mid \mathsf{produce}\ V \mid \mathsf{let}\ x\ \mathsf{be}\ V\ \mathsf{in}\ t \mid \mathsf{succ}\ t \mid \mathsf{pred}\ t \\
&\mid \mathsf{let}\ (x, y) = V\ \mathsf{in}\ t \\
V &\coloneqq x \mid () \mid n \in \mathbb{N} \mid r \in \mathbb{R} \mid \mathsf{thunk}\ \ t \mid (V_1, V_2) \\
T &\coloneqq \mathsf{produce}\ \ V \mid \lambda x\,.\,t \mid \mathsf{force}\ \ x \mid \mathsf{ifZero}\ x\ \mathsf{then}\ t\ \mathsf{else}\ u \\
C &\coloneqq [\,] \mid \lambda x\,.\,C \mid C\ V \mid \mathsf{ifZero}\ V\ \mathsf{then}\ C\ \mathsf{else}\ u \mid \mathsf{ifZero}\ V\ \mathsf{then}\ t\ \mathsf{else}\ C \\
&\mid (x \leftarrow C); u \mid (x \leftarrow t); C \mid \mathsf{let}\ x\ \mathsf{be}\ V\ \mathsf{in}\ C \mid \mathsf{succ}\ C \mid \mathsf{pred}\ C \\
&\mid \mathsf{let}\ (x, y) = V\ \mathsf{in}\ C
\end{aligned}
$$

Fig. 1. Types and Terms of CBPV

- A generalization of the effect simulation problem for cost semantics using a novel logical relations technique for denotational relational reasoning. (§4.1)
- An operational cost semantics for which we prove the expected cost semantics adequate. (§4.3)
- A universal property of the expected cost monad as the minimal submonad of the continuation monad that can accommodate subprobability distributions and cost. (§5)

We also justify the applicability of our semantics through use-cases that illustrate how the expected cost semantics can be used to reason about expected cost of stochastic processes and randomized algorithms.

2 cert : A PROBABILISTIC COST-AWARE METALANGUAGE

In this section we introduce the type system, equational theory and operational semantics of **cert**. The language is a Call-By-Push-Value (CBPV) [29] calculus extended with operations for cost, probabilistic sampling and recursion. For the sake of presentation, we introduce it in parts: we begin by going over the core CBPV calculus and then present its extensions with effectful operations and lists, respectively. The operational semantics is defined as a weighted Markov chain that accounts for the cost and output distributions of programs. It is defined using the standard measure-theoretic treatment of operational semantics for continuous distributions, cf. Vákár et al. [40].

We chose CBPV as the core of **cert** due to its type and term-level separation of values and computations. Such a separation allows for a fine grained control over the execution of programs, providing a uniform treatment of different evaluation strategies such as Call-By-Name (CBN) and Call-By-Value (CBV).

Figure 1 depicts the CBPV syntax. Note that the base types 1, $\mathbb{N}$ and $\mathbb{R}$ are value types, the product types is also a value and arrow types are computation types that receive a value type as input and a computation type as output. 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 [32], while U is used to represent suspended — or thunked — computations.

$$\frac{}{\Gamma_1, x : \tau, \Gamma_2 \vdash_v x : \tau} \qquad \frac{n \in \mathbb{N}}{\Gamma \vdash_v n : \mathbb{N}} \qquad \frac{r \in \mathbb{R}}{\Gamma \vdash_v r : \mathbb{R}} \qquad \frac{}{\Gamma \vdash_v () : 1}$$

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

$$\frac{\mathsf{op} \in O(\tau, \tau')}{\Gamma \vdash \mathsf{op} : \tau \longrightarrow F\tau'} \qquad \frac{\Gamma, x : \tau \vdash_c t : \overline{\tau}}{\Gamma \vdash_c \lambda x.\, t : \tau \longrightarrow \overline{\tau}} \qquad \frac{\Gamma \vdash_v V : \tau \qquad \Gamma \vdash_c t : \tau \longrightarrow \overline{\tau}}{\Gamma \vdash_c t\ V : \overline{\tau}}$$

$$\frac{\Gamma \vdash_v V : \tau}{\Gamma \vdash_c \mathsf{produce}\ V : F\tau} \qquad \frac{\Gamma \vdash_c t : \overline{\tau}}{\Gamma \vdash_v \mathsf{thunk}\ t : U\overline{\tau}} \qquad \frac{\Gamma \vdash_v V : U\overline{\tau}}{\Gamma \vdash_c \mathsf{force}\ V : \overline{\tau}}$$

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

Fig. 2. CBPV typing rules

This two-level approach also manifests itself at the type judgment level, where the judgment $\Gamma \vdash_v V : \tau$ 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 only introduce values to the context. The if-then-else operation checks if the guard V is 0, in which case it returns the first branch, and otherwise it returns the second branch. The language is also parametric on a set of operations O which contains arithmetic functions. 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 $F\tau$.

The syntax differs a bit from the monadic semantics of effects, but, as it is widely known, every strong monad over a Cartesian closed category can interpret the CBPV calculus, as we describe in Appendix A and described in Section 12 of [29].

In Figure 1 there is also a grammar for terminal computations and computation contexts C. These are standard and the latter represents computations with a single hole $[\,]$ that may be filled in by a computation t by a non-capture-avoiding substitution, which we denote by $C[t]$.

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 extend 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 and operational semantics.

```
fix $f : \mathsf{list}(\tau) \to F\mathbb{N}$. $\lambda l : \mathsf{list}(\tau)$.
case $l$ of
 | nil ⇒
   produce 0
 | (hd, tl) ⇒
   $n \leftarrow$ (force $f$) $tl$
   produce $(1 + n)$
```

```
fix $f : \mathsf{list}(\tau) \to F(\mathsf{list}(\tau) \times \mathsf{list}(\tau))$.
$\lambda l : \mathsf{list}(\tau)$ .
$\lambda p : \tau \to F\mathbb{N}$.
case $l$ of
 | nil ⇒
   produce (nil, nil)
 | (hd, tl) ⇒
   $n \leftarrow p\ hd$
   $(l_1, l_2) \leftarrow$ (force $f$) $p\ tl$
   if $n$ then
      produce (cons $hd\ l_1, l_2$)
   else
      produce ($l_1$, cons $hd\ l_2$)
```

Fig. 3. Length function length (left) and filter function biFilter (right).

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:

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

The operation uniform uniformly samples a real number from the interval $[0, 1]$ and fix is the familiar fixed-point operator used for defining recursive programs. In interest of reducing visual pollution and simplifying the presentation, charge $V; t$ desugars to $(x \leftarrow \mathsf{charge}\, V); t$, when x is not used in the body of t, and we will assume that the cost monoid is $\mathbb{N}$.

With the uniform distribution primitive it is possible to define uniform distributions over discrete sets which, given a natural number n, outputs a uniform distribution rand n over the set $\{0, \dots, n-1\}$. This can be desugared to the program $\lambda n.\ x \leftarrow \mathsf{uniform}; \mathsf{produce}\ (\lfloor nx \rfloor) : \mathbb{N} \to F\mathbb{N}$, where $\lfloor \cdot \rfloor$ is the floor function. Biased coins $\oplus_p$ desugar to $\lambda p.\ x \leftarrow \mathsf{uniform}; x \leq p : \mathbb{R} \to F\mathbb{N}$, where $\leq : \mathbb{R} \to \mathbb{R} \to F\mathbb{N}$ is the comparison function that returns 0 if the first argument is less or equal to the second argument and 1 otherwise.

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

$$\cdot \vdash_c \mathsf{fix}\, x.\, (\mathsf{produce}\ 0) \oplus_{0.5} ((y \leftarrow \mathsf{force}\ x); \mathsf{produce}\ (1 + y)) : F\mathbb{N},$$

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 the typing rule of recursive definitions, the variable x is a thunk, meaning that it must be forced before executing it.

By having fine-grained control over which operations have a cost, it is possible to orchestrate your program with charge c operations in order to encode different cost models. 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

$$\mathsf{fix}\, x.\, \mathsf{charge}\ 1; (\mathsf{produce}\ 0) \oplus_{0.5} (y \leftarrow \mathsf{force}\ x; \mathsf{produce}\ (1 + y)) : F\mathbb{N}$$

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:

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

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.

$$\begin{aligned}
\tau &\coloneqq \cdots \mid \mathsf{list}(\tau)\\
V &\coloneqq \cdots \mid \mathsf{nil} \mid \mathsf{cons}\, V_1\, V_2\\
t &\coloneqq \cdots \mid (\mathsf{case}\, x\, \mathsf{of}\ \mathsf{nil} \Rightarrow t \mid \mathsf{cons}\, x\, xs \Rightarrow u)\\
T &\coloneqq \cdots \mid (\mathsf{case}\, x\, \mathsf{of}\ \mathsf{nil} \Rightarrow t \mid \mathsf{cons}\, x\, xs \Rightarrow u)\\
C &\coloneqq \cdots \mid (\mathsf{case}\, x\, \mathsf{of}\ \mathsf{nil} \Rightarrow C \mid \mathsf{cons}\, x\, xs \Rightarrow u) \mid (\mathsf{case}\, x\, \mathsf{of}\ \mathsf{nil} \Rightarrow t \mid \mathsf{cons}\, x\, xs \Rightarrow C)
\end{aligned}$$

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

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

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 length that computes the length of a list and a binary version of the familiar filter function biFilter that outputs two lists, one for the true elements and one for the false elements, are, respectively, defined in the left and right parts of Figure 3. Note that since we have adopted a $\mathbb{N}$-valued if-statement, the predicate p above outputs a natural number. Furthermore, since the recursion operation adds a thunk to the context, in order to call the recursive function you must first force its execution.

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

$$\begin{array}{c}\beta\text{-LAW}\\ \dfrac{\Gamma, x : \tau \vdash_c t : \overline{\tau} \qquad \Gamma \vdash_v V : \tau}{\Gamma \vdash t\{V/x\} = (\lambda x.\ t)\ V : \overline{\tau}}\end{array} \qquad \begin{array}{c}\eta\text{-LAW}\\ \dfrac{\Gamma \vdash_c t : \tau \to \overline{\tau}}{\Gamma \vdash_c (\lambda x.\ t\ x) \ = t : \tau \to \overline{\tau}}\end{array} \qquad \begin{array}{c}0\textsc{Mon}\\ \dfrac{\Gamma \vdash_c t : \overline{\tau}}{\Gamma \vdash (\mathsf{charge}\,0; t) = t : \overline{\tau}}\end{array}$$

$$\begin{array}{c}\textsc{ActMon}\\ \dfrac{}{\Gamma \vdash \mathsf{charge}\,c; \mathsf{charge}\,d = \mathsf{charge}\,c + d : F1}\end{array} \qquad \begin{array}{c}\textsc{ThunkForce}\\ \dfrac{\Gamma \vdash_c t : \overline{\tau}}{\Gamma \vdash \mathsf{force}\ (\mathsf{thunk}\ (t)) = t : \overline{\tau}}\end{array}$$

$$\begin{array}{c}\textsc{IfZ}\\ \dfrac{\Gamma \vdash_c t : \overline{\tau} \quad \Gamma \vdash_c u : \overline{\tau}}{\Gamma \vdash \mathsf{ifZero}\,0\,\mathsf{then}\,t\,\mathsf{else}\,u = t : \overline{\tau}}\end{array} \qquad \begin{array}{c}\textsc{IfS}\\ \dfrac{\Gamma \vdash_c t : \overline{\tau} \quad \Gamma \vdash_c u : \overline{\tau}}{\Gamma \vdash \mathsf{ifZero}\,(n+1)\,\mathsf{then}\,t\,\mathsf{else}\,u = u : \overline{\tau}}\end{array}$$

$$\begin{array}{c}\textsc{Fix}\\ \dfrac{\Gamma, x : U\overline{\tau} \vdash_c t : \overline{\tau}}{\Gamma \vdash (\mathsf{fix}\,x.\,t) = t\{x/\mathsf{thunk}\ (\mathsf{fix}\,x.\,t)\} : \overline{\tau}}\end{array}$$

Fig. 4. Equational Theory (Selected Rules)

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 other rules which are standard in CBPV languages shown in Appendix A. The first two rules are the familiar β and η-rules for the arrow type, the 0Mon and ActMon rules are the monoid equations for the charge operation. The rule ThunkForce says that forcing a thunked computation is the same thing as running the computation, the rules IfZ and IfS explain how if-statements interact with natural numbers and the rule Fix is the fixed point equation that unfolds one recursive call of the recursive computation t.

2.4 Operational Semantics

Since we are interested in modeling the cost of running programs, we will define an operational cost semantics which is closer to the execution model of programs. When defining semantics for probabilistic languages with continuous distributions, one must be careful to define it so that it is a measurable function.

In this section, we begin by showing the the **cert** syntax can be equipped with a measurable space structure that makes the natural syntax operations, such as substitution, measurable. Then, the operational semantics can be defined as a Markov kernel over the syntax, as it is usually done in probabilistic operational semantics for continuous distributions [13, 40]. After defining the operational semantics, we conclude by proving the subject reduction property.

Syntax Spaces and Kernels. Before defining the operational semantics, we need some definitions from measure theory.

Definition 2.4. A measurable space is a pair $(X, \Sigma_X \subseteq \mathcal{P}(X))$, where X is a set and Σ is a σ-algebra, i.e. a collection of subsets that contains the empty set and is closed under complements and countable union.

Definition 2.5. A measurable function $f : (X, \Sigma_X) \longrightarrow (Y, \Sigma_Y)$ is a function $f : X \longrightarrow Y$ such that for every $A \in \Sigma_Y$, $f^{-1}(A) \in \Sigma_X$.

Definition 2.6. A subprobability distribution over a measurable space (X, Σ_X) is a function $\mu : \Sigma_X \longrightarrow [0, 1]$ such that $\mu(\emptyset) = 0$, $\mu(X) \leq 1$ and $\mu(\uplus_{n\in\mathbb{N}} A_n) = \sum_{n\in\mathbb{N}} \mu(A_n)$.

The operational semantics will be modeled as a (sub)Markov kernel, a generalization of transition matrices and Markov chains.

Definition 2.7. A subMarkov kernel between measurable spaces (X, Σ_X) and (Y, Σ_Y) is a function $f : X \times \Sigma_Y \longrightarrow [0, 1]$ such that

- For every $x : X$, $f(x, -) : \Sigma_Y \longrightarrow [0, 1]$ is a subprobability distribution
- For every $A : \Sigma_Y$, $f(-, A) : X \longrightarrow [0, 1]$ is a measurable function

We denote the set of computation terms by Λ, the set of values by $\mathcal{V}al$ and the set of terminal computations by T. Let t (resp. V) be a computation (resp. value), fix the term traversal order left-to-right and let $z_1, z_2, \dots z_n, \dots$ be a sequence of distinct and ordered variables disjoint from the set of term variables. The traversal order gives rise to a canonical enumeration of t's (resp. V's) occurrences of numerals $r \in \mathbb{R}$, which we denote by the sequence $r_1, r_2, \dots, r_n$. By substituting these occurrences by the variables $z_1, z_2, \dots, z_n$, we obtain the term $t\{z_1, \dots, z_n/r_1, \dots, r_n\}$ (resp. $V\{z_1, \dots, z_n/r_1, \dots, r_n\}$). Let Λ_n (resp. $\mathcal{V}al_n$) be the set of such substituted terms with exactly n numerals.

Note that the sets Λ_n and $\mathcal{V}al_n$ are countable and that there are bijections $\Lambda \cong \Sigma_{n:\mathbb{N},t:\Lambda_n}\mathbb{R}^n$ and $\mathcal{V}al \cong \Sigma_{n:\mathbb{N},t:\mathcal{V}al_n}\mathbb{R}^n$, where Σ is the dependent sum operation: for instance, every computation term t can be decomposed into a sequence of its numerals $r_1, \dots, r_n$ and substituted term $t\{z_1, \dots, z_n/r_1, \dots, r_n\}$, and, conversely, every sequence of numerals and substituted term t can be mapped to the term $t\{r_1, \dots, r_n/z_1, \dots, z_n\}$ — note the reversed order of substitution.

Therefore, we can equip Λ with the coproduct σ-algebra: $(\Lambda, \Sigma_\Lambda) = \Sigma_{n:\mathbb{N},t:\Lambda_n}(\mathbb{R}^n, \Sigma_{\mathbb{R}^n})$. More concretely, a subset $A \subset \Lambda$ is measurable if, and only if,

$$\forall n \in \mathbb{N}, t : \Lambda_n, \{(r_1, \dots, r_n) \in \mathbb{R}^n \mid t\{r_1, \dots, r_n/z_1, \dots, z_n\} \in A\} \in \Sigma_{\mathbb{R}^n}$$

The measurable space structure of $\mathcal{V}al$ is defined using a similar coproduct.

Given a pair of a context Γ and a computation type $\overline{\tau}$, the measurable spaces $\Lambda^{\Gamma\vdash\overline{\tau}}$ and $T^{\Gamma\vdash\overline{\tau}}$ are the subspaces of well-typed computations and terminal computations under context Γ and output type $\overline{\tau}$, respectively. Given a value type τ, the measurable space $\mathcal{V}al^{\Gamma\vdash\tau} \subseteq \mathcal{V}al$ is the subspace of well-typed values under context Γ and output type τ. The measurable space structures of $\Lambda^{\Gamma\vdash\overline{\tau}}$, T, $T^{\Gamma\vdash\overline{\tau}}$ and $T^{\Gamma\vdash\tau}$ are defined using the appropriate subspace σ-algebras. The following metatheoretic lemmas are useful and standard.

Lemma 2.8. *If* $\Gamma, x : \tau \vdash_c t : \overline{\tau}$ *and* $\Gamma \vdash_v V : \tau$ *then* $\Gamma \vdash_c t\{V/x\} : \overline{\tau}$.

Lemma 2.9 (c.f. Lemma 3.7 of [13])**.** *For every variable x, the function* $\cdot\{\cdot/x\} : \Lambda \times \mathcal{V}al \longrightarrow \Lambda$ *is measurable.*

Therefore, for every context Γ, computation type $\overline{\tau}$ and value type τ, the substitution function restricts to measurable functions $\cdot\{\cdot/x\} : \Lambda^{\Gamma,x:\tau\vdash\overline{\tau}} \times \mathcal{V}al^{\Gamma\vdash\tau} \longrightarrow \Lambda^{\Gamma\vdash\overline{\tau}}$.

Operational Kernels. In order to capture the cost of running a computation, we define *costful* kernels as a subMarkov kernel $X \times \Sigma_{\mathbb{N}\times Y} \longrightarrow [0,1]$. Given two costful kernels f and g, their composition $f \circ g : X \times \Sigma_{\mathbb{N}\times Z} \longrightarrow [0,1]$ is defined, with slight abuse of notation, as:

$$(g \circ f)(x, n_1 + n_2, C) = \int_Y g(y, n_2, C) f(x, n_1, -)(\mathrm{d}y)$$

In plain terms, every term reduces to a subprobability distribution over costs and terminal computations. Their composition is defined so that the probability that the composition cost is $n_1 + n_2$ is the probability that the input f will cost n_1 and the continuation g will cost n_2, i.e. the product of both events averaged out using an integral. We use the Haskell syntax $\gg=$ for kernel composition.

We can now define the operational semantics as the limit of a sequence of approximate semantics given by costful kernels $\Downarrow_n$: $\Lambda \times \Sigma_{\mathbb{N}\times T} \longrightarrow [0,1]$. The approximate semantics $\Downarrow_n$ are defined by recursion on n, where the base case is defined as $\Downarrow_0 = \bot$, i.e. the 0 measure. When $n > 0$, the recursive definition is depicted in Figure 5.

Though we are using the familiar relational definition of operational semantics, they are functional in nature. We use the notation of Vákár et al. [40], where the inference rule

$$\frac{k_1(t)w_1 \qquad k_2(t, w_1)w_2 \qquad \cdots \qquad k_n(t, w_1, \ldots, w_n)v}{l(t)f(t, w_1, \ldots, w_n, v)}$$

denotes the kernel $k_1(t) \gg= (\lambda w_1.\, k_2(t, w_1) \gg= \ldots \delta_{f(t,w_1,\ldots,w_n,v)})$. We also simplify the presentation by using *guarded* kernel composition. For instance, in the β-reduction rule, whenever t reduces to something which is not a λ-abstraction, the kernel composition loops. Besides the non-standard presentation, the semantics is a fairly standard CBPV big-step semantics [29]. For example, terminal computations cannot reduce any further, so they output a point mass distribution over 0 cost and themselves. In the case of effectful operations, the charge operation steps to a point mass distribution over () and the cost; the sample operation reduces to an independent distribution of the point mass distribution at 0 and the Lebesgue uniform measure λ on the interval $[0,1]$.

It follows by a simple induction that the semantics $\Downarrow_n$ is monotonic in n. Since the space of subprobability distributions forms a CPO, we can define the semantics as the supremum of its finite approximations $\Downarrow = \bigsqcup_n \Downarrow_n$. As usual, it is possible to prove subject reduction by induction on well-typed terms.

Lemma 2.10 (Subject reduction). *If* $\Gamma \vdash_c t : \overline{\tau}$, *then the composition* $\Downarrow \circ\, \iota : \Lambda^{\Gamma\vdash\overline{\tau}} \longrightarrow P_{\leq 1}(\mathbb{N} \times T)$ *factors as* $\Lambda^{\Gamma\vdash\overline{\tau}} \longrightarrow P_{\leq 1}(\mathbb{N} \times T^{\Gamma\vdash\overline{\tau}}) \hookrightarrow P_{\leq 1}(\mathbb{N} \times T)$, *where* $\iota : \Lambda^{\Gamma\vdash\overline{\tau}} \hookrightarrow \Lambda$ *is the inclusion function. More colloquially, well-typedness is stable under the operational semantics.*

3 DENOTATIONAL SEMANTICS

This section presents two concrete denotational semantics to our language:

- A cost semantics that serves as a denotational baseline for compositionally computing the cost distribution of probabilistic programs.
- An expected cost semantics that, while it cannot reason about as many quantitative properties of cost as the cost semantics, such as tail-bounds and higher-moments, it provides a compositional account to the *expected cost.*

Both semantics will be defined over the category $\omega\mathbf{Qbs}$ of ω-quasi Borel spaces [40], a Cartesian closed category that admits a probabilistic powerdomain of subprobability distributions $P_{\leq 1}$. 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 monad it is possible to define the expected cost to be the expected value of the cost distribution $P_{\leq 1}(\mathbb{C})$.

$$\frac{}{\text{produce } V \Downarrow_n \delta_{(0,\text{produce } V)}} \qquad \frac{t \Downarrow_n \mu}{\text{force } (\text{thunk } t) \Downarrow_n \mu} \qquad \frac{}{\text{uniform} \Downarrow_n \delta_0 \otimes \lambda} \qquad \frac{}{\lambda x.\, t \Downarrow_n \delta_{(0,\lambda x.\, t)}}$$

$$\frac{t \Downarrow_n \lambda x.\, u \qquad u\{V/x\} \Downarrow_{n-1} \mu}{t\ V \Downarrow_n \mu} \qquad \frac{}{\text{charge}\, r \Downarrow_n \delta_{(r,\text{produce } ())}}$$

$$\frac{t \Downarrow_n \text{produce } V \qquad u\{V/x\} \Downarrow_{n-1} \mu}{(x \leftarrow t); u \Downarrow_n \mu} \qquad \frac{t\{\text{thunk fix}\, x.\, t/x\} \Downarrow_{n-1} \mu}{\text{fix}\, x.\, t \Downarrow_n \mu} \qquad \frac{t \Downarrow_n \mu}{\text{ifZero } 0 \text{ then } t \text{ else } u \Downarrow_n \mu}$$

$$\frac{u \Downarrow_n \mu}{\text{ifZero } (n+1) \text{ then } t \text{ else } u \Downarrow_n \mu} \qquad \frac{t\{V_1, V_2/x_1, x_2\} \Downarrow_{n-1} \mu}{\text{let } (x_1, x_2) = (V_1, V_2) \text{ in } t \Downarrow_n \mu}$$

$$\frac{t \Downarrow_{n-1} \mu}{\text{case nil of nil} \Rightarrow t \mid \text{cons}\, x\, xs \Rightarrow u \Downarrow_n \mu} \qquad \frac{u\{V_1, V_2/x, xs\} \Downarrow_{n-1} \mu}{\text{case } (\text{cons}\, V_1\, V_2) \text{ of nil} \Rightarrow t \mid \text{cons}\, x\, xs \Rightarrow u \Downarrow_n \mu}$$

Fig. 5. Big-Step Operational Semantics

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. We start this section by going over important definitions and constructions for ω**Qbs**, we then define the cost semantics, followed by the expected cost semantics.

3.1 ω-quasi Borel spaces

We now introduce the semantic machinery used in the interpretation of **cert**. Due to requirement of higher-order functions, probability and unbounded recursion, we are somewhat limited in terms of which semantic domain to use. We use the category of ω-quasi Borel spaces, a domain-theoretic version of quasi-Borel spaces [19].

Definition 3.1 ([40]). 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 $\omega\mathbf{Qbs}$ has ω-quasi Borel spaces as objects and measurable functions as morphisms.

Theorem 3.4 (Section 3.3 of Vákár et al. [40])**.** *The category $\omega\mathbf{Qbs}$ is Cartesian closed.*

Furthermore, there is a full and faithful functor $\mathbf{Sbs} \to \omega\mathbf{Qbs}$, where $\mathbf{Sbs}$ is the category of standard Borel spaces and measurable functions (c.f. Proposition 15 [19]). More concretely, if you interpret a program that has as inputs and output standard Borel spaces, its denotation in $\omega\mathbf{Qbs}$ 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 [40], $\omega\mathbf{Qbs}$ 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 $\omega\mathbf{Qbs}$, 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 Monads. It is possible to construct probabilistic powerdomains in $\omega\mathbf{Qbs}$, 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. We are assuming familiarity with basic concepts from category theory such as monads and use the notation $(T, \eta^T, (-)^\#_T)$, where T is an endofunctor, $\eta^T_A : A \to TA$ and $(-)^\#_T : (A \Rightarrow TB) \to (TA \Rightarrow TB)$ are the unit and bind natural transformations, respectively. 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 [32]. When it is clear from the context, we will simply write η and $(-)^\#$, without the sub and superscript, respectively.

Lemma 3.5 (Section 4.5 of Vákár et al. [40])**.** *The category $\omega\mathbf{Qbs}$ admits strong commutative monads P and $P_{\leq 1}$ of probability and sub-probability distributions, respectively.*

Categorically, $P_{\leq 1}$ 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 $\mathbf{Meas}$, 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. A more detailed presentation of this construction will be given in Section 5.

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 $\int_A$.

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 3.6. 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 $f : A \to \mathbb{C} \times B$, $f^\#(c, a) = (c + (\pi_1 \circ f)(a), (\pi_2 \circ f)(a))$, where $\pi_i : A_1 \times A_2 \to A_i$ is the i-th projection.

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.

$$[\![\mathsf{charge}\ c]\!]_{CS} = \delta_{(c,())} \qquad [\![\mathsf{uniform}]\!]_{CS} = \delta_0 \otimes \lambda \qquad [\![\mathsf{fix}\,x.\,t]\!]_{CS} = \bigsqcup_n [\![t]\!]^n_{CS}\,(\bot)$$

Fig. 6. Cost semantics of operations

3.2 A probabilistic cost semantics

Contrary to the deterministic case, the cost of a probabilistic computation is not a single value, it is a distribution over 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 3.7. *If $T : C \to C$ 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 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 effectful operations, whose semantics are depicted in Figure 6, and for the cost monoid, which is interpreted as the additive natural numbers $(\mathbb{N}, 0, +)$.

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

Definition 3.8. 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 measurable subset $X \subseteq [0, \infty]$ the expected distribution formula can be restricted to distributions over X. In particular, it is possible to restrict this function to have $P_{\leq 1}(\mathbb{N})$ as its domain.

Example 3.9 (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.

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

$$[\![\mathsf{charge}\; c]\!]_{EC} = (c, \delta_{()}) \qquad [\![\mathsf{uniform}]\!]_{EC} = (0, \lambda) \qquad [\![\mathsf{fix}\, x.\, t]\!] = \bigsqcup_n [\![t]\!]^n_{EC}(\bot)$$

Fig. 7. Expected cost semantics of operations

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. In the next section we show how to encode this trick in the semantics itself, significantly simplifying the computation of the expected value.

Theorem 3.10. *(c.f. Appendix A) For every computation t and value V, $[\![t\{V/x\}]\!]_{CS} = [\![t]\!]_{CS}([\![V]\!]_{CS})$.*

Theorem 3.11. *(c.f. Appendix A) For every computation context C, if $[\![t]\!]_{CS} = [\![u]\!]_{CS}$ then $[\![C[t]]\!]_{CS} = [\![C[u]]\!]_{CS}$.*

3.3 A semantics for expected cost

The semantics just proposed can compositionally compute the cost distribution of programs, but compositionality is broken when computing its expected cost. Indeed, after computing the distribution, we must compute the expected cost of an arbitrarily complex distribution. We fix this by defining a novel expected cost monad.

We achieve this by defining a monad structure on the functor $[0, \infty] \times P_{\leq 1}$: every computation will be denoted by an extended positive real number, i.e. its expected cost, and a subprobability distribution over its output. The monad's 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. Formally, given an ω**Qbs** morphism $f : X \to [0, \infty] \times P_{\leq 1}Y$, 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. The proof can be found in Appendix F. □

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 7, 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). Unfolding the expected cost $\pi_1 \circ [\![\mathsf{geom}]\!]_{EC}$ gives us the fixed point equation $E = 1 + (1 - \frac{1}{2})E$, i.e. $E = 2$. This can be readily generalized to arbitrary $p \in [0, 1]$, giving the equation $E_p = 1 + (1 - p)E_p$.

As we have noted in the previous section, the cost semantics can be used to reason about the expected cost by using Definition 3.8. 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

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

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 state some basic definitions and lemmas that allows us to describe precisely how the cost and expected cost semantics relate.

Definition 3.15. A monad morphism is a natural transformation $\gamma : T \longrightarrow S$, where $(T, \eta^T, (-)^\#_T)$ and $(S, \eta^S, (-)^\#_S)$ are monads over the same category, such that $\gamma \circ \eta^T = \eta^S$ and $(\gamma \circ g)^\#_S \circ \gamma = \gamma \circ g^\#_T$, for every $g : A \longrightarrow TB$.

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

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 non-termination explained in the previous section. This semantics also validates the following compositionality properties.

Theorem 3.18. *(c.f. Appendix A) For every computation t and value V, $[\![t\{V/x\}]\!]_{EC} = [\![t]\!]_{EC}\,([\![V]\!]_{EC})$.*

Theorem 3.19. *(c.f. Appendix A) For every computation context C, if $[\![t]\!]_{EC} = [\![u]\!]_{EC}$ then $[\![C[t]]\!]_{EC} = [\![C[u]]\!]_{EC}$.*

4 SOUNDNESS THEOREMS

We have proposed four ways of reasoning about expected cost: by using the equational theory, the operational semantics or by using either of the denotational semantics. Something which is not clear at first is the extent of how much these semantics are reasoning about the same property. We begin this section by proving soundness properties of the expected cost semantics with respect to the denotational cost semantics by stating and proving a generalization of the *effect simulation problem*. Due to subtle interactions between cost, probability and non-termination, we provide separate analyses for the probabilistic and subprobabilitic cases.

In order to justify that the denotational semantics is reasoning about an operational notion of cost, we prove standard soundness and adequacy results of the denotational cost semantics with respect to the operational cost semantics. We also show that both cost and expected cost semantics are sound with respect to the equational theory.

4.1 Denotational Soundness

The soundness property we are interested in is the following: given a closed program $\cdot \vdash_c t : F\tau$, then the expected value for the second marginal of $[\![t]\!]_{CS}$ is equal to $\pi_1([\![t]\!]_{EC})$. In the literature, similar properties have been called the *effect simulation problem* and many semantic techniques have been developed for solving it, such as $\top\top$-lifting [25].

Unfortunately, the available categorical techniques for effect simulation have two limitations. The first one is that it only proves properties about programs with ground type context and output, c.f. Theorem 7 [25]. The second one is the lack of existence of an appropriate monad morphism $\beta : P_{\leq 1}(\mathbb{N} \times -) \longrightarrow [0, \infty] \times P_{\leq 1}$. Since we are trying to relate the expected value of the first marginal with the weight given by the expected cost monad, β would have to be equal to E, as defined above, which according to Lemma 3.17, is not a monad morphism.

We deal with both of these issues by defining a two-level logical relation. This structure mimics the value/computation types distinction present in CBPV and, when compared to previous work [25], gives stronger soundness properties.

Theorem 4.1. *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 $\pi_1([\![t]\!]^c_{EC})$. Furthermore, when restricted to the recursion-free fragment of* **cert**, *these costs are equal.*

Proof. This theorem follows by a logical relations argument that we detail in Appendix B. □

4.2 Equational Soundness

In the previous section, we have argued that in the presence of unbounded recursion, the cost semantics has some undesirable properties which are not present in 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, showing that there is still some harmony between them.

Theorem 4.2. *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. Once again, the full equational theory is shown in Figure 13 and the full proof can be found in Appendix F. □

It is useful to understand the extent in which these equational theories differ. For instance, the cost semantics validates the equation $\bot; t = \bot = t; \bot$. 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 $\mathsf{fix}\, x.\, \mathsf{charge}\, 1; x = \bot$. 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 4.3 (c.f. Appendix F). *The cost semantics validates the commutativity equation.*

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 increases the cost by c and then diverges.

When it comes to modeling the cost of real programs, these two programs should not be the same since, for instance, if the charge operation is modeling a monetary cost, such as a call to an API, only the second program will cost something. Fortunately, the expected cost monad is not commutative, making it a more physically-justified monad.

Lemma 4.4. *The monad $[0, \infty] \times P_{\leq 1}$ is not commutative.*

Proof. When $c > 0$, the terms above are a counterexample: $[\![t]\!]_{EC} = (0, 0) \neq (c, 0) = [\![u]\!]_{EC}$ □

That being said, assuming that the program t in the commutativity equation has no cost and terminates with probability 1, the expected cost semantics validates the commutativity equation, and maximally so.

Theorem 4.5. *The commutativity equation is sound in the expected cost model if, and only if, t terminates with probability 1 and has 0 expected cost.*

Proof. In Appendix F, we prove this by showing that the probability monad P is the center of $[0,\infty] \times P_{\leq 1}$ [6], a concept generalizing the center of monoids and algebraic theories [43]. □

4.3 Operational Soundness and Adequacy

We conclude this section by proving some metatheoretic properties of the operational semantics and show how it relates to the denotational cost semantics. The main results of this section is the adequacy of the operational semantics with respect to the cost semantics and the cost adequacy theorem of the operational semantics with respect to the expected cost semantics.

A consequence of Lemma 2.10 is that it becomes possible to compose the operational and denotational semantics and obtain morphisms $[\![\Downarrow]\!]^{\Gamma\vdash\overline{\tau}} : \Lambda^{\Gamma\vdash\overline{\tau}} \to ([\![\Gamma]\!]_{CS} \to [\![\overline{\tau}]\!]_{CS})$, as explained in Section 6.7 of Vákár et al. [40]. In particular, for closed programs, $[\![\Downarrow]\!]^{\cdot\vdash\overline{\tau}} : \Lambda^{\cdot\vdash\overline{\tau}} \to ([\![\overline{\tau}]\!]_{CS})$. We now state the soundness theorem.

Theorem 4.6 (Soundness). *For every closed computation $\cdot \vdash_c t : \overline{\tau}$, $[\![\Downarrow(t)]\!]_{CS} \leq [\![t]\!]_{CS}$.*

Proof. As usual, since the operational semantics is defined as the supremum of $\Downarrow_n$, the proof follows by induction on n and t. The proof can be found in Appendix C. □

The next step is proving the adequacy theorem which we now state.

Theorem 4.7 (Adequacy). *For every closed computation $\cdot \vdash_c t : F\tau$, $[\![t]\!]_{CS} \leq [\![\Downarrow(t)]\!]_{CS}$.*

Proof. The proof follows by a logical relations argument and can be found in Appendix D. □

Corollary 4.8 (Cost Adequacy). *For every closed computation $\cdot \vdash_c t : F\tau$, where τ is a ground type,*

$$\pi_1([\![t]\!]_{EC}) \leq \mathbb{E}(P_{\leq 1}(\pi_1)([\![\Downarrow(t)]\!]_{CS})$$

Proof. This theorem is a consequence of the adequacy theorem above and Corollary B.9. □

5 PRE-EXPECTATION SOUNDNESS

We conclude the technical development of the semantics by proving a canonicity property of the expected cost monad with respect to the pre-expectation monad — frequently used for expected cost analysis [1, 23]. The main theorem of this section requires more sophisticated categorical machinery [24] which will not impact the other sections of the paper. In this section we show that the expected cost monad is the minimal submonad of the continuation monad $(X \to [0,\infty]) \to [0,\infty]$, hereafter denoted as $K_{[0,\infty]}$, that can accommodate subprobability distribution and expected cost analysis.

One of the major strengths of using continuation monads for modeling effects is that they are extremely flexible and, therefore, can encode a wide variety of impure computations by choosing appropriate response types. Unfortunately, when using such general semantics, one loses the structures and invariants of "tailor made" semantics, making reasoning about programs less direct.

In fact, indirections caused by continuation semantics are well-known in the compiler literature, where the administrative reductions added by continuation-passing style makes it harder to recover information from the source program. Indeed, some authors [9, 31] advocate against the use of continuation-passing style precisely because of this obfuscation.

In the context of expected-cost analysis, one such indirection can be seen in Avanzini et al. [3], where the soundness of their cost analysis hinges on proving that the interpretation of programs can be rewritten as the sum of their expected cost and an integration operator — i.e. a (sub)probability distribution. Their proof relies on syntactic invariants of their language and does not hold for the entire continuation semantics, making it brittle when it comes to language extensions. In contrast, our semantics validates a similar decomposition property by definition of the expected cost monad.

The canonicity property is established through the language of factorization systems. We begin by reviewing the theory of factorization systems for monad morphisms and go over how such a structure is used to construct the statistical powerdomain in ω**Qbs**. We then show how this construction can be extended to the expected cost monad. As an application, we conclude this section by providing a more robust and conceptual proof of Lemma 7.1 of Avanzini et al. [3].

Monad Structures and Factorization Systems. Factorization systems capture classes of morphisms that can uniquely factor any morphism. The most familiar example is that of injections and surjections, where every function can be uniquely factored as the composition of a surjection followed by an injection. However, in order to properly generalize this idea, one needs to be a bit more careful:

Definition 5.1 (e.g. Section 5.5.1 of Borceux [5])**.** A (orthogonal) factorization system over a category C is a pair $(\mathcal{E}, \mathcal{M})$, where both $\mathcal{E}$ and $\mathcal{M}$ are classes of morphisms of C such that

- Every isomorphism belongs to both $\mathcal{E}$ and $\mathcal{M}$
- Both classes are closed under composition
- For every $e : \mathcal{E}$, $m : \mathcal{M}$ and morphisms f and g such that the appropriate diagram commutes, there is a unique h making the following diagram commute:

$$\begin{tikzcd}
A \arrow[r, "f"] \arrow[d, "e"'] & B \arrow[d, "m"] \\
C \arrow[r, "g"'] \arrow[ur, dotted, "h"] & D
\end{tikzcd}$$

- Every morphism f in C can be factored as $f = m \circ e$, where $m : \mathcal{M}$ and $e : \mathcal{E}$.

Note that under these axioms, the factorization is guaranteed to be unique up-to isomorphism. If every morphism in $\mathcal{M}$ is monic, then the factorization system is said to be *epi-mono*.

Example 5.2. As mentioned above, in the category **Set**, the pair $(\{f \mid f \text{ is surjective}\}, \{f \mid f \text{ is injective}\})$ is a factorization system. Given a function $f : A \to B$, its factorization is $A \to f(A) \to B$, where $f(A)$ is the image of f and the function $f(A) \to B$ is the set inclusion.

Example 5.3 (Example 2.4 of Kammar and McDermott [24])**.** In the category **CPO**, the pair $(\{f \mid f \text{ has dense image}\}, \{f \mid f \text{ is order-reflecting}\})$ is a factorization system.

A natural extension of the factorization system above can be defined for the category ω**Qbs**. We now recall some definitions and a theorem from [24] that play important role in how the probability monad in ω**Qbs** is defined.

Definition 5.4. A monad structure is an endofunctor S equipped with the monad natural transformations η and $(-)^{\#}$ but without the necessity of satisfying the monad laws.

Definition 5.5. A monad structure morphism between monad structures S_1 and S_2 over the same categories is a natural transformation $S_1 \to S_2$ such that the monad morphism laws hold. We also call them *premonad morphisms.*

Theorem 5.6 (Th. 2.5 of Kammar and McDermott [24][2])**.** *Let $(\mathcal{E}, \mathcal{M})$ be an epi-mono factorization structure, S a monad structure, T a monad and $\gamma : S \to T$ a premonad morphism. If the factorization system is closed under S then γ can be factored as $S \to M \to T$, where M is a monad, both morphisms are premonad morphisms and each component of these morphisms are elements of $\mathcal{E}$ and $\mathcal{M}$.*

[2] The proof is explained right after Th. 2.6.

This theorem was used by Vákár et al. [40] when defining their statistical powerdomain as the factor of a monad morphism into the pre-expectation monad $K_{[0,\infty]}$. Intuitively, the monad structures we are interested in are those where the monad laws hold up-to an equivalence relation and the factored monads are, roughly, the quotiented monad structures and satisfy a minimality universal property given by the uniqueness of factorizations.

Canonicity. We conclude this section by showing that the expected cost monad is the minimal monad that can accommodate cost and subprobability distributions. We extend the analysis done by Vákár et al. [40] on using Theorem 5.6 to define and prove the minimality of their statistical powerdomain. We now review its construction and then apply it to our semantics.

The "random elements" functor $[0,1] \to -_\perp$, where $(-)_\perp$ is the partiality monad, can be equipped with a monad structure similar to, but distinct from, the reader monad, as explained in Section 4.2 of [40]. The subprobability monad in ω**Qbs** is then defined using the following lemma:

Lemma 5.7. *There is a premonad morphism* $([0,1] \to -_\perp) \to K_{[0,1]}$.

Proof. The proof is nearly identical to Sec. 4.2 of Vákár et al. [40], with the exception that they are using a different, but analog, monad structure. □

The subprobability monad $P_{\leq 1}$ is defined as the factorization of the premonad morphism above as $([0,1] \to -_\perp) \overset{\psi}{\twoheadrightarrow} P_{\leq 1} \hookrightarrow K_{[0,1]}$. The component of the natural transformation above maps a pair $(f : [0,1] \to A_\perp, g : A \to [0,1])$ to $\int_{\mathrm{dom}(f)} (g \circ f)\, \mathrm{d}\mathcal{U}$, where $\mathcal{U}$ is the Lebesgue uniform measure on $[0,1]$ and $\mathrm{dom}(f)$ is the domain of f.

In analogy with the previous results, we can now prove a similar lemma for the expected cost measure monad.

Lemma 5.8. *There is a monad structure on the functor* $[0,\infty] \times ([0,1] \to -_\perp)$ *and a premonad morphism* $\gamma : [0,\infty] \times ([0,1] \to -_\perp) \to K_{[0,\infty]}$.

Proof. There are similarities between the expected cost monad and the monad structure which is given by $\eta(a) = (0, \lambda r.\, a)$ and $f^\#(r,g) = \left(r + \int_{\mathrm{dom}(g)} (\pi_1 \circ f \circ g)\, \mathrm{d}\mathcal{U}, (\pi_2 \circ f)^\#_{[0,1]\to -_\perp}(g)\right)$. The components of the monad structure morphism are defined as $\gamma_A(r,f) = \lambda k.\, r + \int (k \circ f)\, \mathrm{d}\mathcal{U}$. The proof that this is indeed a monad morphism follows by unfolding the definitions and Lemma 5.7. □

By Theorem 5.6, and the fact that the factorization system in ω**Qbs** is stable under products, the premonad morphism γ above can be factored as $[0,\infty] \times ([0,1] \to -_\perp) \to M \to K_{[0,\infty]}$.

We now show the sense in which $[0,\infty] \times P_{\leq 1}$ is canonical.

Theorem 5.9. *The factor M is isomorphic to the monad* $[0,\infty] \times P_{\leq 1}$.

Proof. The proof can be found in Appendix F. □

The canonicity of the expected cost monad is given by it being the smallest submonad of the continuation monad that can accommodate expected cost and subprobability distributions. A consequence of this fact is the following:

Corollary 5.10. *There is a monad morphism* $\varphi : [0,\infty] \times P_{\leq 1} \to K_{[0,\infty]}$ *which is component-wise order-reflecting.*

Intuitively, while the (sub)probability monad corresponds to the linear continuation monad, the expected cost monad corresponds to the affine continuation monad. An application of the results of this section is that it is possible to recover a purely denotational proof of Lemma 7.1 of [3] that is more robust with respect to language extensions.

```
ET = fix f : ℕ → F1. λn : ℕ.
if n then
   produce ()
else
   (force f) (n − 1);
   charge 1;
   (produce () ⊕ (force f) n)
```

$$[\![\mathrm{ET}]\!]_{EC} = \bigsqcup_n F^n(\bot), \text{ where}$$

$$\begin{aligned}
F = {} & \lambda\langle T_1, T_2\rangle.\, \lambda n. \\
& \quad \text{if } n \text{ then} \\
& \quad\quad (0, \delta_{()}) \\
& \quad \text{else} \\
& \quad\quad (E_1, E_2)
\end{aligned}$$

$$E_1 = T_1(n-1) + \mid T_2(n-1) \mid (1 + \frac{1}{2}T_1(n))$$

$$E_2 = \frac{\mid T_2(n-1) \mid}{2}(\delta_{()} + T_2(n))$$

Fig. 8. Expected coin tosses and its expected cost semantics.

Theorem 5.11. *For every program* $\cdot \vdash_c t : F\mathbb{R}$, $[\![t]\!]_{pre} = \varphi([\![t]\!]_{EC})$. *More explicitly,* $[\![t]\!]_{pre}(f) = \pi_1([\![t]\!]_{EC}) + \int f \,\mathrm{d}(\pi_2([\![t]\!]_{EC}))$

Proof. The proof is a variation of Theorem 7 in Katsumata [25]. □

Furthermore, by using the fact that the monad morphism is order-reflecting and, therefore, an injection, we can also prove, for ground type closed computations t_1 and t_2, if $[\![t_1]\!]_{pre} = [\![t_2]\!]_{pre}$, then $[\![t_1]\!]_{EC} = [\![t_2]\!]_{EC}$. However, for non-ground types, this result is false. For instance, the weak commutativity equation of Theorem 4.5 is not validated by the continuation monad (c.f. Example 5.11 of Carette et al. [6]).

6 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 randomized algorithms and recursive stochastic processes, illustrating the versatility of **cert**. Due to space constraints, other examples and a more detailed analysis of these examples can be found in Appendix E.

6.1 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 probabilistic program in Figure 8, where its expected cost semantics is also written down.

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 ⊕, respectively.

The denotational semantics of this program is also shown in Figure 8. We use the notation $\langle T_1, T_2\rangle$ to denote the pairing of functions $T_1 : \mathbb{N} \to [0, \infty]$ and $T_2 : \mathbb{N} \to P_{\leq 1}(1)$. The cost component of the denotation adds the cost of running the program with $n-1$ as input, while its distribution part must multiply the continuation, which costs 1 for the charge operation plus $\frac{1}{2}T_1(n)$, for the

```
$\mathsf{fix}\, f : \mathsf{list}(\tau) \longrightarrow F(\mathsf{list}(\tau)).$
$\lambda l : \mathsf{list}(\tau).$
$\lambda pred : \tau \times \tau \longrightarrow F\mathbb{N}.$
case $l$ of
 | nil $\Rightarrow$
   produce nil
 | (hd, tl) $\Rightarrow$
   $len \leftarrow length\, l$
   $r \leftarrow$ rand $len$
   $(pivot, l') \leftarrow$ nthDrop $r\, l$
   $(l_1, l_2) \leftarrow biFilter\ pred\ l'$
   $less \leftarrow$ force $f\ l_1$
   $greater \leftarrow$ force $f\ l_2$
   produce $(less +\!\!+\, pivot :: greater)$
```

(a) Randomized parametric quicksort

```
$qck_{\mathbb{N}} = \mathsf{fix}\, f : \mathbb{N} \longrightarrow F\mathbb{N}.\, \lambda n : \mathbb{N}.$
if $n$ then
  produce 0
else
  charge $(n - 1)$
  $x \leftarrow$ rand $n$
  (force $f$) $x$
  (force $f$) $(n - x - 1)$
  produce $n$
```

(a) Natural number quicksort

Fig. 10. Quicksort algorithms

recursive call. The distribution part T_2 is the usual probabilistic semantics. Before we reason about the cost of this program, we must show that it terminates with probability 1.

Lemma 6.1. *For every $n : \mathbb{N}$, $\pi_2 \circ \llbracket ET \rrbracket_{EC}(n) = \delta_{()}$, i.e. it terminates with probability* 1.

Proof. The proof follows by induction on n. When $n = 0$, $\llbracket ET \rrbracket_{EC}(0) = (0, \delta_{()})$. For the inductive case, assume $n > 0$. Unfolding the recursive definition gives us

$$(\pi_2 \circ \llbracket ET \rrbracket_{EC}(n + 1)) = \frac{|\llbracket ET \rrbracket_{EC}(n)|}{2}(\delta_{()} + (\pi_2 \circ \llbracket ET \rrbracket_{EC}(n + 1)),$$

by the induction hypothesis $|\llbracket ET \rrbracket_{EC}(n)| = 1$, which gives us that $\pi_2 \circ \llbracket ET \rrbracket_{EC}(n + 1) = \delta_{()}$. □

By unfolding the semantics and using the lemma above, we get the following recurrence relation:

$$T_1(0) = 0$$

$$T_1(n + 1) = 1 + T_1(n) + \frac{1}{2}T_1(n + 1)$$

Which, by inspection, has the closed-form solution $T_1(n) = 2(2^n - 1)$.

6.2 Randomized Quicksort

In Figure 10 we present a program that implements a randomized quicksort parametric on a total order on the type τ. For the purposes of this analysis, we assume that the input list only has distinct elements.

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 the cost is given as a function mapping list lengths to cost.

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 $qck_{\mathbb{N}} : \mathbb{N} \to \mathbb{R} \times P_{\leq 1}\mathbb{N}$ using the program in Figure 14 that corresponds to the quicksort structure assuming that the input is a natural number.

Lemma 6.2 (c.f. Appendix F). *Assuming that $p : \tau \times \tau \to F\mathbb{N}$ is a total order and the the input list has no repetitions, the following program equation holds.*

$$\begin{array}{ll} n \leftarrow \mathsf{length}\, l & l' \leftarrow \mathsf{quicksort}\,(\mathsf{charge}\, 1; p)\, l \\ \mathsf{qck}_{\mathbb{N}}\, n & \mathsf{length}\, l' \end{array} \quad = $$

We now conclude our analysis by using the program equation above and the following lemma which is proved by induction on n.

Lemma 6.3. *For every $n : \mathbb{N}$, $(\pi_2 \circ \mathsf{qck}_{\mathbb{N}})(n) = \delta_n$, i.e. it terminates with probability* 1.

By unfolding the definition of $\pi_1 \circ [\![qck_{\mathbb{N}}]\!]_{EC}$, we obtain the following expression $\mathsf{fix}\,(f \mapsto n \mapsto \mathsf{ifZero}\, n\, \mathsf{then}\, 0\, \mathsf{else}\, (n-1) + \sum_i \frac{1}{n}(f(i) + f(n-i)))$ which can be further simplified to the recurrence relation:

$$T(0) = 0$$

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

This allows us to conclude that quickSort has an expected cost of $O(n\log(n))$, which can be proved by induction and using the observation that for monotonic functions f, $\sum_{i=1}^{n-1} f(i) \leq \int_1^n f(x)\,\mathrm{d}x$.

6.3 Stochastic Convex Hull

We conclude this section by going over a stochastic convex hull algorithm [15]. To the best of our knowledge, this example is outside of reach of other logic/PL approaches to expected cost analysis, due to its combination of continuous distributions, modular cost-probability interaction and non-trivial probabilistic reasoning. This stochastic variant of the algorithm has linear expected cost analysis.

In Figure 11 we go over the implementation of this algorithm in **cert**. Its main body convexHull has three components. First, a list of points is sampled independently and uniformly from the square $[0,1]^2$, then a preprocessing stage sieve removes points that "obviously" are not in the convex hull and then we call the traditional Graham scan algorithm — c.f. Section 3.3.2 of Preparata and Shamos [37].

For the sake of presentation, we make a few simplifying assumptions: we assume that there is an operation iQ that checks to see if a point is inside a given convex quadrilateral and we assume that there is an operation clockOrNot $p_1\, p_2\, p_3$ that checks whether the lines p_1p_2 and p_2p_3 turn counterclockwise or not. Finally, we assume the existence of a parametric binary total order relation $\sqsubseteq : \mathbb{R}^2 \to \mathbb{R}^2 \to \mathbb{R}^2 \to F\mathbb{N}$ such that $\sqsubseteq\ p$ orders two points comparing their angle between the x-axis and their respective straight line with p at the origin.

Theorem 6.4. *(c.f. Appendix E) The stochastic convex hull algorithm has linear expected run time.*

Proof. Due to space constraints, we have moved the careful analysis to the appendix. Something quite appealing about this semantics is that the analysis of this algorithms follows quite closely its "textbook" analysis. □

```
unifList = fix $f.\lambda n.$
ifZero $n$ then
   produce nil
else
   $l \leftarrow$ unifList $(n-1)$
   $x \leftarrow$ uniform
   $y \leftarrow$ uniform
   produce (cons $(x, y)$ $l$)

sieve = $\lambda l.$
$p_1 \leftarrow \min(\lambda(x,y).\ x+y)\ l$
$p_2 \leftarrow \min(\lambda(x,y).\ x-y)\ l$
$p_3 \leftarrow \min(\lambda(x,y).\ -x+y)\ l$
$p_4 \leftarrow \min(\lambda(x,y).\ -x-y)\ l$
filter (charge 1; iQ $p_1\ p_2\ p_3\ p_4$) $l$
```

```
scan = fix $f\ \lambda l.\ \lambda stk.$
charge 1
case $(l, stk)$ of
 | nil, _ $\Rightarrow$ produce $stk$
 | $(x :: xs)$, nil $\Rightarrow f\ xs\ [x]$
 | $(x :: xs), [p] \Rightarrow f\ xs\ [x, p]$
 | $(x :: xs), (p1 :: p2 :: ps) \Rightarrow$
   if clockOrNot $p_2\ p_1\ x$ then
      $f\ (x :: xs)\ (p2 :: ps)$
   else
      $f\ xs\ (x :: stk)$
```

```
graham = $\lambda l.$
$p \leftarrow$ min_xy $l$
$l' \leftarrow$ quicksort (charge 1; $\sqsubseteq\ p$)$l$
scan $l'$ nil

convexHull = $\lambda n.$
$l \leftarrow$ unifList $n$
$l' \leftarrow$ sieve $l$
graham $l'$
```

Fig. 11. Stochastic convex hull algorithm

7 RELATED WORK

Type Theories for Cost Analysis. Recent work [17, 35] have developed (in)equational theories for reasoning about costs of programs inside a modal dependently-typed CBPV metalanguage. Their framework can reason about monadic effects by using the writer monad transformer, similarly to **cert**'s cost semantics, but, due to being inside a total dependent type theory, it can only represent total programs and finitely supported distributions— i.e. it cannot represent the geometric distribution. Furthermore, it is unclear how one would go about extending their framework with continuous distributions.

Other work has focused in designing type theories for doing relational reasoning of programs [8, 38]. 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 common functional idioms like 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, which in the context of type-based reasoning is an important property to have.

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 handle functional programming, though their language is restricted to a CBV semantics and does not validate useful program equations, such as Theorem 4.5. Furthermore, using continuation-passing style to reason about programs creates undesired gaps between the transformed and original programs which are avoided when using the expected cost monad's direct-style reasoning.

Automatic Resource Analysis. One fruitful research direction has been the automatic amortized resource analysis (AARA) [20, 21, 34] which uses a type system to annotate programs with their cost and automatically infer the cost of the program. These techniques have been extended to reason about recursive types [18], probabilistic programs [41] and programs with local state [30].

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 [22] in deterministic programs, though it is still unclear if the same technique can be extended to the probabilistic setting, meaning that they cannot analyze 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 [42]. This language allows users to annotate type signatures with cost-bounds and the type checking algorithm will 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, [39] provides a language for representing probabilistic recurrence relations and a tool for analyzing 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.

Leutgeb el al. [28] 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 develop a weakest pre-condition calculus for reasoning about the expected cost of programs. Again, they can only reason about first-order imperative programs.

8 CONCLUSION

In this work we have presented **cert**, a metalanguage for reasoning about expected cost of recursive probabilistic programs. It extends the existing work of [26] to the probabilistic setting. We have proposed two different semantics, 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.

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] Francis Borceux. 1994. *Handbook of categorical algebra: volume 1, Basic category theory.* Vol. 1. Cambridge University Press.

[6] Titouan Carette, Louis Lemonnier, and Vladimir Zamdzhiev. 2023. Central submonads and notions of computation: Soundness, completeness and internal languages. In *Logic in Computer Science (LICS).*

[7] 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).*

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

[9] Youyou Cong, Leo Osvald, Grégory M Essertel, and Tiark Rompf. 2019. Compiling with Continuations, or without? Whatever. *Proceedings of the ACM on Programming Languages* 3, ICFP (2019), 1–28.

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

[11] 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).*

[12] 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.*

[13] Thomas Ehrhard, Michele Pagani, and Christine Tasson. 2017. Measurable cones and stable, measurable functions: a model for probabilistic higher-order programming. *Proceedings of the ACM on Programming Languages* POPL (2017).

[14] Andrzej Filinski. 1996. *Controlling Effects.* Ph. D. Dissertation. School of Computer Science, Carnegie Mellon University.

[15] Mordecai Golin and Robert Sedgewick. 1988. Analysis of a simple yet efficient convex hull algorithm. In *Proceedings of the fourth annual symposium on Computational geometry.*

[16] Ronald L. Graham. 1972. An efficient algorithm for determining the convex hull of a finite planar set. *Inform. Process. Lett.* (1972).

[17] Harrison Grodin, Yue Niu, Jonathan Sterling, and Robert Harper. 2023. Decalf: A Directed, Effectful Cost-Aware Logical Framework. In *Principles of Programming Languages (POPL).*

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

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

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

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

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

[23] 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).*

[24] Ohad Kammar and Dylan McDermott. 2018. Factorisation systems for logical relations and monadic lifting in type-and-effect system semantics. *Electronic notes in theoretical computer science* (2018).

[25] Shin-ya Katsumata. 2013. Relating computational effects by $\top\top$-lifting. *Information and Computation* (2013).

[26] 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).*

[27] 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).*

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

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

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

[31] Luke Maurer, Paul Downen, Zena M Ariola, and Simon Peyton Jones. 2017. Compiling without continuations. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 482–494.

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

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

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

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

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

[37] Franco P Preparata and Michael I Shamos. 2012. *Computational geometry: an introduction*. Springer Science & Business Media.

[38] 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)*.

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

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

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

[42] 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)*.

[43] Gavin C. Wraith. 1970. Algebraic Theories. Lecture notes, Aarhus University.

A MONADIC SEMANTICS OF CBPV

Let C be a Cartesian closed category and $T : C \longrightarrow C$ 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 $\mu : T^2 \longrightarrow T$, the multiplication, replaces the bind natural transformation. The monad laws under this definition become:

$$\begin{tikzcd}
T^3 \arrow[r, "T\mu"] \arrow[d, "\mu_T"'] & T^2 \arrow[d, "\mu"] \\
T^2 \arrow[r, "\mu"'] & T
\end{tikzcd}
\qquad
\begin{tikzcd}
T \arrow[r, "T\eta"] \arrow[dr, "1"'] & T^2 \arrow[d, "\mu"] & T \arrow[l, "\eta_T"'] \arrow[dl, "1"] \\
& T &
\end{tikzcd}$$

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 C object and $\alpha : TA \longrightarrow A$ is a morphism, such that

$$\begin{tikzcd}
A \arrow[r, "\eta_A"] \arrow[dr, "id_A"'] & TA \arrow[d, "\alpha"] \\
& A
\end{tikzcd}
\qquad
\begin{tikzcd}
T^2A \arrow[r, "\mu_A"] \arrow[d, "T\alpha"'] & TA \arrow[d, "\alpha"] \\
TA \arrow[r, "\alpha"'] & A
\end{tikzcd}$$

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, where the algebra axioms follow from the monad laws.

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

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

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

The idea is that values are interpreted as objects in 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:

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

$$\begin{aligned}
\llbracket F\tau \rrbracket^c &= (T \llbracket \tau \rrbracket^v, \mu_{\llbracket \tau \rrbracket^v}) \\
\llbracket \tau \longrightarrow \overline{\tau} \rrbracket^c &= (\llbracket \tau \rrbracket^v \Rightarrow \llbracket \overline{\tau} \rrbracket^c, \alpha_{\llbracket \tau \rrbracket^v \to \llbracket \overline{\tau} \rrbracket^c})
\end{aligned}$$

$$\textsc{Var}\ \frac{}{\Gamma_1 \times (\tau \times \Gamma_2) \xrightarrow{! \times \pi_1} \tau} \qquad \textsc{If}\ \frac{\Gamma \xrightarrow{V} \mathbb{N} \quad \Gamma \xrightarrow{t} \overline{\tau} \quad \Gamma \xrightarrow{u} \overline{\tau}}{\Gamma \xrightarrow{\langle id;V \rangle;[t,(!;u)]} \overline{\tau}} \qquad \textsc{Abstraction}\ \frac{\Gamma \times \tau \xrightarrow{t} \overline{\tau}}{\Gamma \xrightarrow{\Lambda_\Gamma;\tau \Rightarrow t} \tau \Rightarrow \overline{\tau}} \qquad \textsc{Application}\ \frac{\Gamma \xrightarrow{V} \tau \quad \Gamma \xrightarrow{t} : \tau \to \overline{\tau}}{\Gamma \xrightarrow{\langle t,V \rangle;ev} : \overline{\tau}}$$

$$\textsc{Produce}\ \frac{\Gamma \xrightarrow{V} \tau}{\Gamma \xrightarrow{V;\eta_\tau} T\tau} \qquad \textsc{Sequencing}\ \frac{\Gamma \xrightarrow{t} T\tau' \qquad \Gamma \times \tau' \xrightarrow{u} (\overline{\tau}, \alpha_{\overline{\tau}})}{\Gamma \xrightarrow{\langle id_\Gamma,t \rangle;st;Tu;\alpha_{\overline{\tau}}} (\overline{\tau}, \alpha_{\overline{\tau}})} \qquad \textsc{Thunk}\ \frac{\Gamma \xrightarrow{t} (\overline{\tau}, \alpha_{\overline{\tau}})}{\Gamma \xrightarrow{t} \overline{\tau}} \qquad \textsc{Force}\ \frac{\Gamma \xrightarrow{V} \overline{\tau}}{\Gamma \xrightarrow{V} (\overline{\tau}, \alpha_{\overline{\tau}})}$$

$$\textsc{Let}\ \frac{\Gamma \xrightarrow{V} \tau' \qquad \Gamma \times \tau' \xrightarrow{t} \overline{\tau}}{\Gamma \xrightarrow{\langle id_\Gamma,V \rangle;t} \overline{\tau}} \qquad \textsc{Pair}\ \frac{\Gamma \xrightarrow{t_1} \tau_1 \quad \Gamma \xrightarrow{t_2} \tau_2}{\Gamma \xrightarrow{\langle t_1,t_2 \rangle} \tau_1 \times \tau_2} \qquad \textsc{Unpair}\ \frac{\Gamma \xrightarrow{V} \tau_1 \times \tau_2 \quad \Gamma \times \tau_1 \times \tau_2 \xrightarrow{t} \overline{\tau}}{\Gamma \xrightarrow{\langle id,V \rangle;t} \overline{\tau}}$$

Fig. 12. CBPV monadic semantics

It is also possible to give semantics to the terms of the language as depicted in Figure 12. 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.

This semantics validates the following compositionality properties. Note that we assume, without loss of generality, that the free variable x is the first one in the context.

Theorem A.5. *For every computation* $x : \tau', \Gamma \vdash_c t : \overline{\tau}$ *and values* $x : \tau', \Gamma \vdash_v V : \tau, \Gamma \vdash_v V' : \tau'$, $[\![t\{V/x\}]\!] = [\![t]\!] \circ \langle [\![V]\!], id \rangle$ *and* $[\![V'\{V/x\}]\!] = [\![V']\!]_{CS} \circ \langle [\![V]\!], id \rangle$.

Proof. The proof follows by mutual structural induction on the typing derivations of t and V'.

Variable : This case follows on case analysis on whether the substituted variable is equal to the term or not.

Pair : Using the equality $(V_1, V_2)\{V/x\} = (V_1\{V/x\}, V_2\{V/x\})$, the induction hypothesis and the universal property of products, we can conclude.

Unpair : Using the equality let $(y_1, y_2) = V'$ in $t\{V/x\} =$ let $(y_1, y_2) = V'\{V/x\}$ in $t\{V/x\}$, the induction hypothesis and the bifunctoriality of the Cartesian product, we can conclude.

If : Follows from the universal property of coproducts, the induction hypothesis, the naturality of of the diagonal morphism $A \to A \times A$ and the substitution definition.

Constants : Follows by unfolding definitions and using the fact that constants do not have free variables.

Thunk : Using the equality thunk $t\{V/x\} =$ thunk $(t\{V/x\})$, the definition $[\![\text{thunk } t]\!]^v = [\![t]\!]^c$ and the inductive hypothesis we can conclude.

Force : Using the equality force $V'\{V/x\} =$ force $(V'\{V/x\})$, the definition $[\![\text{force } V']\!]^v = [\![V']\!]^c$ and the inductive hypothesis we can conclude.

Produce : This case follows from the equality $(\text{produce } V')\{V/x\} = \text{produce } (V'\{V/x\})$, the definition $\llbracket \text{produce } V' \rrbracket^c = \eta \circ \llbracket V' \rrbracket^v$ and the inductive hypothesis.

Let : Using the equation $(\text{let } y \text{ be } V' \text{ in } t)\{V/x\} = \text{let } y \text{ be } (V'\{V/x\}) \text{ in } (t\{V/x\})$ and the induction hypothesis we can write the following equations:

$$\begin{aligned}
&\llbracket \text{let } y \text{ be } (V'\{V/x\}) \text{ in } (t\{V/x\}) \rrbracket = \\
&\llbracket (t\{V/x\}) \rrbracket \circ \langle \llbracket (V'\{V/x\}) \rrbracket, id \rangle = \\
&(\llbracket t \rrbracket \circ \langle \llbracket V \rrbracket, id \rangle) \circ \langle \llbracket V' \rrbracket \circ \langle \llbracket V \rrbracket, id \rangle, id \rangle = \\
&\llbracket t \rrbracket \circ \langle \llbracket V' \rrbracket, id \rangle \circ \langle \llbracket V \rrbracket, id \rangle
\end{aligned}$$

Note that the last equation holds up-to the symmetric natural isomorphisms $A \times B \cong B \times A$.

Abstraction : This case follows by using the equality $\lambda y.\ t\{V/x\} = \lambda y.\ (t\{V/x\})$, the induction hypothesis and the naturality of the Cartesian closed isomorphism $C(A \times B, C) \cong C(A, B \Rightarrow C)$.

Application : This case follows from the equality $(t\ V')\{V/x\} = (t\{V/x\})\ (V'\{V/x\})$, the induction hypothesis and the universal property of Cartesian products.

Sequencing : This case follows from the equation $y \leftarrow t; u\{V/x\} = y \leftarrow (t\{V/x\}); (u\{V/x\})$, the induction hypothesis and the naturality of the monad strength. These are summarized by the commutative diagram below.

$$\begin{tikzcd}
\Gamma \arrow[r, "{\langle V, id \rangle}"] \arrow[dr, bend right, "{\llbracket t\{V/x\} \rrbracket}"'] & \tau' \times \Gamma \arrow[r, "id \times \Delta"] & \tau' \times \Gamma \times \Gamma \arrow[r, "{\llbracket t \rrbracket \times id}"] & T\tau \times \Gamma \arrow[r, "{id \times \langle V, id \rangle}"] \arrow[d, "st"] & T\tau \times (\tau' \times \Gamma) \arrow[d, "st"] \\
& T\tau \times \Gamma \arrow[rr, "st"] & & T(\tau \times \Gamma) \arrow[r, "{T(id \times \langle V, id \rangle)}"] \arrow[dr, "{T(\llbracket u\{V/x\} \rrbracket)}"'] & T(\tau \times \tau' \times \Gamma) \arrow[d, "{T\llbracket u \rrbracket}"] \\
& & & & T\overline{\tau}
\end{tikzcd}$$

□

Theorem A.6. *For every computation context C, if $\llbracket t \rrbracket = \llbracket u \rrbracket$ then $\llbracket C[t] \rrbracket = \llbracket C[u] \rrbracket$.*

Proof. The proof follows by induction on the context C. □

A.1 Equational presentation of cert

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

In Figure 13 we present the non-structural equations of **cert**. The left-hand side is present in every CBPV calculus with natural numbers and recursion, where the recursion equation is the last one. The right-hand side is split in two blocks: the first block are the barycentric algebra equations, the second one are the monoid equations and the last one are the list equations.

B DENOTATIONAL SOUNDNESS PROOF

We begin by defining relational a probabilistic relational lifting in ω**Qbs**.

Definition B.1. 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 $P_{\leq 1}$. This definition can be restricted to the probability monad P.

$$\begin{aligned}
&\mathsf{ifZero}\ 0\ \mathsf{then}\ t\ \mathsf{else}\ u \equiv t\\
&\mathsf{ifZero}\ (n+1)\ \mathsf{then}\ t\ \mathsf{else}\ u \equiv u\\
&t \equiv \mathsf{ifZero}\ x\ \mathsf{then}\ t\ \mathsf{else}\ t\\[1em]
&(\lambda x.\ t)\ V \equiv t\{V/x\}\\
&\mathsf{let}\ x\ \mathsf{be}\ V\ \mathsf{in}\ t \equiv t\{V/x\}\\
&t \equiv \lambda x.\ t\ x\\
&x \leftarrow t; (\lambda y.\ u) \equiv \lambda y.\ (x \leftarrow t; u)\\
&\mathsf{force}\ (\mathsf{thunk}\ \ t) \equiv t\\
&\mathsf{thunk}\ (\mathsf{force}\ \ V) \equiv V\\
&x \leftarrow (\mathsf{produce}\ V); t \equiv t\{V/x\}\\
&x \leftarrow t; \mathsf{produce}\ x \equiv t\\
&\mathsf{fix}\, x.\, t = t\{(\mathsf{thunk}\ (\mathsf{fix}\, x.\, t))/x\}
\end{aligned}$$

$$\begin{aligned}
&t \oplus_0 u \equiv t\\
&t \oplus_p u \equiv u \oplus_{1-p} t\\
&t \oplus_p t \equiv t\\
&t \oplus_p (u \oplus_q t') \equiv (t \oplus_{\frac{p(1-q)}{1-pq}} u) \oplus_{pq} t'\\[1em]
&\mathsf{charge}\ n; \mathsf{charge}\ m \equiv \mathsf{charge}\ (n+m)\\
&\mathsf{charge}\ n; \mathsf{charge}\ m \equiv \mathsf{charge}\ m; \mathsf{charge}\ n\\
&\mathsf{charge}\ 0; t \equiv t\\[1em]
&\mathsf{case}\ \mathsf{nil}\ \mathsf{of}\ \mathsf{nil} \Rightarrow t \mid \mathsf{cons}\ x\ xs \Rightarrow u \equiv t\\
&\mathsf{case}\ (\mathsf{cons}\ V_1\ V_2)\ \mathsf{of}\ \mathsf{nil} \Rightarrow t \mid \mathsf{cons}\ x\ xs \Rightarrow u \equiv u\{V_1, V_2/x, xs\}\\
&t \equiv \mathsf{case}\ y\ \mathsf{of}\ \mathsf{nil} \Rightarrow t\{\mathsf{nil}/y\} \mid \mathsf{cons}\ x\ xs \Rightarrow t\{\mathsf{cons}\ x\ xs/y\}\\
&\mathsf{let}\ (x_1, x_2) = (V_1, V_2)\ \mathsf{in}\ t \equiv t\{V_1, V_2/x_1, x_2\}\\
&t \equiv \mathsf{let}\ (x_1, x_2) = V\ \mathsf{in}\ t\{(V_1, V_2)/z\}
\end{aligned}$$

Fig. 13. **cert** equational theory

We now define our logical relations as two families of relations, one for value types and one for computation types:

$$\begin{aligned}
&\mathcal{V}_\tau \subseteq [\![\tau]\!]^v_{CS} \times [\![\tau]\!]^v_{EC} && C_{\overline{\tau}} \subseteq [\![\overline{\tau}]\!]^c_{CS} \times [\![\overline{\tau}]\!]^c_{EC}\\
&\mathcal{V}_{\mathbb{N}} = \{(n,n) \mid n \in \mathbb{N}\} && C_{F\tau} = \{((r,v),\mu) \mid \mathbb{E}(\mu_1) \le r \wedge v\mathcal{V}^\#_\tau \mu_2\}\\
&\mathcal{V}_{U\overline{\tau}} = C_{\overline{\tau}} && C_{\tau\to\overline{\tau}} = \{(f_1, f_2) \mid \forall x_1, x_2, x_1 \mathcal{V}_\tau x_2 \Rightarrow f_1(x_1) C_{\overline{\tau}} f_2(x_2)\}\\
&\mathcal{V}_{\tau_1\times\tau_2} = \mathcal{V}_{\tau_1} \times \mathcal{V}_{\tau_2}\\
&\mathcal{V}_{\mathsf{list}(\tau)} = \mathsf{list}(\mathcal{V}_\tau)
\end{aligned}$$

Where $\mathsf{list}(\mathcal{V}_\tau)$ relates two lists if, and only if, the lists have the same length and are componentwise related by $\mathcal{V}_\tau$. In order to prove the fundamental theorem of logical relations, we first show the following lemmas, where the first one follows by induction:

Lemma B.2. *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 B.3. *For every type τ (resp. $\overline{\tau}$), there is a set M_τ (resp. $M_{\overline{\tau}}$) and partial order $\le$ such that the triple $(\mathcal{V}_\tau, M_\tau, \le)$ (resp. $(C_{\overline{\tau}}, M_{\overline{\tau}}, \le)$) is an ω-quasi Borel space 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 $\le$ 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 $(\mathcal{V}_\tau, M, \le)$ is an ω-quasi Borel space. The injection into $[\![\tau]\!]^v_{CS} \times [\![\tau]\!]^v_{EC}$ being a morphism follows by construction. □

Lemma B.4. *If $(r, \nu)\ C_{F\tau}\ \mu$ then for every pair of functions $f_1 : [\![\tau]\!]_{EC} \to \mathbb{R}$ and $f_2 : [\![\tau]\!]_{CS} \to \mathbb{R}$, such that for every $a_1\ \mathcal{V}_\tau\ a_2$, $f_1(a_1) \le f_2(a_2)$, $\int f_1 \,\mathrm{d}\nu \le \int f_2 \,\mathrm{d}\mu_2$, where μ_2 is the second marginal of μ.*

Proof. Since by assumption $(r, \nu)\ C_{F\tau}\ \mu$, there is a coupling γ over the support of $\mathcal{V}_\tau$, which allows us to conclude:

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

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

At first, it is reasonable to postulate that the expected cost semantics should coincide with the cost semantics. Unfortunately, it does not hold in the subprobabilistic case, as alluded to in Lemma 3.17.

Example B.5. Consider the programs

$$t = \mathsf{charge}\ 2; \mathsf{produce}\ 0$$
$$u = \lambda x.\ \mathsf{ifZero}\ x\ \mathsf{then}\ (\bot \oplus (\mathsf{charge}\ 4; \mathsf{produce}\ 0))\ \mathsf{else}\ \bot$$

By unfolding definitions, we can show $E([\![x \leftarrow t; u\ x]\!]^c_{CS}) = (3, \frac{1}{2}\delta_0) \neq (4, \frac{1}{2}\delta_0) = [\![x \leftarrow t; u\ x]\!]^c_{EC}$.

In the probabilistic case we can prove stronger soundness theorems. Consider the recursion-free fragment of **cert** and the alternative logical relation for $F\tau$ types:

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

Since the subprobabilistic and probabilistic soundness proofs are nearly identical, we will only present the subsprobabilistic one and explicitly mention where they differ.

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.6. *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 $\mathcal{V}_{\tau_1} \times C_{F\tau_2}$, has its output still be restricted to $C_{\overline{\tau}}$.*

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

$F\tau$: In order to prove $f_1^{\#}(r, \nu)\ C_{F\tau'}\ f_2^{\#}(\mu)$ 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 $P_{\le 1}(\mathcal{V}_{\tau'}) \hookrightarrow P_{\le 1}([\![\tau']\!]^v_{CS} \times [\![\tau']\!]^v_{EC})$.

By unfolding the definitions, we get

$$\pi_1(f_1^{\#}(r, \nu)) = r + \int (\pi_1 \circ f_1) \,\mathrm{d}\nu$$
$$\mathbb{E}(f_2^{\#}(\mu)) = \int \int n \|f_2(a)\| \mu(\mathrm{d}n, \mathrm{d}a) + \int n \,\mathrm{d}(f_2^{\#}(\mu)_1)$$

In the second expression, the left hand side term being summed 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 $\int(\pi_1 \circ f_1)\,\mathrm{d}\nu - \int n\,\mathrm{d}(f_2^{\#}(\mu))$, and both should be greater than 0. The first inequality is immediate:

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

For the second expression, assuming $\forall a' \; \mathcal{V}_{\tau'} \; a, \mathbb{E}(f_2(a)) \leq (\pi_1 \circ f_1)(a')$, we can apply Lemma B.4 and use the equality $\int n\,\mathrm{d}(f_2^{\#}(\mu)_1) = \int \mathbb{E}(f_2(a)_1)\,\mathrm{d}\mu_2$.

By adding these two inequalities we obtain exactly the first condition of the relation $\mathcal{C}_{F\tau}$. In the probabilistic case every inequality is an equality, since $\|f(a)\| = 1$ and the inequalities in the definition of $\mathcal{C}_{F\tau}$ 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 morphism $g : \mathcal{V}_\tau \rightarrow P_{\leq 1}(\mathcal{V}_{\tau'})$ 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, $f_1(v_1)\; \mathcal{C}_{F\tau'}\; f_2(v_2)$.

Given this function, we define the coupling $g^{\#}(\mu')$, where μ' is the coupling given by the "witness" of $(r, \nu)\; \mathcal{C}_{F\tau}\; \mu$. 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

$\tau \rightarrow \overline{\tau}$**:** 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 $st; T(f); \alpha_{\tau\rightarrow\overline{\tau}}$ that can be instatiated to both f_1 and f_2.

By definition of $\mathcal{C}_{\tau\rightarrow\overline{\tau}}$, in order to define a morphism $\mathcal{V}_{\tau_1} \times \mathcal{V}_{\tau_2} \rightarrow \mathcal{C}_{\tau\rightarrow\overline{\tau}}$, it suffices to defines its transpose $\mathcal{V}_\tau \times (\mathcal{V}_{\tau_1} \times \mathcal{V}_{\tau_2}) \rightarrow \mathcal{C}_{\overline{\tau}}$. Since the algebra structure of $\alpha_{\tau\rightarrow\overline{\tau}}$ is defined as $\eta; id_\tau \Rightarrow (st; T(ev); \alpha_{\overline{\tau}})$, we want to show that 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 induction hypothesis. This equation holds, up to isomorphism, by the following commutative diagram:

$$\begin{tikzcd}
\tau \times (\tau_1 \times T\tau_2) \arrow[r, "id_\tau \times st"] \arrow[dd, "a"'] & \tau \times T(\tau_1 \times \tau_2) \arrow[r, "\tau \times Tf"] \arrow[dd, "st"'] & \tau \times T(\tau \Rightarrow \overline{\tau}) \arrow[rr, "\tau \times \eta"] \arrow[d, equal] & & \tau \times (\tau \Rightarrow (\tau \times T(\tau \Rightarrow \overline{\tau}))) \arrow[d, dashed, "id_\tau \times (id_\tau \Rightarrow (st; Tev; \alpha_{\overline{\tau}}))"] \arrow[dll, "ev"] \\
 & & \tau \times T(\tau \Rightarrow \overline{\tau}) \arrow[d, "st"] & & \tau \times (\tau \Rightarrow \overline{\tau}) \arrow[d, "ev"] \\
(\tau \times \tau_1) \times T\tau_2 \arrow[r, "st; T(a^{-1})"'] & T(\tau \times (\tau_1 \times \tau_2)) \arrow[r, "T(\tau \times f)"'] & T(\tau \times (\tau \Rightarrow \overline{\tau}) \arrow[r, "Tev"'] & T\overline{\tau} \arrow[r, "\alpha_{\overline{\tau}}"'] & \overline{\tau}
\end{tikzcd}$$

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. □

We are interested in the case where the type τ_1 will be a context Γ. We now state the denotational soundness theorem:

Theorem B.7. *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*

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

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 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 in the theorem statement. We go over the most interesting cases:

Comp This case follows from Lemma B.6.
Fix This theorem follows from the induction hypothesis and from the fact that the relations $\mathcal{C}^c_{\overline{\tau}}$ are closed under suprema of ascending chains.
Produce First apply the induction hypothesis to V and assume that $[\![V]\!]_{CS} = v_1$ and $[\![V]\!]_{EC} = v_2$. By construction, $\eta^{T_1}(v_1)\ \mathcal{C}^v_{F\tau}\ \eta^{T_2}(v_2)$, 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. □

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

Corollary B.8. *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* $[\![t]\!]^c_{CS}$ *greater than* $\pi_1([\![t]\!]^c_{EC})$.

In the recursion-free case, the definition of $\mathcal{C}_{F\tau}$ gives us a stronger soundness property.

Corollary B.9. *The recursion-free 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* $\pi_1([\![t]\!]^c_{EC})$.

C OPERATIONAL SOUNDNESS PROOF

We now prove the soundness theorem.

Proof. Proof by induction on n and on t. The base case $n = 0$ is trivial because $\perp$ is the least element and such an element is preserved by the algebra structure. The inductive ones follow basically from the inductive hypothesis and the CBPV equational theory.

Terminal : The evaluation rules for terminal computations T output the unit of the monad, which allows us to conclude $[\![\Downarrow_n(T)]\!] = (x \leftarrow (\delta_{(0,T)}); [\![x]\!]_{CS}) = [\![T]\!]_{CS}$.
Charge : $[\![\Downarrow_n(\mathsf{charge}\ r)]\!] = \delta_{(r,())} = [\![\mathsf{charge}\ r]\!]_{CS}$.
Sampling : $[\![\Downarrow_n(\mathsf{uniform})]\!] = \delta_0 \otimes \lambda = [\![\mathsf{uniform}]\!]_{CS}$.

Seq :

$$\begin{aligned}
&\llbracket \Downarrow_n (x \leftarrow t; u) \rrbracket = \\
&(\mathsf{produce}\ V) \leftarrow\Downarrow_n (t); y \leftarrow\Downarrow_{n-1} (u\{V/x\}); \llbracket y \rrbracket_{CS} \leq \\
&(\mathsf{produce}\ V) \leftarrow\Downarrow_n (t); \llbracket u\{V/x\} \rrbracket_{CS} = \\
&(\mathsf{produce}\ V) \leftarrow\Downarrow_n (t); x \leftarrow \llbracket \mathsf{produce}\ V \rrbracket_{CS}; \llbracket u \rrbracket_{CS} = \\
&x \leftarrow (y \leftarrow\Downarrow_n (t); \llbracket y \rrbracket_{CS}); \llbracket u \rrbracket_{CS} \leq \\
&x \leftarrow \llbracket t \rrbracket_{CS}; \llbracket u \rrbracket_{CS} = \llbracket x \leftarrow t; u \rrbracket_{CS}
\end{aligned}$$

App :

$$\begin{aligned}
&\llbracket \Downarrow_n (t\, V) \rrbracket = (\lambda x.\ t') \leftarrow\Downarrow_n (t); y \leftarrow\Downarrow_{n-1} (t'\{V/x\}); \llbracket y \rrbracket_{CS} \leq \\
&(\lambda x.\ t') \leftarrow\Downarrow_n (t); \llbracket (\lambda x.\ t')\, V \rrbracket_{CS} = f \leftarrow\Downarrow_n (t); \llbracket f \rrbracket_{CS}\, (\llbracket V \rrbracket_{CS}) = \\
&(f \leftarrow\Downarrow_n (t); \llbracket f \rrbracket_{CS})(\llbracket V \rrbracket_{CS}) \leq \llbracket t \rrbracket_{CS}\, (\llbracket V \rrbracket_{CS}) = \llbracket t\, V \rrbracket_{CS}
\end{aligned}$$

Fix : By unfolding the operational semantics,

$$\begin{aligned}
&\llbracket \Downarrow_n (\mathsf{fix}\, x.\, t) \rrbracket = \\
&\llbracket \Downarrow_{n-1} (t\{\mathsf{thunk}\ \mathsf{fix}\, x.\, t/x\}) \rrbracket \leq \\
&\llbracket t\{\mathsf{thunk}\ \mathsf{fix}\, x.\, t/x\} \rrbracket_{CS} = \\
&\llbracket \mathsf{fix}\, x.\, t \rrbracket_{CS}
\end{aligned}$$

IzZ0 : By unfolding the operational semantics,

$$\begin{aligned}
&\llbracket \Downarrow_n (\mathsf{ifZero}\ 0\ \mathsf{then}\ t\ \mathsf{else}\ u) \rrbracket = \\
&\llbracket \Downarrow_n (t) \rrbracket \leq \llbracket t \rrbracket_{CS} = \\
&\llbracket \mathsf{ifZero}\ 0\ \mathsf{then}\ t\ \mathsf{else}\ u \rrbracket_{CS}
\end{aligned}$$

IzZS : By unfolding the operational semantics,

$$\begin{aligned}
&\llbracket \Downarrow_n (\mathsf{ifZero}\ n+1\ \mathsf{then}\ t\ \mathsf{else}\ u) \rrbracket = \\
&\llbracket \Downarrow_n (u) \rrbracket \leq \llbracket u \rrbracket_{CS} = \\
&\llbracket \mathsf{ifZero}\ n+1\ \mathsf{then}\ t\ \mathsf{else}\ u \rrbracket_{CS}
\end{aligned}$$

UnPair :

$$\begin{aligned}
&\llbracket \Downarrow_n (\mathsf{let}\ (x_1, x_2) = (V_1, V_2)\ \mathsf{in}\ t) \rrbracket = \\
&\llbracket \Downarrow_{n-1} (t\{V_1, V_2/x_1, x_2\}) \rrbracket \leq \\
&\llbracket t \rrbracket_{CS}\, (\llbracket V_1 \rrbracket_{CS}, \llbracket V_2 \rrbracket_{CS}) = \\
&\llbracket \mathsf{let}\ (x_1, x_2) = (V_1, V_2)\ \mathsf{in}\ t) \rrbracket_{CS}
\end{aligned}$$

caseNil :

$$\begin{aligned}
&\llbracket \Downarrow_n (\mathsf{case}\ \mathsf{nil}\ \mathsf{of}\ \mathsf{nil} \Rightarrow t \mid \mathsf{cons}\, x\, xs \Rightarrow u) \rrbracket = \\
&\llbracket \Downarrow_n (t) \rrbracket \leq \llbracket t \rrbracket_{CS} = \\
&\llbracket \mathsf{case}\ \mathsf{nil}\ \mathsf{of}\ \mathsf{nil} \Rightarrow t \mid \mathsf{cons}\, x\, xs \Rightarrow u) \rrbracket_{CS}
\end{aligned}$$

caseCons :

$$\begin{aligned}
&\llbracket \Downarrow_n (\mathsf{case\,cons}\, V_1\, V_2 \,\mathsf{of\,nil} \Rightarrow t \mid \mathsf{cons}\, x\, xs \Rightarrow u)\rrbracket = \\
&\llbracket \Downarrow_{n-1} (u\{V_1, V_2/x_1, x_2\})\rrbracket \leq \\
&\llbracket u\{V_1, V_2/x_1, x_2\})\rrbracket_{CS} = \\
&\llbracket \mathsf{case\,cons}\, V_1\, V_2 \,\mathsf{of\,nil} \Rightarrow t \mid \mathsf{cons}\, x\, xs \Rightarrow u\rrbracket_{CS}
\end{aligned}$$

□

D OPERATIONAL ADEQUACY PROOF

As it is usually the case with adequacy proofs, it follows by a logical relations argument. Before defining it, we define a relation lifting for the subprobability cost monad.

Definition D.1. If $\mathcal{R} \subseteq A \times B$ is a binary relation, its lifting $\widetilde{\mathcal{R}} \subseteq P_{\leq 1}(\mathbb{N} \times A) \times P_{\leq 1}(\mathbb{N} \times B)$ is defined as $\mu \widetilde{\mathcal{R}} \nu$ if, and only if, for every $f : \mathbb{N} \times A \longrightarrow [0,1]$ and $g : \mathbb{N} \times B \longrightarrow [0,1]$ such that whenever $a\mathcal{R}b$, $g(n,b) \leq f(n,a)$, for every $n : \mathbb{N}$, then $\left(\int g \, \mathrm{d}\nu\right) \leq \left(\int f \, \mathrm{d}\mu\right)$.

$$\begin{array}{rcl}
\triangleright_\tau & \subseteq & \mathcal{V}al^{\cdot \vdash_v \tau} \times \llbracket \tau \rrbracket_{CS} \\
n \triangleright_{\mathbb{N}} n & \iff & \top \\
r \triangleright_{\mathbb{R}} r & \iff & \top \\
() \triangleright_1 () & \iff & \top \\
c \triangleright_{\mathbb{C}} c & \iff & \top \\
(V_1, V_2) \triangleright_{\tau_1 \times \tau_2} (x, y) & \iff & (V_1 \triangleright_{\tau_1} x) \wedge (V_2 \triangleright_{\tau_2} y) \\
V \triangleright_{\mathsf{list}(\tau)} x & \iff & V\, \mathsf{list}(\triangleright_\tau)\, x \\
(\mathsf{thunk}\ t) \triangleright_{U\overline{\tau}} x & \iff & \Downarrow(t) \blacktriangleleft_{\overline{\tau}} x
\end{array}$$

$$\begin{array}{rcl}
\blacktriangleleft_{\overline{\tau}} & \subseteq & P_{\leq 1}(\mathbb{N} \times T^{\cdot \vdash_c \overline{\tau}}) \times \llbracket \overline{\tau} \rrbracket_{CS} \\
\mu \blacktriangleleft_{F\tau} \nu & \iff & \mu \widetilde{\triangleright_\tau} \nu \\
\mu \blacktriangleleft_{\tau \to \overline{\tau}} f & \iff & (\forall a V, V \triangleright_\tau a \Rightarrow \\
 & & (\lambda x\,.\, t) \leftarrow \mu; \Downarrow(t\{V/x\}) \blacktriangleleft_{\overline{\tau}} f(a))
\end{array}$$

The relation $\mathsf{list}(\triangleright_\tau)$ holds if, and only if, both lists have the same length and are elementwise related by $\triangleright_\tau$. Next, we define an extension of the logical relations $\triangleright_\tau$ to contexts.

Definition D.2. Let $\Gamma = x_1 : \tau_1, \ldots, x_n : \tau_n$, $\cdot \vdash_v V_i : \tau_1$ and $\gamma : \llbracket \Gamma \rrbracket^v_{CS}$. We say that $(V_1, \ldots, V_n) \triangleright_\Gamma \gamma$ if, and only if, $V_i \triangleright_{\tau_i} \pi_i(\gamma)$, for every $i \in \{1, \ldots, n\}$. We will use the letter G to denote the list of values of $(V_1, \ldots, V_n)$.

In order to prove the fundamental theorem of logical relations we require a couple of lemmas that are proved by induction.

Lemma D.3. *Let T be the subprobability cost monad $P_{\leq 1}(\mathbb{N} \times -)$, Γ be a context, τ_1 and $\overline{\tau}$ types, $\Gamma, x : \tau_1 \vdash t : \overline{\tau}$ a computation and $f : \llbracket \Gamma \rrbracket^v_{CS} \times \llbracket \tau_1 \rrbracket^v_{CS} \longrightarrow \llbracket \overline{\tau} \rrbracket^c_{CS}$ a ω**Qbs** morphism such that for every $\cdot \vdash_v V : \tau_1$ with $V \triangleright_{\tau_1} v$, $\Downarrow(t\{G, V/\Gamma, x\}) \blacktriangleleft_{\overline{\tau}} f(\gamma, v)$, then $\Downarrow(x \leftarrow u; t\{G/\Gamma\}) \blacktriangleleft_{\overline{\tau}} (\alpha_{\overline{\tau}} \circ Tf(\gamma))(\nu)$, whenever $\Downarrow(u) \blacktriangleleft_{F\tau_1} \nu$ and $G \triangleright_\Gamma \gamma$.*

Proof. Fix $G \triangleright_\Gamma \gamma$. The proof follows by induction on the computation type $\overline{\tau}$.

$F\tau$: This proof follows by unfolding the definitions. Given the definition of the relational lifting for the cost monad, let $h : \mathbb{N} \times T^{\cdot \vdash_c F\tau} \longrightarrow [0,1]$ and $g : \mathbb{N} \times \llbracket \tau \rrbracket_{CS} \longrightarrow [0,1]$ be functions such that for every $V'' \rhd_\tau v''$ and $n : \mathbb{N}$, $g(n, v'') \leq h(n, v'')$. Therefore, we have to show that

$$\int g(n+n', y)\,\mathrm{d}((n', v) \leftarrow \nu; f(\gamma, v)) \leq \int f(n+n', y)\,\mathrm{d}((n', V') \leftarrow\!\Downarrow\!(u); \Downarrow\!(t\{G, V'/\Gamma, x\}))$$

Using the equational theory of CBPV and the commutativity equation, the expression above is equivalent to

$$(n', v) \leftarrow \nu; \int g(n+n', y)\,\mathrm{d}(f(\gamma, v)) \leq (n', V') \leftarrow\!\Downarrow\!(u); \int f(n+n', y)\,\mathrm{d}(\Downarrow\!(t\{G, V'/\Gamma, x\}))$$

By assumption, for every $V \rhd_{\tau_1} v$, $\int g\,\mathrm{d}(f(v)) \leq \int f\,\mathrm{d}(\Downarrow\!(t\{V/x\}))$. We conclude this case by using the assumption $\Downarrow\!(u) \blacktriangleleft_{F\tau_1} \nu$ and the functions $g'(n', v) = \int g(n+n', y)\,\mathrm{d}(f(\gamma, v))$ and $h'(n', V) = \int h(n+n', y)\,\mathrm{d}(\Downarrow\!(t\{G, V/\Gamma, x\}))$. Since, by construction, g' and h' satisfy the property that whenever $V_1 \rhd_{\tau_1} v_1$, $h'(n', v_1) \leq g'(n', V_1)$, for every $n' : \mathbb{N}$, we can show:

$$\begin{aligned}&(n', v) \leftarrow \nu; \int g(n+n', y)\,\mathrm{d}(f(\gamma, v)) = \\ &\int g'\,\mathrm{d}(\nu) \leq \\ &\int h'\,\mathrm{d}(\Downarrow\!(u)) = \\ &(n', V') \leftarrow\!\Downarrow\!(u); \int f(n+n', y)\,\mathrm{d}(\Downarrow\!(t\{G, V'/\Gamma, x\}))\end{aligned}$$

$\tau \to \overline{\tau}$: For this case, let $V' \rhd_\tau v'$. We have to show that $((\lambda y.\, t') \leftarrow\!\Downarrow (x \leftarrow u; t); \Downarrow (t'\{G, V'/\Gamma, y\}) \blacktriangleleft_{\overline{\tau}} x \leftarrow \nu; f(\gamma, x, v')$. Rewriting it, we obtain the following equivalent relation $V \leftarrow\!\Downarrow\!(u); (\lambda y.\, t') \leftarrow\!\Downarrow\!(t); \Downarrow\!(t'\{G, V'/\Gamma, y\}) \blacktriangleleft_{\overline{\tau}} x \leftarrow \nu; f(\gamma, x, v')$. We prove this by using the induction hypothesis for the type $\overline{\tau}$. We choose $\Gamma, x : \tau_1 \vdash t\, V'$ and $v \mapsto f(\gamma, v, v')$ in the inductive hypothesis, which allows us to conclude.

□

Lemma D.4. *For every computation type $\overline{\tau}$ and for every distribution μ, $\mu \blacktriangleleft_{\overline{\tau}} \bot$ and if $\mu \blacktriangleleft_{\overline{\tau}} x_n$, for an ascending chain $x_0 \leq \cdots \leq x_n \leq \cdots$, then $\mu \blacktriangleleft_{\overline{\tau}} \sup_n(x_n)$.*

Proof. The proof follows by induction on $\overline{\tau}$. For the base case, let $f : T^{\cdot \vdash_c F\tau} \to [0,1]$ and $g : [\![\tau]\!]_{CS} \to [0,1]$ be functions such that whenever $V \rhd_\tau x$, $g(n, x) \leq f(V)$. Since $\bot$ in $F\tau$ is the 0, measure, $\int g\,\mathrm{d}0 = 0 \leq \int f\,\mathrm{d}(\Downarrow\!(t))$. The stability under suprema of ascending chains follows from Scott-continuity of integration.

For the case $\tau \to \overline{\tau}$, we use the inductive hypothesis and the fact that the order of functions is given pointwise. □

Theorem D.5 (Fundamental Theorem of Logical Relations). *If $\Gamma \vdash_c t : \overline{\tau}$ (resp. $\Gamma \vdash_v V : \tau$) and $G \rhd_\Gamma \gamma$ then $\Downarrow\!(t\{G/\Gamma\}) \blacktriangleleft_{\overline{\tau}} [\![t]\!]_{CS}(\gamma)$ (resp. $V\{G/\Gamma\} \rhd_\tau [\![V]\!]_{CS}(\gamma)$).*

Proof. The proof follows by mutual induction on the typing derivations of t and V.

Var : Assuming that $x_i\{G/\Gamma\} = V_i$, where we assume that the i-th elements of G and Γ are, respectively, V_i and x_i, which implies the conclusion by the assumption $V_i \rhd v_i$.

Arithmetic constants : Follows by inspection.

Pair : Follows directly from the induction hypotheses.

Charge : By unfolding the definitions, $\Downarrow(\mathsf{charge}\, V\{G/\Gamma\}) = \delta_{(\llbracket V\{G/\Gamma\}\rrbracket,())} = \llbracket \mathsf{charge}\, V\{G/\Gamma\} \rrbracket_{CS}$. Therefore, since $\rhd_1 = \{((),())\}$ and the distributions $\Downarrow(\mathsf{charge}\, V\{G/\Gamma\})$ and $\llbracket \mathsf{charge}\, V\{G/\Gamma\} \rrbracket_{CS}$ are the same, we can conclude by definition of $\blacktriangleleft_{F\tau}$.

Sample : By unfolding the definitions, $\Downarrow(\mathsf{uniform}) = \delta_0 \otimes \lambda = \llbracket \mathsf{uniform} \rrbracket_{CS}$. Therefore, since $\rhd_{\mathbb{R}} = \{(r,r) \mid r \in \mathbb{R}\}$ and the distributions $\Downarrow(\mathsf{uniform})$ and $\llbracket \mathsf{uniform} \rrbracket_{CS}$ are the same, we can conclude by definition of $\blacktriangleleft_{F\tau}$

Fix : Follows mostly from Lemma D.4. We begin by proving that $\Downarrow (\mathsf{fix}\, x.(t\{G/\Gamma\})) \blacktriangleleft_{\overline{\tau}} (\llbracket t \rrbracket_{CS}(\gamma))^n(\bot)$, for every $n : \mathbb{N}$. The proof follows by induction on n and, in order to avoid visual pollution, let $t' = t\{G/\Gamma\}$.

0 : In this case, $(\llbracket t \rrbracket_{CS}(\gamma))^0(\bot) = \bot$, so we can apply Lemma D.4.

$n+1$: For this case, we apply the induction hypothesis and get

$$\Downarrow(\mathsf{fix}\, x.t') \blacktriangleleft_{\overline{\tau}} (\llbracket t \rrbracket_{CS}(\gamma))^n(\bot)$$

Next, using the definition of $\rhd_{U\overline{\tau}}$, we can conclude that

$$\mathsf{thunk}\ (\mathsf{fix}\, x.t') \rhd_{U\overline{\tau}} (\llbracket t \rrbracket_{CS}(\gamma))^n(\bot)$$

Now, we apply the global induction hypothesis and conclude

$$\Downarrow(\mathsf{fix}\, x.t') = \Downarrow(t'\{\mathsf{thunk}\ (\mathsf{fix}\, x.t')/x\}) \blacktriangleleft_{\overline{\tau}} (\llbracket t \rrbracket_{CS}(\gamma))^{n+1}(\bot)$$

Therefore, by Lemma D.4 $\Downarrow(\mathsf{fix}\, x.t') \blacktriangleleft_{\overline{\tau}} \bigsqcup_n (\llbracket t \rrbracket_{CS}(\gamma))^n(\bot)$.

Abstraction : Follows directly from the equation $(\lambda x.\ t)\{G/\Gamma\} = \lambda x.\ (t\{G/\Gamma\})$, the equation $\Downarrow(\lambda x.\ t\{G/\Gamma\}) = \delta_{(0,\lambda x.\ t)}$ and the induction hypothesis. We will now show show that $\Downarrow(\lambda x.\ t\{G/\Gamma\}) \blacktriangleleft_{\tau \to \overline{\tau}} \llbracket t \rrbracket_{CS}(\gamma)$. Let $V \rhd_\tau v$, we have to show

$$((\lambda x.\ t') \leftarrow \Downarrow(\lambda x.\ t\{G/\Gamma\}); \Downarrow(t'\{V/x\})) \blacktriangleleft_{\overline{\tau}} \llbracket t \rrbracket_{CS}(\gamma, v)$$

The LHS of that expression is equal to $\Downarrow (t\{G, V/\Gamma, x\})$. We conclude by applying the induction hypothesis to $\Gamma, x : \tau \vdash_c t : \overline{\tau}$ and $V \rhd_\tau v$.

Application : Follows directly from equation $(t\ V)\{G/\Gamma\} = (t\{G/\Gamma\})\ V\{G/\Gamma\}$, the induction hypothesis and the definition of $\blacktriangleleft_{\tau \to \overline{\tau}}$.

Produce : We conclude by the induction hypothesis for a value V, the equalities $\Downarrow(V\{G/\Gamma\}) = \delta_{(0,V\{G/\Gamma\})}$ and $\llbracket \mathsf{produce}\ V\{G/\Gamma\} \rrbracket_{CS} = \delta_{(0,\llbracket V \rrbracket_{CS}(\llbracket G \rrbracket_{CS}))}$, and the fact that for every measurable function $f : A \to [0,1]$, $\int f\, \mathrm{d}(\delta_x) = f(x)$.

Force : Follows from the fact that the only well-typed closed programs of type $U\overline{\tau}$ are those of the form $\mathsf{thunk}\ t$, for some computation t, the equation $\Downarrow(\mathsf{force}\ \mathsf{thunk}\ t) = \Downarrow(t)$ and the induction hypothesis.

Thunk : Follows directly from the substitution equality $(\mathsf{thunk}\ t)\{G/\Gamma\} = \mathsf{thunk}\ (t\{G/\Gamma\})$, the induction hypothesis and the definition of $\rhd_{U\overline{\tau}}$.

List Case : Using the distributivity of substitution, the fact that closed values of type list are either the empty list or the cons of a list, and the induction hypothesis, we can conclude.

Seq : First, use the distributivity of substitution $(x \leftarrow t; u)\{G/\Gamma\} = x \leftarrow t\{G/\Gamma\}; u\{G/\gamma\}$. We conclude by applying Lemma D.3 and noting that its assumptions are exactly the induction hypotheses applied to $t\{G/\Gamma\}$ and $u\{G/\Gamma\}$. □

E EXAMPLES

E.1 Random Walks

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 = $\mu f : \mathbb{N} \to \mathbb{N} \to F1.\, \lambda i : \mathbb{N}\; j : \mathbb{N}.$
if $i = j$ then
  produce ()
else
  charge 1;
  if $i$ then
    (force $f$) 1 $j$
  else
    ((force $f$) $(i-1)$ $j$) $\oplus$ ((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. This is basically the mathematical specification of the one-dimensional random walk with barrier. As such, we know that it terminates with probability 1, c.f Section 1.6 of [36]).

By unfolding the denotational semantics, we obtain that the cost expression is given by the fixed point of the operator

```
$\lambda F : \mathbb{N} \to \mathbb{N} \to [0, \infty].\, \lambda i : \mathbb{N} j : \mathbb{N}.$
if $i = j$ then
  $0$
elseif $i = 0$ then
  $1 + F(1, j)$
else
  $1 + \frac{1}{2}(F(i-1, j) + F(i+1, j))$
```

It is now possible to compute the expected value on the number of rounds that are necessary in order to reach your target, which following the expression above, 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 [36] 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.

E.2 Stochastic Convex Hull

In computational geometry, finding the convex hull of a set of points in space is an important algorithm that has been thoroughly studied. In this case study, we go over a variant of this algorithm that has a linear expected runtime cost.

```
unifList = fix $f.\lambda n.$
ifZero $n$ then
   produce nil
else
   $l \leftarrow$ unifList $(n-1)$
   $x \leftarrow$ uniform
   $y \leftarrow$ uniform
   produce (cons $(x, y)$ $l$)

sieve = $\lambda l : \mathbb{N}.$
$p_1 \leftarrow \min(\lambda(x, y) . x + y)\, l$
$p_2 \leftarrow \min(\lambda(x, y) . x - y)\, l$
$p_3 \leftarrow \min(\lambda(x, y) . -x + y)\, l$
$p_4 \leftarrow \min(\lambda(x, y) . -x - y)\, l$
filter (charge 1; iQ $p_1\ p_2\ p_3\ p_4$) $l$
```

```
scan = fix $f. \lambda l. \lambda stk.$
charge 1
case $(l, stk)$ of
 | nil, _ $\Rightarrow$ produce $stk$
 | $(x :: xs)$, nil $\Rightarrow f\ xs\ [x]$
 | $(x :: xs), [p] \Rightarrow f\ xs\ [x, p]$
 | $(x :: xs), (p1 :: p2 :: ps) \Rightarrow$
   if clockOrNot $p_2\ p_1\ x$ then
      $f\ (x :: xs)\ (p2 :: ps)$
   else
      $f\ xs\ (x :: stk)$
```

```
graham = $\lambda l.$
$(x, y) \leftarrow$ min_xy $l$
$l' \leftarrow$ quicksort (charge 1; $\sqsubseteq\ p$)$l$
scan $l'$ nil

convexHull = $\lambda n : \mathbb{N}.$
$l \leftarrow$ unifList $n$
$l' \leftarrow$ sieve $l$
graham $l'$
```

Fig. 14. Stochastic convex hull algorithm

This algorithm can be divided into three separate components. The first one generates a uniformly and independently sampled list in the square $[0, 1]^2$. The second one sieves the original list of points so that points that are “obviously” not a part of the convex hull are eliminated. The last part is any convex hull algorithm that runs in $O(n \log n)$. It is important to note that this algorithm only has this time complexity under the assumption that the input list is uniformly distributed. Indeed, the time complexity of this algorithm hinges on the following lemma.

Lemma E.1. *(Th. 2.2 of Golin [15]) Let $P \subseteq [0, 1]^2$ be an independent and uniformly distributed finite set of n points. There is a square R which is inside P's convex hull such that the size of the set of points in P outside of R is $O(\sqrt{n})$.*

We can also prove that convexHull terminates with probability 1.

Lemma E.2. *The total mass of $\pi_2(\llbracket \text{convexHull} \rrbracket_{EC})$ is* 1.

Proof. The only component that might be problematic is the scan function, which terminates since it is structurally recursive on the lexicographic order on lists. □

Lemma E.3. *The graham-scan function has runtime $O(n \log n)$, where n is the length of the input.*

Proof. By unfolding the denotational semantics, we see that its cost is given by adding the cost of the min_{xy}, quicksort and scan functions. By the quicksort case study, its cost is $O(n \log n)$. We are assuming that the cost of min_{xy} is linear on the length of the list. Finally, the cost of scan is also linear, though the proof is a bit more involved, so we will not go over it and, instead, will point to a standard analysis of Graham scan [16]. Therefore, the overall cost is bounded above by $O(n \log n)$. □

Theorem E.4. *The stochastic convex hull algorithm has linear expected run time.*

Proof. The cost analysis is given by the expected cost of the sieve function plus the average of the graham function. The cost of the sieve function is $O(n)$, since it iterates over the input list 5 times. The graham function has its cost bounded by the sorting function, which costs $O(n' \log n')$, where n' is the length of the output from the sieve function. By Lemma E.1, the output list has an expected length of $O(\sqrt{n})$. Thus, assuming that the n points in l have been sampled uniformly and independently from $[0, 1]^2$ and $\mu = (\pi_2 \circ [\![\text{sieve}]\!]_{EC})(l)$, the cost structure becomes:

$$\begin{aligned}
&(\pi_1 \circ [\![\text{sieve}]\!]^{\#}_{EC})((n)) + \int (\pi_1 \circ [\![\text{graham}]\!]_{EC})(l')\,\mathrm{d}\mu(l') \leq \\
&O(n) + \int (\pi_1 \circ [\![\text{graham}]\!]_{EC})(l')\,\mathrm{d}\mu(l') \leq \\
&O(n) + \int \log(\text{length}(l'))\text{length}(l')\,\mathrm{d}\mu(l') \leq \\
&O(n) + \log(n) \int \text{length}(l')\,\mathrm{d}\mu(l') \leq \\
&O(n) + O(\sqrt{n}) \log n \leq O(n)
\end{aligned}$$

□

F PROOFS OF MISCELLANEOUS LEMMAS AND THEOREMS

F.1 Proof of Theorem 3.12

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:

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

While the last law requires a bit more work:

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

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

F.2 Proof of Theorem 4.2

Proof. Since both semantics are the monadic semantics of CBPV, we know by Proposition 121 of [29] that they satisfy the equations that do not reference the sampling and cost operations. We only have to prove that the equations that are specific to them are satisfied by both semantics.

- $\oplus_0$ **unit** : $[\![t \oplus_0 u]\!]_{CS} = [\![t]\!]_{CS} + 0\,[\![u]\!]_{CS} = [\![t]\!]_{CS}$ and $[\![t \oplus_0 u]\!]_{EC} = [\![t]\!]_{EC} + 0\,[\![u]\!]_{EC} = [\![t]\!]_{EC}$, where addition and scalar multiplication for the expected cost semantics are defined componentwise.
- $\oplus_p$ **symmetry** : $[\![t \oplus_p u]\!]_{CS} = (1-p)\,[\![t]\!]_{CS} + p\,[\![u]\!]_{CS} = [\![u \oplus_{1-p} t]\!]_{CS}$ and $[\![t \oplus_p u]\!]_{EC} = (1-p)\,[\![t]\!]_{EC} + p\,[\![u]\!]_{EC} = [\![u \oplus_{1-p} t]\!]_{EC}$.
- $\oplus_p$ **idempotent** : $[\![t \oplus_p t]\!]_{CS} = (1-p)\,[\![t]\!]_{CS} + p\,[\![t]\!]_{CS} = [\![t]\!]_{CS}$ and $[\![t \oplus_p t]\!]_{EC} = (1-p)\,[\![t]\!]_{EC} + p\,[\![t]\!]_{EC} = [\![t]\!]_{EC}$

$\oplus_p \oplus_q$ **associativity** :

$$\begin{aligned}
&\left[\!\left[t \oplus_p (u \oplus_q t') \right]\!\right]_{CS} = \\
&(1-p)\, [\![t]\!]_{CS} + p((1-q)\, [\![u]\!]_{CS} + q\, [\![t']\!]_{CS}) = \\
&(1-p)\, [\![t]\!]_{CS} + p(1-q)\, [\![u]\!]_{CS} + pq\, [\![t']\!] = \\
&([\![t]\!]_{CS} \oplus_{\frac{p(1-q)}{1-pq}} [\![u]\!]_{CS}) \oplus_{pq} [\![t']\!]_{CS}
\end{aligned}$$

The reasoning for the expected cost semantics is analog.

charge n **monoid action** : $[\![\text{charge}\, n; \text{charge}\, m]\!]_{CS} = \delta_{(n+m,())} = [\![\text{charge}\,(n+m)]\!]_{CS}$ and for the expected cost semantics $[\![\text{charge}\, n; \text{charge}\, m]\!]_{EC} = (n+m, \delta_{()}) = [\![\text{charge}\,(n+m)]\!]_{EC}$.

charge n **commutativity** : $[\![\text{charge}\, n; \text{charge}\, m]\!]_{CS} = \delta_{(n+m,())} = [\![\text{charge}\, m; \text{charge}\, n]\!]_{CS}$ and $[\![\text{charge}\, n; \text{charge}\, m]\!]_{EC} = (n+m, \delta_{()}) = [\![\text{charge}\, m; \text{charge}\, n]\!]_{EC}$.

charge 0 **unit** : $[\![\text{charge}\, 0; t]\!]_{CS} = [\![t]\!]^{\#}_{CS} (\delta_{(0,())} = [\![t]\!]_{CS}$ and $[\![\text{charge}\, 0; t]\!]_{EC} = (0 + (\pi_1 \circ [\![t]\!]_{EC})(()), (\pi_2 \circ [\![t]\!]_{EC})(())) = ((\pi_1 \circ [\![t]\!]_{EC})(()), (\pi_2 \circ [\![t]\!]_{EC})(())) = [\![t]\!]_{EC}$ □

F.3 Proof of Theorem 4.3

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

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

F.4 Proof of Theorem 4.5

We will formalize the maximality of the equation using the concept of the center of a monad, which we now start to define.

Definition F.1 ([6]). Let $X : \mathcal{C}$ be an object in a Cartesian category and $T : \mathcal{C} \to \mathcal{C}$ a monad. A central cone at X is a pair (Z, ι), where $\iota : Z \to TX$ is a morphism making the following diagram commute for every Y:

$$\begin{tikzcd}
Z \times TY \arrow[r, "\iota"] \arrow[d, "\iota"'] & TX \times TY \arrow[r, "st'_{X,TY}"] & T(X \times TY) \arrow[d, "T(st_{X,Y})"] \\
TX \times TY \arrow[d, "st_{TX,Y}"'] & & T^2(X \times Y) \arrow[d, "\mu_{X\times Y}"] \\
T(TX \times Y) \arrow[r, "T(st'_{X,Y})"'] & T^2(X \times Y) \arrow[r, "\mu_{X\times Y}"'] & T(X \times Y)
\end{tikzcd}$$

This definition is the categorification of choosing elements of TX that commute over every element of TY, for every Y. These cones can be naturally organized as a category, by defining it as the appropriate subcategory of the slice category $\mathcal{C}/TX$.

Definition F.2 ([6]). A monad $T : \mathcal{C} \to \mathcal{C}$ is centralizable if for every object $X : \mathcal{C}$, there exists a terminal central cone on X.

Something quite nice about this definition is that if for every X, there is a terminal central cone $\mathcal{Z}(X)$, this assignment $X \mapsto \mathcal{Z}(X)$ extends to a commutative submonad of T [6]. With this definition, we can now restate Theorem 4.5 as follows.

Theorem F.3. *The center of* $[0,\infty] \times P_{\leq 1}$ *is the probability monad* P.

Proof. For every $X : \omega\mathbf{Qbs}$, we define $\iota(\mu) = (0, \mu)$. To show that this is indeed a central cone, we observe that the upper leg of the central cone commutative diagram is the function $f(\mu, (r, \nu)) = (r, \mu \otimes \nu)$, where $\mu \otimes \nu$ is the product distribution on μ and ν. A direct calculation shows that the lower leg is equal to $(r, \mu \otimes \nu)$ as well.

In order to show that it is the terminal object, let $\iota : Z \longrightarrow [0,\infty] \times P_{\leq 1}X$ be a central cone. Let $z : Z$ and assume that $\iota(z) = (r', \mu)$. The upper leg then becomes $(r + r'\nu(Y), \mu \otimes \nu)$ while the lower leg is $(r' + r\mu(X), \mu \otimes \nu)$. Since this equation has to hold for every r and ν, it holds if, and only if, $r' = 0$ and $\mu(X) = 1$, i.e. its total mass is 1. Therefore, $\iota : Z \longrightarrow [0,\infty] \times P_{\leq 1}X$ factors through the inclusion $PX \hookrightarrow [0,\infty] \times P_{\leq 1}X$, concluding the proof. □

F.5 Proof of Theorem 5.9

Proof. The proof follows by showing that there are monad structure morphisms $[0,\infty] \times ([0,1] \longrightarrow -) \twoheadrightarrow [0,\infty] \times P_{\leq 1}$ and $\varphi : [0,\infty] \times P_{\leq 1} \hookrightarrow K_{[0,\infty]}$. The proof is concluded by the uniqueness of factorizations.

The first monad morphism is $(id \times \psi)$, where ψ is the premonad morphism $([0,1] \longrightarrow -_{\bot}) \twoheadrightarrow P_{\leq 1}$, and the proof that this transformation is component-wise image-dense follows from the observation that the product order in $\omega\mathbf{Qbs}$ is given pairwise and ψ is, by assumption, dense in its image. The monad morphisms axioms follow from:

$$(id \times \psi)(\eta(a)) = (id \times \psi)(0, \lambda r.\, a) = (0, \delta_a) = \eta(a)$$

$$\begin{aligned}
&(((id \times \psi) \circ f)^{\#} \circ (id \times \psi))(r, g) = ((id \times \psi) \circ f)^{\#}(r, \psi(g), (\psi \circ \pi_2 \circ f)^{\#}(g)) = \\
&(r + \int (\pi_1 \circ f)\, \mathrm{d}(\psi(g)), (\psi \circ (\pi_2 \circ f)^{\#}))(g) = \\
&(r + \int (\pi_1 \circ f \circ g)\, \mathrm{d}(\lambda), (\psi \circ (\pi_2 \circ f)^{\#})(g)) = ((id \times \psi) \circ f^{\#})(r, g)
\end{aligned}$$

The second monad morphism has components $\varphi(r, \mu) = \lambda f.\, r + \int f\, \mathrm{d}\mu$. The proof that this is indeed order reflecting is a direct consequence of $P_{\leq 1} \hookrightarrow K_{[0,\infty]}$ being order reflecting.

The proof that this is indeed a monad morphism follows, once again, from a series of direct calculations. □

F.6 Proof of Lemma 6.2

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. Next, assume that the length is greater than 0. We can prove the following program equality:

$$\begin{array}{l} \mathit{len} \leftarrow \mathit{length}\, l \\ r \leftarrow \text{rand}\ \mathit{len} \\ \mathit{pivot} \leftarrow l[r] \\ \boxed{(l_1, l_2) \leftarrow \mathit{biFilter}\ (\lambda n.\ \text{charge}\ 1; n \leq \mathit{pivot})\ (\text{drop}\ l)} \\ l_1' \leftarrow \text{quicksort}\ l_1 \\ l_2' \leftarrow \text{quicksort}\ l_2 \\ \text{length}\ (l_1' +\!\!+\ \mathit{pivot} :: l_2') \end{array} = \begin{array}{l} \mathit{len} \leftarrow \mathit{length}\, l \\ r \leftarrow \text{rand}\ \mathit{len} \\ \mathit{pivot} \leftarrow l[r] \\ \boxed{\begin{array}{l} \text{charge}\ (\mathit{len} - 1) \\ (l_1, l_2) \leftarrow \mathit{biFilter}\ (\lambda n.\ n \leq \mathit{pivot})\ (\text{drop}\ l) \end{array}} \\ l_1' \leftarrow \text{quicksort}\ l_1 \\ l_2' \leftarrow \text{quicksort}\ l_2 \\ \text{length}\ (l_1' +\!\!+\ \mathit{pivot} :: l_2') \end{array}$$

Next, we want to consider the interaction of the regular quicksort algorithm and applying the length function to it. Furthermore, by using the additive properties of the length function and the strong induction hypotheses, we get the following program equation:

$$\begin{array}{l} \mathit{len} \leftarrow \mathit{length}\, l \\ r \leftarrow \text{rand}\ \mathit{len} \\ \mathit{pivot} \leftarrow l[r] \\ \text{charge}\ (\mathit{len} - 1) \\ (l_1, l_2) \leftarrow \mathit{biFilter}\ (\lambda n.\ n \leq \mathit{pivot})\ (\text{drop}\ l) \\ \boxed{\begin{array}{l} l_1' \leftarrow \text{quicksort}\ l_1 \\ l_2' \leftarrow \text{quicksort}\ l_2 \\ \text{length}\ (l_1' +\!\!+\ \mathit{pivot} :: l_2') \end{array}} \end{array} = \begin{array}{l} \mathit{len} \leftarrow \mathit{length}\, l \\ r \leftarrow \text{rand}\ \mathit{len} \\ \mathit{pivot} \leftarrow l[r] \\ \text{charge}\ (\mathit{len} - 1) \\ (l_1, l_2) \leftarrow \mathit{biFilter}\ (\lambda n.\ n \leq \mathit{pivot})\ (\text{drop}\ l) \\ \boxed{\begin{array}{l} n_1 \leftarrow (\text{length}\ l_1) \\ n_2 \leftarrow (\text{length}\ l_2) \\ n_1' \leftarrow \text{qck}_{\mathbb{N}}\ n_1 \\ n_2' \leftarrow \text{qck}_{\mathbb{N}}\ n_2 \\ \text{produce}\ (n_1' + 1 + n_2') \end{array}} \end{array}$$

Since the pivot is chosen uniformly at random, we can deduct denotationally that the following equations hold:

$$\begin{array}{l} \mathit{len} \leftarrow \mathit{length}\, l \\ r \leftarrow \text{rand}\ \mathit{len} \\ \mathit{pivot} \leftarrow l[r] \\ \text{charge}\ (\mathit{len} - 1) \\ (l_1, l_2) \leftarrow \mathit{biFilter}\ (\lambda n.\ n \leq \mathit{pivot})\ (\text{drop}\ l) \\ n_1 \leftarrow (\text{length}\ l_1) \\ n_2 \leftarrow (\text{length}\ l_2) \\ n_1' \leftarrow \text{qck}_{\mathbb{N}}\ n_1 \\ n_2' \leftarrow \text{qck}_{\mathbb{N}}\ n_2 \\ \text{produce}\ (n_1' + 1 + n_2') \end{array} = \begin{array}{l} \mathit{len} \leftarrow \text{length}\ l \\ r \leftarrow \text{rand}\ \mathit{len} \\ \text{charge}\ (\mathit{len} - 1) \\ n_1'' \leftarrow \text{qck}_{\mathbb{N}}\ r \\ n_2'' \leftarrow \text{qck}_{\mathbb{N}}\ (\mathit{len} - r - 1) \\ \text{produce}\ (n_1'' + 1 + n_2'') \end{array} = \begin{array}{l} \mathit{len} \leftarrow \text{length}\ l \\ \text{qck}_{\mathbb{N}}\ \mathit{len} \end{array}$$

The last equation holds under the inductive hypothesis that the list l is non-empty. This concludes the inductive case and the proof. □