Available online at www.sciencedirect.com

ScienceDirect

Electronic Notes in Theoretical Computer Science 347 (2019) 121–141

Electronic Notes in Theoretical Computer Science

www.elsevier.com/locate/entcs

On Bisimilarity in Lambda Calculi with Continuous Probabilistic Choice

Ugo Dal Lago[2]

University of Bologna
Bologna, Italy

Francesco Gavazzo[3]

IMDEA Software Institute
Madrid, Spain

Abstract

Applicative bisimiliarity is a coinductively-defined program equivalence in which programs are tested as argument-passing processes. Starting with the seminal work by Abramsky, applicative bisimiliarity has been proved to be a powerful technique for higher-order program equivalence. Recently, applicative bisimiliarity has also been generalised to lambda calculi with algebraic effects, and with discrete probabilistic choice in particular. In this paper, we show that applicative bisimiliarity behaves well in a lambda-calculus in which probabilistic choice is available in a more general form, namely through an operator for sampling of values from *continuous* distributions. Our main result shows that applicative bisimilarity is sound for contextual equivalence, hence providing a new reasoning principle for higher-order probabilistic languages.

Keywords: Probabilistic Computation, λ-calculus, Continuous Distributions.

1 Introduction

Program equivalence and refinement are central concepts in program semantics: giving meaning to programs has the positive effect of allowing one to *compare* programs and to dub them *equivalent* whenever having the same semantics. But what comes first, the chicken or the egg? After all, *any* equivalence relation between program phrases implicitly gives meaning to a program through their equivalence class.

Among the many possible notions of program equivalence, Morris' style contextual equivalence [37] has been considered as somehow canonical, being the largest,

[1] Thanks to the ANR projects 14CE250005 ELICA and 16CE250011 REPAS.

[2] Email: ugo.dallago@unibo.it

[3] Email: francesco.gavazzo@gmail.com

https://doi.org/10.1016/j.entcs.2019.09.007

and thus coarsest, adequate and compatible relation between programs: *by construction*, one cannot do any better than that when trying to identify programs behaving the same. As such, contextual equivalence is accepted as the reference notion of equivalence in an higher-order scenario. When proving that two programs are actually equivalent, however, the universal quantification over all contexts (i.e. over all program environments) in the definition of contextual equivalence makes the proof burdensome, if not impossible.

This has stimulated the development of alternative techniques for program equivalence which, in one way or another, *get rid of* the aforementioned universal quantification. Examples are logical relations [42,54,48], applicative bisimilarity [1], and denotational semantics [52,53]. Traditionally, the first step in studying the nature of these notions of program equivalence consists, again, in proving compatibility and adequacy, from which inclusion in contextual equivalence easily follows. In some cases this comes more or less by definition, while in other cases, like the one of applicative bisimilarity, proofs of compatibility are nontrivial, and are typically carried out in indirect ways, by methodologies like the so-called Howe's method [21]. Starting from Abramsky's pioneering work on applicative bisimilarity [1], coinduction has been proved to be a useful methodology for program equivalence in the context of pure λ-calculi.

All these (and others) techniques, give purely functional programming languages a clean semantics which makes it relatively easy to reason about them. Injecting various forms of effects (such as I/O, exceptions, or nondeterminism) complicates the matter considerably. Some of the aforementioned techniques for program equivalence have been proved to be remarkably robust in this respect, with contributions ranging from extensions encompassing *specific* notions of effect [6,14,29,12,24] to more abstract notions in which effects *of a certain kind* (e.g. algebraic effects [43,44]) are proved to work well in a given semantic framework [11,20,5,55,23].

One kind of effect which has received a lot of attention recently [56,56], but whose impact to the traditional techniques for program equivalence has been studied only marginally, is *continuous* probabilistic choice, i.e. the operation of sampling from continuous distributions. This by itself poses a challenge to the definition of an operational semantics, and thus to operationally-based techniques. Concerning the latter, the only contributions the authors are aware of are those by Culpepper et al. [10,61] on logical relations for a probabilistic λ-calculus with continuous distributions and scoring.

In this paper, we show that applicative (bi)simulation remains a sound methodology for program equivalence and refinement in presence of sampling from continuous distributions. We do it for a λ-calculus endowed with a very liberal recursive type system.

Noticeably, probabilistic bisimulation has been first defined in the abstract setting of discrete probabilistic transition systems, also known as labelled Markov chains [28], and later generalised to Markov decision processes [7]. The main technical difficulty one faces when trying to turn probabilistic bisimulation into a notion of equivalence for higher-order *programs* consists in coming up with a sensible notion of *lifting*, allowing to turn a relation between programs into a relation between program (sub)distributions [39]: there is a fundamental tension between the properties one typically gets from the aforementioned abstract notions of bisimilarity, and those one *needs* in the proofs of congruence. This paper shows that there is indeed a point in which the two meet.

The rest of the paper is organised as follows. We begin our analysis with an example of a simple program transformation. Proving correctness of such a transformation, however, is nontrivial and requires the design of suitable operational techniques. We then introduce our main vehicle calculus, namely a probabilistic call-by-value λ-calculus with sum and recursive types (Section 2), as well as its operational semantics (Section 2). Section 4 introduces the relational calculus we will use to study notions of program equivalence and refinement. We instantiate such a framework to define contextual equivalence and approximation (Section 5), and applicative (bi)simulation (Section 6). We prove soundness of applicative (bi)similarity in Section 6. We conclude this paper with a short discussion on full abstraction (Section 7).

1.1 Warming Up

Before entering into the technicalities of our anlaysis, we discuss a simple, yet nontrivial example of program equivalence in presence of continuous distributions. This way, we also introduce some features of the languages we are going to study.

We consider higher-order functional languages enriched with primitives for real-valued (measurable) functions and for sampling from continuous (and discrete) distributions. For instance, the program $\mathtt{normal}\ \mu\ \sigma$ samples a real number from the normal distribution with mean (encoded by the numeral) μ and standard deviation σ. Similarly, we can write a program $\mathtt{poisson}\ \mu$ which samples a real number from the Poisson distribution with rate μ.

We can also take advantage of the higher-order nature of functional languages, and use arbitrary programs e_μ, e_σ to compute the mean and standard deviation of the normal distribution:

$$\mathtt{let}\ x_\mu = e_\mu\ \mathtt{in}\ (\mathtt{let}\ x_\sigma = e_\sigma\ \mathtt{in}\ (\mathtt{normal}\ x_\mu\ x_\sigma^2)).$$

Notice that e_μ, e_σ may be defined sampling from other distributions as in:

$$\mathtt{let}\ x_\mu = (\mathtt{poisson}\ e_{\lambda_1})\ \mathtt{in}\ (\mathtt{let}\ x_\sigma = (\mathtt{poisson}\ e_{\lambda_2})\ \mathtt{in}\ (\mathtt{normal}\ x_\mu\ x_\sigma^2)).$$

This immediately raises the question of whether the order in which we evaluate (independent) expressions matter. That is, we ask whether the equivalence

$$\mathtt{let}\ x_1 = e_1\ \mathtt{in}\ (\mathtt{let}\ x_2 = e_2\ \mathtt{in}\ e_3) \equiv \mathtt{let}\ x_2 = e_2\ \mathtt{in}\ (\mathtt{let}\ x_1 = e_1\ \mathtt{in}\ e_3)$$

holds, assuming the variable x_1 not to be free in e_2, and similarity for x_2 and e_1. Such an equivalence is known as *commutativity*, and has been studied in [56,57] by means of denotational methods, and in [10,61] by means of logical relations (in both cases, however, the target language also includes primitives for conditioning).

Let us now look at a more sophisticated example. Consider the following instances of the combinators `map` `append`, and `zipwith`:

$$\begin{aligned}\mathtt{map} &: (\mathtt{unit} \to \mathtt{real}) \to \mathit{list}(\mathtt{unit}) \to \mathit{list}(\mathtt{real})\\ \mathtt{append} &: \mathit{list}(\mathtt{real}) \to \mathit{list}(\mathtt{real}) \to \mathit{list}(\mathtt{real})\\ \mathtt{zipwith} &: (\mathtt{real} \to \mathtt{real}) \to \mathit{list}(\mathtt{real}) \to \mathit{list}(\mathtt{real}) \to \mathit{list}(\mathtt{real})\end{aligned}$$

where `map` is the usual combinator for mapping functions on lists, `append` is the usual function for appending lists, and `zipwith` is a combinator taking as input a binary operation and two lists (which, for the sake of the argument, we assume to have the same length), and returning the list obtained by the pointwise application of the operation to the elements occurring at the same position in both lists. Let now zs be a list of n elements of type `unit` (we use zs to generate lists of random numbers), and let us write $\lambda.e : \mathtt{unit} \to \mathtt{real}$ for the thunk of a program $e : \mathtt{real}$. Is the following program transformation, which avoids a (possibly expensive) list traversing correct?

$$\lambda x.\left(\begin{array}{l}\mathtt{let}\ xs = \mathtt{map}\ (\lambda.\ \mathtt{normal}\ e_{\mu_1}\ e^2_{\sigma_1})\ zs\\ \qquad ys = \mathtt{map}\ (\lambda.\ \mathtt{normal}\ e_{\mu_2}\ e^2_{\sigma_2})\ zs\\ \ \mathtt{in}\ \mathtt{append}\ x\ (\mathtt{zipwith}\ (+)\ xs\ ys)\end{array}\right)$$

$$\equiv\ \lambda x.\left(\begin{array}{l}\mathtt{let}\ xs = \mathtt{map}\ (\lambda.\ \mathtt{normal}\ (e_{\mu_1}+e_{\mu_2})\ (e^2_{\sigma_1}+e^2_{\sigma_2}))\ zs\\ \ \mathtt{in}\ \mathtt{append}\ x\ xs\end{array}\right)$$

Proving the correctness of the above transformation requires to combine denotational reasoning about probability measures with operational reasoning about program behaviour. On the one hand, the two programs above are values, so that looking at their computational behaviour in isolation is useless: in order to understand what these programs do we have to test them against any possible input the environment can pass them (which might have a probabilistic behaviour itself). On the other hand, such programs are somehow related by basic results in statistics (notably, given random variables with normal probability distribution $X_1 \sim \mathcal{N}(\mu_1, \sigma_1^2)$ and $X_2 \sim \mathcal{N}(\mu_2, \sigma_2^2)$, we have $X_1 + X_2 \sim \mathcal{N}(\mu_1 + \mu_2, \sigma_1^2 + \sigma_2^2)$). As we will see, the techniques we develop in this paper give a handy proof of correctness of the above program transformation.

Now that the reader has some insights on the kind of problems we aim to solve in this work, we can start our formal analysis of program equivalence and refinement.

2 A Calculus with Continuous Probabilities: Syntax

The target language of this work is a probabilistic fine-grain call-by-value [34] with finite sums and recursive types reminiscent of probabilistic PCF [16,40]. The syntax and static semantics of $\Lambda_{\mathtt{P}}$ is defined in Figure 1.

Types of $\Lambda_{\mathtt{P}}$ are built starting from a countable set of type variables (denoted by the letter α in Figure 1) and the basic type `real` for real numbers, using finite sum, arrow, and recursive type constructors. In particular, in a type of the form $\sum_{i \in I} \tau_i$ we assume the letter I to stand for a finite set whose elements are denoted by $\hat{\imath}, \hat{\jmath}, \ldots$. We write `void` and `unit` for the empty sum type and $\mathtt{void} \to \mathtt{void}$, respectively.

Terms of $\Lambda_{\mathtt{P}}$ are divided in two classes: values and expressions (also called computations). Intuitively, a value is the result of a computation, whereas an expression is a value producer, i.e. a term that once evaluated may produce a value (the evaluation process might not terminate) as well as side effects (the latter being probabilistic). Accordingly, computations must be explicitly sequenced by means of the sequencing constructor $\mathtt{let}\ x = -\ \mathtt{in}\ -$.

We use judgments of the form $\Gamma \vdash^{\Lambda} e : \tau$ for expressions, and $\Gamma \vdash^{\mathcal{V}} v : \tau$ for values. In a judgment of the form $\Gamma \vdash^{\Lambda} e : \tau$ (resp. $\Gamma \vdash^{\mathcal{V}} v : \tau$), Γ denotes an environment, i.e. a finite sequence $x_1 : \tau_1, \ldots, x_n : \tau_n$ of distinct variables with associated *closed* types (we denote by $\cdot$ the empty environment), τ is a *closed* type, and e (resp. v) is an expression (resp. value) with free variables among Γ. Notice that we work with closed types only. We use the letter e (possibly with sup- and subscripts) to denote expressions, and v (possibly with sup- and subscripts) to denote values. We also refer to sequents $\Gamma \vdash^{\Lambda} \tau$ as computation or expression sequents, and to $\Gamma \vdash^{\mathcal{V}} \tau$ as value sequents. When the distinction between values and expressions is not relevant, we generically refer to terms.

Each real number r is represented in $\Lambda_{\mathtt{P}}$ by the value $\mathtt{r}$. Additionally, $\Lambda_{\mathtt{P}}$ is parametric with respect to a $\mathbb{N}$-indexed family $\mathcal{C}$ of *countable* sets $\mathcal{C}_n$, each of which contains (symbols for) measurable functions from $\mathbb{R}^n$ to $\mathbb{R}$. In particular, we assume standard real-valued arithmetic operations to be in $\mathcal{C}$. We use letters $F, G, \ldots$ to denote elements in $\mathcal{C}_n$. Notice that a function $F \in \mathcal{C}_n$ has values as arguments, rather than expressions. Indeed, we can encode the expression $F(e_1, \ldots, e_n)$ as [4] $\mathtt{let}\ x_1 = e_1\ \mathtt{in}\ (\mathtt{let}\ x_2 = e_2\ \mathtt{in}\ \ldots (\mathtt{let}\ x_n = e_n\ \mathtt{in}\ F(x_1, \ldots, x_n)))$.

Finally, the expression `sample` stands for the uniform distribution over the unit interval, which is nothing but the Lebesgue measure λ on $[0, 1]$. As it is shown in e.g. [40,16], starting from `sample` it is possible to define several probabilistic measures (e.g. binomial, geometric, and exponential distribution) using functions in $\mathcal{C}$.

Example 2.1 Given closed terms e_μ, e_σ of type `real` (encoding mean μ and standard deviation σ) we represent the normal distribution with mean μ and standard deviation σ as the $\Lambda_{\mathtt{P}}$ expression $\mathtt{normal}\ e_\mu\ e_\sigma$, where the term $\mathtt{normal}_{\mathtt{std}}$ encodes the normal distribution with mean 0 and standard deviation 1. Notice that the

[4] This encoding reflects the standard semantics of operations, where to evaluate an expression of the form $F(e_1, \ldots, e_n)$ one first sequentially evaluates its arguments, proceeding from left to right.

expressions e_μ, e_σ may be themselves defined in terms of other distributions:

$$\mathtt{normal}_{\mathtt{std}} \triangleq \mathtt{let}\ x = \mathtt{sample}\ \mathtt{in}\ (\mathtt{let}\ y = \mathtt{sample}\ \mathtt{in}\ (\sqrt{-2\log(x)}\cos(2\pi y)))$$
$$\mathtt{normal}\ e_\mu\ e_\sigma \triangleq \mathtt{let}\ x_\mu = e_\mu,\ x_\sigma = e_\sigma, y = \mathtt{normal}_{\mathtt{std}}\ \mathtt{in}\ ((x_\sigma * y) + x_\mu).$$

We adopt the standard syntactical conventions as in [3], notably the so-called variable convention. In particular, we denote by $FV(e)$ (resp. $FV(v)$) the collection of free variables of e (resp. v) (notice that, e.g., in a term of the form $\mathtt{case}\ v\ \mathtt{of}\ \{\mathtt{fold}\ x \to e\}$ the variable x is bound in e). We refer to closed expressions as *programs*. We denote by $e[v/x]$ (resp. $v'[v/x]$) the capture-free substitution of the value v for all free occurrences of x in e (resp. v') and identify terms up to renaming of bound variables. We extend the aforementioned conventions to types. For instance, we denote by $\tau_1[\tau_2/\alpha]$ the result of capture-avoiding substitution of the type τ_2 for the type variable α in τ_1. Finally, we use the notation Λ and $\mathcal{V}$ to denote the collections of all typable expressions and values, respectively.

$$\frac{}{\Gamma, x:\tau \vdash^{\mathcal{V}} x:\tau} \qquad \frac{\Gamma \vdash^{\mathcal{V}} v:\tau}{\Gamma \vdash^{\Lambda} v:\tau} \qquad \frac{r \in \mathbb{R}}{\Gamma \vdash^{\mathcal{V}} \mathtt{r}:\mathtt{real}} \qquad \frac{}{\Gamma \vdash^{\Lambda} \mathtt{sample}:\mathtt{real}} \qquad \frac{\Gamma \vdash^{\mathcal{V}} v_1:\mathtt{real} \quad \cdots \quad \Gamma \vdash^{\mathcal{V}} v_n:\mathtt{real} \quad F \in \mathcal{C}_n}{\Gamma \vdash^{\Lambda} F(v_1,\ldots,v_n):\mathtt{real}}$$

$$\frac{\Gamma, x:\tau_1 \vdash^{\Lambda} e:\tau_2}{\Gamma \vdash^{\mathcal{V}} \lambda x.e:\tau_1 \to \tau_2} \qquad \frac{\Gamma \vdash^{\mathcal{V}} v_1:\tau_1 \to \tau_2 \quad \Gamma \vdash^{\mathcal{V}} v_2:\tau_2}{\Gamma \vdash^{\Lambda} v_1 v_2:\tau_2} \qquad \frac{\Gamma \vdash^{\Lambda} e_1:\tau_1 \quad \Gamma, x:\tau_1 \vdash^{\Lambda} e_2:\tau_2}{\Gamma \vdash^{\Lambda} \mathtt{let}\ x = e_1\ \mathtt{in}\ e_2:\tau_2} \qquad \frac{\Gamma \vdash^{\mathcal{V}} v:\tau_{\hat{\imath}}}{\Gamma \vdash^{\mathcal{V}} \langle \hat{\imath}, v\rangle : \sum_{i\in I}\tau_i}$$

$$\frac{\Gamma \vdash^{\mathcal{V}} v:\sum_{i\in I}\tau_i \quad \Gamma, x:\tau_i \vdash^{\Lambda} e_i:\tau\ (\forall i \in I)}{\Gamma \vdash^{\Lambda} \mathtt{case}\ v\ \mathtt{of}\ \{\langle i,x\rangle \to e_i\}:\tau} \qquad \frac{\Gamma \vdash^{\mathcal{V}} v:\tau[\mu\alpha.\tau/\alpha]}{\Gamma \vdash^{\mathcal{V}} \mathtt{fold}\ v:\mu\alpha.\tau} \qquad \frac{\Gamma \vdash^{\mathcal{V}} v:\mu\alpha.\tau_1 \quad \Gamma, x:\tau_1[\mu\alpha.\tau_1/\alpha] \vdash^{\Lambda} e:\tau_2}{\Gamma \vdash^{\Lambda} \mathtt{case}\ v\ \mathtt{of}\ \{\mathtt{fold}\ x \to e\}:\tau_2}$$

Fig. 1. Static semantics of $\Lambda_{\mathtt{P}}$.

3 A Calculus with Continuous Probabilities: Semantics

The dynamic semantics of $\Lambda_{\mathtt{P}}$ is given by a type-indexed family of evaluation functions $[\![-]\!]_\tau$ mapping programs of type τ to sub-probability measures over values of type τ. The collection of such measures can be abstractly described using the *sub-Giry monad* $\mathcal{G}$ [18], so that the evaluation function $[\![-]\!]_\tau$ will map a program $e \in \Lambda_\tau$ to a sub-probability measure $[\![e]\!]_\tau \in \mathcal{G}(\mathcal{V}_\tau)$. Additionally, since $\mathcal{G}X$ carries a measurable space structure whenever X does, we would like $[\![-]\!]_\tau$ to be a measurable function. For that to make sense, however, we first need to make Λ a measurable space itself. We follow [57,50,16] and rely on the measurable space structure of $\mathbb{R}^n$ and coproducts in **Meas**, the category of measurable spaces[5]. Before entering into the details we recall some useful mathematical preliminaries.

3.1 Mathematical Preliminaries: Stochastic Kernels and the Giry Monad

In this section we recall some mathematical preliminaries which are needed in order to give $\Lambda_{\mathtt{P}}$ dynamic semantics. Unfortunately, there is no hope to be comprehensive,

[5] Another approach is given in [8], where the set of terms inherits a measurable structure from a syntax-oriented metric.

and thus we assume the reader to be familiar with basic measure theory [45], domain theory [2], and category theory [35].

We denote by **Meas** the category of measurable spaces and measurable functions, by **Set** the category of sets and functions, and by ω**Cppo** the category of ω-cppo and continuous functions. Given a measurable space (X, Σ_X) with a measure μ on it and a function $f : X \to [0, \infty]$, we denote by $\int_X f \mathrm{d}\mu$ or $\int_X f(x)\mu(\mathrm{d}x)$ the number in $[0, \infty]$ obtained by Lebesgue integrating f over X with respect to μ (when clear from the context, we will omit the subscript X).

In order to deal with nontermination, we work with sub-probability distributions rather than with full distributions. For instance, the measure $[\![\Omega]\!]$ associated to the purely divergent expression Ω will naturally assign 0 to any measurable sets of values. Mathematically, this means working with the sub-Giry monad rather than with the (full) Giry monad.

Definition 3.1 The sub-Giry functor $\mathcal{G} : \mathbf{Meas} \to \mathbf{Meas}$ maps a measurable space (X, Σ_X) to $(\mathcal{G}X, \Sigma_{\mathcal{G}X})$, where $\mathcal{G}X$ is the set of all sub-probability measures on X, and $\Sigma_{\mathcal{G}X}$ is the least σ-algebra making the Σ_X-indexed family of maps $\mathsf{eval}_A : \mathcal{G}(X) \to [0, 1]$, defined by $\mathsf{eval}_A(\nu) = \nu(A)$, measurable. Measurable functions $f : X \to Y$ are mapped to $\mathcal{G}f : \mathcal{G}X \to \mathcal{G}Y$, where $(\mathcal{G}f)(\nu) \in \mathcal{G}X = \nu(f^{-1}(-))$.

The functor $\mathcal{G}$ carries a monad structure, with unit $\eta_X : X \to \mathcal{G}X$ given by Dirac measures (recall that for any $x \in X$, the Dirac measure δ_x on X is defined as $\delta_x(A) = 1$ if $x \in A$, and $\delta_x(A) = 0$, otherwise), and multiplication $\mu_X : \mathcal{G}\mathcal{G}X \to \mathcal{G}X$ defined as $\mu_X(M)(A) = \int_{\mathcal{G}X} \mathsf{eval}_A(x)M(\mathrm{d}x)$, for $A \in \Sigma_X$. Equivalently, we can use $\mathcal{G}$ to build a Kleisli triple [35] defining[6] $f^\dagger(\nu)(B) = \int f(x)(B)\nu(\mathrm{d}x)$, for $f : X \to \mathcal{G}Y$. Actually, Fubini's Theorem implies that $\mathcal{G}$ has the structure of a commutative monad, with double strength $\mathsf{dst}_{X,Y} : \mathcal{G}X \times \mathcal{G}Y \to \mathcal{G}(X \times Y)$ mapping measures $\mu \in \mathcal{G}X$, $\nu \in \mathcal{G}Y$ to their product measure $\mu \otimes \nu$.

A measurable function $f : X \to \mathcal{G}Y$ is also called a (stochastic) *kernel* [38] from X to Y, as we can uncurry f obtaining a map $f' : X \times \Sigma_Y \to [0, 1]$. We write **Kern** for the Kleisli category of $\mathcal{G}$. The category **Kern** enjoys several nice properties [38]. In particular, the collection of measurable functions from X to $\mathcal{G}Y$ carries an ω-cppo structure under the pointwise order $\sqsubseteq$ (i.e. $f \sqsubseteq g$ if and only if $\forall x \in X.\ \forall A \in \Sigma_Y.\ f(x)(A) \leq g(x)(A)$). The bottom element $\bot$ is defined as the always zero distribution, whereas the least upper bound of an ω-chain $(f_n)_{n \geq 0}$ is defined as $\sup_n f_n$. Additionally, the monotone convergence theorem and the very definition of Lebesgue integration give the following identities: $\int \sup_n f_n \mathrm{d}\nu = \sup_n \int f_n \mathrm{d}\nu$, $\int f d(\sup_n \nu_n) = \sup_n \int f \mathrm{d}\nu_n$. Abstractly, this means that **Kern** is ω**Cppo**-enriched [26]. Finally, we recall that **Meas** is cartesian (although not cartesian closed) and that **Kern** has countable coproducts. For a countable family (X_i, Σ_{X_i}) of measurable spaces we denote by $\coprod_i (X_i, \Sigma_{X_i})$ its coproduct.

[6] Notice that the function mapping an element x in X to $f(x)(A) \in [0, 1]$ is integrable.

3.2 Evaluation Semantics

We define a (type-indexed) map $[\![-]\!]_\tau : \Lambda_\tau \to \mathcal{G}(\mathcal{V}_\tau)$ associating to each program $e \in \Lambda_\tau$ a (sub)probability measure over values in $\mathcal{V}_\tau$. We rely on the ω**Cppo**-enrichment of **Kern** to deal with non-termination.

Definition 3.2 Define the type-indexed family of ($\mathbb{N}$-indexed) maps $[\![-]\!]^{(n)}_\tau : \Lambda_\tau \to \mathcal{G}(\mathcal{V}_\tau)$ as follows (for readability, we omit type annotations in $[\![-]\!]^{(n)}$):

$$\begin{aligned}
[\![e]\!]^{(0)}(A) &= 0\\
[\![v]\!]^{(n+1)}(A) &= \delta_v(A)\\
[\![F(\mathtt{r}_1,\ldots,\mathtt{r}_n)]\!]^{(n+1)}(A) &= [\![\mathtt{F}(\mathtt{r}_1,\ldots,\mathtt{r}_\mathtt{n})]\!]^{(n)}(A)\\
[\![\mathtt{sample}]\!]^{(n+1)}(A) &= \lambda\{r \mid \mathtt{r} \in A\}\\
[\![(\lambda x.e)v]\!]^{(n+1)}(A) &= [\![e[v/x]]\!]^{(n)}(A)\\
[\![\mathtt{let}\ x = e_1\ \mathtt{in}\ e_2]\!]^{(n+1)}(A) &= \int [\![e_2[v/x]]\!]^{(n)} [\![e_1]\!]^{(n)}(\mathrm{d}v)\\
[\![\mathtt{case}\ \langle \hat{\imath}, v\rangle\ \mathtt{of}\ \{\langle i, x\rangle \to e_i\}]\!]^{(n+1)}(A) &= [\![e_{\hat{\imath}}[v/x]]\!]^{(n)}(A)\\
[\![\mathtt{case}\ (\mathtt{fold}\ v)\ \mathtt{of}\ \{\mathtt{fold}\ x \to e\}]\!]^{(n+1)}(A) &= [\![e[v/x]]\!]^{(n)}(A).
\end{aligned}$$

Lemma 3.3 *For any type τ and natural number n, the map $[\![-]\!]^{(n)}_\tau$ is a kernel.*

Proof. [Sketch] First of all, we notice that for the statement of Lemma 3.3 to make sense, we first need to make Λ_τ (as well as $\mathcal{V}_\tau$) a measurable space. This can be done as in [16,57], taking advantages of coproducts in **Meas**. By observing that for any expression $\Gamma, x : \tau_1 \vdash^\Lambda e : \tau_2$, the map $e[-/x] : \mathcal{V}_{\Gamma\vdash\tau_1} \to \Lambda_{\Gamma\vdash\tau_2}$ is measurable, one can easily show $[\![-]\!]^{(n)}$ to be measurable. □

Additionally, we see that $[\![e]\!]^{(n)}_\tau$ is actually an ω-chain in $\mathcal{G}(\mathcal{V}_\tau)$, so that we can define $[\![e]\!]_\tau$ as $\sup_n [\![e]\!]^{(n)}_\tau$. In particular, due to ω**Cppo**-enrichment of **Kern**, we can replace all the occurrences of $[\![-]\!]^{(n)}$ in Definition 3.2 with $[\![-]\!]$. As usual, where possible we omit type subscripts in $[\![-]\!]$.

Having defined the operational behaviour of programs, we can finally move to the study of notions of program equivalence and refinement.

4 Probabilistic Equivalence and Refinement

In the next sections we define notions of program equivalence and refinement for $\Lambda_\mathtt{P}$. Specifically, we introduce (probabilistic) contextual equivalence and approximation, and (probabilistic) applicative (bi)similarity. Our main result states that applicative similarity (resp. bisimilarity) is a sound proof technique for contextual approximation (resp. equivalence).

Studying notions of program equivalence and refinement, it is useful to fix a proper notation and vocabulary for program relations. We do so following [29,19].

4.1 Relational Calculus

We use letter R, $S, \ldots$ to range over relations and write $\mathsf{Rel}(X, Y)$ for the set of relations over X and Y. We call elements of $\mathsf{Rel}(X, X)$ *endorelations*. Given a relation $R \subseteq X \times Y$ and set $A \subseteq X$, we denote by $R[A]$ the R-image of A, i.e. $\{y \in Y \mid \exists x \in A.\ (x, y) \in R\}$. For relations $R \subseteq X \times Y$, $S \subseteq Y \times Z$, we write $S \circ R$ for their composition, and denote by $=_X$ the equality relation on X. As usual, an endorelation $R \in \mathsf{Rel}(X, X)$ is reflexive if $=_X \subseteq R$, transitive if $R \circ R \subseteq R$, and symmetric if $R^\top \subseteq R$, where $R^\top$ denotes the converse or transpose of R. We also recall that $\mathsf{Rel}(X, X)$ carries a complete lattice structure with order given by set-theoretic inclusion.

Relational reasoning about programs oftentimes require to reason about open terms. It is thus useful to work with families of relations relating expressions typable within the same sequent. We refer to such (families of) relations as *term relations*.

Definition 4.1 A term relation R associates to each computation sequent $\Gamma \vdash^\Lambda \tau$ (resp. value sequent $\Gamma \vdash^\mathcal{V} \tau$) a relation $\Gamma \vdash^\Lambda - R - : \tau$ (resp. $\Gamma \vdash^\mathcal{V} - R - : \tau$) between its inhabitants.

Formally, a term relation is thus an element in the set $\mathsf{Rel} = \prod_{\Gamma \vdash \tau} \mathsf{Rel}(\Lambda_{\Gamma \vdash \tau}, \Lambda_{\Gamma \vdash \tau}) \times \mathsf{Rel}(\mathcal{V}_{\Gamma \vdash \tau}, \mathcal{V}_{\Gamma \vdash \tau})$. The latter inherits a complete lattice structure from $\mathsf{Rel}(\Lambda_{\Gamma \vdash \tau}, \Lambda_{\Gamma \vdash \tau})$ and $\mathsf{Rel}(\mathcal{V}_{\Gamma \vdash \tau}, \mathcal{V}_{\Gamma \vdash \tau})$ pointwise, this way allowing one to define term relations both inductively and coinductively. Concerning notation, we write $R \subseteq S$ if:

$$\forall \Gamma, e_1, e_2.\ \Gamma \vdash^\Lambda e_1 \mathrel{R} e_2 \implies \Gamma \vdash^\Lambda e_1 \mathrel{S} e_2, \quad \forall \Gamma, v_1, v_2.\ \Gamma \vdash^\mathcal{V} v_1 \mathrel{R} v_2 \implies \Gamma \vdash^\mathcal{V} v_1 \mathrel{S} v_2.$$

Example 4.2 The identity term relation is defined by the (0-ary) rules: $\Gamma \vdash^\Lambda e =_\Lambda e : \tau$, $\Gamma \vdash^\mathcal{V} v =_\mathcal{V} v : \tau$. Given term relations R, S, we define $S \circ R$ as follows:

$$\frac{\Gamma \vdash^\Lambda e_1 \mathrel{R} e_2 : \tau \quad \Gamma \vdash^\Lambda e_2 \mathrel{S} e_3 : \tau}{\Gamma \vdash^\Lambda e_1 \mathrel{(S \circ R)} e_3 : \tau} \qquad \frac{\Gamma \vdash^\mathcal{V} v_1 \mathrel{R} v_2 : \tau \quad \Gamma \vdash^\mathcal{V} v_2 \mathrel{S} v_3 : \tau}{\Gamma \vdash^\mathcal{V} v_1 \mathrel{(S \circ R)} v_3 : \tau}$$

The notions of a *preorder* and *equivalence* term relation are defined as usual. In fact, we say that a term relation R is reflexive if $= \subseteq R$, transitive if $R \circ R \subseteq R$, and symmetric if $R^\top \subseteq R$ (where $R^\top$ is defined in the obvious way).

Oftentimes we will be interested in defining relations between programs only. We denote by Rel_0 the subspace $\prod_\tau \mathsf{Rel}(\Lambda_\tau, \Lambda_\tau) \times \mathsf{Rel}(\mathcal{V}_\tau, \mathcal{V}_\tau)$ of Rel, and refer to its elements as *closed term relations*. Every closed term relation $R \in \mathsf{Rel}_0$ induces a term relation, called the *open extension* of R, as follows: given $\Gamma \vdash^\Lambda e : \tau$, let γ range over closed substitutions for Γ (i.e. for any variable $(x_i : \tau_i) \in \Gamma$, $\gamma(x_i) \in \mathcal{V}_{\tau_i}$). Define:

$$\Gamma \vdash^\Lambda e_1 \mathrel{R} e_2 : \tau \iff \forall \gamma.\ \cdot \vdash^\Lambda e_1\gamma \mathrel{R} e_2\gamma : \tau.$$

A similar definition can be given for values. Given a closed term relation R, we sometimes write $(R_\Lambda, R_\mathcal{V})$ to denote the (type-indexed families of) relations defining its action on expressions and values.

We tacitly require term relations to be closed under weakening, meaning that the following hold.

$$\frac{\Gamma \vdash^{\Lambda} e_1 \mathrel{R} e_2 : \tau}{\Gamma, x : \tau' \vdash^{\Lambda} e_1 \mathrel{R} e_2 : \tau} \qquad \frac{\Gamma \vdash^{\mathcal{V}} v_1 \mathrel{R} v_2 : \tau}{\Gamma, x : \tau' \vdash^{\mathcal{V}} v_1 \mathrel{R} v_2 : \tau}$$

Notice that the open extension of a closed relation is indeed closed under weakening.

In order to formalise compositional reasoning about program behaviour, we introduce the notion of a compatible term relation, i.e. of a term relation closed under term constructors of $\Lambda_{\mathtt{P}}$.

Definition 4.3 Given a term relation R, its compatible refinement $\widehat{R}$ is defined by the rules in Figure 2. A term relation R is compatible if $\widehat{R} \subseteq R$, and a closed term relation is compatible if its open extension is.

$$\frac{}{\Gamma, x : \tau \vdash^{\mathcal{V}} x \mathrel{\widehat{R}} x : \tau} \quad \frac{\Gamma \vdash^{\mathcal{V}} v_1 \mathrel{\widehat{R}} v_2 : \tau}{\Gamma \vdash^{\Lambda} v_1 \mathrel{\widehat{R}} v_2 : \tau} \quad \frac{}{\Gamma \vdash^{\mathcal{V}} \mathtt{r} \mathrel{\widehat{R}} \mathtt{r} : \mathtt{real}} \quad \frac{}{\Gamma \vdash^{\Lambda} \mathtt{sample} \mathrel{\widehat{R}} \mathtt{sample} : \mathtt{real}}$$

$$\frac{\Gamma \vdash^{\mathcal{V}} v_1 \mathrel{R} v_1' : \mathtt{real} \quad \cdots \quad \Gamma \vdash^{\mathcal{V}} v_n \mathrel{R} v_n' : \mathtt{real}}{\Gamma \vdash^{\Lambda} F(v_1, \ldots, v_n) \mathrel{\widehat{R}} F(v_1', \ldots, v_n') : \mathtt{real}} \quad \frac{\Gamma, x : \tau_1 \vdash^{\Lambda} e_1 \mathrel{R} e_2 : \tau_2}{\Gamma \vdash^{\mathcal{V}} \lambda x.e_1 \mathrel{\widehat{R}} \lambda x.e_2 : \tau_1 \to \tau_2} \quad \frac{\Gamma \vdash^{\mathcal{V}} v_1 \mathrel{R} v_1' : \tau_1 \to \tau_2 \quad \Gamma \vdash^{\mathcal{V}} v_2 \mathrel{R} v_2' : \tau_1}{\Gamma \vdash^{\Lambda} v_1 v_2 \mathrel{\widehat{R}} v_1' v_2' : \tau_2}$$

$$\frac{\Gamma \vdash^{\Lambda} e_1 \mathrel{R} e_1' : \tau_1 \quad \Gamma, x : \tau_1 \vdash^{\Lambda} e_2 \mathrel{R} e_2' : \tau_2}{\Gamma \vdash^{\Lambda} \mathtt{let}\ x = e_1\ \mathtt{in}\ e_2 \mathrel{\widehat{R}} \mathtt{let}\ x = e_1'\ \mathtt{in}\ e_2' : \tau_2} \quad \frac{\Gamma \vdash^{\mathcal{V}} v_1 \mathrel{R} v_2 : \tau_{\hat{\imath}}}{\Gamma \vdash^{\mathcal{V}} \langle \hat{\imath}, v_1 \rangle \mathrel{\widehat{R}} \langle \hat{\imath}, v_2 \rangle : \sum_{i \in I} \tau_i} \quad \frac{\Gamma \vdash^{\mathcal{V}} v_1 \mathrel{R} v_2 : \tau[\mu\alpha.\tau/\alpha]}{\Gamma \vdash^{\mathcal{V}} \mathtt{fold}\ v_1 \mathrel{\widehat{R}} \mathtt{fold}\ v_2 : \mu\alpha.\tau}$$

$$\frac{\Gamma \vdash^{\mathcal{V}} v \mathrel{R} v' : \sum_{i \in I} \tau_i \quad \Gamma, x : \tau_i \vdash^{\Lambda} e_i \mathrel{R} e_i' : \tau \quad (\forall i \in I)}{\Gamma \vdash^{\Lambda} \mathtt{case}\ v\ \mathtt{of}\ \{\langle i, x \rangle \to e_i\} \mathrel{\widehat{R}} \mathtt{case}\ v'\ \mathtt{of}\ \{\langle i, x \rangle \to e_i'\} : \tau} \quad \frac{\Gamma \vdash^{\mathcal{V}} v \mathrel{R_{\mathcal{V}}} v' : \mu\alpha.\tau_1 \quad \Gamma, x : \tau_1[\mu\alpha.\tau_1/\alpha] \vdash^{\Lambda} e \mathrel{R} e' : \tau_2}{\Gamma \vdash^{\Lambda} \mathtt{case}\ v\ \mathtt{of}\ \{\mathtt{fold}\ x \to e\} \mathrel{\widehat{R}} \mathtt{case}\ v'\ \mathtt{of}\ \{\mathtt{fold}\ x \to e'\} : \tau_2}$$

Fig. 2. Compatible refinement.

Notice that $\widehat{R}$ is indeed a term relation (notably, $\widehat{R}$ is closed under weakening). Definition 4.3 induces a monotone endofuction $\widehat{-}$ on Rel which distributes over composition ($\widehat{S \circ R} = \widehat{S} \circ \widehat{R}$) and transpose ($\widehat{R^{\top}} = (\widehat{R})^{\top}$). In particular, a term relation is compatible if and only if it is a pre-fixed point of $\widehat{-}$. It is not hard to prove that the identity term relation of Example 4.2 is a pre-fixed point of $\widehat{-}$, and actually the least such. As a consequence, any compatible relation is reflexive.

Lemma 4.4 *The collection of compatible λ-term relations ordered by $\subseteq$ has a complete lattice .*

Lemma 4.4 allows us to define compatible λ-term relations both inductively and coinductively. Another central notion dealing with term relation is the one of *substitutivity.*

Definition 4.5 (i) A term relation R is *value-substitutive* if the following hold:

$$\frac{\Gamma, x : \tau_1 \vdash^{\Lambda} e_1 \mathrel{R} e_2 : \tau_2 \quad \cdot \vdash^{\mathcal{V}} v : \tau_1}{\Gamma \vdash^{\Lambda} e_1[v/x] \mathrel{R} e_2[v/x] : \tau_2} \qquad \frac{\Gamma, x : \tau_1 \vdash^{\mathcal{V}} v_1 \mathrel{R} v_2 \quad \cdot \vdash^{\mathcal{V}} v : \tau_1}{\Gamma \vdash^{\mathcal{V}} v_1[v/x] \mathrel{R} v_2[v/x] : \tau_2}$$

(ii) A term relation R is *substitutive* if the following hold:

$$\frac{\Gamma, x : \tau_1 \vdash^{\Lambda} e_1 \mathrel{R} e_2 : \tau_2 \quad \cdot \vdash^{\mathcal{V}} v_1 \mathrel{R} v_2 : \tau_1}{\Gamma \vdash^{\Lambda} e_1[v_1/x] \mathrel{R} e_2[v_2/x] : \tau_2}$$

$$\frac{\Gamma, x : \tau_1 \vdash^{\mathcal{V}} u_1 \mathrel{R} u_2 : \tau_2 \quad \cdot \vdash^{\mathcal{V}} v_1 \mathrel{R} v_2 : \tau_1}{\Gamma \vdash^{\mathcal{V}} u_1[v_1/x] \mathrel{R} u_2[v_2/x] : \tau_2}$$

A closed relation is (value) substitutive if its open extension is. Moreover, we notice that the open extension of a closed term relation is trivially value-substitutive.

5 Contextual Approximation and Equivalence

Contextual equivalence [37] equates programs that behave the same in any possible environment. The notion of an environment is formalised by means of *contexts*, which are, roughly speaking, terms with a hole to be filled in with the program we aim to test. In order to avoid syntactic bureaucracy, we follow [29,19] and give a property-based definition of contextual equivalence and approximation based on the notion of *(pre)adequacy*. For readability, from now on we write $\mathtt{R}$ for $\mathcal{V}_{\mathtt{real}} \in \Sigma_{\mathtt{real}}$.

Definition 5.1 Given a term relation R, we say that R is *preadequate* (resp. *adequate*) if:

$$\cdot \vdash^{\Lambda} e_1 \mathrel{R} e_2 : \mathtt{real} \implies [\![e_1]\!](\mathtt{R}) \leq [\![e_2]\!](\mathtt{R});$$

$$(\text{resp. } \cdot \vdash^{\Lambda} e_1 \mathrel{R} e_2 : \mathtt{real} \implies [\![e_1]\!](\mathtt{R}) = [\![e_2]\!](\mathtt{R})).$$

Definition 5.2 Contextual approximation $\preceq^{\mathsf{ctx}}$ (resp. contextual equivalence $\simeq^{\mathsf{ctx}}$) is the largest compatible and preadequate (resp. adequate) term relation.

Definition 5.2 is not well-posed, as it is not clear whether such largest term relations exist. In fact, (pre)adequacy is not a monotone property, meaning that we cannot define $\preceq^{\mathsf{ctx}}$ (resp. $\simeq^{\mathsf{ctx}}$) as the greatest fixed point of the (non-monotone) endofuction on Rel induced by Definition 5.2. Nonetheless, we see that the the union of compatible (pre)adequate term relations is itself a compatible (pre)adequate term relation.

Additionally, Definition 5.2 gives a proof principle for contextual approximation (resp. equivalence) resembling a coinduction proof principle. In order to prove that a term relation R is included in $\preceq^{\mathsf{ctx}}$ (resp. $\simeq^{\mathsf{ctx}}$), it is sufficient to show that R is preadequate (resp. adequate) and compatible. Using such a proof principle it is a straightforward exercise to prove that $\preceq^{\mathsf{ctx}}$ is a precongruence and that $\simeq^{\mathsf{ctx}}$ is a congruence term relation.

Example 5.3 Contextual equivalence gives the structural equalities in Figure 3, as well the quasi-denotational law: $\forall e_1, e_2 \in \Lambda_\tau.\ [\![e_1]\!] = [\![e_2]\!] \implies \cdot \vdash^{\Lambda} e_1 \simeq^{\mathsf{ctx}} e_2 : \tau$. The quasi-denotational law can be used to export powerful results from measure theory, for instance (we slightly abuse notation by desequencing some computations):

$$\begin{aligned}&\texttt{let } x_1 = (\texttt{normal } e_{\mu_1}\ e^2_{\sigma_1}) \texttt{ in } (\texttt{let } x_2 = (\texttt{normal } e_{\mu_2}\ e^2_{\sigma_2}) \texttt{ in } (x_1 + x_2))\\ &\qquad \simeq^{\mathsf{ctx}} \texttt{normal } (e_{\mu_1} + e_{\mu_2})\ (e^2_{\sigma_1} + e^2_{\sigma_2})\end{aligned}$$

Combining such laws with the structural equivalences in Figure 3 we can prove non-trivial program equivalences. For instance, recalling the program transformation studied in Section 1.1, we see that contextual equivalence gives correctness of the following program transformation, which avoid a list traversing.

$$\left(\begin{array}{l}\texttt{let } xs = \texttt{map } (\lambda.\ \texttt{normal } e_{\mu_1}\ e^2_{\sigma_1})\ zs\\ \quad\ ys = \texttt{map } (\lambda.\ \texttt{normal } e_{\mu_2}\ e^2_{\sigma_2})\ zs\\ \ \texttt{in append } x\ (\texttt{zipwith } (+)\ xs\ ys)\end{array}\right) \simeq^{\mathsf{ctx}} \left(\begin{array}{l}\texttt{let } xs = \texttt{map } (\lambda.\ \texttt{normal } (e_{\mu_1} + e_{\mu_2})\ (e^2_{\sigma_1} + e^2_{\sigma_2}))\ zs\\ \ \texttt{in append } x\ xs\end{array}\right)$$

Similarly, we can prove the correctness of the program transformation studied in Section 1.1. Proving such equivalences and laws from first principles, however, is highly non-trivial. We will obtain these results relying on our notion of applicative bisimulation.

$$\begin{array}{rll}(\lambda x.e)v &\simeq^{\mathsf{ctx}} e[v/x] &\\ \texttt{let } x = v \texttt{ in } e &\simeq^{\mathsf{ctx}} e[v/x] &\\ \texttt{let } x = e_1 \texttt{ in } e_2 &\simeq^{\mathsf{ctx}} e_2 & (x \notin FV(e_2))\\ \texttt{let } x = e \texttt{ in } (\texttt{let } y = e' \texttt{ in } e'') &\simeq^{\mathsf{ctx}} \texttt{let } y = e' \texttt{ in } (\texttt{let } x = e \texttt{ in } e'') & (x \notin FV(e'), y \notin FV(e))\end{array}$$

Fig. 3. Structural identities.

6 Applicative (Bi)simulation

Abramsky's applicative bisimilarity [1] tests programs as argument-passing processes, i.e. by allowing the environment to interact with a program by passing it arbitrary inputs only. In a probabilistic setting, however, the result of a program is not a value, but a subdistribution of values, meaning that we have to refine the conceptual apparatus behind applicative bisimilarity to take into account probabilistic behaviours.

Following [58,22,33], we look at notions of (bi)simulation in terms of relation lifting operations. Suppose we are given a term relation R such that $\vdash^{\Lambda} e_1\ R\ e_2$. How should we relate $[\![e_1]\!], [\![e_2]\!] \in \mathcal{G}\mathcal{V}_\tau$? Answering such a question means nothing more than finding a proper way to lift $R_{\mathcal{V}}$ to $\mathcal{G}\mathcal{V}_\tau \times \mathcal{G}\mathcal{V}_\tau$.

6.1 Probabilistic Relation Lifting

From an abstract perspective, what we need in order to define applicative (bi)similarity is to find a lifting from $\mathsf{Rel}(X, Y)$ to $\mathsf{Rel}(\mathcal{G}X, \mathcal{G}Y)$. The literature on probabilistic equivalence abounds of such notions of lifting (the interested reader can consult [39] for an overview, and [25] for a more technical treatment). For our purposes, the kind of lifting we are interested in is one that allows applicative (bi)similarity:

(i) To be definable by coinduction;

(ii) To be a preorder (resp. equivalence) term relation;

(iii) To be a compatible and (pre)adequate term relation, and thus a sound proof technique for contextual approximation/equivalence.

The collection of notions of lifting satisfying these requirements has been identified in [11] with the one of *relators* or *lax extension* [4,58].

Unfortunately, finding non-trivial relators for the Giry monad has been proved to be extremely difficult, as the latter lacks some fundamental structural properties (notably, it does not preserve weak pullbacks [39]). Nonetheless, as first argued in [60], restricting the attention from arbitrary binary relations to *reflexive endorelations* allows one to define well-behaved notions of relation lifting for $\mathcal{G}$.

At this point it is instructive to recall the standard notion of relation lifting proposed for $\mathcal{G}$ (see, e.g., [25]), and the kind of problems one faces when working with it. We also remark that the notions of lifting we are going to analyse are actually meant to model notions of program refinement, rather than equivalence. In fact, applicative bisimilarity can be defined in terms of applicative similarity, and compatibility of the former follows from compatibility of the latter.

Definition 6.1 Given measurable spaces (X, Σ_X), (Y, Σ_Y), and a relation $R \subseteq X \times Y$ between them, we define $\Gamma R \subseteq \mathcal{G}X \times \mathcal{G}Y$ as follows: $\Gamma R = \{(\nu_1, \nu_2) \mid \forall A \in \Sigma_X.\ \forall B \in \Sigma_Y.\ R\,[A] \subseteq B \implies \nu_1(A) \leq \nu_2(B)\}$.

The map $\Gamma : \mathsf{Rel}(X, Y) \to \mathsf{Rel}(\mathcal{G}X, \mathcal{G}Y)$ satisfies many interesting properties (see e.g. [25]), but fails to satisfy the inclusion $\Gamma S \circ \Gamma R \subseteq \Gamma(S \circ R)$, which we refer to as *quasi transitivity*. Quasi transitivity ensures applicative similarity to be transitive. Most importantly, it is a central ingredient in Howe's method [21,41], the standard technique used to prove applicative (bi)similarity to be compatible.

A counterexample to quasi transitivity of Γ is given in [25] (Example 4.15), taking advantage of the sets X and Y to be distinct (notably, taking X and Y to be the discrete and indiscrete space over the two-elements set, respectively). However, as we are interested in term relations, we can look at endorelations only. Moreover, as observed in [60], working with reflexive endorelations (which still form a complete lattice), it is possible to refine Γ in such a way to guarantee quasi transitivity.

Definition 6.2 Given a measurable space (X, Σ_X), and a reflexive relation $R \subseteq X \times X$, define $\Gamma R \subseteq \mathcal{G}X \times \mathcal{G}X$ as follows[7]: $\Gamma R = \{(\nu_1, \nu_2) \mid \forall A \in \Sigma_X.\ R\,[A] \subseteq A \implies \nu_1(A) \leq \nu_2(A)\}$.

Lemma 6.3 *Let $\mathsf{Rel}_=(X, X)$ denote the complete lattice of endorelations on X. Then $\Gamma : \mathsf{Rel}_=(X, X) \to \mathsf{Rel}_=(\mathcal{G}X, \mathcal{G}X)$ satisfies the following properties, for all $R, S \in \mathsf{Rel}_=(X, X)$.*

$$=_{\mathcal{G}X} \subseteq \Gamma(=_X) \qquad \text{(quasi reflexivity)}$$

$$\Gamma S \circ \Gamma R \subseteq \Gamma(S \circ R). \qquad \text{(quasi transitivity)}$$

$$R \subseteq S \implies \Gamma R \subseteq \Gamma S. \qquad \text{(monotonicity)}$$

[7] Notice that R being reflexive, $R[A] \subseteq A$ implies $R[A] = A$.

The proof of Lemma 6.3 is straightforward. Remarkably, we can give another characterisation of Γ.

Proposition 6.4 *Given $\mu, \nu \in \mathcal{G}X$ and $R \in \mathsf{Rel}_{=}(X, X)$, we have:*

$$\mu \, \Gamma R \, \nu \iff \forall f : (X, R) \xrightarrow{1} ([0,1], \leq). \int_X f \mathrm{d}\mu \leq \int_X f \mathrm{d}\nu,$$

where $f : (X, R) \xrightarrow{1} ([0,1], \leq)$ means that f is a measurable function (notice that $[0,1] \cong \mathcal{G}\{*\}$) such that $x \, R \, y$ implies $f(x) \leq f(y)$.

The proof of Proposition 6.4 goes essentially as in [25] (Proposition 4.4). See also [51]. Using Proposition 6.4 we can show that Γ is well-behaved with respect to the monad structure of $\mathcal{G}$, and thus that it nicely interacts with sequencing (observe that $[\![\mathtt{let}\ x = e_1\ \mathtt{in}\ e_2]\!](A) = \int [\![e_2[v/x]]\!](A) [\![e_1]\!](\mathrm{d}v) = [\![e_2[-/x]]\!]^\dagger [\![e_1]\!](A)$).

Lemma 6.5 *Given $R \in \mathsf{Rel}_{=}(X, X)$, $S \in \mathsf{Rel}_{=}(Y, Y)$, and measurable functions $f, g : X \to Y$, the following hold, for all $x, y \in X$ and $\mu, \nu \in \mathcal{G}X$*

$$x \, R \, y \implies \delta_x \, \Gamma R \, \delta_y \qquad \text{(dirac)}$$

$$(\forall x_1, x_2 \in X.\ x_1 \, R \, x_2 \implies f(x_1) \, \Gamma S \, g(x_2)) \implies (\mu \, \Gamma R \, \nu \implies f^\dagger(\mu) \, \Gamma S \, g^\dagger(\nu)) \qquad \text{(bind)}$$

Finally, Γ nicely interacts with the ω**Cppo**-enrichment of $\mathcal{G}$, allowing the following induction principle.

Lemma 6.6 *For any relation $R \in \mathsf{Rel}_{=}(X, X)$, any ω-chain $(\mu_n)_{n \geq 0}$ in $\mathcal{G}X$, and any $\nu \in \mathcal{G}X$, if $\mu_n \, \Gamma R \, \nu$ holds for all $n \geq 0$, then $\sup_n \mu_n \, \Gamma R \, \nu$.*

We can now rely on Γ to define a suitable notion of applicative (bi)simulation.

6.2 Applicative Similarly and Bisimilarity

We are now ready to formally define our notion of an applicative (bi)simulation.

Definition 6.7 A *reflexive* closed term relation $R = (R_\Lambda, R_\mathcal{V})$ is an applicative simulation if the following conditions hold:

$$\begin{aligned}
\cdot \vdash^\Lambda e_1 \, R \, e_2 : \tau &\implies [\![e_1]\!] \, \Gamma R_\mathcal{V} \, [\![e_2]\!] && \text{(app eval)}\\
\cdot \vdash^\mathcal{V} \mathtt{r}_1 \, R \, \mathtt{r}_2 : \mathtt{real} &\implies \mathtt{r}_1 = \mathtt{r}_2 && \text{(app num)}\\
\cdot \vdash^\mathcal{V} \lambda x.e_1 \, R \, \lambda x.e_2 : \tau_1 \to \tau_2 &\implies \forall v \in \mathcal{V}_\tau.\ \cdot \vdash^\Lambda e_1[v/x] \, R \, e_2[v/x] : \tau_2 && \text{(app abs)}\\
\cdot \vdash^\mathcal{V} \langle \hat{\imath}, v_{\hat{\imath}} \rangle \, R \, \langle \hat{\jmath}, v_{\hat{\jmath}} \rangle &\implies \hat{\imath} = \hat{\jmath} \ \wedge \ \cdot \vdash^\mathcal{V} v_{\hat{\imath}} \, R \, v_{\hat{\jmath}} : \tau_{\hat{\imath}} && \text{(app inj)}\\
\cdot \vdash^\mathcal{V} \mathtt{fold}\, v_1 \, R \, \mathtt{fold}\, v_1 : \mu\alpha.\tau &\implies \cdot \vdash^\mathcal{V} v_1 \, R \, v_2 : \tau[\mu\alpha.\tau/\alpha]. && \text{(app fold)}
\end{aligned}$$

Applicative similarity $\preceq^\mathsf{A}$ is defined as the largest applicative simulation, whereas applicative bisimilarity $\simeq^\mathsf{A}$ is the largest symmetric applicative simulation.

To see that $\preceq^{\mathsf{A}}$ indeed exists, we define the endofunction $[-]$ on the complete lattice Rel_0 as follows:

$$\begin{aligned}
\cdot \vdash^{\Lambda} e_1 \, [R] \, e_2 &\iff [\![e_1]\!] \, \Gamma R_{\mathcal{V}} \, [\![e_2]\!] \\
\cdot \vdash^{\mathcal{V}} \mathtt{r}_1 \, [R] \, \mathtt{r}_2 : \mathtt{real} &\iff \mathtt{r}_1 = \mathtt{r}_2 \\
\cdot \vdash^{\mathcal{V}} \lambda x.e_1 \, [R] \, \lambda x.e_2 : \tau_1 \to \tau_2 &\iff \forall v \in \mathcal{V}_{\tau}. \ \cdot \vdash^{\Lambda} e_1[v/x] \, R \, e_2[v/x] : \tau_2 \\
\cdot \vdash^{\mathcal{V}} \langle \hat{\imath}, v_{\hat{\imath}} \rangle \, [R] \, \langle \hat{\jmath}, v_{\hat{\jmath}} \rangle &\iff \hat{\imath} = \hat{\jmath} \ \wedge \ \cdot \vdash^{\mathcal{V}} v_{\hat{\imath}} \, R \, v_{\hat{\jmath}} : \tau_{\hat{\imath}} \\
\cdot \vdash^{\mathcal{V}} \mathtt{fold}\, v_1 \, [R] \, \mathtt{fold}\, v_1 : \mu\alpha.\tau &\iff \cdot \vdash^{\mathcal{V}} v_1 \, R \, v_2 : \tau[\mu\alpha.\tau/\alpha].
\end{aligned}$$

We see that R is an applicative simulation if and only if $R \subseteq [R]$. Besides, Γ being monotone, $[-]$ is monotone as well, and thus it has a greatest fixed point which is $\preceq^{\mathsf{A}}$.

Remark 6.8 Notice that we have not required $R_{\mathcal{V}} \subseteq R_{\Lambda}$ for an applicative simulation $R = (R_{\Lambda}, R_{\mathcal{V}})$. However, it is straightforward to see that given such a simulation R, we can always extend R_{Λ} adding the pairs of values $(v_1, v_2) \in R_{\mathcal{V}}$ still obtaining a simulation. This directly follows from condition (dirac). In particular, $\preceq^{\mathsf{A}}_{\mathcal{V}} \subseteq \preceq^{\mathsf{A}}_{\Lambda}$.

Applicative similarity being defined coinductively, it comes with an associated coinduction proof principle, which can be formally given as follows:

$$\frac{\exists R.\ R \subseteq [R] \qquad \cdot \vdash^{\Lambda} e_1 \, R \, e_2 : \tau}{\cdot \vdash^{\Lambda} e_1 \preceq^{\mathsf{A}} e_2 : \tau} \qquad \frac{\exists R.\ R \subseteq [R] \qquad \cdot \vdash^{\mathcal{V}} v_1 \, R \, v_2 : \tau}{\cdot \vdash^{\mathcal{V}} v_1 \preceq^{\mathsf{A}} v_2 : \tau}$$

An easy application of the coinduction proof principle allows us to prove $\preceq^{\mathsf{A}}$ to be a preorder term relation.

Lemma 6.9 *Applicative similarity is reflexive and transitive.*

Proof. [Sketch] By coinduction, relying on the properties (quasi reflexivity) and (quasi transitivity). □

Additionally, we see that $\simeq^{\mathsf{A}} = \preceq^{\mathsf{A}} \cap (\preceq^{\mathsf{A}})^{\top}$, so that $\simeq^{\mathsf{A}}$ is an equivalence term relation. In order to ensure compositionality, we need to prove $\preceq^{\mathsf{A}}$ and $\simeq^{\mathsf{A}}$ to be compatible term relations. We first prove that applicative similarity is compatible, i.e. $\widehat{\preceq^{\mathsf{A}}} \subseteq \preceq^{\mathsf{A}}$. To achieve such a goal, we use Howe's technique [21,41].

Definition 6.10 Given a closed term relation R, the Howe extension of R is the term relation R^{H} defined as the least solution to the equation:

$$\rho = R^{o} \circ \widehat{\rho},$$

where R^{o} denotes the open extension of R. Such a solution exists, since both $-^{o}$ and $\widehat{-}$ are monotone endofunctions. More explicitly, we can inductively define R^{H} by:

$$\frac{\Gamma \vdash^{\Lambda} e_1 \, \widehat{R^{H}} \, e_3 : \tau \quad \Gamma \vdash^{\Lambda} e_3 \, R \, e_2 : \tau}{\Gamma \vdash^{\Lambda} e_1 \, R^{H} \, e_2 : \tau} \qquad \frac{\Gamma \vdash^{\mathcal{V}} v_1 \, \widehat{R^{H}} \, v_3 : \tau \quad \Gamma \vdash^{\mathcal{V}} v_3 \, R \, v_2 : \tau}{\Gamma \vdash^{\mathcal{V}} v_1 \, R^{H} \, v_2 : \tau}$$

Notice that R^H is not only the least solution to the equation $\rho = R^o \circ \widehat{\rho}$, but actually to the inclusion $R^o \circ \widehat{\rho} \subseteq \rho$ [29,32]. The Howe extension of a reflexive and transitive (closed) term relation enjoys several interesting properties. In particular, it is a reflexive, compatible, and substitutive term relation.

Lemma 6.11 *Let R be reflexive and transitive closed term relation. Then the following hold:*

(i) $R^o \subseteq R^H$.

(ii) $R^o \circ R^H \subseteq R^H$

(iii) R^H *is compatible, and thus reflexive.*

(iv) R^H *is substitutive.*

The proof of Lemma 6.11 is standard, and the reader is referred to [41,29] for details. We are finally ready to prove our first main result, namely that applicative similarity is a precongruence term relation. To achieve such a goal, we prove that $(\preceq^{\mathsf{A}})^H$ (restricted to closed terms) is an applicative simulation, and it is thus contained in $\preceq^{\mathsf{A}}$. Since by Lemma 6.11 we have the opposite inclusion, we will conclude $(\preceq^{\mathsf{A}})^H = \preceq^{\mathsf{A}}$.

Lemma 6.12 (Key Lemma) *Given a reflexive and transitive closed term relation R, if R is an applicative simulation, then so is R^H (restricted to closed expressions and values).*

Proof. [Sketch] By Lemma 6.11, R^H is reflexive, and thus a possible candidate simulation. It is easy to see that $R^H_{\mathcal{V}}$ (restricted to closed values) satisfies conditions (app num)-(app fold). It thus remain to prove (app eval), i.e. $\cdot \vdash^{\Lambda} e_1 \, R \, e_2 : \tau \implies [\![e_1]\!] \, \Gamma R^H_{\mathcal{V}} \, [\![e_2]\!]$. Since $[\![e_1]\!] = \sup_n [\![e_1]\!]^{(n)}$ we can appeal to Lemma 6.6, and prove the statement: $\cdot \vdash^{\Lambda} e_1 \, R^H \, e_2 : \tau \implies \forall n \geq 0. \; [\![e_1]\!]^{(n)} \, \Gamma R^H_{\mathcal{V}} \, [\![e_2]\!]$.

We proceed by induction on n. The base case is trivial. The inductive step is proved by case analysis on the derivation of $\cdot \vdash^{\Lambda} e_1 \, R^H \, e_2 : \tau$, which in turn gives a case analysis on $\widehat{R^H}$. A central role is played by the inclusion $\Gamma R \circ \Gamma R^H \subseteq \Gamma R^H$ (which directly follows from Lemma 6.11 by monotonicity of Γ). Sequencing is handled as in [11], relying on property (bind), whereas the cases for measurable functions and numerals follows by condition (dirac). All other cases follow by induction hypothesis, relying on substitutivity of R^H. □

Theorem 6.13 *Applicative similarity is a precongruence term relation and a sound proof technique for contextual approximation.*

Proof. It is sufficient to show that $\preceq^{\mathsf{A}}$ is compatible and preadequate. By Lemma 6.12 and Lemma 6.11 we know that $(\preceq^{\mathsf{A}})^H = \preceq^{\mathsf{A}}$ when restricted to closed terms. This also holds on arbitrary terms, since $(\preceq^{\mathsf{A}})^H$ is substitutive and, dealing with closed values only, sequential and simultaneous substitution coincide. As a consequence, the open extension of applicative similarity coincides with $(\preceq^{\mathsf{A}})^H$, and thus it is compatible. To see that $\preceq^{\mathsf{A}}$ is preadequate, simply observe that $\preceq^{\mathsf{A}}_{\mathcal{V}}[\mathtt{R}] = \mathtt{R}$. □

Finally, since $\simeq^{\mathsf{A}} = \preceq^{\mathsf{A}} \cap (\preceq^{\mathsf{A}})^{\top}$, we see that $\simeq^{\mathsf{A}}$ is compatible and adequate, and thus a sound proof technique for contextual equivalence.

Theorem 6.14 *Applicative bisimilarity $\simeq^{\mathsf{A}}$ is a congruence term relation and a sound proof technique for contextual equivalence.*

Finally, we observe that we can use $\simeq^{\mathsf{A}}$ to prove the equalities and laws in Example 5.3. In particular, the commutativity equation follows by commutativity of $\mathcal{G}$, whereas the soundness of the program transformation of Section 1.1 can be proved combining the quasi-denotational law with congruence properties of $\simeq^{\mathsf{A}}$, by passing the two programs an arbitrary value as input.

7 Digression: Towards Full Abstraction

Theorem 6.14 states that applicative bisimilarity is sound for contextual equivalence: is it also fully abstract (i.e. $\simeq^{\mathsf{A}} = \simeq^{\mathsf{ctx}}$)? In [9] this question is answered in the affirmative for a λ-calculus with sampling from *discrete* probability distributions. Such a result is proved going through *testing equivalence* [15,28], showing that tests can be implemented as λ-term contexts (meaning that testing equivalence includes contextual equivalence), on one hand, and that applicative bisimilarity coincides with testing equivalence, on the other hand.

The latter result has been proved in full generality in [60] relying on nontrivial results from domain theory, category theory, and measurable space theory. Remarkably, the equivalence between bisimilarity and testing equivalence holds not only for Markov chains, but also Markov processes, hence suggesting the possibility of proving full abstraction of applicative bisimilarity also in the continuous case. In fact, it is not hard to see that the operational semantics of Λ_{P} induces a (labelled) Markov process on terms, so that one naturally obtains notions of bisimilarity and testing equivalence for it. However, there is a fundamental difference between such a Markov process and the ones studied in [60]. The former has *uncountably* many labels, whereas the latter requires the of labels to be *countable*.

Indeed, given a state $\lambda x.e$ of the Markov process, in order to model applicative bisimulation we need to consider transitions of the form $\lambda x.e \xrightarrow{v} e[v/x]$, meaning that the set of labels of the Markov process needs to contain (at least) all closed values, which are uncountably many.

A natural way to fix such a problem is to work with (uncountably many) forms of 'countable' bisimilarity relations $\simeq^{\mathsf{A}}_{E}$, where E is a *countable* set of labels of the Markov process, and to work with $\bigcap_E \simeq^{\mathsf{A}}_{E}$ in place of $\simeq^{\mathsf{A}}$. Showing $\bigcap_E \simeq^{\mathsf{A}}_{E}$ to be well-behaved, however, turns out to be highly nontrivial and to require the (abstract) identity $\Gamma(\bigcap_{i\in I} R_i) = \bigcap_{i\in I} \Gamma R$. Up to this point, the authors do not know whether such an identity holds, and leave further investigations on full abstraction as future work.

8 Conclusion

In this paper, we gave a notion of applicative bisimilarity for higher-order languages endowed with an operator performing sampling from *continuous* distributions. We proved that applicative bisimilarity is adequate and compatible, and thus sound for contextual equivalence. As far as the authors know, this is the first contribution on coinductive notions of equivalence for continuous probabilistic λ-calculi. We also gave evidence on the effectiveness of the introduced methodology by proving some nontrivial example equivalences. It is also interesting to notice that we can modify our framework to prove soundness of *open* (also known as *normal form*) bisimilarity [49,30], along the lines of [27] (although we should remark that 'standard' open bisimilarity is not very well-suited for typed languages [31]).

Related Work

Higher-order programming languages featuring sampling from continuous distributions have received quite some attention in the last ten years, due to their use as idioms for bayesian programming. This has stimulated the study of operational [8,46,40] and denotational [56,57,16,59] kinds of semantics for these languages. Recently, contextual equivalence and logical relations have been introduced and proved to coincide both in presence and in absence of full recursion by Culpepper et al. [10,61]. Such logical relations require to test programs against any possible evaluation context, a feature reflected by their characterisation as CIU equivalence [36]. This makes reasoning with and about logical relations quite difficult. However, it is important to stress that such logical relations are defined for languages with conditioning (which makes program equivalence and semantics considerably harder), a feature not present in $\Lambda_{\mathtt{P}}$.

Future Work

A feature which is absent here, but which is desirable in fully-fledged probabilistic programming is conditioning, for example as expressed as a form of *scoring*. Coming up with an operational semantics for an extension of our λ-calculus with a scoring operator would be relatively easy, through the notion of s-finite kernel [56,57]. The latter, however, seem not to carry a monad structure. A better choice might be to work with the monad $\mathcal{G}(\mathbb{R}^{\infty}_{\geq 0} \times -)$ associating to each expression a probability measures over (measurable sets of) pairs score-value. However, finding a way to compare such measures seems nontrivial. Of course, one can define a relation lifting for $\mathcal{G}(\mathbb{R}^{\infty}_{\geq 0} \times -)$ composing Γ with the 'canonical' relation lifting for $\mathbb{R}^{\infty}_{\geq 0} \times -$, the latter being nothing but an instance of the output monad. Such a lifting, however, is too fine-grained, as related values are required to have the same score. A better lifting of a relation R might be obtained requiring the expectations of the scores in R-closed measurable subsets of $\mathbb{R}^{\infty}_{\geq 0} \times -$ to be equal. Proving such a notion of lifting to be well-behaved is, however, nontrivial.

Finally, a further extension of the present work is the design of behavioural distances for probabilistic languages, possibly along the lines of [47,13,17].

References

[1] Abramsky, S., *The lazy lambda calculus*, in: D. Turner, editor, *Research Topics in Functional Programming* (1990), pp. 65–117.

[2] Abramsky, S. and A. Jung, *Domain theory*, in: *Handbook of Logic in Computer Science* (1994), pp. 1–168.

[3] Barendregt, H., "The lambda calculus: its syntax and semantics," Studies in logic and the foundations of mathematics, North-Holland, 1984.

[4] Barr, M., *Relational algebras*, Lect. Notes Math. **137** (1970), pp. 39–55.

[5] Biernacki, D., M. Piróg, P. Polesiuk and F. Sieczkowski, *Handle with care: relational interpretation of algebraic effects and handlers*, PACMPL **2** (2018), pp. 8:1–8:30.

[6] Bizjak, A. and L. Birkedal, *Step-indexed logical relations for probability*, in: *Proc. of FOSSACS 2015*, 2015, pp. 279–294.

[7] Blute, R., J. Desharnais, A. Edalat and P. Panangaden, *Bisimulation for labelled markov processes*, in: *Proc. of LICS*, 1997, pp. 149–158.

[8] Borgström, J., U. D. Lago, A. D. Gordon and M. Szymczak, *A lambda-calculus foundation for universal probabilistic programming*, in: *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming, ICFP 2016, Nara, Japan, September 18-22, 2016*, 2016, pp. 33–46.

[9] Crubillé, R. and U. Dal Lago, *On probabilistic applicative bisimulation and call-by-value lambda-calculi*, in: *Proc. of ESOP 2014*, 2014, pp. 209–228.

[10] Culpepper, R. and A. Cobb, *Contextual equivalence for probabilistic programs with continuous random variables and scoring*, in: *Proceedings of ESOP 2017*, 2017, pp. 368–392.

[11] Dal Lago, U., F. Gavazzo and P. Levy, *Effectful applicative bisimilarity: Monads, relators, and howe's method*, in: *Proc. of LICS 2017*, 2017, pp. 1–12.

[12] Dal Lago, U., D. Sangiorgi and M. Alberti, *On coinductive equivalences for higher-order probabilistic functional programs*, in: *Proc. of POPL 2014*, 2014, pp. 297–308.

[13] de Amorim, A., M. Gaboardi, J. Hsu, S. Katsumata and I. Cherigui, *A semantic account of metric preservation*, in: *Proc. of POPL 2017*, 2017, pp. 545–556.

[14] De Liguoro, U. and A. Piperno, *Non deterministic extensions of untyped lambda-calculus*, Inf. Comput. **122** (1995), pp. 149–177.

[15] De Nicola, R. and M. Hennessy, *Testing equivalence for processes*, in: *Automata, Languages and Programming, 10th Colloquium, Barcelona, Spain, July 18-22, 1983, Proceedings*, 1983, pp. 548–560.

[16] Ehrhard, T., M. Pagani and C. Tasson, *Measurable cones and stable, measurable functions: a model for probabilistic higher-order programming*, PACMPL **2** (2018), pp. 59:1–59:28.

[17] Gavazzo, F., *Quantitative behavioural reasoning for higher-order effectful programs: Applicative distances*, in: *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2018, Oxford, UK, July 09-12, 2018*, 2018, pp. 452–461.

[18] Giry, M., *A categorical approach to probability theory*, in: B. Banaschewski, editor, *Categorical Aspects of Topology and Analysis* (1982), pp. 68–85.

[19] Gordon, A., *A tutorial on co-induction and functional programming*, in: *Workshops in Computing* (1994), pp. 78–95.

[20] Goubault-Larrecq, J., S. Lasota and D. Nowak, *Logical relations for monadic types*, Mathematical Structures in Computer Science **18** (2008), pp. 1169–1217.

[21] Howe, D., *Proving congruence of bisimulation in functional programming languages*, Inf. Comput. **124** (1996), pp. 103–112.

[22] Hughes, J. and B. Jacobs, *Simulations in coalgebra*, Theor. Comput. Sci. **327** (2004), pp. 71–108.

[23] Johann, P., A. Simpson and J. Voigtländer, *A generic operational metatheory for algebraic effects*, in: *Proc. of LICS 2010* (2010), pp. 209–218.

[24] Jones, C., "Probabilistic non-determinism," Ph.D. thesis, University of Edinburgh, UK (1990).

[25] Katsumata, S., T. Sato and T. Uustalu, *Codensity lifting of monads and its dual*, Logical Methods in Computer Science **14** (2018).

[26] Kelly, G. M., *Basic concepts of enriched category theory*, Reprints in Theory and Applications of Categories (2005), pp. 1–136.

[27] Lago, U. D. and F. Gavazzo, *Effectful normal form bisimulation*, in: *Proc. of ESOP 2019*, 2019, to appear.

[28] Larsen, K. G. and A. Skou, *Bisimulation through probabilistic testing*, in: *Proceedings of POPL 1989*, 1989, pp. 344–352.

[29] Lassen, S., "Relational Reasoning about Functions and Nondeterminism," Ph.D. thesis, Dept. of Computer Science, University of Aarhus (1998).

[30] Lassen, S. B., *Bisimulation in untyped lambda calculus: Böhm trees and bisimulation up to context*, Electr. Notes Theor. Comput. Sci. **20** (1999), pp. 346–374.

[31] Lassen, S. B. and P. B. Levy, *Typed normal form bisimulation*, in: *Computer Science Logic, 21st International Workshop, CSL 2007, 16th Annual Conference of the EACSL, Lausanne, Switzerland, September 11-15, 2007, Proceedings*, 2007, pp. 283–297.

[32] Levy, P., *Infinitary howe's method*, Electr. Notes Theor. Comput. Sci. **164** (2006), pp. 85–104.

[33] Levy, P., *Similarity quotients as final coalgebras*, in: *Proc. of FOSSACS 2011*, LNCS **6604**, 2011, pp. 27–41.

[34] Levy, P., J. Power and H. Thielecke, *Modelling environments in call-by-value programming languages*, Inf. Comput. **185** (2003), pp. 182–210.

[35] MacLane, S., "Categories for the Working Mathematician," Springer-Verlag, 1971.

[36] Mason, I. A. and C. L. Talcott, *Equivalence in functional languages with effects*, J. Funct. Program. **1** (1991), pp. 287–327.

[37] Morris, J., "Lambda Calculus Models of Programming Languages," Ph.D. thesis, MIT (1969).

[38] Panangaden, P., *The category of markov kernels*, Electr. Notes Theor. Comput. Sci. **22** (1999), pp. 171–187.

[39] Panangaden, P., "Probabilistic bisimulation," Cambridge Tracts in Theoretical Computer Science, Cambridge University Press, 2011 p. 290–326.

[40] Park, S., F. Pfenning and S. Thrun, *A probabilistic language based upon sampling functions*, in: *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2005, Long Beach, California, USA, January 12-14, 2005*, 2005, pp. 171–182.

[41] Pitts, A., *Howe's method for higher-order languages*, in: D. Sangiorgi and J. Rutten, editors, *Advanced Topics in Bisimulation and Coinduction*, Cambridge Tracts in Theoretical Computer Science **52**, Cambridge University Press, 2011 pp. 197–232.

[42] Plotkin, G., *Lambda-definability and logical relations* (1973), technical Report SAI-RM-4, School of A.I., University of Edinburgh.

[43] Plotkin, G. D. and J. Power, *Adequacy for algebraic effects*, in: *Proc. of FOSSACS 2001*, 2001, pp. 1–24.

[44] Plotkin, G. D. and J. Power, *Notions of computation determine monads*, in: *Proc. of FOSSACS 2002*, 2002, pp. 342–356.

[45] Pollard, D., R. Gill and B. Ripley, "A User's Guide to Measure Theoretic Probability," Cambridge Series in Statistical and Probabilistic Mathematics, Cambridge University Press, 2002.

[46] Ramsey, N. and A. Pfeffer, *Stochastic lambda calculus and monads of probability distributions*, in: *Proc. of POPL 2002*, 2002, pp. 154–165.

[47] Reed, J. and B. Pierce, *Distance makes the types grow stronger: a calculus for differential privacy*, in: *Proc. of ICFP 2010*, 2010, pp. 157–168.

[48] Reynolds, J., *Types, abstraction and parametric polymorphism*, in: *IFIP Congress*, 1983, pp. 513–523.

[49] Sangiorgi, D., *The lazy lambda calculus in a concurrency scenario*, Inf. Comput. **111** (1994), pp. 120–153.

[50] Sangiorgi, D., N. Kobayashi and E. Sumii, *Environmental bisimulations for higher-order languages*, ACM Trans. Program. Lang. Syst. **33** (2011), pp. 5:1–5:69.

[51] Sato, T., *Approximate relational hoare logic for continuous random samplings*, Electr. Notes Theor. Comput. Sci. **325** (2016), pp. 277–298.

[52] Scott, D., *Outline of a mathematical theory of computation*, Technical Report PRG02, OUCL (1970).

[53] Scott, D. and C. Strachey, *Toward a mathematical semantics for computer languages*, Technical Report PRG06, OUCL (1971).

[54] Sieber, K., *Reasoning about sequential functions via logical relations*, in: M. P. Fourman, P. T. Johnstone and A. M. Pitts, editors, *Applications of Categories in Computer Science*, London Mathematical Society Lecture Note Series **177**, Cambridge University Press, 1992 pp. 258–269.

[55] Simpson, A. and N. Voorneveld, *Behavioural equivalence via modalities for algebraic effects*, in: *Proc. of ESOP 2018*, 2018, pp. 300–326.

[56] Staton, S., *Commutative semantics for probabilistic programming*, in: *Programming Languages and Systems - 26th European Symposium on Programming, ESOP 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings*, 2017, pp. 855–879.

[57] Staton, S., H. Yang, F. D. Wood, C. Heunen and O. Kammar, *Semantics for probabilistic programming: higher-order functions, continuous distributions, and soft constraints*, in: *Proceedings of the 31st Annual ACM/IEEE Symposium on Logic in Computer Science, LICS '16, New York, NY, USA, July 5-8, 2016*, 2016, pp. 525–534.

[58] Thijs, A., "Simulation and fixpoint semantics," Rijksuniversiteit Groningen, 1996.

[59] Vákár, M., O. Kammar and S. Staton, *A domain theory for statistical probabilistic programming*, PACMPL **3** (2019), pp. 36:1–36:29.

[60] Van Breugel, F., M. Mislove, J. Ouaknine and J. Worrell, *Domain theory, testing and simulation for labelled markov processes*, Theor. Comput. Sci. **333** (2005), pp. 171–197.

[61] Wand, M., R. Culpepper, T. Giannakopoulos and A. Cobb, *Contextual equivalence for a probabilistic language with continuous random variables and recursion*, PACMPL **2** (2018), pp. 87:1–87:30.