Semantics of Probabilistic Programs using s-Finite Kernels in Coq

Reynald Affeldt, Cyril Cohen, Ayumu Saito

Semantics of Probabilistic Programs using s-Finite Kernels in Coq

Reynald Affeldt
National Institute of Advanced Industrial Science and Technology (AIST)
Tokyo, Japan

Cyril Cohen
Université Côte d'Azur and Inria
Sophia Antipolis, France

Ayumu Saito
Department of Mathematical and Computing Science,
Tokyo Institute of Technology
Tokyo, Japan

Abstract

Probabilistic programming languages are used to write probabilistic models to make probabilistic inferences. A number of rigorous semantics have recently been proposed that are now available to carry out formal verification of probabilistic programs. In this paper, we extend an existing formalization of measure and integration theory with s-finite kernels, a mathematical structure to interpret typing judgments in the semantics of a probabilistic programming language. The resulting library makes it possible to reason formally about transformations of probabilistic programs and their execution.

CCS Concepts: • **Theory of computation** → **Logic and verification**; **Program verification**; *Denotational semantics*; • **Mathematics of computing** → *Probabilistic algorithms.*

Keywords: Coq, measure theory, integration theory, probabilistic programming language

ACM Reference Format:
Reynald Affeldt, Cyril Cohen, and Ayumu Saito. 2023. Semantics of Probabilistic Programs using s-Finite Kernels in Coq. In *Proceedings of the 12th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP '23), January 16–17, 2023, Boston, MA, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3573105.3575691

1 Introduction

There exist several probabilistic programming languages (Hakaru [26], Anglican [37], etc.). They are used to write probabilistic models and these models are used to perform probabilistic inference. A probabilistic programming language typically provides specific instructions for (1) sampling from a prior distribution, (2) scoring a particular event (i.e., record its likelihood or weighting it in a Monte Carlo simulation), and (3) normalizing a posterior probability distribution (to get the result of the probabilistic inference or to serve for a new sampling, i.e., a "nested query"). For the sake of concreteness, let us explain an example of probabilistic model. We borrow this example from Staton [31, Sect. 4.3]. It is about inferring whether or not today is the weekend according to the number of buses passing by:

```
1  normalize (
2    let x = sample (bernoulli (2 / 7)) in
3    let r = if x then 3 else 10 in
4    let _ = score (r ^ 4 / 4! * e ^ (- r)) in
5    return x)
```

- At line 2, we select a boolean number according to a Bernoulli probability measure. This represents whether or not it is the weekend.
- At line 3, we model the fact that there are three buses per hour on weekends and ten buses per hour otherwise.
- At line 4, we record the fact that we observe four buses in one hour. We assume that buses arrive as a Poisson process with rate `r`; the expression inside the `score` instruction is the Poisson distribution function.
- Line 1 normalizes the measure to a probability measure.

The instructions `sample`, `score`, and `normalize` arguably form the core of a typical probabilistic programming language.

The semantics of probabilistic programming languages is an active topic of research. In particular, an important mathematical notion that has emerged is that of a kernel [14, 21] which generalizes the notion of measure. Recently, Staton proposed to use s-finite kernels, i.e., kernels that can be expressed as a countable sum of finite kernels [30]. Thanks to s-finite kernels, one can provide a compositional semantics for (first-order) probabilistic programming languages by interpreting instructions as measures. One can also reason about the validity of transformations such as the reordering (or "commutativity") of instructions. S-finite kernels have been used for semantics [12, 25, 30, 31] and as a topic of interest on their own, since as a mathematical structure they seem

CPP '23, January 16–17, 2023, Boston, MA, USA

ACM ISBN 979-8-4007-0026-2/23/01…$15.00
https://doi.org/10.1145/3573105.3575691

to have been overlooked so far [27, 36]. Among the recent developments in the semantics of probabilistic programming languages, we can also cite quasi-Borel spaces [18], a mathematical structure to support higher-order functions.

How to turn the research results on the semantics of probabilistic programming languages into practical tools for their formal verification is a recent topic of interest. Shan [29] has been advocating for the use of equational reasoning to reason about probabilistic programs; this idea has been experimented in the Coq proof assistant but the formalization is an axiomatization that assumes that all functions are measurable and all measures are s-finite [17]. Zhang and Amin [38] provide a Coq formalization of a probabilistic programming language with scoring, general recursion and nested queries. However, in their Coq formalization, the notions of measure and integral are axiomatized, and all issues of measurability are ignored. Affeldt et al. [6, 8] have been formalizing monadic equational reasoning to reason in Coq about programs with effects. But as far as the probability effect is concerned, they only support the `sample` instruction and discrete probabilities. Hirata at al. [19] have been experimenting with the formalization in Isabelle/HOL of quasi-Borel spaces; here again, they only support the `sample` instruction.

In summary, related work indicates that, as of today, there is no complete formalization of a probabilistic programming language. In particular, as far as Coq is concerned, progress is hampered by the lack of a good formalization of measure theory.

Our contribution in this paper is to provide a formalization of s-finite kernels with an application to the formalization of the semantics of a probabilistic programming language in Coq. This language supports sampling, scoring, and normalization, and the formalization is axiom-free, in the sense that no result about measurability or integration is left unproved[1]. To that aim, we improve an existing Coq formalization of measure and integration theory [2]. Furthermore, we use our formal semantics to reason about program transformations and symbolic execution of probabilistic programs.

Paper Outline. Sect. 2 recalls basic definitions and results about measure and integration theory. Sect. 3 provides background information about the formalization of measure and integration theory in the MathComp-Analysis library, including recent extensions. In Sect. 4, we formalize kernels, s-finite kernels, finite kernels, subprobability kernels, and probabilistic kernels as a hierarchy of mathematical structures. In Sect. 5, we formalize a key result: the proof that s-finite kernels are stable by composition. In Sect. 6, we use the theory of s-finite kernels to formalize the semantics of a probabilistic programming language. In Sect. 7, we show how to reason formally about the properties of probabilistic programming languages. We review related work in Sect. 8 and conclude in Sect. 9.

[1]Yet, it should be said that our work is based on MathComp-Analysis [1] that augments Coq with standard axioms for classical reasoning: functional and proposition extensionality, the law of excluded middle, and the axiom of choice [4, Sect. 5].

2 Mathematical Preliminaries

Let us recall the basics of measure and integration theory.

A σ-algebra on a set X is a collection of subsets of X that contains $\varnothing$ and that is closed under complement and countable union. We note Σ_X for such a σ-algebra and call *measurables* the sets in Σ_X. The standard σ-algebra on $\mathbb{R}$ is the smallest σ-algebra containing the intervals: the Borel sets. Given two σ-algebras Σ_X and Σ_Y, the product σ-algebra is the smallest σ-algebra generated by $\{A \times B \,|\, X \in \Sigma_X, Y \in \Sigma_Y\}$.

A (non-negative) *measure* is a function $\mu : \Sigma_X \to [0, \infty]$ such that $\mu(\varnothing) = 0$ and $\mu(\bigcup_i A_i) = \sum_i \mu(A_i)$ for pairwise-disjoint measurable sets A_n where the sum is countable. The latter property is called σ-additivity. A *probability measure* on Σ_X is a measure μ such that $\mu(X) = 1$. The standard measure on $\mathbb{R}$ is the Lebesgue measure, which is such that the length of an interval (a, b) is $b - a$. The Dirac measure δ_x is defined by $\delta_x(U) = [x \in U]$ (using the Iverson bracket notation). The countable addition of measures forms a measure. Given a measure μ and a measurable set D, $\mu|_D \stackrel{\text{def}}{=} \lambda U.\mu(U \cap D)$ is a measure: the restriction of μ to D. A measure μ is σ-finite over A when there is countable family of measurable sets F such that $A = \bigcup_i F_i$ and for all i, $\mu(F_i) < \infty$.

A function $f : \Sigma_X \to \Sigma_Y$ is *measurable* when for all measurable sets B, $f^{-1}(B)$ is also measurable. A *simple function* is a measurable function with a finite image. In particular, a simple function can be written as the sum of *indicator functions*: $f = \sum_{y \in \text{range}(f)} y \mathbb{1}_{f^{-1}\{y\}} x$. The approximation theorem states that, given a non-negative measurable function f, there is a non-decreasing sequence of non-negative simple function f_i that converges pointwise towards f.

We can integrate a measurable function f w.r.t. a measure μ over D to get an extended real number denoted by $\int_{x \in D} f\, x (\mathbf{d}\, \mu)$. Integration satisfies the monotone convergence theorem, i.e., $\int_{x \in D} f\, x (\mathbf{d}\, \mu) = \lim_i \int_{x \in D} f_i\, x (\mathbf{d}\, \mu)$ where f_i is an increasing sequence of simple functions converging towards f, a non-negative measurable function.

Given a non-negative measurable binary function f and two σ-finite measures μ_1 and μ_2, we have Tonelli-Fubini's theorem: $\int_x \int_y f(x, y) (\mathbf{d}\, \mu_2) (\mathbf{d}\, \mu_1) = \int_y \int_x f(x, y) (\mathbf{d}\, \mu_1) (\mathbf{d}\, \mu_2)$.

As for the mathematical definitions of kernels, we will introduce them along their formalization in Coq.

3 Preliminaries on MathComp-Analysis

We provide an overview of the formalization of measure and integration theory in MathComp-Analysis to understand the rest of this paper.

3.1 Basic Definitions about Set Theory and Topology

The type `set X` is for sets of objects of type `X`. We can write `A a` or `a \in A` to state that `a` belongs to the set `A`. The full set of objects of type `X` is denoted by `[set: X]`; it is a notation for `setT`. The singleton $\{a\}$ is denoted by `[set a]`. Set inclusion is denoted by `` `<=` ``. The preimage of the set `A` by `f` is denoted by `f @^-1` A`. Sets can be defined by comprehension using the notation `[set x | P]`, for the set of objects `x` such that `P` holds. The expression `xsection A x` is for the x-section $A|_x$, i.e., the set of y's such that $(x, y) \in A$.

The convergence of a sequence `u` towards `a` is denoted by `u --> a` [4, Sect. 2.3]. The type of a sequence of objects of type `A` is denoted by `A^nat`. The type `{nonneg R}` is for non-negative numeric types, where `R` is a numeric type among `numFieldType` for numeric fields, `realType` for real numbers, etc. Non-negative numeric types can be built with the constructor `NngNum` which takes a proof that its argument is non-negative. The type `\bar R` is for extended real numbers, where `R` is expected to be a numeric type. Infinite values are denoted by `-oo` and `+oo` and `r%:E` represents the injection of `r` of type `R` into `\bar R`. The function `fine` returns the numeric value of a finite extended real numbers and `0` otherwise.

We also make use of standard MathComp [24] notations. The notation `f ^~ y` is for the function $\lambda x.f\,x\,y$. The notation `\o` is for function composition. The projections of a pair are denoted by `.1` and `.2`. Boolean operations are denoted by `~~` for the negation and `||` for the disjunction. The notation `n%:R` is for injecting the natural number `n` into a ring type; `%R` is the scope of rings.

3.2 Basic Measure and Integration Theory

Most details about the formalization of measure and integration theory in MathComp-Analysis can be found in related work [2]. This section also documents recent technical additions to MathComp-Analysis.

The type of σ-algebra's is `measurableType d`. More precisely, given `T` of type `measurableType d` and `U` of type `set T`, `measurable U` asserts that `U` belongs to the σ-algebra corresponding to `T`. The parameter `d` controls the display of the `measurable` predicate, so that `measurable U` is printed as `d.-measurable U`. This is useful to disambiguate the local context of a proof in the presence of several σ-algebra's. See [2, Sect. 3.4] for details. Given `T` of type `measurableType d`, a measure on `T` is denoted by `{measure set T -> \bar R}`, where `R` has type `realType`. The Dirac measure is denoted by `dirac a A` (with notation `\d_a A`). The measure corresponding to the function $\lambda\_.0$ is `mzero`. The sum of two measures `m1` and `m2` is `measure_add m1 m2`. The measure corresponding to $\lambda x.r\,\mu(x)$ where r is a non-negative number is `mscale r mu` where `r` has type `{nonneg R}`. The measure corresponding to the sum $\sum_{i=n}^{\infty} f_i$ is `mseries f n`. The restriction of the measure `mu` to a measurable set `D` with `mD : measurable D` is `mrestr mu mD`. The type of a `R`-valued probability measure over the measurable type `X` is `probability X R`. The predicate that characterizes a measure `mu` that is σ-finite over `D` is `sigma_finite D mu`.

We write `measurable_fun D f` for a measurable function `f` with domain `D`. The type of non-negative simple functions with domain `D` (a set) and codomain `R` (a `realType`) is denoted by `{nnsfun D >-> R}`.

Finally, the notation for the integral $\int_{x\in D} f(x)(\mathbf{d}\,\mu)$ is `\int[mu]_(x in D) f x`.

4 Formalization of a Hierarchy of Kernels

In this section, we formalize kernels as a hierarchy of mathematical structures. For that purpose, we use Hierarchy-Builder [15], a Coq extension that automates the creation of *packed classes*, a methodology to create hierarchies of mathematical structures using canonical structures [16]. In short, Hierarchy-Builder extends Coq with commands that generate the boilerplate code for packed classes while checking that the hierarchies of mathematical structures produced in this way are well-formed. This section provides enough details to serve as an introduction to Hierarchy-Builder. See also [2, Sect. 3.1] for an overview of Hierarchy-Builder.

4.1 Definition of Kernels and Circularity

A *kernel* $X \rightsquigarrow Y$ is a function $k : X \to \Sigma_Y \to [0, \infty]$ such that

(i) for all x, $k\,x$ is a measure and
(ii) for all measurable U, $x \mapsto k\,x\,U$ is a measurable function.

We can see k as a family of measures indexed by X.

A kernel $k : X \rightsquigarrow Y$ is a *finite kernel* when there is a finite bound r such that for all x, $k\,x\,Y < r$. This is a uniform upper bound, i.e., the same r for all x. Let us denote the type of finite kernels by $X \overset{\text{fin}}{\rightsquigarrow} Y$.

A kernel $k : X \rightsquigarrow Y$ is an *s-finite kernel* when there is a sequence s of finite kernels such that $k = \sum_{i=0}^{\infty} s_i$. Let us denote by $X \overset{\text{s-fin}}{\rightsquigarrow} Y$ the type of s-finite kernels. It is important to notice that the definition of s-finite kernels uses the one of finite kernels and that finite kernels *are* s-finite kernels. In terms of hierarchy of mathematical structures, this means that s-finite kernels are more general than finite kernels and that they should come first. We explain in this section how to handle this circularity; a naive encoding would be the source of non forgetful inheritance issues [3].

A kernel $k : X \rightsquigarrow Y$ is a *subprobability kernel* when $\sup_{x\in X} k\,x\,Y \le 1$ and it is a *probability kernel* when for all x, we have $k\,x\,Y = 1$. Let us denote the type of subprobability kernels by $X \overset{\text{subprob}}{\rightsquigarrow} Y$ and the one of probability kernels by $X \overset{\text{prob}}{\rightsquigarrow} Y$.

The kernels we have defined above are organized as a hierarchy of mathematical structures depicted in Fig. 1 that we will now formalize.

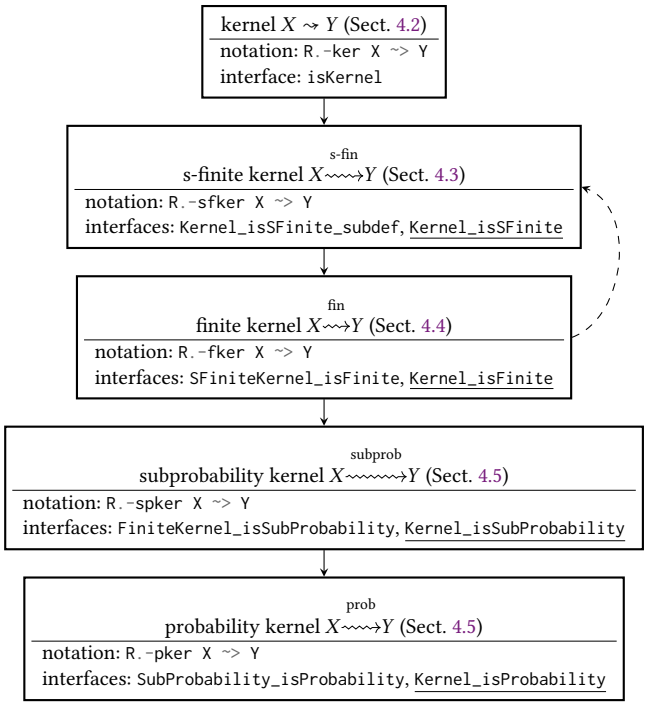

Figure 1. Hierarchy of Kernels. Solid arrows show inheritance relations, the dashed arrow records the use of the definition of finite kernel inside the definition of s-finite kernel. Among the interfaces, factories are underlined.

4.2 Formalization of Kernels

We saw in Sect. 4.1 that a kernel $X \rightsquigarrow Y$ is a function $k : X \to \Sigma_Y \to [0, \infty]$ that satisfies two conditions. We formalize this definition using a *mixin*, i.e., an interface. The fact that for all x, $k\,x$ is a measure (condition (i)) is captured by the type of k; the fact that for all measurable U, $x \mapsto k\,x\,U$ is a measurable function (condition (ii)) is captured by the field measurable_kernel (where we use advantageously the notation ^~ from MathComp—Sect. 3.1).

```
HB.mixin Record isKernel
  d d' (X : measurableType d) (Y : measurableType d')
  (R : realType) (k : X -> {measure set Y -> \bar R})
 :=  { measurable_kernel : forall U,
       measurable U -> measurable_fun setT (k ^~ U) }.
```

The Hierarchy-Builder commands are all prefixed with HB.

The mathematical structure of kernels is formalized as a sigma-type using the isKernel mixin:

```
HB.structure Definition Kernel
  d d' (X : measurableType d) (Y : measurableType d')
  (R : realType) := { k & isKernel _ _ X Y R k }.
```

When processing this definition Hierarchy-Builder generates a function isKernel.Build that can be used to create instances of kernels. Hereafter, we use the Coq notation R.-ker X ~> Y for the type of kernels $X \rightsquigarrow Y$.

Example of Instantiation. Let us define the kernels corresponding to countable sums of kernels. The first step is to define a symbol kseries for the appropriate family of measures. This definition merely lifts the countable sum of measures mseries (Sect. 3.2):

```
Context d d' (X : measurableType d)
  (Y : measurableType d') (R : realType).
Variable k : (R.-ker X ~> Y)^nat.
Definition kseries : X -> {measure set Y -> \bar R} :=
  fun x => mseries (k ^~ x) 0.
```

The measure structure of mseries is provided by MathComp-Analysis. It is inferred automatically by Coq without any additional syntax thanks to the reversible coercion mechanism available since Coq 8.16 [35].

To equip kseries with the structure of kernel, we need to show that the condition (ii) of kernels holds:

```
Lemma measurable_fun_kseries (U : set Y) :
  measurable U -> measurable_fun setT (kseries ^~ U).
Proof. (* see [5] *) Qed.
```

Finally, we instantiate the kernel structure using the function isKernel.Builder mentioned above:

```
HB.instance Definition _ := isKernel.Build _ _ _ _ _
  kseries measurable_fun_kseries.
```

The symbol kseries is now equipped with the structure of kernel, as indicated by the HB.about command:

```
> HB.about kseries.
  HB: kseries is canonically equipped with mixins:
      - kernel.Kernel
```

4.3 Formalization of s-Finite Kernels

We saw in Sect. 4.1 that an s-finite kernel is a kernel that can be expressed as a countable sum of finite kernels that are themselves s-finite kernels. Some care is needed to deal with this circular definition

Characterization of Finite Kernels. We do not define finite kernels at this point; we only introduce a predicate (measure_fam_uub) to characterize them:

```
Context d d' (X : measurableType d)
  (Y : measurableType d') (R : numFieldType).
Variable k : X -> {measure set Y -> \bar R}.
Definition measure_fam_uub :=
  exists r, forall x, k x [set: Y] < r%:E.
```

Note that since a measure is non-negative, the witness r is positive by definition. This characterization of finite kernels is enough to produce a definition of s-finite kernels.

Definition of s-Finite Kernels. The mixin for an s-finite kernel k has a field (sfinite_subdef below) that states the existence of a sequence s of kernels such that k is equal to kseries s. The fact that each kernel composing the sequence is finite is captured using the predicate measure_fam_uub:

```
HB.mixin Record Kernel_isSFinite_subdef
   d d' (X : measurableType d) (Y : measurableType d')
   (R : realType) (k : X -> {measure set Y -> \bar R})
```

```
:= { sfinite_subdef : exists2 s : (R.-ker X ~> Y)^nat,
   forall n, measure_fam_uub (s n) &
   forall x U, measurable U -> k x U = kseries s x U }.
```

The structure of s-finite kernels is defined as a family of measures `k` that is a kernel and that also satisfies the mixin `Kernel_isSFinite_subdef`:

```
HB.structure Definition SFiniteKernel
  d d' (X : measurableType d) (Y : measurableType d')
  (R : realType) :=
  {k of @Kernel _ _ _ _ _ k &
        Kernel_isSFinite_subdef _ _ X Y R k }.
```

Hereafter, the Coq notation for an s-finite kernel $X \overset{\text{s-fin}}{\rightsquigarrow} Y$ is `R.-sfker X ~> Y`.

Even though this definition of s-finite kernels is perfectly valid, it is not exactly in the expected form because it does not use the definition of finite kernels undefined at this point. This is improved in the next section.

4.4 Completing the Formalization of Finite Kernels

To define finite kernels, we provide a mixin for a family of measures which is uniformly upper bounded (using the predicate `measure_fam_uub` introduced in Sect. 4.3):

```
HB.mixin Record SFiniteKernel_isFinite
  d d' (X : measurableType d) (Y : measurableType d')
  (R : realType) (k : X -> {measure set Y -> \bar R})
:= { measure_uub : measure_fam_uub k }.
```

The structure of finite kernels is a sigma-type that uses the structure `SFiniteKernel` of the previous section and the above mixin `SFiniteKernel_isFinite`:

```
HB.structure Definition FiniteKernel
  d d' (X : measurableType d) (Y : measurableType d')
  (R : realType) :=
 {k of @SFiniteKernel _ _ _ _ _ k &
       SFiniteKernel_isFinite _ _ X Y R k }.
```

Hereafter, the Coq notation for a finite kernel $X \overset{\text{fin}}{\rightsquigarrow} Y$ is `R.-fker X ~> Y`.

The above definition of finite kernels seems to require upon instantiation to prove that a finite kernel is actually an s-finite kernel. This is actually not the case because we can use a generic proof of this fact to produce a simpler interface, called a *factory* (a mixin being really just a special case of factory):

```
HB.factory Record Kernel_isFinite
  d d' (X : measurableType d) (Y : measurableType d')
  (R : realType) (k : X -> {measure set Y -> \bar R})
of isKernel _ _ _ _ _ k := {
  measure_uub : measure_fam_uub k }.
```

After declaring this interface, we use Hierarchy-Builder commands (see factories and builders in [15]) to generate a function `Kernel_isFinite.Build` to build instances of finite kernels using the simpler interface while preserving the inheritance relations. See [5] for details.

Improving the Definition of s-Finite Kernels. Now that we have a proper definition of finite kernels we can rework the definition of s-finite kernels (Sect. 4.3). Concretely, instead of using the predicate `measure_fam_uub`, we use the type `R.-fker X ~> Y` in the following factory:

```
HB.factory Record Kernel_isSFinite
  d d' (X : measurableType d) (Y : measurableType d')
  (R : realType) (k : X -> {measure set Y -> \bar R})
  of isKernel _ _ _ _ _ k := {
sfinite : exists s : (R.-fker X ~> Y)^nat,
  forall x U, measurable U -> k x U = kseries s x U }.
```

We have thus recovered the textbook definition of s-finite kernels while preserving the hierarchical organization of the definitions of kernels.

We believe that the way we deal with the circularity between the definitions of finite kernel and s-finite kernel is a pattern that is more widely applicable when formalizing mathematics. We could think at least of the definition of schemes and affine schemes as another (unrelated) example. It might be worth investigating an extension of Hierarchy-Builder to deal with this pattern in a more succinct fashion.

4.5 Probability Kernels

We complete our hierarchy of kernels with subprobability and probability kernels. The mixin below is for subprobability kernels; the field `sprob_kernel` captures the definition seen in Sect. 4.1:

```
HB.mixin Record FiniteKernel_isSubProbability
  d d' (X : measurableType d) (Y : measurableType d')
  (R : realType) (k : X -> {measure set Y -> \bar R})
:= { sprob_kernel :
  ereal_sup [set k x [set: Y] | x in setT] <= 1}.
```

A subprobability kernel is a finite kernel with the above interface:

```
HB.structure Definition SubProbabilityKernel
  d d' (X : measurableType d) (Y : measurableType d')
  (R : realType) :=
 {k of @FiniteKernel _ _ _ _ _ k &
       FiniteKernel_isSubProbability _ _ X Y R k }.
```

Finally, the mixin below defines probability kernels as the families of kernels such that the full set always has measure 1:

```
HB.mixin Record SubProbability_isProbability
  d d' (X : measurableType d) (Y : measurableType d')
  (R : realType) (k : X -> {measure set Y -> \bar R})
:= { prob_kernel : forall x, k x [set: Y] = 1}.
```

A probability kernel is a subprobability kernel with the above interface:

```
HB.structure Definition ProbabilityKernel
  d d' (X : measurableType d) (Y : measurableType d')
  (R : realType) :=
 {k of @SubProbabilityKernel _ _ _ _ _ k &
       SubProbability_isProbability _ _ X Y R k }.
```

Again, it is not necessary to instantiate intermediate interfaces to instantiate a probability kernel because there is a generic proof that the property of probability kernel implies the property of subprobability and finite kernels. We can therefore provide a factory which corresponds to the expected textbook definition:

```
HB.factory Record Kernel_isProbability
  d d' (X : measurableType d) (Y : measurableType d')
  (R : realType) (k : X -> {measure set Y -> \bar R})
 of isKernel _ _ X Y R k :=
  { prob_kernel : forall x, k x setT = 1 }.
```

We use Hierarchy-Builder commands to generate a function `Kernel_isProbability.Build` to build instances of probability kernel using the latter interface. See [5] for details.

4.6 Examples of s-Finite Kernels

We illustrate furthermore the process of instantiating structures with examples of s-finite kernels we will use later in this paper.

Deterministic Kernels. A measurable function f gives rise to a probability kernel defined by $x \mapsto \delta_{(f\,x)}$ [36, Example 1]. In Coq, given a measurable function `f`, we define a family of measures `kdirac` using the `dirac` measure:

```
Variable f : X -> Y.
Definition kdirac (mf : measurable_fun setT f)
    : X -> {measure set Y -> \bar R} :=
  fun x => dirac (f x).
```

The proof `mf` is a phantom type: it is not used directly in the definition but it is required for type inference. Indeed, since `measurable_fun` is not a class and since there is no structure of measurable functions in MathComp-Analysis, instances with which we would like to endow `dirac` cannot be found. That is why we define `kdirac`, a wrapper around `dirac` that is designed to contain a non-inferable proof that is used only for the instances but not directly in the definition[2]. To show that `kdirac` indeed forms a probability kernel, we can use the factory of Sect. 4.5. First, we show that `kdirac` forms a kernel (condition (ii), Sect. 4.2):

```
Hypothesis mf : measurable_fun setT f.
Let measurable_fun_kdirac U : measurable U ->
  measurable_fun setT (kdirac mf ^~ U).
Proof. (* see [5] *) Qed.
HB.instance Definition _ := isKernel.Build _ _ _ _ _
  (kdirac mf) measurable_fun_kdirac.
```

Second, we show that the measure of the full set is 1 for each measure:

```
Let kdirac_prob x : kdirac mf x setT = 1.
Proof. by rewrite /kdirac/= diracT. Qed.
```

Last, we use this proof to instantiate the structure of probability kernel:

```
HB.instance Definition _ := Kernel_isProbability.Build
  _ _ _ _ _ (kdirac mf) kdirac_prob.
```

Addition of Kernels. Similarly, the fact that the addition of kernels (resp. s-finite kernels/finite) is a kernel (resp. s-finite/finite kernels) can be obtained by lifting properties of measures. We first define the addition of kernels `kadd` by lifting the addition of measures `measure_add` (Sect. 3.2):

```
Context d d' (X : measurableType d)
  (Y : measurableType d') (R : realType).
Variables k1 k2 : R.-ker X ~> Y.
Definition kadd : X -> {measure set Y -> \bar R} :=
  fun x => measure_add (k1 x) (k2 x).
```

Then, we prove that `kadd` verifies the properties of kernels (Sect. 4.2), of s-finite kernels (Sect. 4.3), and of finite kernels (Sect. 4.4) when its arguments respectively do. We have to be careful to declare to Coq that addition preserves s-finite kernels first, even though we need the proof it preserves finite kernels for that. Indeed, otherwise the instance for finite kernels would also make the addition s-finite, hence in the inference rule the arguments of the addition would be required to be finite even to decide s-finiteness, which is not the desired requirement. Finally, we use these proofs to instantiate the respective structures. See [5] for details.

5 Composition of s-Finite Kernels

The main result that makes it possible to use s-finite kernels for program semantics is their stability by composition. This is a consequence of a lemma by Staton [30, Lemma 3]. We explain its formal proof, highlighting the technicalities about measurability. We focus on the latter because related work [30, 31, 36] does not provide details about them and because related formalizations [17, 38] indicate that measurability results in general tend to be axiomatized.

5.1 Definition of Composition and Stability Proof

Given a kernel $l : X \rightsquigarrow Y$ and a kernel $k : X \times Y \rightsquigarrow Z$, we define the composition $l \boxed{;} k$ as

$$\int_y k\,(x,y)\,U(\mathbf{d}\,l\,x).$$

This translates directly in MathComp-Analysis to:

```
Definition kcomp x U := \int[l x]_y k (x, y) U.
```

We use the notation `l \; k` for `kcomp l k`. When l and k are s-finite, $l \boxed{;} k$ is an s-finite kernel $X \overset{\text{s-fin}}{\rightsquigarrow} Z$:

Theorem 5.1 ([30, Lemma 3])**.** *Given two s-finite kernels l and k, their composition is an s-finite kernel, and, given a non-negative measurable function f, we have for all x:*

$$\int_z f\,z(\mathbf{d}\,l \boxed{;} k\,x) = \int_y \left(\int_z f\,z(\mathbf{d}\,k\,(x,y)) \right) (\mathbf{d}\,l\,x) \qquad (1)$$

In the rest of this section, we explain the formalization of the proof of Theorem 5.1. For the sake of clarity, we decompose the proof in the following steps:

[2] This is another use case of phantom types as used for example in MathComp to define `horner_morph`.

1. When `l` and `k` are kernels, for all `x`, `(l \; k) x` is a measure. The only difficulty is to prove σ-additivity (Sect. 2). Staton gives a proof in 3 lines that translates to a Coq script of less than 8 lines using lemmas from MathComp-Analysis's measure and integration theory [2]. See [5, lemma `kcomp_sigma_additive`].
2. When `l` and `k` are finite kernels, `l \; k` is a finite kernel. For the finiteness property, it suffices to take as an upper bound for `l \; k` the product of the upper bounds of the two finite kernels `l` and `k` (see [5, lemma `mkcomp_finite`]). The difficulty rather lies in proving the measurability property, i.e.:
```
Lemma measurable_fun_kcomp_finite U :
  measurable U ->
  measurable_fun setT ((l \; k) ^~ U).
```
We defer this part of the proof to Sect. 5.2.
3. When `l` and `k` are s-finite kernels, `l \; k` is an s-finite kernel. For the s-finiteness property, Staton gives a proof in 5 lines [30, Lemma 3] that translates to a Coq script of less than 23 lines [5, lemma `mkcomp_sfinite`]. This is essentially about the manipulation of iterated operators and their commutativity with integrals and can be carried out formally using standard lemmas. Again, the difficulty rather lies in proving the measurability property. However, it is similar to Step 2. and reuses its lemmas, so that the next section (Sect. 5.2) should provide enough insights for formalization; see [5] for details.
4. Finally, the proof of equation (1) is an application of the monotone convergence theorem that follows the usual pattern: the lemma is first proved for indicator functions, then for simple functions, and finally for measurable functions using the approximation theorem and the monotone convergence theorem (see Sect. 2). It should be noted however that this requires to prove that the function $\lambda y. \int_z f_n\, z(\mathbf{d}\, k(x,y))$ is measurable for all x and n (where f_i's form the approximation of f) which can be proved as consequence of the lemmas from Step 2. See [5, lemma `integral_kcomp`].

5.2 Composition of Finite Kernels

The proof that the composition of finite kernels is a kernel is not detailed by Staton [30, Sect. 3.2]:

> "The measurability of each $(k_i \star l_j)(-,U) : X \to [0,\infty]$ follows from the general fact that for any measurable function $f : X \times Y \to [0,\infty]$, the function $\int_Y l_j(-,dy) f(-,y) : X \to [0,\infty]$ is measurable (e.g. [28, Thm. 20(ii)]). [...] This step crucially uses the fact that each measure $l_j(x,-)$ is finite."

Staton does not provide details either in a more recent version of his paper [31], nor do Vakar and Ong who also send the reader back to Pollard's book [36, Thm. 5].

The needed lemma can be taken to be the following one, the measurability of $\lambda x. \int_y k(x,y)(\mathbf{d}\, l\, x)$ under appropriate hypotheses:

```
Lemma measurable_fun_integral_finite_kernel d d'
  (X : measurableType d) (Y : measurableType d')
  (R : realType) (l : R.-fker X ~> Y)
  (k : X * Y -> \bar R) (k0 : forall z, 0 <= k z)
  (mk : measurable_fun setT k) :
measurable_fun setT (fun x => \int[l x]_y k (x, y)).
```

It can be proved by, first, transforming the goal about the integral into a goal about x-sections (see Sect. 3) and, second, applying a lemma akin to the "fundamental lemma" of Lebesgue integration.

From Integral to x-Sections. Instead of proving the measurability of $\lambda x. \int_y k\,(x,y)(\mathbf{d}\, l\, x)$, one can consider an approximation of k by an increasing sequence of non-negative simple functions k_n converging pointwise towards k (using the approximation theorem—Sect. 2) and prove the measurability of $\lambda x. l\, x\, (k_n^{-1}\{r\}|_x)$ for all n and r:

```
Lemma measurable_fun_xsection_integral
 (l : X -> {measure set Y -> \bar R})
 (k : X * Y -> \bar R)
 (k_ : ({nnsfun X * Y >-> R})^nat)
 (ndk_ : nondecreasing_seq k_)
 (k_k : forall z, EFin \o (k_ ^~ z) --> k z) :
(forall n r, measurable_fun setT
 (fun x => l x (xsection (k_ n @^-1` [set r]) x))) ->
measurable_fun setT (fun x => \int[l x]_y k (x, y)).
```

The proof uses the monotone convergence theorem to reduce the problem to the problem of the measurability of each $\lambda x. \int_y k_n(x,y)(\mathbf{d}\, l\, x)$ which are the same functions as $\lambda x. \sum_{r \in \mathrm{range}(k_n)} \int_y r \mathbb{1}_{k_n^{-1}\{r\}}(x,y)(\mathbf{d}\, l\, x)$ in virtue of the properties of simple functions. The latter are more easily seen to be measurable.

The Fundamental Lemma. The measurability of the measure of x-sections is the matter of the following lemma:

```
Context d d' (X : measurableType d)
  (Y : measurableType d') (R : realType).
Variable k : R.-fker X ~> Y.
Let phi A := fun x => k x (xsection A x).
Lemma measurable_fun_xsection_finite_kernel A :
  A \in measurable -> measurable_fun setT (phi A).
```

This is a variant for kernels of the so-called "fundamental lemma" [23] of Lebesgue integration. It is qualified as such because it is an important lemma to prove Fubini's theorem and because it is non-trivial compared to most proofs of measurability of functions seen in standard undergraduate textbooks on Lebesgue integration.

The idea of the proof is to show that the set `XY` defined as

```
[set A | measurable A /\ measurable_fun setT (phi A)]
```

is equal to the set of measurables associated with `X * Y`, i.e., its product σ-algebra.

This boils down to another lemma that shows that it is sufficient to prove that, for all x, there is a (finite) upper-bound for the measure of all measurables. The upper bound provided by the finite kernel k is an appropriate witness. For the sake of completeness, here is the formal statement:

```
Variable k : R.-ker X ~> Y.
Variables (D : set Y) (mD : measurable D).
Let kD x := mrestr (k x) mD.
Let phi A := fun x => kD x (xsection A x).
Let XY := [set A | measurable A /\
                   measurable_fun setT (phi A)].
Lemma measurable_prod_subset_xsection_kernel :
    (forall x, exists M, forall X, measurable X ->
                                   kD x X < M%:E) ->
  measurable `<=` XY.
```

It is stated for a kernel k and uses restrictions of measures to a measurable subset D of Y (Sect. 3.2). Its proof relies on the properties of *monotone classes* that have been used in MathComp-Analysis to prove Fubini's theorem [2].

Comparison with Lebesgue Integration. It is informative to compare the lemma we proved for kernels to the corresponding lemma used in MathComp-Analysis to prove Fubini's theorem:

```
Lemma measurable_fun_fubini_tonelli_F d1 d2
  (T1 : measurableType d1) (T2 : measurableType d2)
  (R : realType) (m2 : {measure set T2 -> \bar R})
  (sf_m2 : sigma_finite [set: T2] m2)
  (f : T1 * T2 -> \bar R) (f0 : forall x, 0 <= f x)
  (mf : measurable_fun setT f) :
 measurable_fun setT (fun x => \int[m2]_y f (x, y)).
```

There are differences because of which the lemma is not directly usable for kernels. The lemma is stated in terms of measures (which are less general than kernels) that are σ-finite (rather than finite or s-finite). In the conclusion, the measure m2 does not depend on x while in the case of kernels (lemma measurable_fun_integral_finite_kernel), l x depends on x in general.

6 Semantics of a Probabilistic Programming Language

In this section, we propose an encoding of the semantics of a probabilistic programming language with sampling, scoring, and normalization. It forms a subset of the probabilistic programming language of Staton [30, 31] and it is rich enough to write examples such as the one we explained in Sect. 1.

6.1 Staton's Probabilistic Programming Language

Let us recall the syntax of the language proposed by Staton [30, Sect. 3.1] [31, Sect. 4.1, 4.3]. In this language, types are ranged over by $\mathbf{A}, \mathbf{B}, \ldots$:

$$\mathbf{A}, \mathbf{B} \quad ::= \quad \mathbf{R} \mid P(\mathbf{A}) \mid 1 \mid \mathbf{A} \times \mathbf{B} \mid \textstyle\sum_{i \in I} \mathbf{A}_i$$

$\mathbf{R}$ is for the type of real numbers. $P(\mathbf{A})$ is for the type of distributions over $\mathbf{A}$. 1 is for a type with one element. $\mathbf{A} \times \mathbf{B}$ is for the cartesian product. $\sum_{i \in I} \mathbf{A}_i$ (where I is countable) is for a sum; for example, $1 + 1$ corresponds to the type *bool* of boolean numbers.

Terms are ranged over by $t, u, \ldots$:

$$\begin{aligned} t, u \quad ::= \quad & (i, t) \mid \mathtt{if}\, t\, \mathtt{then}\, t_1\, \mathtt{else}\, t_2 \mid \mathtt{tt} \mid (t_0, t_1) \mid \pi_j(t) \mid \\ & f(t) \mid x \mid \mathtt{ret}\, t \mid \mathtt{let}\, x := t\, \mathtt{in}\, u \mid \\ & \mathtt{sample}(t) \mid \mathtt{score}(t) \mid \mathtt{normalize}(t) \end{aligned}$$

Terms can be tagged (i, t); for example, $(1, 1)$ represents the boolean number true and $(2, 1)$ represents the boolean number false. If-then-else branching is for branching; this is a simple version of the case branching instruction by Staton. tt is the element of type 1. (t_0, t_1) is a pair whose projections are accessed with π_0 and π_1. The symbol f is a constant that stands for a measurable function. Variables are ranged over by x (and y, etc.). Last we have return, sequencing, and the three instructions specific to probabilistic programming languages that we will explain below in more details.

Typing judgments distinguish *deterministic* and *probabilistic* terms. Typing contexts are tuples $(x_1 : \mathbf{A}_1; \ldots; x_n : \mathbf{A}_n)$ ranged over by Γ. The typing judgment is $\Gamma \vdash_{\mathrm{d}} t : \mathbf{A}$ for deterministic terms and $\Gamma \vdash_{\mathrm{p}} t : \mathbf{A}$ for probabilistic terms. We reproduce here the typing rules for sums (tagged terms), products, variables, and measurable functions. They are all about deterministic terms.

$$\frac{\Gamma \vdash_{\mathrm{d}} t : \mathbf{A}_i}{\Gamma \vdash_{\mathrm{d}} (i, t) : \sum_{i \in I} \mathbf{A}_i} \qquad \frac{}{\Gamma \vdash_{\mathrm{d}} \mathtt{tt} : 1}$$

$$\frac{\Gamma \vdash_{\mathrm{d}} t_0 : \mathbf{A} \quad \Gamma \vdash_{\mathrm{d}} t_1 : \mathbf{B}}{\Gamma \vdash_{\mathrm{d}} (t_0, t_1) : \mathbf{A} \times \mathbf{B}} \qquad \frac{\Gamma \vdash_{\mathrm{d}} t : \mathbf{A}_0 \times \mathbf{A}_1}{\Gamma \vdash_{\mathrm{d}} \pi_i t : \mathbf{A}_i}$$

$$\frac{}{\Gamma, x : \mathbf{A}, \Gamma' \vdash_{\mathrm{d}} x : \mathbf{A}} \qquad \frac{\Gamma \vdash_{\mathrm{d}} t : \mathbf{A}}{\Gamma \vdash_{\mathrm{d}} f\, t : \mathbf{B}}\, f \text{ is measurable}$$

The typing rules for the other instructions illustrate the interplay between deterministic and probabilistic terms:

$$\frac{\Gamma \vdash_{\mathrm{d}} t : \mathbf{A}}{\Gamma \vdash_{\mathrm{p}} \mathtt{ret}\, t : \mathbf{A}} \qquad \frac{\Gamma \vdash_{\mathrm{p}} t : \mathbf{A} \quad \Gamma, x : \mathbf{A} \vdash_{\mathrm{p}} u : \mathbf{B}}{\Gamma \vdash_{\mathrm{p}} \mathtt{let}\, x := t\, \mathtt{in}\, u : \mathbf{B}}$$

$$\frac{\Gamma \vdash_{\mathrm{d}} t : \mathit{bool} \quad \Gamma \vdash_z t_1 : \mathbf{A} \quad \Gamma \vdash_z t_2 : \mathbf{A}}{\Gamma \vdash_z \mathtt{if}\, t\, \mathtt{then}\, t_1\, \mathtt{else}\, t_2 : \mathbf{A}}\, z \in \{d, p\}$$

$$\frac{\Gamma \vdash_{\mathrm{d}} t : P(\mathbf{A})}{\Gamma \vdash_{\mathrm{p}} \mathtt{sample}(t) : \mathbf{A}} \qquad \frac{\Gamma \vdash_{\mathrm{d}} t : \mathbf{R}}{\Gamma \vdash_{\mathrm{p}} \mathtt{score}(t) : 1}$$

$$\frac{\Gamma \vdash_{\mathrm{p}} t : \mathbf{A}}{\Gamma \vdash_{\mathrm{d}} \mathtt{normalize}(t) : \mathbf{R} \times P(\mathbf{A}) + 1 + 1}$$

We will comment more precisely about these last rules in the next section.

Basic Idea for the Semantics. Following Staton, we define a function $[\![\cdot]\!]$ to interpret the syntax of types, the syntax of terms, and the contexts respectively to measurable spaces, measurable functions or kernels, and product spaces. For

example, the measurable space corresponding to $\mathbf{R}$ is $[\![\mathbf{R}]\!]$, the measurable space $\mathbb{R}$ of real numbers with its Borel sets. A context $\Gamma = (x_1 : \mathbf{A}_1; \ldots; x_n : \mathbf{A}_n)$ is interpreted by the product space $[\![\Gamma]\!] \stackrel{\text{def}}{=} \prod_{i=1}^{n} [\![\mathbf{A}_i]\!]$. Deterministic terms $\Gamma \vdash_{\mathrm{d}} t : \mathbf{A}$ are interpreted by measurable functions $[\![\Gamma]\!] \to [\![\mathbf{A}]\!]$ and probabilistic terms $\Gamma \vdash_{\mathrm{p}} t : \mathbf{A}$ are interpreted by (s-finite) kernels $[\![\Gamma]\!] \overset{\text{s-fin}}{\rightsquigarrow} [\![\mathbf{A}]\!]$.

6.2 Formalization of Instructions

To represent types, typing contexts, and deterministic terms, we use and extend MathComp-Analysis [2] which provides several measurable spaces to represent types, product spaces to represent typing contexts by nested products, and measurable functions to represent deterministic terms. For probabilistic terms, we use our formalization of s-finite kernels and of their composition (Sections 4 and 5).

6.2.1 Instructions for Control-Flow.

Return. As seen in Sect. 6.1, the return instruction turns a deterministic term into a probabilistic one. Its semantics is defined using deterministic kernels (Sect. 4.6). Assuming that the semantics of the deterministic term t is represented by the measurable function `f`, we define the semantics of `ret` t as follows:

```
Definition ret (f : X -> Y)
    (mf : measurable_fun setT f) : R.-sfker X ~> Y :=
  kdirac mf.
```

Sequence. As seen in Sect. 6.1, sequencing applies only to probabilistic terms. We take the semantics of $\texttt{let}\, x := t\, \texttt{in}\, u$ to be $l \boxed{;} k$ of type $[\![\Gamma]\!] \overset{\text{s-fin}}{\rightsquigarrow} [\![\mathbf{B}]\!]$ where $l : [\![\Gamma]\!] \overset{\text{s-fin}}{\rightsquigarrow} [\![\mathbf{A}]\!]$ is the semantics of t and $k : [\![\Gamma; x : \mathbf{A}]\!] \overset{\text{s-fin}}{\rightsquigarrow} [\![\mathbf{B}]\!]$ is the semantics of u. The composition is indeed an s-finite kernel when the semantics of t and u are in virtue of Theorem 5.1. This choice of encoding means that we do not have a genuine notion of variable to stand for x. Occurrences of x inside u need to be translated to appropriate functions that access the environment $[\![\Gamma; x : \mathbf{A}]\!]$. This actually corresponds to the use of constant symbols f for measurable functions in Staton's language. Indeed, since typing contexts are represented by nested products, these access functions are $\mathtt{snd}^{[k \neq 1]} \circ \mathtt{fst}^{n-k}$ for access to the kth variable in a context Γ of size $n > 1$ ($1 \leq k \leq n$). These functions are measurable.

Branching. Branching for deterministic terms can be represented directly by measurable functions. Let us consider branching for probabilistic terms. The semantics we want is intuitively obvious: behave as one or the other branch according to whether or not the condition holds. This can be achieved using composition of s-finite kernels [30, Sect. 3.2]. First, we build two s-finite kernels for both branches. Given a kernel `k`, `kiteT k` defines a family of measures that behaves like `k` when the condition is true and like `mzero` otherwise:

```
Definition kiteT (xb : X * bool)
  : {measure set Y -> \bar R} :=
if xb.2 then k xb.1 else mzero.
```

Similarly, we define `kiteF k` for when the condition is false:

```
Definition kiteF (xb : X * bool)
  : {measure set Y -> \bar R} :=
if ~~ xb.2 then k xb.1 else mzero.
```

We can then equip `kiteT` and `kiteF` with the structures of kernel, s-finite kernel, and finite kernel (provided that `k` is respectively a kernel, an s-finite kernel, or a finite kernel). See [5, module `ITE`].

Finally we can define branching by simply using composition (Sect. 5.1) and the addition of s-finite kernels (Sect. 4.6):

```
Definition kite : R.-sfker T ~> T' :=
  kdirac mf \; kadd (kiteT u1) (kiteF u2).
```

This is an s-finite kernel by construction.

6.2.2 Language-specific Instructions.

Sampling. The sampling instruction takes as parameter a measurable function whose codomain is the measurable space of probability measures.

Before formalizing the semantics of sampling, we formalize the measure space of probability measures using MathComp-Analysis. The semantics of $P(\mathbf{A})$ (that we saw in Sect. 6.1) is the set of probability measures on $\mathbf{A}$ together with the σ-algebra generated by the sets of probability measures μ such that $\mu(U) < r$ for all measurable sets U and $r \in [0; 1]$. First, we define the set of probability measures `mu` such that `mu U < r%:E` for any `U` and `r`:

```
Definition mset U r :=
  [set mu : probability T R | mu U < r%:E].
```

We then use `mset` to define the generator of the σ-algebra of probability measures:

```
Definition pset : set (set (probability T R)) :=
  [set mset U r | r in `[0%R,1%R] & U in measurable].
```

Last, we use the `salgebraType` of MathComp-Analysis to generate the σ-algebra of probability measures:

```
Definition pprobability : measurableType pset.-sigma :=
  salgebraType pset.
```

The sampling instruction for a measurable function `P` is the kernel defined by `P` itself:

```
Variable P : X -> pprobability Y R.
Definition kprobability (mP : measurable_fun setT P)
  : X -> {measure set Y -> \bar R} := P.
```

The type of `P` makes it clear that sampling may depend on the program execution. To show that `kprobability mP` is a kernel, we need to show that `kprobability mP ^~ U` is a measurable function for any measurable set `U`:

```
Let measurable_fun_kprobability U : measurable U ->
  measurable_fun setT (kprobability mP ^~ U).
```

It suffices to show that the preimage by `P ^~ U` of any interval (of extended real numbers) $]-\infty, r[$ where r is a

real number is measurable. This is actually true by construction of `pprobability Y R`. Furthermore, we can equip `kprobability mP` with the structure of probability kernel thanks to the properties of probability measures.

When `P` is a (constant) probability measure, we can finally define the `sample` instruction as follows:

```
Definition sample (P : pprobability Y R)
    : R.-pker X ~> Y :=
  kprobability (measurable_fun_cst P).
```

For illustration, let us consider sampling from Bernoulli probability measures. The Bernoulli probability measure for a probability $0 \le p \le 1$ is defined by $p\delta_1 + (1-p)\delta_0$. In Coq, let us be given `p`, a non-negative number (type `{nonneg R}`, see Sect. 3.1) smaller than 1 (proof `p1`). Let `onem_nonneg p1` be the non-negative number $1 - p$ of type `{nonneg R}`. The following function can be used to define the Bernoulli probability measure corresponding to `p`:

```
Definition bernoulli := measure_add
  (mscale p (dirac true))
  (mscale (onem_nonneg p1) (dirac false)).
```

Indeed, `bernoulli` can be equipped with the structure of probability measure thanks to the properties of Dirac measures, of scaling of measures, and of addition of measures (see [5, lemma `bernoulli_setT`] for details). Let `bernoulli` be this probability measure. Then the semantics for sampling from the Bernoulli measure with probability $\frac{2}{7}$ is the expression `sample (bernoulli p27)`, provided that `p27` is a proof that $\frac{2}{7} \le 1$ (the fact that $\frac{2}{7}$ is a non-negative number is automatically inferred in MathComp-Analysis).

Scoring. Intuitively, the semantics of `score` is to scale the measure by a non-negative number. Concretely, it is a family of measures on a measurable space with one element. Let us recall Staton's definition of the semantics of `score` [30, Sect. 3.2]:

$$[\![\mathtt{score}(t)]\!]_{\gamma,U} \stackrel{\text{def}}{=} \begin{cases} |[\![t]\!]_\gamma| & \text{if } U = \{()\} \\ 0 & \text{if } U = \varnothing \end{cases}$$

Here, $\gamma \in [\![\Gamma]\!]$ is a valuation for variables. For a deterministic term t, $[\![t]\!]_\gamma$ is defined by induction on the structure of typing rules. For a probabilistic term $\Gamma \vdash_{\mathrm{p}} t : \mathbf{A}$, we have $[\![t]\!]_{\gamma,U} \in [0, \infty]$ with $U \in \Sigma_{\mathbf{A}}$. The parameter for `score` is typically the distribution function (in the discrete case) or the density function (in the continuous case) of a probability measure. It can also be 0 to encode a *hard contraint* (otherwise it is a *soft constraint*).

We can advantageously formalize the semantics of `score` as the combination of the scaling measure and of the Dirac measure:

```
Context d (T : measurableType d) (R : realType).
Variable f : T -> R.
Definition mscore t : {measure set unit -> \bar R} :=
  let p := NngNum (normr_ge0 (f t)) in
  mscale p (dirac tt).
```

(The expression `normr_ge0 x` is a proof that the absolute value of `x` is non-negative.) This definition shows almost directly that the execution of `score` multiplies the measure by a non-negative number. We use `mscore` to define `kscore` for the kernel structure of `score`:

```
Variable f : T -> R.
Definition kscore (mf : measurable_fun setT f)
    : T -> {measure set unit -> \bar R} :=
  mscore f.
```

However, proving that `score` is an s-finite kernel is a bit involved because even though each measure always returns a finite number, there is no uniform bound. We can however provide a family of finite kernels `k` indexed by a natural number `i` [30, Sect. 3.2]:

```
Definition k (mf : measurable_fun setT f) i t U :=
    if i%:R%:E <= mscore f t U < i.+1%:R%:E then
      mscore f t U
    else
      0.
```

Using `k`, we can rewrite `score` as a countable sum of finite kernels:

```
Let sfinite_kscore
  exists k : (R.-fker T ~> unit)^nat,
    forall x U, measurable U ->
      kscore mf x U = mseries (k ^~ x) 0 U.
```

See [5, module `SCORE`] for the complete proofs.

Normalization. Normalization turns a kernel into a probability measure by computing the measure of the full set (line 3) and dividing each measure by it (line 5):

```
1  (* f is a kernel R.-sfker X ~> Y *)
2  Definition mnormalize x U :=
3    let evidence := f x [set: Y] in
4    if (evidence == 0) || (evidence == +oo) then P U
5    else f x U * (fine evidence)^-1%:E.
```

When then measure of the full set is 0 or ∞, we are facing an exceptional case. To take care of these situations, we use a default probability (`P` at line 4). As we saw in the typing rules of Sect. 6.1, Staton deals with exceptional cases by adding the sum 1 + 1 to the return type of `normalize`. In MathComp, it has however proved more practical to deal with such exceptional cases with a default value rather than using an `option` type.

To finalize the semantics of normalization, we prove that that the function `mnormalize` can be equipped with the structure of probability measure, i.e., the type `probability Y R` (Sect. 3.2). See [5] for the formalization.

7 Formal Reasoning about Probabilistic Programs

Using the formal semantics in terms of s-finite kernels of the previous section (Sect. 6), it is now possible to provide formal proofs for properties stated in related work [30–32] or axioms used in other formalizations [17]. Since we did not

formalize a syntax, we state these properties directly at the semantic level. The property that was the main motivation for the use of s-finite kernels is commutativity (Sect. 7.1). We can also recover a number of rewriting laws to perform symbolic execution (Sect. 7.2).

7.1 Commutativity

7.1.1 Tonelli-Fubini's Theorem for s-Finite Measures.

The commutativity of sequencing is a consequence of a variant of Tonelli-Fubini's theorem for *s-finite measures*. Tonelli-Fubini's theorem does not hold for arbitrary measures; it holds for σ-finite measures but they are not closed under composition and pushforward, which is why the approach using s-finiteness is needed. Let `finite_measure` be a predicate that holds for a measure `mu` when `mu setT < +oo`. We define s-finite measures on the model of s-finite kernels (Sect. 4.3) with the following predicate:

```
Definition sfinite_measure d (T : measurableType d)
    (R : realType) (mu : set T -> \bar R) :=
  exists2 s : {measure set T -> \bar R}^nat,
    forall n, finite_measure (s n) &
    forall U, measurable U ->
      mu U = mseries s 0 U.
```

Let us now consider two s-finite measures `m1` and `m2` and a non-negative binary measurable function `f`. One can show that integrations of `f` over `m1` and `m2` commute:

```
Context d d' (X : measurableType d)
  (Y : measurableType d') (R : realType).
Variables (m1 : {measure set X -> \bar R})
  (sfm1 : sfinite_measure m1).
Variables (m2 : {measure set Y -> \bar R})
  (sfm2 : sfinite_measure m2).
Variables (f : X * Y -> \bar R)
          (f0 : forall xy, 0 <= f xy)
          (mf : measurable_fun setT f).
Lemma sfinite_fubini :
  \int[m1]_x \int[m2]_y f (x, y) =
  \int[m2]_y \int[m1]_x f (x, y).
```

See [5, lemma `sfinite_fubini`] for the formalization of [30, Prop. 5]. This is a consequence of Tonelli-Fubini's theorem for σ-finite measures which we borrowed from MathComp-Analysis [2, Sect. 6.5].

7.1.2 Commutativity using Tonelli-Fubini's Theorem.

Let us consider the following program transformation (written in pseudo-code):

```
let x := t in          ↔    let y := u in
let y := u in               let x := t in
ret (x, y)                  ret (x, y)
```

In the programs above, `x` is not free in `u` and `y` is not free in `t`. We want to show commutativity, i.e., that these two programs have the same semantics.

Let us encode both programs. We first declare an s-finite kernel `t` of type $Z \overset{\text{s-fin}}{\rightsquigarrow} X$ and an s-finite kernel `u` of type $Z \overset{\text{s-fin}}{\rightsquigarrow} Y$. To represent the second occurrences of `t` and `u`, we introduce two s-finite kernels `t'` and `u'` resp. of type $Z \times Y \overset{\text{s-fin}}{\rightsquigarrow} X$ and $Z \times X \overset{\text{s-fin}}{\rightsquigarrow} Y$. We capture the fact that `u` does not depend on `x` in the program on the left by using `u'` instead of `u` together with the following hypothesis:

```
forall x z, u z = u' (z, x)
```

In the absence of syntax, this is the semantic equivalent of the condition that `x` is not free in `u`. We add the similar hypothesis for `t` and `t'`. Using this encoding, we can apply Tonelli-Fubini's theorem for s-finite measures to prove the equality between both semantics:

```
Lemma letinC z A : measurable A ->
  letin t
  (letin u'
  (ret R (measurable_fun_pair var2of3 var3of3))) z A
= letin u
  (letin t'
  (ret R (measurable_fun_pair var3of3 var2of3))) z A.
Proof. (* see [5] *) Qed.
```

To represent variables we use access functions (as explained in Sect. 6.2.1). When inside the second `letin` in the program on the left, the environment is of the form $Z \times X \times Y$. The notation `var2of3` is for an access function to the second-to-last variable in a triple and `var3of3` is an access function to the last variable. Since the environment is of the form $Z \times Y \times X$ in the program on the right, the pair `(x, y)` is represented differently. It remains, as we proved, that both semantics are equal.

7.2 Symbolic Computation

7.2.1 Rewriting Laws.

Using the semantics, we can establish a number of generic program equations that can be used as rewriting laws. On can find such equations in related work, e.g., [30, Sect. 4.2], [32, Sect. 4]. The simplest example is the following identity law:

```
let x := return t in k x    ↔    k t
```

Let us assume that the semantics of (the deterministic term) `t` is given by the measurable function `f`. Then the corresponding formal lemma is:

```
Lemma letin_retk
  (f : X -> Y) (mf : measurable_fun setT f)
  (k : R.-sfker X * Y ~> Z)
  x U : measurable U ->
  letin (ret mf) k x U = k (x, f x) U.
```

The proof is by unfolding the definitions of `letin` and `ret` and appealing to the properties of the integral w.r.t. the Dirac measure.

Let us provide another example that illustrates the semantics of `score`. The execution in sequence of two scorings using functions f and g is equivalent to scoring using the function $\lambda x. f\,x \cdot g\,x$:

```
Lemma score_score (f : R -> R) (g : R * unit -> R)
    (mf : measurable_fun setT f)
```

```
    (mg : measurable_fun setT g) :
  letin (score mf) (score mg) =
  score (measurable_funM mf
        (measurable_fun_prod2 tt mg)).
```

In this lemma, `measurable_funM` is a proof that the product of two real number-valued measurable functions is measurable and `measurable_fun_prod2 tt mg` is a proof that `g ^~ tt` is measurable. This property is one of the illustrations in [32, Sect. 4] and one of the axioms in [17]. See [5, file `prob_lang.v`] for more examples of rewriting laws.

7.2.2 Application of Rewriting Laws.

Using rewriting laws such as the ones we saw in Sect. 7.2.1, we can symbolically execute programs and perform probabilistic reasoning. Let us consider the example of Sect. 1 reproduced here for convenience:

```
normalize (
  let x = sample (bernoulli (2 / 7)) in
  let r = if x then 3 else 10 in
  let _ = score (r ^ 4 / 4! * e ^ (- r)) in
  return x)
```

Recall that we have introduced the Bernoulli probability measure in Sect. 6.2.2. For the sake of generality, Let us assume that we are given a measurable function `h` of type `R -> R` to serve as a probability distribution/density function instead of the Poisson distribution function (`mh` is a proof that `h` is measurable). Let `k3` be the (proof of measurability of the) constant function that returns 3 and similarly for `k10`. Then we can encode the semantics of the above program using the instructions defined in Sect. 6 as follows:

```
Definition kstaton_bus : R.-sfker T ~> mbool :=
  letin (sample (bernoulli p27))
  (letin
    (letin (ite var2of2 (ret k3) (ret k10))
      (score (measurable_fun_comp mh var3of3)))
    (ret var2of3)).
Definition staton_bus := normalize kstaton_bus.
```

Like in Sect. 7.1.2, we use (measurable) functions instead of genuine variables to access the environment (here the functions `var2of2`, `var2of3`, and `var3of3`) . We can instantiate this program with, for example, the Poisson distribution function `poisson` ($\lambda k\, r. \frac{r^k}{k!} e^{-r}$) or the exponential density function `exp_density` ($\lambda x\, r. r\, e^{-rx}$). For the sake of illustration, let us instantiate `staton_bus` with $\lambda r. \frac{r^4}{4} e^{-r}$ as in our running example (`poisson4`/`mpoisson4` below). Using rewriting laws, we compute the resulting probability measure:

```
Lemma staton_busE P (t : R) U :
  let N := ((2 / 7) * poisson4 3 +
            (5 / 7) * poisson4 10)%R in
  staton_bus mpoisson4 P t U =
  ((2 / 7)%:E * (poisson4 3)%:E * \d_true U +
   (5 / 7)%:E * (poisson4 10)%:E * \d_false U)
  * N^-1%:E.
Proof. (* see [5] *) Qed.
```

This way, we recover the fact that the probability that it is the weekend is $\frac{\frac{2}{7}\frac{3^4}{4!}e^{-3}}{N} = \frac{0.048009}{0.0615208} = 0.780369$ and 0.219631 otherwise.

8 Related Work

We have already cited in Sect. 1 related work about s-finite kernels and their application to the semantics of probabilistic programming languages. Let us comment further about formalization experiments using proof assistants.

Heimerdinger and Shan experiment formal verification in Coq with axioms that encode a DSL with random choice and scoring [17]. The axioms assume that all functions are measurable and that all measures are s-finite. With our encoding of Staton's language, we can formally prove needed axioms (such as multiplying scores). The authors investigate correctness proofs of many program transformations, such as disintegration, that we plan to tackle in future work.

Zhang and Amin [38] give a formal semantics to a core probabilistic programming language with scoring, general recursion, and nested queries, and they formalize in Coq the fact that logical relatedness implies contextual equivalence. However, they axiomatize the notions of measure and integral, and ignore all issues of measurability. In contrast, our formalization is axiom-free (with the proviso of footnote 1).

There are several experiments of verifications of machine learning that involve the formalization of some probabilistic programming language, although not featuring the combined use of sampling, scoring, and normalization. In their formalization of PAC learnability for decision stumps in Lean, Tassarotti et al. represent stochastic procedures using the Giry monad [33]. Interestingly, they already observe with this simpler language the hurdle of dealing with measurability proofs. In their formal verification of generalization guarantees in Coq, Bagnall and Stewart encode a denotational semantics in which a program is interpreted as the expected value of a real number-valued valuation function w.r.t. the distribution of its results [10]. It is limited to discrete distributions and includes some axioms about probability theory (not of a nature to jeopardize soundness though, since some of them such as Pinkser's inequality are available in other work [7]).

More generally, the formalization of probabilistic programs is a long-standing topic for proof assistants. Hurd verified probabilistic algorithms in HOL, most notably the Miller–Rabin probabilistic primality test [20]. The probability theory that he developed on this occasion was a subset of what MathComp-Analysis proposes today. Audebaud and Paulin-Mohring verified randomized algorithms in Coq but the measure theory they relied on had some limitations (discrete distributions only, etc.) [9]. Bidlingmaier et al. have extended Audebaud and Paulin-Mohring's work to deal with continuous distributions [13]. Yet, with scoring and normalization in addition to sampling, Staton's language aimed

more at describing probability distributions than actual algorithms.

Our work extends MathComp-Analysis [1] in several ways. We needed a number of technical lemmas about measurable functions, measures and integration, and we also introduced probability measures (Bernoulli probability measures, etc.) (see Sections 3.2 and 6.2.2). More importantly, we introduced kernels, and also finite/s-finite measures, whereas Affeldt and Cohen focus on σ-finite measures, to formalize Fubini's theorem in particular [2], as it is customary in mathematics textbooks.

9 Conclusion

This paper provides an original and concrete application of the formalization of measure theory to the semantics of programming languages in Coq. We formalized kernels as a hierarchy of mathematical structures, showing in particular how to deal with a circularity between the definition of finite kernels and s-finite kernels. We developed the formal theory of s-finite kernels, most notably the fact that they are stable by composition. On this occasion, we focused on the formalization of measurability proofs which are non-trivial and, as a matter of fact, axiomatized in related work in Coq. We used this theory to formalize the semantics of a probabilistic programming language, used this semantics to establish properties of probabilistic programs, and, finally, used these properties to formally prove correctness of program transformations and perform symbolic execution.

Future Work. We plan to use our work to mechanize the equational reasoning approach advocated for by Shan [29] as an extension of related work on monadic equational reasoning [6, 8]. We are also investigating the formalization of a syntax for the language for which we have formalized a semantics so that writing programs and specifications becomes easier. More generally, we also plan to develop further the theory of s-finite measures that we started formalizing for the purpose of this work since they are occasionally used in probability theory (e.g., [22]).

Acknowledgments

The authors would like to thank Shin'ya Katsumata for his help, the members of the KAKENHI grant 22H00520 for their comments, and the anonymous reviewers for insightful comments that improved this paper. The first and the third author acknowledge the support of the JSPS KAKENHI grant 22H00520.

References

[1] Reynald Affeldt, Yves Bertot, Cyril Cohen Marie Kerjean, Assia Mahboubi, Damien Rouhling, Pierre Roux, Kazuhiko Sakaguchi, Zachary Stone, Pierre-Yves Strub, and Laurent Théry. 2022. MathComp-Analysis: Mathematical Components compliant Analysis Library. https://github.com/math-comp/analysis. Since 2017. Version 0.5.4.

[2] Reynald Affeldt and Cyril Cohen. 2022. Measure Construction by Extension in Dependent Type Theory with Application to Integration. arXiv:2209.02345 [cs.LO]

[3] Reynald Affeldt, Cyril Cohen, Marie Kerjean, Assia Mahboubi, Damien Rouhling, and Kazuhiko Sakaguchi. 2020. Competing Inheritance Paths in Dependent Type Theory: A Case Study in Functional Analysis. In *10th International Joint Conference on Automated Reasoning (IJCAR 2020), Paris, France, July 1–4, 2020 (Lecture Notes in Computer Science, Vol. 12167)*. Springer, 3–20. https://doi.org/10.1007/978-3-030-51054-1_1 Part II.

[4] Reynald Affeldt, Cyril Cohen, and Damien Rouhling. 2018. Formalization Techniques for Asymptotic Reasoning in Classical Analysis. *J. Formaliz. Reason.* 11, 1 (2018), 43–76. https://doi.org/10.6092/issn.1972-5787/8124

[5] Reynald Affeldt, Cyril Cohen, and Ayumu Saito. 2022. Semantics of Probabilistic Programs using s-Finite Kernels in Coq. MathComp-Analysis Pull Request https://github.com/math-comp/analysis/pull/749.

[6] Reynald Affeldt, Jacques Garrigue, David Nowak, and Takafumi Saikawa. 2021. A trustful monad for axiomatic reasoning with probability and nondeterminism. *J. Funct. Program.* 31 (2021), e17. https://doi.org/10.1017/S0956796821000137

[7] Reynald Affeldt, Manabu Hagiwara, and Jonas Sénizergues. 2014. Formalization of Shannon's Theorems. *J. Autom. Reason.* 53, 1 (2014), 63–103. https://doi.org/10.1007/s10817-013-9298-1

[8] Reynald Affeldt, David Nowak, and Takafumi Saikawa. 2019. A Hierarchy of Monadic Effects for Program Verification Using Equational Reasoning. In *13th International Conference on Mathematics of Program Construction (MPC 2019), Porto, Portugal, October 7–9, 2019 (Lecture Notes in Computer Science, Vol. 11825)*, Graham Hutton (Ed.). Springer, 226–254. https://doi.org/10.1007/978-3-030-33636-3_9

[9] Philippe Audebaud and Christine Paulin-Mohring. 2009. Proofs of randomized algorithms in Coq. *Sci. Comput. Program.* 74, 8 (2009), 568–589. https://doi.org/10.1016/j.scico.2007.09.002

[10] Alexander Bagnall and Gordon Stewart. 2019. Certifying the True Error: Machine Learning in Coq with Verified Generalization Guarantees. In *33rd AAAI Conference on Artificial Intelligence (AAAI 2019)*. Association for the Advancement of Artificial Intelligence (AAAI), 2662–2669. https://par.nsf.gov/servlets/purl/10138645

[11] Gilles Barthe, Joost-Pieter Katoen, and Alexandra Silva (Eds.). 2020. *Foundations of Probabilistic Programming*. Cambridge University Press. https://doi.org/10.1017/9781108770750

[12] Benjamin Bichsel, Timon Gehr, and Martin T. Vechev. 2018. Fine-Grained Semantics for Probabilistic Programs. In *27th European Symposium on Programming on Programming Languages and Systems (ESOP 2018), Thessaloniki, Greece, April 14–20, 2018 (Lecture Notes in Computer Science, Vol. 10801)*. Springer, 145–185. https://doi.org/10.1007/978-3-319-89884-1_6

[13] Martin E. Bidlingmaier, Florian Faissole, and Bas Spitters. 2021. Synthetic topology in Homotopy Type Theory for probabilistic programming. *Math. Struct. Comput. Sci.* 31, 10 (2021), 1301–1329. https://doi.org/10.1017/S0960129521000165

[14] Johannes Borgström, Ugo Dal Lago, Andrew D. Gordon, and Marcin Szymczak. 2016. A lambda-calculus foundation for universal probabilistic programming. In *21st ACM SIGPLAN International Conference on Functional Programming (ICFP 2016), Nara, Japan, September 18–22, 2016*. ACM, 33–46. https://doi.org/10.1145/2951913.2951942

[15] Cyril Cohen, Kazuhiko Sakaguchi, and Enrico Tassi. 2020. Hierarchy Builder: Algebraic hierarchies Made Easy in Coq with Elpi (System Description). In *5th International Conference on Formal Structures for Computation and Deduction (FSCD 2020), June 29–July 6, 2020, Paris, France (Virtual Conference) (LIPIcs, Vol. 167)*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 34:1–34:21. https://doi.org/10.4230/LIPIcs.FSCD.2020.34

[16] François Garillot, Georges Gonthier, Assia Mahboubi, and Laurence Rideau. 2009. Packaging Mathematical Structures. In *22nd International Conference on Theorem Proving in Higher Order Logics (TPHOLs 2009), Munich, Germany, August 17–20, 2009 (Lecture Notes in Computer Science, Vol. 5674)*. Springer, 327–342. https://doi.org/10.1007/978-3-642-03359-9_23

[17] Matthew Heimerdinger and Chung-chieh Shan. 2019. Verified Equational Reasoning on a Little Language of Measures. Workshop on Languages for Inference (LAFI 2019), Cascais, Portugal, January 15, 2019.

[18] Chris Heunen, Ohad Kammar, Sam Staton, and Hongseok Yang. 2017. A convenient category for higher-order probability theory. In *32nd Annual ACM/IEEE Symposium on Logic in Computer Science (LICS 2017), Reykjavik, Iceland, June 20–23, 2017*. IEEE Computer Society, 1–12. https://doi.org/10.1109/LICS.2017.8005137

[19] Michikazu Hirata, Yasuhiko Minamide, and Tetsuya Sato. 2022. Program Logic for Higher-Order Probabilistic Programs in Isabelle/HOL. In *16th International Symposium on Functional and Logic Programming (FLOPS 2022), Kyoto, Japan, May 10-12, 2022 (Lecture Notes in Computer Science, Vol. 13215)*. Springer, 57–74. https://doi.org/10.1007/978-3-030-99461-7_4

[20] Joe Hurd. 2001. *Formal verification of probabilistic algorithms*. Ph. D. Dissertation. Computer Laboratory, University of Cambridge.

[21] Dexter Kozen. 1985. A probabilistic PDL. *J. Comput. System Sci.* 30, 2 (1985), 162–178. https://doi.org/10.1016/0022-0000(85)90012-1

[22] Günter Last and Mathew Penrose. 2017. *Lectures on the Poisson Process*. Cambridge University Press.

[23] Daniel Li. 2016. *Intégration et applications—Cours et exercices corrigés*. Eyrolles.

[24] Assia Mahboubi and Enrico Tassi. 2021. *Mathematical Components*. Zenodo. https://doi.org/10.5281/zenodo.4457887

[25] Praveen Narayanan. 2019. *Verifiable and Reusable Conditioning*. Ph. D. Dissertation. School of Informatics, Computing, and Engineering, Indiana University. https://scholarworks.iu.edu/dspace/handle/2022/24645w

[26] Praveen Narayanan, Jacques Carette, Wren Romano, Chung-chieh Shan, and Robert Zinkov. 2016. Probabilistic Inference by Program Transformation in Hakaru (System Description). In *13th International Symposium on Functional and Logic Programming (FLOPS 2016), Kochi, Japan, March 4–6, 2016 (Lecture Notes in Computer Science, Vol. 9613)*. Springer, 62–79. https://doi.org/10.1007/978-3-319-29604-3_5

[27] Luke Ong and Matthijs Vákár. 2018. Radon-Nikodým derivatives and disintegration for s-finite measures: some semantic bases for probabilistic metaprogramming. Seminar 113: Metaprogramming for Statistical Machine Learning, Shonan Village Centre.

[28] D. Pollard. 2002. *A user's guide to measure theoretic probability*. CUP.

[29] Chung-chieh Shan. 2018. Equational reasoning for probabilistic programming. POPL TutorialFest.

[30] Sam Staton. 2017. Commutative Semantics for Probabilistic Programming. In *26th European Symposium on Programming (ESOP 2017), Uppsala, Sweden, April 22–29, 2017 (Lecture Notes in Computer Science, Vol. 10201)*. Springer, 855–879. https://doi.org/10.1007/978-3-662-54434-1_32

[31] Sam Staton. 2020. *Probabilistic Programs as Measures*. 43–74. https://doi.org/10.1017/9781108770750.003 Chapter in [11].

[32] Sam Staton, Hongseok Yang, Frank D. Wood, Chris Heunen, and Ohad Kammar. 2016. Semantics for probabilistic programming: higher-order functions, continuous distributions, and soft constraints. In *the 31st Annual ACM/IEEE Symposium on Logic in Computer Science (LICS 2016), New York, NY, USA, July 5–8, 2016*. ACM, 525–534. https://doi.org/10.1145/2933575.2935313

[33] Joseph Tassarotti, Koundinya Vajjha, Anindya Banerjee, and Jean-Baptiste Tristan. 2021. A formal proof of PAC learnability for decision stumps. In *10th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP 2021), Virtual Event, Denmark, January 17–19, 2021*, Catalin Hritcu and Andrei Popescu (Eds.). ACM, 5–17. https://doi.org/10.1145/3437992.3439917

[34] The Coq Development Team. 2022. *The Coq Proof Assistant Reference Manual*. Inria. Available at https://coq.inria.fr. Version 8.16.0.

[35] The Coq Development Team. 2022. Reversible Coercions. Section in [34]. https://coq.inria.fr/refman/addendum/implicit-coercions.html#reversible-coercions Coq Pull Request 15693.

[36] Matthijs Vákár and Luke Ong. 2018. On S-Finite Measures and Kernels. https://doi.org/10.48550/ARXIV.1810.01837

[37] Frank D. Wood, Jan-Willem van de Meent, and Vikash Mansinghka. 2014. A New Approach to Probabilistic Programming Inference. In *7th International Conference on Artificial Intelligence and Statistics (AISTATS 2014) Reykjavik, Iceland, April 22–25, 2014 (JMLR Workshop and Conference Proceedings, Vol. 33)*. JMLR.org, 1024–1032. http://proceedings.mlr.press/v33/wood14.html

[38] Yizhou Zhang and Nada Amin. 2022. Reasoning about "reasoning about reasoning": semantics and contextual equivalence for probabilistic programs with nested queries and recursion. *Proc. ACM Program. Lang.* 6, POPL (2022), 1–28. https://doi.org/10.1145/3498677

Received 2022-09-21; accepted 2022-11-21