Semantics of Probabilistic Programs using s-Finite Kernels in Coq
Reynald Affeldt, Cyril Cohen, Ayumu Saito

To cite this version:
Reynald Affeldt, Cyril Cohen, Ayumu Saito. Semantics of Probabilistic Programs using s-Finite Kernels in Coq. CPP 2023 - Certified Programs and Proofs, Jan 2023, Boston, United States. 10.1145/3573105.3575691. hal-03917948

HAL Id: hal-03917948
https://inria.hal.science/hal-03917948
Submitted on 2 Jan 2023

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L’archive ouverte pluridisciplinaire HAL, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d’enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.
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

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
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 et 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. 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 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.

## 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 $\emptyset$ and that is closed under complement and countable union. We note $\Sigma_X$ for such a σ-algebra and call measurable the sets in $\Sigma_X$. The standard σ-algebra on $\mathbb{R}$ is the smallest σ-algebra containing the intervals: the Borel sets. Given two σ-algebras $\Sigma_X$ and $\Sigma_Y$, the product σ-algebra is the smallest σ-algebra generated by $\{(A \times B) | A \in \Sigma_X, B \in \Sigma_Y\}$.

A (non-negative) measure is a function $\mu : \Sigma_X \rightarrow [0, \infty]$ such that $\mu(\emptyset) = 0$ and $\mu(\cup_i A_i) = \sum_i \mu(A_i)$ for pairwise-disjoint measurable sets $A_i$ where the sum is countable. The latter property is called σ-additivity. A probability measure on $\Sigma_X$ is a measure $\mu$ 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 $\delta_x$ is defined by $\delta_x(U) = [x \in U]$ (using the Iveron bracket notation). The countable addition of measures forms a measure. Given a measure $\mu$ and a measurable set $D$, $\mu_D \overset{\text{def}}{=} \lambda U. \mu(U \cap D)$ is a measure: the restriction of $\mu$ to $D$. A measure $\mu$ 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 $f : \Sigma_X \rightarrow \Sigma_Y$ 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: $f = \sum_{y \in \text{range}(f)} y \cdot f^{-1}(y)$. The approximation theorem states that, given a non-negative measurable function $f$, there is a non-decreasing sequence of non-negative simple functions $f_i$ that converges pointwise towards $f$.

We can integrate a measurable function $f$ w.r.t. a measure $\mu$ over $D$ to get an extended real number denoted by $\int_{x \in D} f(x) \, d\mu$. Integration satisfies the monotone convergence theorem, i.e., $\int_{x \in D} f(x) \, d\mu = \lim_{n \to \infty} \int_{x \in D} f_n(x) \, d\mu$ where $f_n$ 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 $\mu_1$ and $\mu_2$, we have Tonelli-Fubini’s theorem: $\int_f \int f(x, y) \, d\mu_1 \, d\mu_2 = \int_f \int f(x, y) \, d\mu_2 \, d\mu_1$.

As for the mathematical definitions of kernels, we will introduce them along their formalization in Coq.
3.1 Basic Definitions about Set Theory and Topology

The type set X is for sets of objects of type X. We can write \( A \vdash a \in \text{set } X \) to state that a belongs to the set \( A \). The full set of objects of type X is denoted by \( \text{set}(X) \); it is a notation for set T. The singleton \( \{a\} \) is denoted by \( \{a\} \). Set inclusion is denoted by \( \subseteq \). The preimage of the set \( A \) by \( f \) is denoted by \( f^{-1}(A) \). Sets can be defined by comprehension using the notation \( \{ x \in \text{set } X \mid P \} \), for the set of objects \( x \) such that \( P \) holds. The expression \( x \in \text{set } A \) is for the x-section \( A \times x \), i.e., the set of \( y \)'s such that \((x, y) \in A\).

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

We also make use of standard MathComp [24] notations. The notation \( f \circ y \) is for the function \( \lambda x. f x y \). The notation \( \& \) is for function composition. The projections of a pair are denoted by \( _1 \) and \( _2 \). Boolean operations are denoted by \( \neg \) for the negation and \( || \) for the disjunction. The notation \( r : E \) 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 \( \sigma \)-algebra's is \text{measurableType } d. More precisely, given \( T \) of type \text{measurableType } d and \( U \) of type \text{set } T, \text{measurable } U \) asserts that \( U \) belongs to the \( \sigma \)-algebra corresponding to \( T \). The parameter \( d \) controls the display of the measurable predicate, so that \text{measurable } U \) is printed as \text{d}_\text{measurable } U \). This is useful to disambiguate the local context of a proof in the presence of several \( \sigma \)-algebra's. See [2, Sect. 3.4] for details. Given \( T \) of type \text{measurableType } d, a measure on \( T \) is denoted by \( \text{measure } T \rightarrow \text{bar } R \), where \( R \) has type \text{realType} \). The Dirac measure is denoted by \( \text{dirac } a \ A \) (with notation \( \& \text{dirac } a \ A \)). The measure corresponding to the function \( \lambda x. 0 \) is \( \text{mzero} \). The sum of two measures \( m1 \) and \( m2 \) is \( \text{mmeasure } \_ \_ \) \( m1 \) \( m2 \). The measure corresponding to \( \lambda x. r \mu(x) \) where \( r \) is a non-negative number is \( \text{mmeasure } r \mu \) \( m \). The measure corresponding to the sum \( \sum_{i=0}^\infty f_i \) is \( \text{mseries } f \ n \). The restriction of the measure \( \mu \) to \text{measurable } D \) with \( D \vdash \text{measurable } D \) is \( \text{mrestr } \mu \) \( mD \). 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 \( \sigma \)-finite over \( D \) is \( \text{sigma}_\text{finite } D \mu \).

We write \text{measurable}_\text{fun } D \rightarrow 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 \( \text{nsfun } D \rightarrow R \).

Finally, the notation for the integral \( \int_{x \in D} f(x) (d\mu) \) is \( \text{\text{\int}}[\text{\text{\mu}}]_x(x \in D) f x \).

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 \( X \rightarrow Y \) is a function \( k : X \rightarrow \Sigma Y \rightarrow [0, \infty] \) 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 \( k : X \rightarrow Y \) is a finite kernel when there is a finite bound \( r \) such that for all \( x \), \( k x Y \leq r \). This is a uniform upper bound, i.e., the same \( r \) for all \( x \). Let us denote the type of finite kernels by \( X^\text{fin}_\rightarrow Y \).

A kernel \( k : X \rightarrow Y \) 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 \( X^\text{fin}_\rightarrow Y \) 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 \( k : X \rightarrow Y \) is a subprobability kernel when \( \sup_{x \in X} k x Y \leq 1 \) 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 \( X^\text{subprob}_\rightarrow Y \) and the one of probability kernels by \( X^\text{prob}_\rightarrow Y \).

The kernels we have defined above are organized as a hierarchy of mathematical structures depicted in Fig. 1 that we will now formalize.
When processing this definition, we saw in Sect. 4.1 that a kernel \( X \rightarrow Y \) is a function \( k : X \rightarrow \Sigma_Y \rightarrow [0, \infty] \) that satisfies two conditions. We formalize this definition using a \texttt{mixin}, i.e., an interface. The fact that for all \( x, k \in k \) is a measure (condition (i)) is captured by the predicate \texttt{measure_fam_uub} (where we use advantageously the notation \( ^\wedge \) from \textsc{MathComp}—Sect. 3.1).

The \textsc{Hierarchy-Builder} commands are all prefixed with \texttt{HB}.

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

\[ \texttt{HB.mixin Record isKernel} \]

\[ \texttt{d d' (X : measurableType d) (Y : measurableType d')} \]

\[ \texttt{(R : realType) (k : X \rightarrow (measure set Y \rightarrow R))} \]

\[ \texttt{:= \{ measurable_kernel : forall U, measurable U \rightarrow measurable_fun setT (k ^\wedge U) \}.} \]

The \textsc{Hierarchy-Builder} commands are all prefixed with \texttt{HB}.

The \textsc{Hierarchy-Builder} commands are all prefixed with \texttt{HB}.

The \textsc{Hierarchy-Builder} commands are all prefixed with \texttt{HB}.

When processing this definition \textsc{Hierarchy-Builder} generates a function \texttt{isKernel.Build} that can be used to create instances of kernels. Hereafter, we use the \texttt{Coq} notation \( \texttt{R.-ker X \rightarrow Y} \) for the type of kernels \( X \rightarrow Y \).

### Example of Instantiation

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

\[ \texttt{Context d d' (X : measurableType d) (Y : measurableType d')} (R : realType). \]

\[ \texttt{Variable k : (R.-ker X \rightarrow Y)^nat.} \]

\[ \texttt{Definition kseries : X \rightarrow (measure set Y \rightarrow \bar R) :=} \]

\[ \texttt{fun x := mseries (k ^\wedge x) 0.} \]

The measure structure of \texttt{mseries} is provided by \textsc{MathComp-Analysis}. It is inferred automatically by \texttt{Coq} without any additional syntax thanks to the reversible coercion mechanism available since \texttt{Coq 8.16}.

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

\[ \texttt{Lemma measurable_fun_kseries (U : set Y) : measurable U \rightarrow measurable_fun setT (kseries ^\wedge U).} \]

\[ \texttt{Proof. (* see [5] *) Qed.} \]

Finally, we instantiate the kernel structure using the function \texttt{isKernel.Builder} mentioned above:

\[ \texttt{HB.instance Definition _ := isKernel.Build _ _ _ _ kseries measurable_fun_kseries.} \]

The symbol \texttt{kseries} is now equipped with the structure of kernel, as indicated by the \texttt{HB} about command:

\[ \texttt{HB: kseries is canonically equipped with mixins:
- kernel Kernel} \]

### 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 \texttt{measure_fam_uub} to characterize them:

\[ \texttt{Context d d' (X : measurableType d) (Y : measurableType d')} (R : realType). \]

\[ \texttt{Variable k : X \rightarrow (measure set Y \rightarrow \bar R).} \]

\[ \texttt{Definition measure_fam_uub :=} \]

\[ \texttt{exists r, forall x, k x [set: Y] < r%.} \]

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 \( \texttt{(sfinite_subdef below)} \) that states the existence of a sequence \( s \) of kernels such that \( k \) is equal to \texttt{kseries} \( s \). The fact that each kernel composing the sequence is finite is captured using the predicate \texttt{measure_fam_uub}:

\[ \texttt{HB.mixin Record Kernel_isSFinite_subdef} \]

\[ \texttt{d d' (X : measurableType d) (Y : measurableType d')} (R : realType) (k : X \rightarrow (measure set Y \rightarrow \bar R)) \]
After declaring this interface, we use the `Hierarchy-Builder` commands (see factories and builders in [15]) to generate a function `Kernel_isFinite`. `Build` builds 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:

```coq
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 := (R : realType) (k : measurableType d)
    (Y : measurableType d’)
    (R : realType) := (k of @Kernel _ _ _ _ _ k &
    Kernel_isSFinite_subdef _ _ X Y R k).

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:

```coq
HB.mixin Record FiniteKernel_isSubProbability
  d d’ (X : measurableType d) (Y : measurableType d’)
  (R : realType) (k : X -> (measure set Y -> \bar R))
  := { sprob_kernel : forall x U, measurable U -> k x U = kseries s x U }.
```

Next, the `Coq` notation for an s-finite kernel `X \xrightarrow{sfinite} Y` is `R.-sfker X -> Y`.

Hereafter, the 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):

```coq
HB.factory Record Kernel_isSubProbability
  d d’ (X : measurableType d) (Y : measurableType d’)
  (R : realType) (k : X -> (measure set Y -> \bar R))
  := { prob_kernel : forall x, k x Y = 1 }.
```

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

```coq
HB.structure Definition SubProbabilityKernel
  d d’ (X : measurableType d) (Y : measurableType d’)
  (R : realType) := {
    sprob_kernel : sfinite := (R : realType) (k : measure set Y -> \bar R))
    of isKernel _ _ _ _ _ k := {
      measure_uub : measure_fam_uub k k }.
```

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

```coq
HB.mixin Record SubProbability_isProbability
  d d’ (X : measurableType d) (Y : measurableType d’)
  (R : realType) := {
    sprob_kernel : forall x U, measurable U -> k x U = kseries s x U }.
```

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

```coq
HB.structure Definition ProbabilityKernel
  d d’ (X : measurableType d) (Y : measurableType d’)
  (R : realType) := {
    prob_kernel : forall x U, measurable U -> k x U = 1 }.
```

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

```coq
HB.factory Record SFiniteKernel_isFinite
  d d’ (X : measurableType d) (Y : measurableType d’)
  (R : realType) := {
    measure_uub : measure_fam_uub 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 \( \text{kdirac} \) using the \( \text{dirac} \) 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 \( \text{mf} \) 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 \( \text{dirac} \) cannot be found. That is why we define \( \text{kdirac} \), a wrapper around \( \text{dirac} \) 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 \( \text{kdirac} \) indeed forms a probability kernel, we can use the factory of Sect. 4.5. First, we show that \( \text{kdirac} \) 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).
HB.instance Definition _ := isKernel.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 \( \text{kadd} \) by lifting the addition of measures \( \text{measure_add} \) (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 \( \text{kadd} \) 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 \( l : X \rightarrow Y \) and a kernel \( k : X \times Y \rightarrow Z \), we define the composition \( l[k] \) as

```
\int_y k(x,y) U(dl x).
```

This translates directly in MathComp-Analysis to:

```
Definition kcomp x U := \int[1 x]_y k (x, y) U.
```

We use the notation \( \int \setminus \) \( k \) for \( \text{kcomp} \) \( 1 \). When \( l \) and \( k \) are s-finite, \( l[k] \) is an s-finite kernel \( X \rightarrow^{sfin} Z \):

**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(\int x k z x) = \int y (\int z f(\int x k z x)) (dl x)
```

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:
1. When 1 and k are kernels, for all \( x \), \( \{ 1 \land \colon k \} x \) is a measure. The only difficulty is to prove \( \sigma \)-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 \( kcomp_{\sigma}\text{-additive} \)].

2. When 1 and k are finite kernels, \( 1 \land \colon k \) is a finite kernel. For the finiteness property, it suffices to take as an upper bound for \( 1 \land \colon k \) the product of the upper bounds of the two finite kernels 1 and k (see [5, lemma \( mkcomp_{\text{finite}} \)]). The difficulty rather lies in proving the measurability property, i.e.:

\[
\text{Lemma measurable\_fun\_kcomp\_finite U : measures U -> measurable\_fun setT \((1 \land \colon k)^{\sim} U\).}
\]

We defer this part of the proof to Sect. 5.2.

3. When 1 and k are \( s \)-finite kernels, \( 1 \land \colon k \) 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 \( mkcomp_{\text{sfinite}} \)]. 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 \( \lambda y. \int_x f_n z(d(k(x, y))) \) is measurable for all \( x \) and \( n \) (where \( f_n \)’s form the approximation of \( f \)) which can be proved as consequence of the lemmas from Step 2. See [5, lemma \( \text{integral}_{\text{kcomp}} \)].

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

\[
\text{Lemma measurable\_fun\_integral\_finite\_kernel d d’ (X : measurableType d) (Y : measurableType d’) (R : realType) (1 : R \text{-}\text{ker} X \rightarrow \text{Y}) (k : X \times Y \rightarrow \text{bar R}) (k_0 : forall z, 0 <= k z) (l : R \rightarrow \text{bar R}) (mk : measurable\_fun setT k) : measurable\_fun setT \((\text{fun x => \int[l x]_y k(x, y))}.\)
\]

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.

\textbf{From Integral to } x\text{-Sections.} Instead of proving the measurability of \( \lambda x. \int_y f(x, y)(d l x) \), 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 \( \lambda x.l x(k_n^{-1}(r)]_1) \) for all \( n \) and \( r \):

\[
\text{Lemma measurable\_fun\_xsection\_integral \((1 : X \rightarrow \{\text{measure set Y \rightarrow \text{bar R}}\}) \((k : X \times Y \rightarrow \text{bar R}) \((k_0 : (\text{nnfun } X \times Y \rightarrow R)^{\text{nat}} \rightarrow \text{nat}) \((\text{ncomp} k_0 : \text{nondecreasing\_seq k_0}) \((k \_ k : \text{forall } z, \text{Ef\_in} \emptyset \,(k_0 \rightarrow k z) \rightarrow k z) : (\text{forall } n r, \text{measurable\_fun setT} \((\text{fun x => \int[l x]_y (k_n \circ^{-1}(r)]_1) ) \rightarrow \text{measurable\_fun setT} \((\text{fun x => \int[l x]_y k(x, y))}.\)
\]

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

\textbf{The Fundamental Lemma.} The measurability of the measure of \( x \)-sections is the matter of the following lemma:

\[
\text{Context d d’ (X : measurableType d) (Y : measurableType d’) (R : realType). Variable k : R \text{-}\text{ker} X \rightarrow \text{Y}. Let phi A := fun x => k x (xsection A x). Lemma measurable\_fun\_xsection\_finite\_kernel A : A \in \text{measurable} \rightarrow \text{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 \( [\text{set A | measurable A /\ measurable\_fun setT} \((\phi A)] \) is equal to the set of measurables associated with \( X \times Y \), i.e., its product \( \sigma \)-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 measurable functions. The upper bound provided by the finite kernel k is an appropriate witness. For the sake of completeness, here is the formal statement:

\[\text{Variable } k : \mathbb{R} \rightarrow \text{ker } X \rightarrow Y. \]

\[\text{Variables } (D : \text{set } Y) \text{ (mD : measurable } D). \]

Let \(kD x := \text{mresr} (k x) \text{ mD}. \quad \text{Let } \phi A := \text{fun } x => kD x (\text{xsection } A x). \]

\[\text{Let } XY := \{\text{set } A | \text{measurable } A \wedge \text{measurable_fun setT } (\phi A)\}. \]

\[\text{Lemma measurable_fun setT } (\exists \text{forall } x, \exists \text{exists } M, \text{forall } X \rightarrow \text{kD } x \text{ } x < M%:E) \rightarrow \text{measurable } (\text{ Y).} \]

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-Analsys 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:

\[\text{Lemma measurable_fun_fubini_tonelli_F } d1 \text{ d2 } \]
\[(\text{T1 : measurableType d1) } \text{ (T2 : measurableType d2) } \]
\[(\text{R : realType) } \text{ (m2 : measure set T2 \rightarrow } \text{\bar R)} \]
\[(\text{sf}_\text{m2} : \text{sigma_finite } \text{set } T2 \text{ m2}) \]
\[\text{ (f : T1 } \times \text{T2 } \rightarrow \text{\bar R)} (\text{f0 : forall } x, 0 \text{ } \leq \text{ f } x) \]
\[\text{ (mf : measurable_fun setT } f) : \text{measurable_fun setT } (\text{fun } x => \text{\int[m2]}_y f (x, y)). \]

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 \(\sigma\)-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_dir, \(\{x, x\} \) 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 \(A, B, \ldots\):

\[A, B := \mathbb{R} | P(A) | 1 | A \times B | \sum_{i \in I} A_i \]

\(R\) is for the type of real numbers. \(P(A)\) is for the type of distributions over \(A, 1\) is for a type with one element. \(A \times B\)

is for the cartesian product. \(\sum_{i \in I} A_i\) (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 \(t, u, \ldots:\)

\[t, u := (i, t) | (if \text{ then } t \text{ else } t) | tt | (t_0, t_1) | \pi_i(t) | f(t) | x | ret t | let x := t in u | \]
\[\text{sample}(t) | \text{score}(t) | \text{normalize}(t) \]

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 \(\pi_0\) and \(\pi_1\). 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 \((x_1 : A_1; \ldots; x_n : A_n)\) ranged over by \(\Gamma\). The typing judgment is \(\Gamma \vdash_d t : A\) for deterministic terms and \(\Gamma \vdash_p t : A\) for probabilistic terms. We reproduce here the typing rules for sums (tagged terms), products, variables, and measurable functions. They are all about deterministic terms.

\[
\begin{align*}
\Gamma \vdash_d t : A_i & \quad \Gamma \vdash_d (i, t) : \sum_{i \in I} A_i \\
\Gamma \vdash_d t_0 : A & \quad \Gamma \vdash_d t_1 : B \\
\Gamma \vdash_d (t_0, t_1) : A \times B & \\
\Gamma \vdash_d t : A_0 \times A_1 & \\
\Gamma \vdash_d x : A & \quad \Gamma \vdash_d f : B \quad f \text{ is measurable} \\
\Gamma \vdash d \text{let } x := t \text{ in } u : B & \\
\Gamma \vdash d \text{ ret } t : A & \\
\Gamma \vdash d \text{ sample}(t) : A & \\
\Gamma \vdash d \text{ score}(t) : I & \\
\Gamma \vdash d \text{ normalize}(t) : R \times P(A) + 1 + 1 &
\end{align*}
\]

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

**Basic Idea for the Semantics.** Following Staton, we define a function \(\llbracket \cdot \rrbracket\) 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 \( \mathbb{R} \) is \([\mathbb{R}]\), the measurable space \( \mathbb{R} \) of real numbers with its Borel sets. A context \( \Gamma = (x_1 : A_1; \ldots ; x_n : A_n) \) is interpreted by the product space \([\Gamma]\) \(\overset{\text{def}}{=} \prod_{i=1}^{n} [A_i]\). Deterministic terms \( \Gamma \vdash_d t : A \) are interpreted by measurable functions \([\Gamma] \rightarrow [A]\) and probabilistic terms \( \Gamma \vdash_p t : A \) are interpreted by (s-finite) kernels \([\Gamma] \overset{\text{s-fin}}{\rightarrow} [A]\).

### 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 \( \text{ret} \ t \) as follows:

\[
\text{Definition ret} \ (f : X \rightarrow Y) = \text{mdirac} \ f.
\]

**Sequence.** As seen in Sect. 6.1, sequencing applies only to probabilistic terms. We take the semantics of \( \text{let} \ x := t \ \text{in} \ u \) to be \([\Gamma]^k\) of type \([\Gamma]^k \overset{\text{s-fin}}{\rightarrow} [B]\) where \( l : [\Gamma] \overset{\text{s-fin}}{\rightarrow} [A]\) is the semantics of \( t \) and \( k : [\Gamma; x : A]^k \overset{\text{s-fin}}{\rightarrow} [B] \) 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 \([\Gamma; x : A]\). 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 \( \text{snd}^{[k=1]} \circ \text{fst}^{n-k} \) for access to the \( k \)th variable in a context \( \Gamma \) 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 define two s-finite kernels for both branches. Given a kernel \( k, \text{kiteT} \ k \) defines a family of measures that behaves like \( k \) when the condition is true and like \( mzero \) otherwise:

\[
\text{Definition kiteT} \ (xb : X \times \text{bool}) = \begin{cases} (\text{measure set} \ Y \rightarrow \mu \bar{R}) := & \text{if} \ xb.2 \ \text{then} \ xb.1 \ \text{else} \ mzero. \\
\end{cases}
\]

Similarly, we define \( \text{kiteF} \ k \) for when the condition is false:

\[
\text{Definition kiteF} \ k = \begin{cases} (\text{measure set} \ Y \rightarrow \mu \bar{R}) := & \text{if} \ \neg \ xb.2 \ \text{then} \ xb.1 \ \text{else} \ mzero. \\
\end{cases}
\]

We can then equip \( \text{kiteT} \) and \( \text{kiteF} \) 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 \( \text{ITE} \)].

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

\[
\text{Definition kite} : \ R \rightarrow \text{soker T} \rightarrow \text{T'} := \text{mdirac} \ mf \ \text{;} \ \text{add} \ (\text{kiteT} \ u1) \ \text{(kiteF} \ u2).\]

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 \( P(A) \) (that we saw in Sect. 6.1) is the set of probability measures on \( A \) together with the \( \sigma \)-algebra generated by the sets of probability measures \( \mu \) 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 \( r \):

\[
\text{Definition mset U r} := [\text{set} \ \mu : \text{probability} \ T \ R \ | \ \mu(U) < r \%:E].
\]

We then use \( \text{mset} \) to define the generator of the \( \sigma \)-algebra of probability measures:

\[
\text{Definition pset} : \text{set} \ (\text{set} \ (\text{probability} \ T \ R)) := [\text{set} \ \text{mset} U r \ | \ r \ \text{in} \ [0;1] \ % : E \ \text{in} \ \text{measurable}].
\]

Last, we use the \( \text{salgebraType} \) of MathComp-Analysis to generate the \( \sigma \)-algebra of probability measures:

\[
\text{Definition pprobability} : \text{measurableType} \text{pset.-sigma} := \text{salgebraType} \ pset.
\]

The sampling instruction for a measurable function \( P \) is the kernel defined by \( P \) itself:

\[
\text{Variable P : X \rightarrow pprobability Y R.}
\]

\[
\text{Definition kprobability} \ (\text{mP} : \text{measurable_fun setT P}) : X \rightarrow (\text{measure set} \ Y \rightarrow \text{bar} \ R) := P.
\]

The type of \( P \) makes it clear that sampling may depend on the program execution. To show that \( \text{kprobability} \ mP \) is a kernel, we need to show that \( \text{kprobability} \ mP \ ^{\sim} \ U \) is a measurable function for any measurable set \( U \).

\[
\text{Let measurable_fun_kprobability U : measurable U \rightarrow measurable_fun_setT} \ (\text{kprobability} \ mP \ ^{\sim} \ U).
\]

It suffices to show that the preimage by \( P \ ^{\sim} \ U \) 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 :=

\[
\begin{cases}
  \text{kprobability } \left(\text{measurable_fun_cst } P\right) & \text{if } P \text{ is a constant } \text{probability measure,} \\
  \text{else } \text{the measure } P & \text{otherwise.}
\end{cases}
\]

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

**Definition** bernoulli := measure_add

\[
\left(\text{mscale } p \left(\text{dirac } \text{true}\right)\right)
\]

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 \(\frac{p}{2}\) is the expression sample (bernoulli \(p27\)), provided that \(p27\) is a proof that \(\frac{p}{2} \leq 1\) (the fact that \(\frac{p}{2}\) is a non-negative number is automatically inferred in MathCompAnalysis).

**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]:

\[
\left(\text{score}(t)\right)_{Y,U} \defeq \begin{cases}
  1 & \text{if } U = \{\}\text{ (the full set)} \\
  0 & \text{if } U = \emptyset
\end{cases}
\]

Here, \(Y \in \Gamma\) is a valuation for variables. For a deterministic term \(t\), \(\left[\epsilon\right]_Y\) is defined by induction on the structure of typing rules. For a probabilistic term \(\Gamma \vdash_p t : A\), we have \(\left[\epsilon\right]_{Y,U} \in [0,\infty]\) with \(U \in \Sigma_A\). 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 constraint (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 \to R.
**Definition** mscore t : (measure set unit \to \text{\bar R}) :=

\[
\begin{cases}
  \text{let } p := \text{NngNum } \left(\text{normr}\_\text{ge0 } f t\right) & \text{in} \\
  \text{mscale } p \left(\text{dirac } \text{tt}\right)
\end{cases}
\]

(The expression \text{normr}\_\text{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 \to R.
**Definition** kscore (mf : measurable_fun setT f) : T \to (measure set unit \to \text{\bar R}) :=

\[
\text{mscore } f.
\]

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, \text{Sect. 3.2}]:

**Definition** k (mf : measurable_fun setT f) i t U :=

\[
\begin{cases}
  \text{if } 1\% R : E \leq \text{mscore } f t U < 1.1\% R : E & \text{then} \\
  \text{mscore } f t U & \text{else} \\
  0
\end{cases}
\]

Using \(k\), we can rewrite score as a countable sum of finite kernels:

**Let** sfinite_kscore

\[
\begin{cases}
  \text{exists } k : \left(\text{R.-\text{fker } T \to \text{unit}}\right)^{\text{nat}}, \\
  \forall t, U, \text{measurable } U \to \text{kscore } mf t U = \text{mseries } k \left(x \to 0\right) \emptyset U
\end{cases}
\]

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):

\[
\begin{align*}
1 & : \left(x \to \text{a kernel } \text{R.-\text{sfker } T \to Y} \right) * \\
2 & \text{Definition } \text{normalize } x U := \\
3 & \text{let } \text{evidence } := f x \left[\text{set: } Y\right] \text{ in} \\
4 & \text{if } (\text{evidence } = \emptyset) \land \text{...) then } P U \\
5 & \text{else } f x U \left(\text{fine evidence}\right)^{-1}\% E
\end{align*}
\]

When then measure of the full set is \(0\) or \(\infty\), 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 \text{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 the function \text{normalize} 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 μ when μ set T < +∞. We define s-finite measures on the model of s-finite kernels (Sect. 4.3) with the following predicate:

\begin{definition}
finite_measure d (T : measurableType d) (R : realType) (mu : measurable_fun setT f).
\end{definition}

\begin{lemma}
sfinite_measure (X : measurableType d) (Y : measurableType d) (R : realType) (mu : measurable_fun setT f).
\end{lemma}

\begin{lemma}
sfinite_measure (X : measurableType d) (Y : measurableType d) (R : realType) (mu : measurable_fun setT f).
\end{lemma}

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 × X × 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 × Y × 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:

\begin{example}
let x := return t in k x ↔ k t
\end{example}

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

\begin{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.
\end{lemma}

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 λx.fx · g x:

\begin{lemma}
score_score (f : R → R) (g : R × unit → R)
(mf : measurable_fun setT f)
\end{lemma}
Then we can encode the semantics of the above program


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:

```coq
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 (which 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:

```coq
Definition kstaton_bus : R -> mbool :=
  letin (score m) (score mg) = score (measurable_funM m)
  (measurable_fun_prod2 tt mg)).
```

This way, we recover the fact that the probability that it is the weekend is \( \frac{5}{7} \) 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 \( \sigma \)-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


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