Experimenting with an Intrinsically-typed Probabilistic Programming Language in Coq[⋆]

Ayumu Saito$^{1,2[0000-0002-3908-0496]}$ and Reynald Affeldt$^{2[0000-0002-2327-953X]}$

[1] Department of Mathematical and Computing Science, Tokyo Institute of Technology, Tokyo, Japan
[2] National Institute of Advanced Industrial Science and Technology (AIST), Tokyo, Japan

Abstract. Although the formalization of probabilistic programs already has several applications in the fields of security proofs and artificial intelligence, formal verification experiments are still underway to support the many features of probabilistic programming. We report on the formalization in the Coq proof assistant of a syntax and a denotational semantics for a probabilistic programming language with sampling, scoring, and normalization. We use dependent types in a crucial way since our syntax is intrinsically-typed and since the semantic values are essentially dependent records. Thanks to the features of Coq, we can use notations that hide the details of type inference when writing examples. The resulting formalization is usable to reason about simple probabilistic programs.

1 Introduction

The formalization of probabilistic programs already has several applications in security (e.g., [8]) or artificial intelligence (e.g., [7]). However, the support to formalize all the features of probabilistic programs is still lacking. For example, the formalization of equational reasoning by Heimerdinger and Shan [14] is axiomatized; the study of nested queries and recursion by Zhang and Amin [30] relies on a partially axiomatized formalization of measure theory. Efforts are underway to improve the formal foundations of probabilistic programming languages. For example, Affeldt et al. have been formalizing in the Coq proof assistant *s-finite kernels* (which are essentially families of measures that lend themselves well to composition [23,24]) to represent the semantics of a first-order probabilistic programming language [4]. Hirata et al. have been formalizing quasi-Borel spaces in Isabelle/HOL to handle higher-order features [15,16].

In this paper, we address the problem of the formalization of the syntax and the denotational semantics of a probabilistic programming language, to reason about programs with sampling, scoring, and normalization, in a proof assistant based on dependent type theory. In such programs, semantic values are typically measurable functions or s-finite kernels. However, a mere formalization of s-finite kernels (such as [4]) does not provide a practical mean to reason about programs

[⋆] This is a preprint with an appendix of a paper to be presented at APLAS 2023.

in the absence of syntax. Indeed, criteria that are easily thought of as syntactic (e.g., the fact that a variable is not free in an expression) need to be recast into semantic terms [4, Sect. 7.1.2]. The evaluation of variables needs to be expressed semantically as measurable functions that access the execution environment by indices akin to de Bruijn indices (see [4, Sections 7.1.2 and 7.2.2]). This situation calls for more formalization experiments of syntax and semantics of probabilistic programming languages.

In the following we provide a formal syntax and denotational semantics for SFPPL, a probabilistic programming language based on s-finite kernels. For syntax formalization, we choose *intrinsic typing* by which the typing rules of the language are embedded into the syntax. This guarantees that one can only write well-typed programs but requires a proof assistant based on dependent type theory such as COQ or AGDA. The idea of intrinsic-typing is well-known but has not been applied to a probabilistic programming language as far as we know. Besides syntax, we also use dependent types in a crucial way to represent semantic values of SFPPL, which are either a measurable function (a dependent pair of a function and a proof that it is measurable) or an s-finite kernel. In addition to dependent types, we exploit other features of COQ to provide a concrete syntax by using bidirectional hints [28], canonical structures [13], and custom entries [27]. This provides a generic approach to represent a programming language inside COQ with intrinsic-typing and a user-friendly syntax. Using this approach, we eventually investigate the formalization of reusable lemmas for the verification of probabilistic programs with SFPPL.

Outline We complete our review of related work in Sect. 2. Section 3 is for preliminaries on measure theory and its formalization in COQ. Section 4 is an overview of the syntax, the typing rules, and the semantics of SFPPL. We split the formalization of the syntax of SFPPL by first explaining the idea of *intrinsically-typed concrete syntax* using a toy language in Sect. 5. We then explain the formalization of the syntax of SFPPL in Sect. 6 and its denotational semantics in Sect. 7. We experiment with the resulting framework by verifying simple programs in Sect. 8 and conclude in Sect. 9.

2 Related Work

To the best of our knowledge, our experiment is the first formalization of a probabilistic programming language using an intrinsically-typed syntax.

The formalization of probabilistic programs in proof assistants is a long-standing topic. In seminal work in HOL, Hurd verifies the Miller-Rabin probabilistic primality test [17]. In COQ, Audebaud and Paulin-Mohring verify randomized algorithms [6] but the measure theory they rely on has some limitations (discrete distributions only, etc.). More recent applications have been targeting artificial intelligence. Bagnall and Stewart encode in COQ 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 [7]; this work is limited to discrete distributions. In Lean, Tassarotti et al. represent stochastic procedures using the Giry monad to formalize PAC learnability for decision stumps [26]. These pieces of work do not feature the combined use of sampling and scoring. We already mentioned in Sect. 1 formalization work in Coq partly relying on axiomatization [14,30]. To address this problem, Affeldt et al. formalize s-finite kernels in Coq, allowing for the support of sampling, scoring, and normalization, without being limited to discrete distributions [4]. They apply their formalization to the encoding of the semantics of a probabilistic programming language. One practical limitation is that they use ad hoc Coq notations to represent variables as De Bruijn indices. In Isabelle/HOL, Hirata et al. formalize quasi-Borel spaces to handle sampling and higher-order features [15], to which they recently add scoring [16]. These last pieces of work do not provide an encoding of syntax.

The encoding of intrinsically-typed syntax in proof assistants based on dependent type theory has also attracted much interest. Benton et al. provide an historical account [10, Sect. 1] along with applications in Coq to a simply-typed language and to the polymorphic lambda calculus. Indeed, this technique is often applied to foundational calculi, e.g., system F in Agda [12]. In Coq, Affeldt and Sakaguchi apply it to a subset of the C programming language [5]. Intrinsically-typed syntax allows for a succinct handling of the many integral types of C. While the encoding of well-formed type contexts in C is a source of difficulty, the absence of let-in expressions simplifies the encoding of an intrinsically-typed syntax for C. Poulsen et al. propose to use intrinsically-typed syntax to write in Agda definitional interpreters for imperative languages. They explain how to deal with mutable state and apply this approach to a subset of Java [20]. Besides encoding of semantics, intrinsically-typed syntax also has other applications such as compiler calculation [18]. We are however not aware of related work applying intrinsically-typed syntax to a probabilistic programming language.

3 Preliminaries: Measure Theory in MathComp-Analysis

3.1 Reminder about Measure 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 Σ_X for such a σ-algebra and call *measurable sets* the sets in Σ_X. For example, the standard σ-algebra on $\mathbb{R}$ is the smallest σ-algebra containing the intervals: the Borel sets. A *measurable space* is a set together with the σ-algebra defining the measurable sets. Given two σ-algebras Σ_X and Σ_Y, the product σ-algebra is the smallest σ-algebra generated by $\{A \times B \mid A \in \Sigma_X, B \in \Sigma_Y\}$.

A (non-negative) *measure* is a function $\mu : \Sigma_X \to [0, \infty]$ such that $\mu(\emptyset) = 0$ and $\mu(\bigcup_i A_i) = \sum_i \mu(A_i)$ for pairwise-disjoint measurable sets A_i, where the sum is countable. This property is called σ-additivity. The Dirac measure δ_x is defined by $\delta_x(U) = [x \in U]$ (using the Iverson bracket notation). A *probability measure* on Σ_X is a measure μ such that $\mu(X) = 1$.

If Σ_X and Σ_Y are two σ-algebras, a *measurable function* $f : X \to Y$ is such that, for all measurable subsets $B \in \Sigma_Y$, the inverse image is a measurable subset $f^{-1}(B) \in \Sigma_X$. If Σ_D is a σ-algebra, we can integrate a measurable function $f : D \to [0, \infty]$ w.r.t. a measure μ over D to get an extended real number denoted by $\int_{x \in D} f\,x (\mathbf{d}\,\mu)$.

A *kernel* $X \rightsquigarrow Y$ is a function $k : X \to \Sigma_Y \to [0, \infty]$ such that for all x, $k\,x$ is a measure and for all measurable sets U, $x \mapsto k\,x\,U$ is a measurable function. 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. When for all x, $k\,x\,Y = 1$, we talk about a *probability kernel*.

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. Given a kernel $l : X \rightsquigarrow Y$ and a kernel $k : X \times Y \rightsquigarrow Z$, the composition of the kernel l and of the kernel k is $x, U \mapsto \int_y k\,(x, y)\,U (\mathbf{d}\,l\,x)$.

3.2 Basics of MathComp-Analysis and its Measure Theory

This paper relies on MathComp-Analysis [1], a library for classical analysis[3] in Coq that provides among others a formalization of measure theory including s-finite kernels.

The type `set T` is for sets of objects of type `T`. The set of all the objects of type `T` is denoted by `setT : set T`. The type `\bar R` is the type `R` extended with two infinity elements. It is typically used when `R` is a numeric type. In particular, the numeric type `realType` corresponds to real numbers, so that when the type of `R` is `realType`, `R` corresponds to $\mathbb{R}$ and `\bar R` corresponds to $\overline{\mathbb{R}} = \mathbb{R} \cup \{+\infty, -\infty\}$. The expression `%:R` injects a natural number into $\mathbb{R}$, `%:E` injects a real number into $\overline{\mathbb{R}}$. Non-negative numeric types are noted `{nonneg R}` where `R` is a numeric type. Given `e : {nonneg R}`, `e%:num` is the projection of type `R`. A function returning unconditionally `c` is represented by `cst c`.

σ-algebra's are represented by objects of type `measurableType d` where `d` is a "display parameter" [2, Sect. 3.2.1]. 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`. The display mechanism is useful to disambiguate goals with several σ-algebras [2, Sect. 3.4]. For example, the display of the product of two measurable types with displays `d1` and `d2` is a measurable type with display `(d1, d2).-prod`.

Given `T` of type `measurableType d`, a non-negative measure on `T` is denoted by `{measure set T -> \bar R}`, where `R` has type `realType`. The Dirac measure is denoted by `dirac a` with notation `\d_a`. The type of a `R`-valued probability measure over the measurable type `T` is `probability T R`. We write `measurable_fun D f` for a measurable function `f` with domain `D`. A kernel $f : X \rightsquigarrow Y$ (resp. an s-finite

[3] MathComp-Analysis adds to the constructive logic of Coq functional and propositional extensionality and the axiom of constructive indefinite description [3, Sect. 5].

kernel $f : X \overset{\text{s-fin}}{\rightsquigarrow} Y$) is noted `R.-ker X ~> Y` (resp. `R.-sfker X ~> Y`) (`R` indicates the support type of extended real numbers).

4 Probabilistic Programming Language using s-Finite Kernels

Before entering the details of formalization, we explain the syntax and the semantics of sfPPL, a probabilistic programming language based on s-finite kernels. The syntax corresponds to [25, Sect. 3] [23, Sect. 3.1] [24, Sect. 4.1, 4.3]. The semantics comes from [23, 24]. It is a simplification because we do not formalize a generic notion of sum types.

The main specificity of sfPPL types is a type for probability distributions:

$$\mathbf{A} ::= \mathbf{U} \mid \mathbf{B} \mid \mathbf{R} \mid P(\mathbf{A}) \mid \mathbf{A}_1 \times \mathbf{A}_2$$

The syntax $\mathbf{U}$ is for a type with one element, $\mathbf{B}$ for boolean numbers, $\mathbf{R}$ for real numbers, $P(\mathbf{A})$ for distributions over $\mathbf{A}$, $\mathbf{A}_1 \times \mathbf{A}_2$ for the cartesian product.

The expressions of sfPPL extend the expressions of a first-order functional language with three instructions specific to probabilistic programming languages:

$$\begin{aligned} e ::= {} & \mathtt{tt} \mid b \mid r \mid f(e_1, \ldots, e_n) \mid (e_1, e_2) \mid \pi_1(e) \mid \pi_2(e) \\ & \mathtt{if}\, e\, \mathtt{then}\, e_1\, \mathtt{else}\, e_2 \mid x \mid \mathtt{return}\, e \mid \mathtt{let}\ x := e_1\ \mathtt{in}\ e_2 \mid \\ & \mathtt{sample}(e) \mid \mathtt{score}(e) \mid \mathtt{normalize}(e) \end{aligned}$$

The syntax `tt` is for the element of type $\mathbf{U}$, b for boolean numbers, r for real numbers. All measurable functions (and arithmetic operations) can be introduced as constants with the syntax $f(e_1, \ldots, e_n)$. Pairs are (e_1, e_2), π_1 and π_2 access their projections. If-then-else branching is self-explanatory. Variables are ranged over by x (y, z, etc.). Last we have return, let-in expressions, and the three instructions specific to probabilistic programming languages: sampling (from a probability measure), scoring (to record that a datum was observed as being drawn from a probability distribution, the parameter is the density of the probability distribution), and normalization (of a measure into a probability measure).

Typing judgments distinguish *deterministic* and *probabilistic* expressions. Typing environments (hereafter, contexts) are tuples $(x_1 : \mathbf{A}_1; \ldots; x_n : \mathbf{A}_n)$ ranged over by Γ. The typing judgment is $\Gamma \vdash_{\mathsf{D}} e : \mathbf{A}$ for deterministic expressions and $\Gamma \vdash_{\mathsf{P}} e : \mathbf{A}$ for probabilistic ones. We reproduce here the typing rules for the basic datatypes, constants, products, projections, and variables.

$$\frac{}{\Gamma \vdash_{\mathsf{D}} \mathtt{tt} : \mathbf{U}} \quad \frac{b \in \mathbb{B}}{\Gamma \vdash_{\mathsf{D}} b : \mathbf{B}} \quad \frac{r \in \mathbb{R}}{\Gamma \vdash_{\mathsf{D}} r : \mathbf{R}} \quad \frac{\Gamma \vdash_{\mathsf{D}} e_i : \mathbf{A}_i}{\Gamma \vdash_{\mathsf{D}} f(e_1, \ldots, e_n) : \mathbf{A}} f \text{ is measurable}$$

$$\frac{\Gamma \vdash_{\mathsf{D}} e_1 : \mathbf{A}_1 \quad \Gamma \vdash_{\mathsf{D}} e_2 : \mathbf{A}_2}{\Gamma \vdash_{\mathsf{D}} (e_1, e_2) : \mathbf{A}_1 \times \mathbf{A}_2} \qquad \frac{\Gamma \vdash_{\mathsf{D}} e : \mathbf{A}_1 \times \mathbf{A}_2}{\Gamma \vdash_{\mathsf{D}} \pi_i(e) : \mathbf{A}_i}$$

$$\frac{x \notin \mathrm{dom}(\Gamma')}{\Gamma, x : \mathbf{A}, \Gamma' \vdash_{\mathsf{D}} x : \mathbf{A}} \quad \frac{\Gamma, \Gamma' \vdash_z e : \mathbf{A}_0 \quad x \notin \mathrm{dom}(\Gamma), x \notin \mathrm{dom}(\Gamma')}{\Gamma, x : \mathbf{A}_1, \Gamma' \vdash_z e : \mathbf{A}_0} z \in \{\mathsf{D}, \mathsf{P}\}$$

These typing rules are mostly about deterministic expressions except the weakening rule that also applies to probabilistic expressions.

The typing rules for the other instructions illustrate the interplay between deterministic and probabilistic expressions. For example, `return` turns a deterministic expression into a probabilistic one. Note that we assume that `normalize` returns a default distribution when the normalization constant is 0 or ∞.

$$\frac{\Gamma \vdash_{\mathsf{D}} e : \mathbf{B} \quad \Gamma \vdash_z e_1 : \mathbf{A} \quad \Gamma \vdash_z e_2 : \mathbf{A}}{\Gamma \vdash_z \texttt{if}\, e\, \texttt{then}\, e_1\, \texttt{else}\, e_2 : \mathbf{A}} z \in \{\mathsf{D}, \mathsf{P}\}$$

$$\frac{\Gamma \vdash_{\mathsf{D}} e : \mathbf{A}}{\Gamma \vdash_{\mathsf{P}} \texttt{return}\, e : \mathbf{A}} \quad \frac{\Gamma \vdash_{\mathsf{P}} e_1 : \mathbf{A}_1 \quad \Gamma, x : \mathbf{A}_1 \vdash_{\mathsf{P}} e_2 : \mathbf{A}_2}{\Gamma \vdash_{\mathsf{P}} \texttt{let}\ x := e_1\ \texttt{in}\ e_2 : \mathbf{A}_2}$$

$$\frac{\Gamma \vdash_{\mathsf{D}} e : P(\mathbf{A})}{\Gamma \vdash_{\mathsf{P}} \texttt{sample}(e) : \mathbf{A}} \quad \frac{\Gamma \vdash_{\mathsf{D}} e : \mathbf{R}}{\Gamma \vdash_{\mathsf{P}} \texttt{score}(e) : \mathbf{U}} \quad \frac{\Gamma \vdash_{\mathsf{P}} e : \mathbf{A}}{\Gamma \vdash_{\mathsf{D}} \texttt{normalize}(e) : P(\mathbf{A})}$$

Let us denote the denotational semantics of sfPPL by a function $[\![\cdot]\!]$ that interprets the syntax of types, of contexts, and of typing judgments resp. to measurable spaces, products of measurable spaces, and measurable functions or s-finite kernels. For example, the measurable space corresponding to $\mathbf{R}$ is $[\![\mathbf{R}]\!]$, the set $\mathbb{R}$ 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 $[\![\Gamma]\!] \overset{\text{def}}{=} \prod_{i=1}^{n} [\![\mathbf{A}_i]\!]$. Deterministic expressions $\Gamma \vdash_{\mathsf{D}} e : \mathbf{A}$ are interpreted by measurable functions $[\![\Gamma]\!] \to [\![\mathbf{A}]\!]$ and probabilistic expressions $\Gamma \vdash_{\mathsf{P}} e : \mathbf{A}$ are interpreted by s-finite kernels $[\![\Gamma]\!] \overset{\text{s-fin}}{\rightsquigarrow} [\![\mathbf{A}]\!]$. In particular, the semantics of $[\![\texttt{let}\ x := e_1\ \texttt{in}\ e_2]\!]$ is the composition (see Sect. 3.1) of a kernel $[\![\Gamma]\!] \overset{\text{s-fin}}{\rightsquigarrow} [\![\mathbf{A}_1]\!]$ corresponding to e_1 and a kernel $[\![\Gamma]\!] \times [\![\mathbf{A}_1]\!] \overset{\text{s-fin}}{\rightsquigarrow} [\![\mathbf{A}_2]\!]$ corresponding to e_2; the result is a kernel of type $[\![\Gamma]\!] \overset{\text{s-fin}}{\rightsquigarrow} [\![\mathbf{A}_2]\!]$ [23].

5 Intrinsically-typed Concrete Syntax for a Toy Language

We recall the notion of intrinsically-typed syntax and introduce the notion of *intrinsically-typed concrete syntax*. For this purpose, we use a subset of sfPPL (Sect. 4) where types are ranged over by $\mathbf{A} ::= \mathbf{U} \mid \mathbf{R}$ and expressions are ranged over by $e ::= \texttt{tt} \mid r \mid x \mid e_1 + e_2 \mid \texttt{let}\ x := e_1\ \texttt{in}\ e_2$. The symbol $+$ represents the addition of real numbers. Typing contexts and typing rules are defined as in Sect. 4 except that both $\vdash_{\mathsf{D}}$ and $\vdash_{\mathsf{P}}$ become $\vdash$ because there are no probabilistic expressions; the only difference is a typing rule for addition: $\Gamma \vdash e_1 : \mathbf{R} \wedge \Gamma \vdash e_2 : \mathbf{R} \to \Gamma \vdash e_1 + e_2 : \mathbf{R}$.

We explain an encoding of this toy language using intrinsically-typed syntax in Sect. 5.1. This syntax enforces the property that only well-typed expressions can be encoded but type-checking can only be automated for ground expressions. In Sect. 5.2, we show that we can solve this problem using Coq's canonical structures. In Sect. 5.3, we explain how to to give our toy language a readable concrete syntax.

5.1 Intrinsically-typed Syntax for a Toy Language

We formalize the basic types **U** and **R** as an inductive type and define a context as a list of pairs of a string and of a type:

```
Inductive typ := Unit | Real. Definition ctx := seq (string * typ).
```

With intrinsically-typed syntax, the expressions are defined by an inductive type indexed by a context and a type:

```
1 Inductive exp : ctx -> typ -> Type :=
2 | exp_unit g : exp g Unit
3 | exp_real g : R -> exp g Real
4 | exp_var g t str : t = lookup Unit g str -> exp g t
5 | exp_add g : exp g Real -> exp g Real -> exp g Real
6 | exp_letin g t1 t2 x : exp g t1 -> exp ((x, t1) :: g) t2 -> exp g t2.
```

The constructors `exp_unit` and `exp_real` build basic data structures. The constructor `exp_var` builds an expression `exp g t` where `t` is the type associated with the string `str` in the context, as proved by the equality at line 4. In particular, given a concrete type and a concrete context, this equality can be checked by the Coq's conversion rule using `erefl`. In the constructor `exp_letin`, a new bound variable is introduced and the context is extended; this is the reason why contexts appear as an index of `exp`. We observe here that intrinsically-typed syntax also means that expressions are well-scoped by construction.

We complete the encoding of the intrinsically-typed syntax by setting the context and type parameters of constructors as implicit (using curly brackets):

```
1 Arguments exp_unit {g}.
2 Arguments exp_real {g}.
3 Arguments exp_var {g t}.
4 Arguments exp_add {g} &.
5 Arguments exp_letin {g} & {t1 t2}.
```

The `&` mark at lines 4 and 5 is a bidirectionality hint: it indicates that Coq should first type-check `g` and propagate the information to type-check the remaining arguments [28].

For example, here is the abstract syntax for `let` $x := 1$ `in` `let` $y := 2$ `in` $x + y$:

```
Example letin_add : exp [::] _ :=
  exp_letin "x" (exp_real 1) (exp_letin "y" (exp_real 2)
   (exp_add (exp_var "x" erefl) (exp_var "y" erefl))).
```

As intended, we only need to pass the outermost context (here the empty context `[::]`) for this expression to type-check. Without bidirectionality hints, the above expression would fail to type-check with the following error message:

```
The term "exp_var "x" (erefl (lookup Unit ?g1 "x"))" has type "exp ?g1
(lookup Unit ?g1 "x")" while it is expected to have type "exp ?g1 Real".
```

In other words, type-checking gets stuck on a hole `?g1` corresponding to the context. This can be fixed by inserting an intermediate context, e.g., by replacing the syntax for the variable x at line 3 by `(@exp_var g _ "x" erefl)`[4] where `g` is the context `[:: ("y", Real); ("x", Real)]` but that somehow defeats the purpose of intrinsically-typed syntax. As a side node, we observe that in AGDA, `letin_add` type-checks using a similar encoding without explicit bidirectionality hints (see Appendix A).

The intrinsically-typed syntax above allows for type-checking ground expressions but fails to type-check expressions where string identifiers are parameters, making it difficult to write generic statements about intrinsically-typed terms.

5.2 Canonical Structures for Intrinsically-typed Syntax

We use canonical structures in the manner of Gonthier et al. [13, Sections 2.3 and 6.1] so that one can write easily generic statements about intrinsically-typed expressions. The idea is to provide an alternative way to construct program variables that triggers a search that builds the context along with type inference.

We define "tagged contexts" (`T` is a decidable type with an element `t0`):

```
Let ctx := seq (string * T). Structure tagged_ctx := Tag {untag : ctx}.
```

We define a structure `find`, parameterized by a string, that contains a tagged context and a proof that the string is associated with some datum:

```
Structure find str t := Find {
  ctx_of : tagged_ctx ;
  ctx_prf : t = lookup (untag ctx_of) str}.
```

Then, we define an alternative way to build variables that is parameterized by a `find` structure:

```
Definition exp_var' str {t : typ} (g : find str t) :=
  @exp_var (untag (ctx_of g)) t str (ctx_prf g).
```

The important point is the use of the projection `ctx_of` that will trigger a search for an appropriate `g : find str t`. We still need to tell COQ how to search for instances of `find`. There are two ways to instantiate this structure. The pair `(str, t)` can be the head of the context, in which case the following lemma provides a way to instantiate the second field of `find`:

```
Lemma ctx_prf_head str t g : t = lookup ((str, t) :: g) str.
```

Otherwise, the pair `(str, t)` might be in the tail of the context:

```
Lemma ctx_prf_tail str t g str' t' : str' != str ->
  t = lookup g str -> t = lookup ((str', t') :: g) str.
```

To account for these two situations, we introduce two definitions that unfold to `Tag`, the constructor for tagged contexts:

[4] In COQ, `@` disables implicit arguments.

```
Definition recurse_tag h := Tag h.
Canonical found_tag h := recurse_tag h.
```

We associate the definition `found_tag` with the situation where the sought variable is in the head of the context and `recurse_tag` with the other situation:

```
Canonical found str t g : find str t :=
  @Find str t (found_tag ((str, t) :: g)) (@ctx_prf_head str t g).
Canonical recurse str t str' t' {H : infer (str' != str)}
    (g : find str t) : find str t :=
  @Find str t (recurse_tag ((str', t') :: untag (ctx_of g)))
    (@ctx_prf_tail str t (untag (ctx_of g)) str' t' H (ctx_prf g)).
```

(The identifier `infer` comes from MathComp-Analysis [1] and provides a proof that two strings are different automatically using type classes.) Since `found_tag` is canonical it will be searched for first, in case of success we will have inferred a correct context, otherwise Coq unfolds `found_tag` to reveal `recurse_tag` and tries to look for the variable in the tail of the context, recursively [13, 31].

Using `exp_var'` instead of `exp_var`, we can rewrite the example of the previous section with just the assumption that the string identifiers are different:

```
Example letin_add (x y : string)
    (xy : infer (x != y)) (yx : infer (y != x)) : exp [::] _ :=
  exp_letin x (exp_real 1) (exp_letin y (exp_real 2)
    (exp_add (exp_var' x _) (exp_var' y _))).
```

We can therefore use `exp_var'` instead of `exp_var`; moreover the former can always be rewritten into the latter:

```
Lemma exp_var'E str t (g : find str t) H : exp_var' str g = exp_var str H.
```

5.3 Intrinsically-typed Concrete Syntax with Custom Entries

Custom entries [27] are a feature of Coq to support autonomous grammars of terms. The definition of a grammar for our toy language starts by declaring an identifier for the custom entry: `Declare Custom Entry expr`. Then we introduce a notation (`[...]`) to delimit expressions written with the `expr` grammar and a notation (`{...}`) to delimit Coq terms that appear inside `expr` expressions:

```
Notation "[ e ]" := e (e custom expr at level 5).
Notation "{ x }" := x (in custom expr, x constr).
```

We can then write the grammar $e ::= \texttt{tt} \mid r \mid x \mid e_1 + e_2 \mid \texttt{let}\ x := e_1\ \texttt{in}\ e_2$:

```
1 Notation "x" := x (in custom expr at level 0, x ident).
2 Notation "x ':R'" := (exp_real x) (in custom expr at level 1).
3 Notation "e1 + e2" := (exp_add e1 e2)
4   (in custom expr at level 2, left associativity).
5 Notation "'let' x ':=' e1 'in' e2" := (exp_letin x e1 e2)
6   (in custom expr at level 3, x constr, e1 custom expr at level 2,
7    e2 custom expr at level 3, left associativity).
```

Line 1 is to allow for the use of Coq identifiers inside `expr` expressions. The other lines are for real numbers, additions, and let-in expressions; they all use the constructors of the syntax. As for variables, we have a notation for `exp_var'`:

```
Notation "# x" := (exp_var' x%string _) (in custom expr at level 1).
```

Using these notations, our running example `let` $x := 1$ `in let` $y := 2$ `in` $x + y$ can be written succinctly and more generally:

```
Example letin_add (x y : string)
    (yx : infer (y != x)) (xy : infer (x != y)) : exp [::] _ := [
  let x := {1}:R in let y := {2}:R in #x + #y].
```

6 Intrinsically-typed Probabilistic Programming Language

We formalize an intrinsically-typed concrete syntax for sfPPL (Sect. 4) on the model of the previous section (Sect. 5).

6.1 Intrinsically-typed Expressions

```
Inductive exp : flag -> ctx -> typ -> Type :=
| exp_unit g : exp D g Unit
| exp_bool g : bool -> exp D g Bool
| exp_real g : R -> exp D g Real
| exp_pair g t1 t2 : exp D g t1 -> exp D g t2 -> exp D g (Pair t1 t2)
| exp_proj1 g t1 t2 : exp D g (Pair t1 t2) -> exp D g t1
| exp_proj2 g t1 t2 : exp D g (Pair t1 t2) -> exp D g t2
| exp_var g str t : t = lookup Unit g str -> exp D g t
| exp_bernoulli g (r : {nonneg R}) (r1 : r%:num <= 1) :
    exp D g (Prob Bool)
| exp_poisson g : nat -> exp D g Real -> exp D g Real
| exp_normalize g t : exp P g t -> exp D g (Prob t)
| exp_letin g t1 t2 str : exp P g t1 -> exp P ((str, t1) :: g) t2 ->
    exp P g t2
| exp_sample g t : exp D g (Prob t) -> exp P g t
| exp_score g : exp D g Real -> exp P g Unit
| exp_return g t : exp D g t -> exp P g t
| exp_if z g t : exp D g Bool -> exp z g t -> exp z g t -> exp z g t
| exp_weak z g h t x : exp z (g ++ h) t ->
    x.1 \notin dom (g ++ h) -> exp z (g ++ x :: h) t.
```

Fig. 1. Expressions of sfPPL.

First, the Coq encoding of the syntax of the types of sfPPL is immediate:

```
Inductive typ := Unit | Bool | Real
  | Pair : typ -> typ -> typ | Prob : typ -> typ.
```

To distinguish between deterministic and probabilistic expressions, we use a flag: `Inductive flag := D | P.` It is better to use a flag than a mutually inductive type because we can have only one constructor for typing rules that do not depend on whether an expression is deterministic or probabilistic.

The constructors for basic datatypes (`exp_unit`, `exp_bool`, `exp_real`), pairs (`exp_pair`) and their projections (`exp_proj1`, `exp_proj2`) should read easily (Fig. 1, lines 2–7). The constructors for variables (`exp_var`, line 8) and for let-in expressions (`exp_letin`, line 13) are essentially the same as in Sect. 5.1. The constructors `exp_bernoulli` and `exp_poisson` (lines 9–11) provide two examples of measurable functions that we explain below. The constructors for return, sampling, scoring, and normalizing are as we explained in Sect. 4. The constructors `exp_if` and `exp_weak` accommodate both the deterministic and the probabilistic cases thanks to a flag. The rule `exp_weak` allows to change the type of an expression by inserting a fresh variable at an arbitrary position in the context.

The constructor `exp_bernoulli` represents a Bernoulli distribution that takes as parameters a non-negative real number `r` and a proof that `r` ≤ 1. Since it is a distribution of boolean numbers, the type of the corresponding expression is `exp D g (Prob Bool)`. Informally, the typing rule could be written:

$$\frac{r \in \mathbb{R} \quad 0 \leq r \leq 1}{\Gamma \vdash_{\mathsf{D}} \texttt{exp\_bernoulli}(r) : P(\mathbf{B})}$$

Given a natural number n and an expression e, the constructor `exp_poisson` represents the likelihood of n for a Poisson distribution with rate e:

$$\frac{n \in \mathbb{N} \quad \Gamma \vdash_{\mathsf{D}} e : \mathbf{R}}{\Gamma \vdash_{\mathsf{D}} \texttt{exp\_poisson}(n, e) : \mathbf{R}}$$

6.2 Intrinsically-typed Concrete Syntax for sfPPL

We use custom entries as in Sect. 5.3 to provide a concrete syntax for sfPPL. Instead of reproducing the complete grammar that can be found online [21], we consider the following illustrative program from [24]:

$$\begin{aligned}\texttt{normalize}(\ &\texttt{let}\ x := \texttt{sample}(\mathit{bernoulli}(2/7))\ \texttt{in}\\ &\texttt{let}\ r := \texttt{if}\ x\ \texttt{then}\ 3\ \texttt{else}\ 10\ \texttt{in}\\ &\texttt{let}\ \_ := \texttt{score}(\mathit{poisson}(4, r))\ \texttt{in}\ \texttt{return}\ x\ )\end{aligned}$$

This program is about inferring whether today is the weekend according to the number of buses passing by. It selects a boolean number from a Bernoulli distribution to represent whether today is the weekend. The if-then-else expression models the fact that there are three buses per hour during the weekend and ten buses per hour otherwise. Scoring records the observation that four buses have been passing by in one hour, assuming buses arrive as a Poisson process with rate r. The resulting measure is eventually normalized. As a Coq term:

```
Definition staton_bus_syntax0 : exp _ [::] _ :=
  [let "x" := Sample {exp_bernoulli (2 / 7%:R)%:nng p27} in
   let "r" := if #{"x"} then return {3}:R else return {10}:R in
   let "_" := Score {exp_poisson 4 [#{"r"}]} in return #{"x"}].
Definition staton_bus_syntax := [Normalize {staton_bus_syntax0}].
```

We use the same delimiters to enter and exit the grammar and the same notation for constants as in Sect. 5.3. Other grammar entries should be intuitive. The Coq expression `p27` is a proof that $2/7 \leq 1$.

7 Denotational Semantics of sfPPL

We formalize a denotational semantics for sfPPL that links the syntax of Sect. 6 to previous work [4]. Intuitively, this is the function $[\![\cdot]\!]$ of Sect. 4. Since the denotations are non-trivial objects (measurable functions and s-finite kernels), we formalize an evaluation function and show that it is a function.

7.1 Interpretation of Types and Contexts

We first provide Coq functions to interpret types, their sequences, and contexts to measurable spaces.

We interpret an object `t : typ` with the function `measurable_of_typ` that returns a measurable type (Sect. 3.2) together with its display in the form of a dependent pair `{d & measurableType d}`. The implementation is by recursion on the structure of `t` and uses the product spaces of MathComp-Analysis. The function `mtyp t` takes the second projection (using `projT2`) of `measurable_of_typ t`. We interpret a list `s : seq typ` with the function `measurable_of_seq` that essentially iterates the function `measurable_of_typ` over `s`. More precisely, given a list $[\mathbf{A}_1; \mathbf{A}_2; \cdots; \mathbf{A}_n]$, it returns a measurable space made of nested products $[\![\mathbf{A_1}]\!] \times ([\![\mathbf{A_2}]\!] \times \cdots ([\![\mathbf{A_n}]\!] \times [\![\mathbf{U}]\!]))$; we use $\mathbf{U}$ to avoid empty spaces. The result of `measurable_seq` is a dependent pair of type `{d & measurableType d}`. When applied to a context `g : ctx`, the function `mctx` returns the second projection of `measurable_seq (map snd g)`.

7.2 Evaluation Relation for sfPPL Expressions

The evaluation of sfPPL expressions takes the form of a mutually inductive relation. The relation `evalD` relates an expression `exp D g t` to a measurable function `f` such that the domain of `f` is the interpretation of `g` and the codomain of `f` is the interpretation of `t`, i.e., its type is `dval R g t := @mctx R g -> @mtyp R t`. The type of `evalD` is therefore:

```
forall g t, exp D g t ->
  forall f : dval R g t, measurable_fun setT f -> Prop.
```

The expression `forall f : dval R g t, measurable_fun setT f` is a dependent pair of a function with a measurability proof; hereafter, `evalD e f mf` stands for `e -D> f ; mf`. Similarly, `evalP` relates an expression `exp P g t` to an s-finite kernel of type `pval R g t := R.-sfker @mctx R g ~> @mtyp R t`. The type of `evalP` is therefore `forall g t, exp P g t -> pval R g t -> Prop` and we note `e -P> k` for `evalP e k`. Let us now explain the main constructors of `evalD` and `evalP`.

The constructors `eval_unit`, `eval_bool`, and `eval_real` (Fig. 2, lines 3–5) relates basic data structures to constant functions (`ktt`, `kb`, and `kr` are notations for the proof that constant functions are measurable).

The constructors for pairs and their projections use results from MathComp-Analysis to build measurability proofs for products (`measurable_fun_prod`, line 8) or to compose measurability proofs (`measurableT_comp`, lines 10, 12).

```
1  Inductive evalD : forall g t, exp D g t ->
2    forall f : dval R g t, measurable_fun setT f -> Prop :=
3  | eval_unit g   : ([TT] : exp D g _)  -D> cst tt ; ktt
4  | eval_bool g b : ([b:B] : exp D g _) -D> cst b ; kb b
5  | eval_real g r : ([r:R] : exp D g _) -D> cst r ; kr r
6  | eval_pair g t1 (e1 : exp D g t1) f1 mf1 t2 (e2 : exp D g t2) f2 mf2 :
7      e1 -D> f1 ; mf1  ->  e2 -D> f2 ; mf2 ->
8    [(e1, e2)] -D> fun x => (f1 x, f2 x) ; measurable_fun_prod mf1 mf2
9  | eval_proj1 g t1 t2 (e : exp D g (Pair t1 t2)) f mf : e -D> f ; mf ->
10   [\pi_1 e] -D> fst \o f ; measurableT_comp measurable_fst mf
11 | eval_proj2 g t1 t2 (e : exp D g (Pair t1 t2)) f mf : e -D> f ; mf ->
12   [\pi_2 e] -D> snd \o f ; measurableT_comp measurable_snd mf
13 | eval_var g x H : let i := index x (dom g) in
14   exp_var x H -D> acc_typ (map snd g) i ; measurable_acc_typ (map snd g) i
15 | eval_bernoulli g (r : {nonneg R}) (r1 : r%:num <= 1) :
16   (exp_bernoulli r r1 : exp D g _) -D> cst (bernoulli r1) ;
17                                      measurable_cst _
18 | eval_poisson g n (e : exp D g _) f mf : e -D> f ; mf ->
19   exp_poisson n e -D> poisson n \o f ;
20                     measurableT_comp (measurable_poisson n) mf
21 | eval_normalize g t (e : exp P g t) k : e -P> k ->
22   exp_normalize e -D> normalize_pt k ; measurable_normalize_pt k
23 | evalD_if g t e f mf (e1 : exp D g t) f1 mf1 e2 f2 mf2 :
24     e -D> f ; mf -> e1 -D> f1 ; mf1 -> e2 -D> f2 ; mf2 ->
25   [if e then e1 else e2] -D> fun x => if f x then f1 x else f2 x ;
26                            measurable_fun_ifT mf mf1 mf2
27 | evalD_weak g h t e x (H : x.1 \notin dom (g ++ h)) f mf : e -D> f ; mf ->
28   (exp_weak _ g h x e H : exp _ _ t) -D> weak g h x f ;
29                                        measurable_weak g h x f mf
```

Fig. 2. Evaluation relation for the deterministic expressions of SFPPL. See Fig. 3 for probabilistic expressions.

The constructor `eval_var` (line 13) defines the evaluation of a variable `x` by first finding its index `i` in the context `g` and produces a measurable function. The function `acc_typ` accesses the interpretation of `g` and returns the element corresponding to the `i`th measurable space of `g`:

```
Fixpoint acc_typ (s : seq typ) n : projT2 (@measurable_of_seq R s) ->
  projT2 (measurable_of_typ (nth Unit s n)) := (* See [21] *).
```

Since the interpretation of the context is a nested product, such a function is built out of projections and is therefore measurable (proof `measurable_acc_type`). This generic way to compute measurable functions that access the environment is an improvement over previous work [4] where accesses were performed using ad hoc Coq notations for just a handful of functions.

The constructor `eval_bernoulli` (line 15) yields a constant function that returns a Bernoulli distribution. This particular method of sampling does not depend on the execution but there is no fundamental limitation to extend `evalD` with nested queries, as long as one provides a proof of measurability.

The constructor `eval_poisson` (line 18) produces a function `poisson n \o f` where `n` is the observation recorded for scoring and `f` is a measurable function that evaluates to the rate of the Poisson process. The expression `poisson n` is the measurable function $\lambda r.\, r^{\mathtt{n}} e^{-r}/\mathtt{n}!$.

For an expression `e : exp P g t` corresponding to an s-finite kernel `k`, the constructor `eval_normalize` (line 21) yields a function `normalize_pt k` going from `mctx g` to a probability measure over `mtyp t`. This function works by lifting a function that normalizes measures using a default probability measure when normalization is not possible [21].

The constructor `evalD_weak` (Fig. 2, line 27) produces a function `weak g h x t` of type `dval R (g ++ h) t -> dval R (g ++ x :: h) t`. The probabilistic version `evalP_weak` (Fig. 3, line 15) is similar but of course evaluates to an s-finite kernel.

The constructor `eval_letin` evaluates a let-in expression by combining the s-finite kernels of the two sub-expressions (Fig. 3, line 4). The function `letin'` has type $X \overset{\text{s-fin}}{\rightsquigarrow} Y \to Y \times X \overset{\text{s-fin}}{\rightsquigarrow} Z \to X \overset{\text{s-fin}}{\rightsquigarrow} Z$ so that it keeps the nesting of measurable spaces in the same order as the contexts where new variables are added by list consing. It is defined by composing the composition of kernels (Sections 3.1 and 4, [4, Sect. 5.1]) with a kernel of type $X \times Y \overset{\text{s-fin}}{\rightsquigarrow} Z \to Y \times X \overset{\text{s-fin}}{\rightsquigarrow} Z$ that swaps the projections of a product space.

We briefly explain the last constructors of Fig. 3. The constructor `eval_sample` (line 5) produces a probability kernel given a measurable function of a type compatible with the functions yielded by the constructors `eval_bernoulli` and `eval_normalize`. The constructor `eval_score` (line 8) yields an s-finite kernel of type $(\mathtt{mctx\ g}) \overset{\text{s-fin}}{\rightsquigarrow} \mathbf{U}$ where `g` is the context of the expression passed to the `Score` expression. The constructor `eval_return` (line 10) produces an s-finite kernel `ret mf` where `ret` is the functional $x \mapsto \delta_{\mathtt{f}(x)}$ formalized as `kdirac` in [4, Sect. 4.6].

```
with evalP : forall g t, exp P g t -> pval R g t -> Prop :=
| eval_letin g t1 t2 str (e1 : exp _ g t1) (e2 : exp _ _ t2) k1 k2 :
    e1 -P> k1 -> e2 -P> k2 ->
    [let str := e1 in e2] -P> letin' k1 k2
| eval_sample g t (e : exp _ _ (Prob t))
  (f : mctx g -> pprobability (mtyp t) R) mf :
    e -D> f ; mf -> [Sample e] -P> sample f mf
| eval_score g (e : exp _ g _) f mf :
    e -D> f ; mf -> [Score e] -P> kscore mf
| eval_return g t (e : exp D g t) f mf :
    e -D> f ; mf -> [return e] -P> ret mf
| evalP_if g t e f mf (e1 : exp P g t) k1 e2 k2 :
    e -D> f ; mf -> e1 -P> k1 -> e2 -P> k2 ->
   [if e then e1 else e2] -P> ite mf k1 k2
| evalP_weak g h t (e : exp P (g ++ h) t) x
  (H : x.1 \notin dom (g ++ h)) f :
    e -P> f -> exp_weak _ g h x e H -P> kweak g h x f
```

Fig. 3. Evaluation relation for the probabilistic expressions of sfPPL. See Fig. 2 for deterministic expressions.

7.3 From the Evaluation Relation to a Function

The evaluation relation of the previous section is actually a function because it is right-unique and left-total. Right-uniqueness can be proved by induction on the evaluation relation:

```
Lemma evalD_uniq g t (e : exp D g t) (u v : dval R g t) mu mv :
  e -D> u ; mu -> e -D> v ; mv -> u = v.
Lemma evalP_uniq g t (e : exp P g t) (u v : pval R g t) :
  e -P> u -> e -P> v -> u = v.
```

Left-totality can be proved by induction on the syntax:

```
Lemma eval_total z g t (e : exp z g t) : (match z with
  | D => fun e => exists f mf, e -D> f ; mf
  | P => fun e => exists k, e -P> k end) e.
```

Thanks to these properties, we can produce a pair of functions `execD` and `execP` written using the constructive indefinite description axiom `cid` of Coq:

```
Definition execD g t (e : exp D g t)
    : {f : dval R g t & measurable_fun setT f} :=
  let: exist _ H := cid (evalD_total e) in existT _ _ (projT1 (cid H)).
Definition execP g t (e : exp P g t) : pval R g t :=
  projT1 (cid (evalP_total e)).
```

Finally, we prove equations for `execD`/`execP` that associate to each expression of sfPPL its result according to `evalD`/`evalP`. In general these equations are recursive, e.g., the execution of a return expression:

```
Lemma execP_return g t (e : exp D g t) :
  execP [return e] = ret (projT2 (execD e)).
```

The proofs of these equations are manual but follow an easy pattern which is a direct application of the equivalence between `execD`/`execP` and `evalD`/`evalP` (see lemmas `execD_evalD`/`execP_evalP` in [21]).

8 Using sfPPL to Reason Formally about Programs

Pair of Samplings The following program samples two values from Bernoulli distributions with parameters 1/2 and 1/3 and returns the pair (`p1S` n is a proof that $1/(n+1) \leq 1$):

```
Definition sample_pair_syntax0 : exp _ [::] _ :=
  [let "x" := Sample {exp_bernoulli (1 / 2)%:nng (p1S 1)} in
   let "y" := Sample {exp_bernoulli (1 / 3)%:nng (p1S 2)} in
   return (#{"x"}, #{"y"})].
```

We can verify that the pair `(true, true)` is returned with probability 1/6:

```
Lemma exec_sample_pair0_TandT :
  @execP R [::] _ sample_pair_syntax0 tt [set (true, true)] = (1 / 6)%:E.
```

Since we compute a pair of boolean numbers and since the context is empty, the result of execution has type `R.-sfker unit ~> bool * bool`. This is why we pass `tt` and, as an event, the pair whose projections are `true`. The proof is by rewriting using equations such as `execP_return` (Sect. 7.3) and, once the semantics is revealed, by using generic lemmas from MathComp-Analysis.

Sampling and Scoring The following program samples a value from a Bernoulli distributions with parameter 1/3 and scores the output with 1/3 or 2/3:

```
Definition bernoulli13_score := [Normalize
  let "x" := Sample {@exp_bernoulli R [::] (1 / 3)%:nng (p1S 2)} in
  let "_" := if #{"x"} then Score {(1 / 3)}:R else Score {(2 / 3)}:R in
  return #{"x"}].
```

We can verify that the resulting probability measure is $\frac{1}{3} \times \frac{1}{3} : \frac{2}{3} \times \frac{2}{3} = 1 : 4$, i.e., the Bernoulli distribution with parameter 1/5:

```
Lemma exec_bernoulli13_score :
  execD bernoulli13_score = execD (exp_bernoulli (1 / 5)%:nng (p1S 4)).
```

The proof is essentially by rewriting [21, file `lang_syntax_examples.v`].

Probabilistic Inference We solve the probabilistic inference problem of Sect. 6.2 by computing the measure corresponding to the execution of `staton_bus_syntax0`. This measure corresponds to the probability measure `true` : `false` $= \frac{2}{7} \times \frac{3^4 e^{-3}}{4!}$: $\frac{5}{7} \times \frac{10^4 e^{-10}}{4!} \approx 0.78 : 0.22$. It can be defined in Coq as a sum of Dirac measures:

```
Let staton_bus_probability U := (2 / 7)%:E * (poisson4 3)%:E * \d_true U +
                                (5 / 7)%:E * (poisson4 10)%:E * \d_false U.
```

We state that execution of `staton_bus_syntax0` yields the expected measure:

```
Lemma exec_staton_bus0 (U : set bool) :
  execP staton_bus_syntax0 tt U = staton_bus_probability U.
```

The proof goes through the intermediate step of computing the semantics of `staton_bus_syntax0` and then shows that this semantics is the expected measure.

Program Transformation We verify the equivalence between the program used just above (`staton_bus_syntax0`) and a slightly modified version in which we only change the associativity of let-in expressions:

```
[let "x" := Sample {exp_bernoulli (2 / 7%:R)%:nng p27} in
 let "_" :=
   let "r" := if #{"x"} then return {3}:R else return {10}:R in
   Score {exp_poisson 4 [#{"r"}]} in
 return #{"x"}].
```

This seemingly trivial modification is actually explained by a non-trivial lemma [23, Lemma 3]. The associativity of let-in expressions can be stated as follows [23, Sect. 4.2]: $[\![\mathtt{let\ x := e_1\ in\ let\ y := e_2\ in\ e_3}]\!] = [\![\mathtt{let\ y := (let\ x := e_1\ in\ e_2)\ in\ e_3}]\!]$. To type-check an equivalent formal statement, we need to be careful about the type of $\mathtt{e}_3$. The typing judgment for $\mathtt{e}_3$ on the left-hand side is of the form $\mathtt{y} : \mathbf{A}_2, \mathtt{x} : \mathbf{A}_1, \Gamma \vdash \mathtt{e}_3 : \mathbf{A}_3$ while on the right-hand side it is $\mathtt{y} : \mathbf{A}_2, \Gamma \vdash \mathtt{e}_3 : \mathbf{A}_3$. In our formalization, we use `exp_weak` (Sect. 6.1) to weaken $\mathtt{e}_3$ of type `exp P [:: (y, t2) & g] t3` to the type `exp P [:: (y, t2); (x, t1) & g] t3`:

```
Lemma letinA g x y t1 t2 t3 (xyg : x \notin dom ((y, t2) :: g))
    (e1 : exp P g t1) (e2 : exp P [:: (x, t1) & g] t2)
    (e3 : exp P [:: (y, t2) & g] t3) :
  forall U, measurable U ->
  execP [let x := e1 in let y := e2 in
         {@exp_weak _ _ [:: (y, t2)] _ _ (x, t1) e3 xyg}] ^~ U =
  execP [let y := let x := e1 in e2 in e3] ^~ U.
```

The notation `f ^~ y` is for the function $\lambda x. f\, x\, y$. We can prove `letinA` using lemmas for `execP` (Sect. 7.3) and previous work [4], and use this lemma to prove the equivalence between the two versions of our probabilistic inference problem.

Commutativity We formalize the commutativity example by Staton [24, Sect. 5.1]:

$$[\![\mathtt{let\ x := e_1\ in\ let\ y := e_2\ in\ return\ (x, y)}]\!] = [\![\mathtt{let\ y := e_2\ in\ let\ x := e_1\ in\ return\ (x, y)}]\!]$$

This kind of commutativity properties was the main motivation for the use of s-finite kernels; it relies on a version of Fubini's theorem for s-finite measures [23]. The property above holds under the condition that `x` is not free in $\mathtt{e}_2$ and `y` is not free in $\mathtt{e}_1$. We can specify these conditions by having $\mathtt{e}_1$ and $\mathtt{e}_2$ of types `exp P g t1` and `exp P g t2` with `x` and `y` not appearing in `dom g`. However, as in the associativity of let-in expressions, we need to weaken $\mathtt{e}_1$ and $\mathtt{e}_2$ appropriately:

```
Lemma letinC g t1 t2 (e1 : exp P g t1) (e2 : exp P g t2)
  (x y : string) (xy : infer (x != y)) (yx : infer (y != x))
  (xg : x \notin dom g) (yg : y \notin dom g) :
  forall U, measurable U ->
  execP [let x := e1 in let y := {exp_weak _ [::] _ (x, t1) e2 xg} in
        return (#x, #y)] ^~ U =
  execP [let y := e2 in let x := {exp_weak _ [::] _ (y, t2) e1 yg} in
        return (#x, #y)] ^~ U.
```

The proof of `letinC` relies on previous work [4, Sect. 7.1.2] and its generic statement relies on our use of canonical structures (Sect. 5.2).

9 Conclusions

To the best of our knowledge, we provide the first formalization of a probabilistic programming language with sampling, scoring, and normalization, using an intrinsically-typed syntax. Our work builds on top of an existing formalization of s-finite kernels [4] that we improve (we hinted at some technical improvements in Sect. 7.2) and, more importantly, that we extend with a syntax and a denotational semantics. We proposed a generic approach to encode intrinsically-typed syntax in Coq using bidirectional hints and canonical structures; combined with Coq's custom entries, this allows for the formalization of a user-friendly concrete syntax (Sect. 5). More specifically to probabilistic programming languages, we explained in Sect. 7 how to handle in the semantics nested product spaces (an important construct in probability theory) and measurable functions and s-finite kernels (which present themselves as dependent records). We formalized a denotational semantics in the form of a function derived from an evaluation relation. We showed that our formalization can be used to reason about simple probabilistic programs by rewriting, covering the examples from [4] and more.

Current and Future Work We have recently added to our framework a formalization of iteration as proposed by Staton [23, Sect. 4.2]. It takes the form of an s-finite kernel `iterate` t `from` x `:=` u that calls t from $x = u$, repeats with $x = u'$ if t returns $u' : \mathbf{A}$ or stops if t returns in $\mathbf{B}$; see [21, file `prob_lang.v`] for the code. This makes it possible to extend sfPPL with loops. As a technical improvement, we are considering the use of deep-embedded binders [19] to avoid strings from the Coq standard library in the concrete syntax. It might also be worth testing whether type classes can be used instead of canonical structures to type-check intrinsically-typed syntax [13, Sect. 7] [19, Sect. 5]. Though not specifically designed for that purpose, MathComp-Analysis turned out to be a good match to formalize the denotational semantics of a probabilistic programming language, which raises the question of its application to the formalization of an operational semantics such as in [11]. We are now investigating application of our approach to more verification examples as a step towards the formalization of equational reasoning for probabilistic programs [14, 22].

Acknowledgements The authors would like to thank the members of the Programming Research Group of the Department of Mathematical and Computing Science at the Tokyo Institute of Technology for their input, and to the anonymous reviewers for many comments that substantially improved this paper. The authors acknowledge the support of the JSPS KAKENHI Grant Number 22H00520.

References

1. Affeldt, R., Bertot, Y., Cohen, C., Kerjean, M., Mahboubi, A., Rouhling, D., Roux, P., Sakaguchi, K., Stone, Z., Strub, P.Y., Théry, L.: MathComp-Analysis: Mathematical components compliant analysis library. `https://github.com/math-comp/analysis` (2023), since 2017. Version 0.6.4
2. Affeldt, R., Cohen, C.: Measure construction by extension in dependent type theory with application to integration. J. Autom. Reason. **67**(3), 28:1–28:27 (2023). `https://doi.org/10.1007/s10817-023-09671-5`
3. Affeldt, R., Cohen, C., Rouhling, D.: Formalization techniques for asymptotic reasoning in classical analysis. J. Formaliz. Reason. **11**(1), 43–76 (2018). `https://doi.org/10.6092/issn.1972-5787/8124`
4. Affeldt, R., Cohen, C., Saito, A.: Semantics of probabilistic programs using s-finite kernels in Coq. In: 12th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP 2023) Boston, MA, USA, January 16–17, 2023. pp. 3–16. ACM (2023). `https://doi.org/10.1145/3573105.3575691`
5. Affeldt, R., Sakaguchi, K.: An intrinsic encoding of a subset of C and its application to TLS network packet processing. J. Formaliz. Reason. **7**(1), 63–104 (2014). `https://doi.org/10.6092/issn.1972-5787/4317`
6. Audebaud, P., Paulin-Mohring, C.: Proofs of randomized algorithms in Coq. Sci. Comput. Program. **74**(8), 568–589 (2009). `https://doi.org/10.1016/j.scico.2007.09.002`
7. Bagnall, A., Stewart, G.: Certifying the true error: Machine learning in Coq with verified generalization guarantees. In: 33rd AAAI Conference on Artificial Intelligence, 31st Conference on Innovative Applications of Artificial Intelligence, 9th Symposium on Educational Advances in Artificial Intelligence, Honolulu, Hawaii, USA, January 27–February 1, 2019. pp. 2662–2669. AAAI Press (2019). `https://doi.org/10.1609/aaai.v33i01.33012662`
8. Barthe, G., Grégoire, B., Béguelin, S.Z.: Formal certification of code-based cryptographic proofs. In: 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2009), Savannah, GA, USA, January 21–23, 2009. pp. 90–101. ACM (2009). `https://doi.org/10.1145/1480881.1480894`
9. Barthe, G., Katoen, J.P., Silva, A. (eds.): Foundations of Probabilistic Programming. Cambridge University Press (2020). `https://doi.org/10.1017/9781108770750`
10. Benton, N., Hur, C., Kennedy, A., McBride, C.: Strongly typed term representations in Coq. J. Autom. Reason. **49**(2), 141–159 (2012). `https://doi.org/10.1007/s10817-011-9219-0`
11. Borgström, J., Lago, U.D., Gordon, A.D., Szymczak, M.: 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. pp. 33–46. ACM (2016). `https://doi.org/10.1145/2951913.2951942`

12. Chapman, J., Kireev, R., Nester, C., Wadler, P.: System F in Agda, for fun and profit. In: 13th International Conference on Mathematics of Program Construction (MPC 2019), Porto, Portugal, October 7–9, 2019. Lecture Notes in Computer Science, vol. 11825, pp. 255–297. Springer (2019). `https://doi.org/10.1007/978-3-030-33636-3_10`
13. Gonthier, G., Ziliani, B., Nanevski, A., Dreyer, D.: How to make ad hoc proof automation less ad hoc. J. Funct. Program. **23**(4), 357–401 (2013). `https://doi.org/10.1017/S0956796813000051`
14. Heimerdinger, M., Shan, C.: Verified equational reasoning on a little language of measures. Workshop on Languages for Inference (LAFI 2019), Cascais, Portugal, January 15, 2019 (Jan 2019)
15. Hirata, M., Minamide, Y., Sato, T.: 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, pp. 57–74. Springer (2022). `https://doi.org/10.1007/978-3-030-99461-7_4`
16. Hirata, M., Minamide, Y., Sato, T.: Semantic foundations of higher-order probabilistic programs in Isabelle/HOL. In: 14th International Conference on Interactive Theorem Proving (ITP 2023), July 31–August 4, 2023, Białystok, Poland. LIPIcs, vol. 268, pp. 18:1–18:18. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2023). `https://doi.org/10.4230/LIPIcs.ITP.2023.18`
17. Hurd, J.: Formal verification of probabilistic algorithms. Ph.D. thesis, Computer Laboratory, University of Cambridge (Dec 2001)
18. Pickard, M., Hutton, G.: Calculating dependently-typed compilers (functional pearl). Proc. ACM Program. Lang. **5**(ICFP), 1–27 (2021). `https://doi.org/10.1145/3473587`
19. Pit-Claudel, C., Bourgeat, T.: An experience report on writing usable DSLs in Coq. In: 7th International Workshop on Coq for Programming Languages (CoqPL 2021) (Jan 2021), available at `https://popl21.sigplan.org/details/CoqPL-2021-papers/7/An-experience-report-on-writing-usable-DSLs-in-Coq`
20. Poulsen, C.B., Rouvoet, A., Tolmach, A., Krebbers, R., Visser, E.: Intrinsically-typed definitional interpreters for imperative languages. Proc. ACM Program. Lang. **2**(POPL), 16:1–16:34 (2018). `https://doi.org/10.1145/3158104`
21. Saito, A., Affeldt, R.: Experimenting with an intrinsically-typed probabilistic programming language in Coq. Part of MathComp-Analysis Pull Request `https://github.com/math-comp/analysis/pull/912` “Application of s-finite kernels to program semantics” (2023), formal development accompanying this paper.
22. Shan, C.: Equational reasoning for probabilistic programming. POPL TutorialFest (2018)
23. Staton, S.: 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, pp. 855–879. Springer (2017). `https://doi.org/10.1007/978-3-662-54434-1_32`
24. Staton, S.: Probabilistic Programs as Measures, pp. 43–74 (2020). `https://doi.org/10.1017/9781108770750.003`, chapter in [9]
25. Staton, S., Yang, H., Wood, F.D., Heunen, C., Kammar, O.: Semantics for probabilistic programming: higher-order functions, continuous distributions, and soft constraints. In: 31st Annual ACM/IEEE Symposium on Logic in Computer Science (LICS 2016), New York, NY, USA, July 5–8, 2016. pp. 525–534. ACM (2016). `https://doi.org/10.1145/2933575.2935313`

26. Tassarotti, J., Vajjha, K., Banerjee, A., Tristan, J.: 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. pp. 5–17. ACM (2021). `https://doi.org/10.1145/3437992.3439917`
27. The Coq Development Team: Custom entries. Inria (2019), chapter Syntax extensions and notation scopes of [29], direct link
28. The Coq Development Team: Bidirectionality hints. Inria (2020), chapter Setting properties of a function's arguments of [29], direct link
29. The Coq Development Team: The Coq Proof Assistant Reference Manual. Inria (2023), available at `https://coq.inria.fr/distrib/current/refman/`. Version 8.17.1
30. Zhang, Y., Amin, N.: Reasoning about "reasoning about reasoning": semantics and contextual equivalence for probabilistic programs with nested queries and recursion. Proc. ACM Program. Lang. **6**(POPL), 1–28 (2022). `https://doi.org/10.1145/3498677`
31. Ziliani, B., Sozeau, M.: A comprehensible guide to a new unifier for CIC including universe polymorphism and overloading. J. Funct. Program. **27**, e10 (2017). `https://doi.org/10.1017/S0956796817000028`

A Intrinsically-typed Syntax for a Toy Language in Agda

In this section, we transpose from Coq to Agda the encoding of the syntax of the toy language that we formalized in Sect. 5.1. First, we define the syntax of types using an inductive type:

```
data typ : Set where
  Unit : typ
  Rat : typ
```

Second, we define contexts exactly as we did in Coq:

```
ctx : Set
ctx = Pair String typ
```

Third, we define a `lookup` function that returns the type paired with a string in a given context. This function is made up of a `findIndex` function and a `nth` functions written on the model of the ones found in the SSReflect library of Coq:

```
index' : String → List String → ℕ
index' x [] = 0
index' x (y :: ys) with x ≟ y
... | yes _ = 0
... | no _ = index' x ys + 1

nth : typ -> List typ -> ℕ -> typ
nth x0 [] n = x0
nth x0 (x :: xs) n with n ≟ 0
... | yes _ = x
... | no _ = nth x0 xs (pred n)

lookup' : List ctx -> String -> typ
lookup' g str = nth Unit (map snd g) (index' str (map fst g))
```

Finally, we define the syntax of the toy language as an inductive type where a context and a type appear as indices:

```
data exp : List ctx -> typ -> Set where
  TT : ∀ {g} -> exp g Unit
  Val : ∀ {g} (r : ℚ) -> exp g Rat
  Var : ∀ {g ty} (str : String) -> ty ≡ lookup' g str -> exp g ty
  Add : ∀ {g} -> exp g Rat -> exp g Rat -> exp g Rat
  Letin : ∀ {g ty1 ty2} str -> exp g ty1 ->
          exp ((str , ty1) :: g) ty2 -> exp g ty2
```

We have declared contexts and types for each constructor are declared as implicit arguments. Note that equality is noted ≡ in Agda. Observe also that we are using the type of rational numbers from the Agda standard library instead of real numbers as we did in Coq.

Eventually, we can type-check in Agda the running example of Sect. 5, i.e., `let` $x := 1$ `in let` $y := 2$ `in` $x + y$:

```
letin_add : exp [] Rat
letin_add = Letin "x" (Val 0ℚ) (Letin "y" (Val 1ℚ)
            (Add (Var "x" refl) (Var "y" refl)))
```

This simple example shows that Agda does not require explicit bidirectionality hints to type-check a ground expression contrary to the Coq encoding of Sect. 5.1.