An Intrinsically-typed Probabilistic Programming Language in Coq (Extended Abstract)

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

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

1 Introduction

The formalization of probabilistic programs already has several applications in security (e.g., [5]) or artificial intelligence (e.g., [4]). 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 [9] is axiomatized; the study of nested queries and recursion by Zhang and Amin [19] relies on a partially axiomatized formalization of measure theory. Efforts are nevertheless underway to improve the formal foundations of probabilistic programming languages. For example, Hirata et al. have been formalizing quasi-Borel spaces in Isabelle/HOL to handle higher-order features [10]. Affeldt et al. have been formalizing s-finite kernels in Coq to provide a semantics for a probabilistic programming language [2].

In this presentation, we address the problem of the formalization of the syntax and of the evaluation of a probabilistic programming language. Our target is a language proposed by Staton [15, 16] whose semantics has already been formalized in Coq [2]. This formalization needs to be improved to provide a practical mean to reason about programs. Indeed, in the absence of syntax, syntactic criteria need to be recast into semantic terms. Also, the evaluation of program variables needs to be expressed semantically, i.e., as measurable functions that access the environment by indices akin to de Bruijn indices (see [2, Sections 7.1.2 and 7.2.2] for concrete examples).

In fact, the nature of probabilistic programs makes the formalization of a syntax and its evaluation not obvious. For example, evaluation does not return standard values but, in the case of the probabilistic programming language by Staton, *s-finite kernels*, which are kernels that can be expressed as countable sums of finite kernels (where a kernel is essentially a family of measures). In this work, we rely on MathComp-Analysis [1], a library for analysis in the Coq proof assistant that already provides a formalization of these notions.

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 is well-know [7, Sect. 1] but has not yet been applied to a probabilistic programming language as far as we know. Besides dependent types, we also exploit other features of the Coq proof assistant to provide a concrete syntax by using *custom entries* [18] and type inference by canonical structures. Using this syntax, we formalize an evaluation relation (which we prove to be a function) that also uses dependent types in a crucial way since the result of an evaluation is essentially a dependent record: either a measurable function or a kernel, depending on whether the evaluated expression is deterministic or probabilistic.

In the following, we give an overview of sfPPL, an archetypal probabilistic programming language based on s-finite kernels. Concretely, we provide a syntax and a functional evaluation relation illustrated by a sample program and by basic equational reasoning rules. The formalization is axiom-free and can be found online[1] (see in particular the file `lang_syntax.v`).

2 Intrinsically-typed Probabilistic Programming Language

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

$$\mathbf{A} \quad ::= \quad \mathbf{U} \mid \mathbf{B} \mid \mathbf{R} \mid P(\mathbf{A}) \mid \mathbf{A}_0 \times \mathbf{A}_1$$

The syntax $\mathbf{U}$ is for a type with one element, $\mathbf{B}$ is for the type of boolean numbers, $\mathbf{R}$ is for the type of real numbers. The syntax $P(\mathbf{A})$ is for the type of distributions over $\mathbf{A}$. The cartesian product is denoted by $\mathbf{A}_0 \times \mathbf{A}_1$. The encoding of this syntax is unsurprising:

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

We represent a (typing) context simply by a list:

```
Definition ctx := seq (string * typ).
```

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

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

[1] https://github.com/AyumuSaito/analysis/tree/lang_syntax

The syntax `tt` is the element of type $\mathbf{U}$, b is for boolean numbers, r is for real numbers. The syntax $f(e_1, \cdots, e_n)$ represents measurable functions. (e_1, e_2) is a pair whose projections are accessed with π_1 and π_2. If-then-else is for boolean branching. Variables are ranged over by x, y, etc. Last we have return (from deterministic to probabilistic expressions), sequencing (for probabilistic expressions only), and the three instructions specific to probabilistic programming languages: sampling (from a probability measure), scoring (for likelihood scores), and normalization (of a measure into a probability measure).

sfPPL distinguishes deterministic and probabilistic expressions by means of two typing judgments $\vdash_D$ and $\vdash_P$. For example, sampling is a probabilistic expression whose parameter is a deterministic expression:

$$\frac{\Gamma \vdash_D e : P(\mathbf{A})}{\Gamma \vdash_P \mathtt{sample}(e) : \mathbf{A}}$$

Since we chose an intrinsically-typed syntax, we formalize the expressions of sfPPL where contexts and types appear as indices (we use a flag instead of a mutually inductive type):

```
Inductive flag := D | P.
Inductive exp : flag -> ctx -> typ -> Type := ...
```

(The context indeed needs to be an index since it is modified by let-in expressions.) For example, the constructor `exp_sample` takes an expression representing a distribution over some type `t` and samples from this distribution an element of type `t`:

```
| exp_sample g t : exp D g (Prob t) -> exp P g t
```

To sample from a concrete distribution, we have a constructor for Bernoulli distributions:

```
| exp_bernoulli g (r : {nonneg R}) (r1 : r%:num <= 1) :
    exp D g (Prob Bool)
```

The non-negative real number $r \leq 1$ is the parameter of the Bernoulli distribution (we use the library for real numbers of MathComp-Analysis). This last constructor corresponds to a measurable function f with the following typing rule:

$$\frac{r \in \mathbb{R} \quad 0 \leq r \leq 1}{\Gamma \vdash_D \mathtt{bernoulli}(r) : P(\mathbf{B})}$$

Using `exp`, only well-typed expressions can be written. However, even in the case of concrete programs (i.e., with fully instantiated contexts), the user sometimes need to make intermediate contexts explicit to type-check abstract syntax. To make it easier to write concrete examples, we use Coq custom entries [18] and canonical structures [8] to provide a concrete syntax that hides the details of inference of contexts. As an example, let us consider the following program in pseudo-code from [15] which models the inference of whether today is the week-end based on the observation that four buses passed by in an hour and knowing that there are three buses per hour on week-ends and ten otherwise:

```
normalize(let x := sample(bernoulli(2/7)) in
          let r := if x then 3 else 10 in
          let _ := score(poisson(4, r)) in
          return x)
```

Using `exp`, custom entries, and canonical structures, the user can get along by writing in Coq:

```
Definition staton_bus_syntax0 : @exp R _ [::] _ := [
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}].
```

The `[]` delimiters enter the grammar of custom entries and `{}` allow to go back to Coq expressions; variables are marked by `#` and real number constants by `:R`.

3 Evaluation of Typed Expressions

The evaluation of sfPPL expression links the syntax from the previous section to the semantics provided by previous work [2].

On paper, the denotational semantics of sfPPL can be represented by a overloaded function $[\![\cdot]\!]$ that evaluates 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 measurable space $\mathbb{R}$ of real numbers with its Borel sets. A context $\Gamma = (x_1 : \mathbf{A}_1; \ldots; x_n : \mathbf{A}_n)$ is interpreted by the product space $[\![\Gamma]\!] \stackrel{\text{def}}{=} \prod_{i=1}^{n} [\![\mathbf{A}_i]\!]$. Deterministic expressions $\Gamma \vdash_D e : \mathbf{A}$ are interpreted by measurable functions $[\![\Gamma]\!] \to [\![\mathbf{A}]\!]$ and probabilistic expressions $\Gamma \vdash_P e : \mathbf{A}$ are interpreted by s-finite kernels $[\![\Gamma]\!] \xrightarrow{\text{s-fin}} [\![\mathbf{A}]\!]$. In particular, the semantics of a let-in expression $[\![\mathtt{let}\, x := t \,\mathtt{in}\, u]\!]$ is the composition of a kernel of type $[\![\Gamma]\!] \xrightarrow{\text{s-fin}} [\![\mathbf{A}_0]\!]$ corresponding to t and of a kernel of type $[\![\Gamma]\!] \times [\![\mathbf{A}_0]\!] \xrightarrow{\text{s-fin}} [\![\mathbf{A}_1]\!]$ corresponding to u. The proof of the fact that this results in a kernel of type $[\![\Gamma]\!] \xrightarrow{\text{s-fin}} [\![\mathbf{A}_1]\!]$ is due to Staton [15] and has been formalized using MathComp-Analysis [2].

To interpret types and contexts, we provide two functions. When applied to a `typ`, the recursive function `mtyp` returns the corresponding measurable space, a `measurableType` in MathComp-Analysis. When applied to a context $(x_1 : \mathbf{A}_1; \cdots; x_n : \mathbf{A}_n)$, the function `mctx` returns a measurable space made by the nested products $[\![\mathbf{A_1}]\!] \times (\cdots \times ([\![\mathbf{A_n}]\!] \times [\![\mathbf{U}]\!]))$ (we use unit to avoid empty spaces).

We define the evaluation of expressions as a binary relation between a deterministic expression and a measurable function (`evalD`, notation `-D>`) or a probabilistic expression and a s-finite kernel (`evalP`, notation `-P>`), the two relations being mutually defined. The relation `evalD` relates a deterministic expression of type `exp D g t` with a measurable function `f`. The domain of `f` is the interpretation of the context `g` and its codomain is the interpretation of the type `t`, i.e., its type is `dval R g t`:

```
Definition dval (R : realType) (g : ctx) (t : typ) :=
  @mctx R g -> @mtyp R t.
```

Similarly, the relation `evalP` relates a probabilistic expression `exp P g t` with a s-finite kernel of type:

```
Definition pval (R : realType) (g : ctx) (t : typ) :=
  R.-sfker @mctx R g ~> @mtyp R t.
```

For example, the evaluation of a variable is defined by the constructor `eval_var` below:

```
Inductive evalD : forall g t, exp D g t ->
forall f : dval R g t, measurable_fun setT f -> Prop := ...
| eval_var g str : let i := index str (dom g) in
  [%str] -D> acc_typ (map snd g) i ;
  macc_typ (map snd g) i ...
```

It starts by finding the index `i` of the variable in the context and produces a function `acc_typ` that goes from the interpretation of the context to the `i`th measurable space to access the execution environment. Since the interpretation of the context is a nested product, such a function is made of projections and is therefore measurable (proof `macc_type`).

The evaluation of a let-in expression is a matter of combining the s-finite kernels of the two sub-expressions using the `eval_letin` constructor:

```
(* evalD cont'd *) with evalP :
forall g t, exp P g t -> pval R g t -> Prop := ...
| eval_letin g t1 t2 str
    (e1 : exp P g t1) (k1 : pval R g t1)
    (e2 : exp P ((str, t1) :: g) t2)
    (k2 : pval R ((str, t1) :: g) t2) :
  e1 -P> k1 -> e2 -P> k2 ->
  [let str := e1 in e2] -P> letin' k1 k2 ...
```

The function `letin'` performs composition of s-finite kernels. It is actually the composition of `letin` from [2] and of an s-finite kernel that swaps projections of a product space so that measurable spaces are nested in the same order as the contexts (where variables are added with list consing).

The relation `evalD`/`evalP` is proved to be in fact a function by establishing afterwards that it can be treated as a total mapping thanks to the two properties of right-uniqueness and left-totality. For example, here follows right-uniqueness for the evaluation of probabilistic expressions:

```
Lemma evalP_uniq g t (e : exp P g t)
  (u v : pval R g t) : e -P> u -> e -P> v -> u = v.
```

Thanks to these properties, we define two functions `execD` and `execP`: for any deterministic expression `e`, `execD e` is a measurable function and for any probabilistic expression `e`, `execP e` is an s-finite kernel. As a result, we can prove equations that recover the functional behavior of evaluation. e.g.:

```
Lemma execP_letin g x t1 t2
    (e1 : exp P g t1) (e2 : exp P ((x, t1) :: g) t2) :
  execP [let x := e1 in e2] =
    letin' (execP e1) (execP e2) :> (R.-sfker _ ~> _).
```

In other words, `execP_letin` interprets let-in expressions, similarly, `execP_sample` interprets `sample`, etc. As an application of these equations, we prove by equational reasoning that the program `staton_bus_syntax0` evaluates to the semantic value `staton_bus_probability` [2, Sect. 7.2.2]:

```
Lemma exec_staton_bus0 (U : set bool) :
  execP staton_bus_syntax0 tt U =
  staton_bus_probability U.
Proof.
(* processing the syntax (11 lines) *)
rewrite 3!execP_letin execP_sample execD_bernoulli...
(* processing the semantics (8 lines) *)
rewrite letin'_sample_bernoulli...
Qed.
```

The execution of `staton_bus_syntax0` is an s-finite kernel $[\![\mathbf{U}]\!]^{\text{s-fin}}\ [\![\mathbf{B}]\!]$, application to `tt` returns a measure, whose normalization is the probability distribution `true` : `false` = $\frac{2}{7} \times \frac{3^4 e^{-3}}{4!} : \frac{5}{7} \times \frac{10^4 e^{-10}}{4!}$.

4 Related Work

Several pieces of work have been using intrinsically-typed syntax to formalize programming languages in proof assistants based on dependent type theory. To the best of our knowledge, none of them is a probabilistic programming language, the focus is rather on standard lambda-calculi (e.g., [7]). An intrinsically-typed syntax has been used to formalize in Coq a subset of the C programming language [3]. An important difference is that contexts in C cannot be extended as in let-in expressions so that they need not appear as an index of the inductive type encoding the abstract syntax, making for a simpler encoding of expressions. Poulsen et al. propose to use intrinsically-typed syntax to write in Agda definitional interpreters for imperative languages and apply this approach to a subset of the Java programming language [13]. They explain how to deal with mutable state whereas sfPPL is functional. Intrinsically-typed syntax is also used to calculate compilers [11]. Our use of canonical structures to defer computations to type inference comes from Gonthier et al. [8]; a similar result can be obtained with type classes [12, Sect. 5].

5 Conclusion

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 and with a denotational semantics. This extended abstract briefly explained how we formalized the abstract syntax, a concrete syntax, and the semantics, semantic values coming from previous work [2]. We are now working on applying equational reasoning to more examples as a step towards the formalization of equational reasoning for probabilistic programs as advocated for by Shan [9, 14].

References

[1] Affeldt, R., Bertot, Y., Cohen, C., Kerjean, M., Mahboubi, A., Rouhling, D., Roux, P., Sakaguchi, K., Stone, Z., Strub, P.-Y., and Théry, L. (2023a). MathComp-Analysis: Mathematical components compliant analysis library. https://github.com/math-comp/analysis. Since 2017. Version 0.6.4.

[2] Affeldt, R., Cohen, C., and Saito, A. (2023b). 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*, pages 3–16. ACM.

[3] Affeldt, R. and Sakaguchi, K. (2014). An intrinsic encoding of a subset of C and its application to TLS network packet processing. *J. Formaliz. Reason.*, 7(1):63–104.

[4] Bagnall, A. and Stewart, G. (2019). 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*, pages 2662–2669. AAAI Press.

[5] Barthe, G., Grégoire, B., and Béguelin, S. Z. (2009). 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*, pages 90–101. ACM.

[6] Barthe, G., Katoen, J.-P., and Silva, A., editors (2020). *Foundations of Probabilistic Programming*. Cambridge University Press.

[7] Benton, N., Hur, C., Kennedy, A., and McBride, C. (2012). Strongly typed term representations in Coq. *J. Autom. Reason.*, 49(2):141–159.

[8] Gonthier, G., Ziliani, B., Nanevski, A., and Dreyer, D. (2013). How to make ad hoc proof automation less ad hoc. *J. Funct. Program.*, 23(4):357–401.

[9] Heimerdinger, M. and Shan, C. (2019). Verified equational reasoning on a little language of measures. Workshop on Languages for Inference (LAFI 2019), Cascais, Portugal, January 15, 2019.

[10] Hirata, M., Minamide, Y., and Sato, T. (2022). Program logic for higher-order probabilistic programs in Isabelle/HOL. In *16th International Symposium on Functional and Logic Programming (FLOPS 2022), Kyoto, Japan, May 10–12, 2022*, volume 13215 of *Lecture Notes in Computer Science*, pages 57–74. Springer.

[11] Pickard, M. and Hutton, G. (2021). Calculating dependently-typed compilers (functional pearl). *Proc. ACM Program. Lang.*, 5(ICFP):1–27.

[12] Pit-Claudel, C. and Bourgeat, T. (2021). An experience report on writing usable DSLs in Coq. In *The 7th International Workshop on Coq for Programming Languages (CoqPL 2021)*. Available at https://popl21.sigplan.org/details/CoqPL-2021-papers/7/An-experience-report-on-writing-usable-DSLs-in-Coq.

[13] Poulsen, C. B., Rouvoet, A., Tolmach, A., Krebbers, R., and Visser, E. (2018). Intrinsically-typed definitional interpreters for imperative languages. *Proc. ACM Program. Lang.*, 2(POPL):16:1–16:34.

[14] Shan, C. (2018). Equational reasoning for probabilistic programming. POPL TutorialFest.

[15] Staton, S. (2017). Commutative semantics for probabilistic programming. In *26th European Symposium on Programming (ESOP 2017), Uppsala, Sweden, April 22–29, 2017*, volume 10201 of *Lecture Notes in Computer Science*, pages 855–879. Springer.

[16] Staton, S. (2020). *Probabilistic Programs as Measures*, pages 43–74. Chapter in [6].

[17] The Coq Development Team (2023a). *The Coq Proof Assistant Reference Manual*. Inria. Available at https://coq.inria.fr. Version 8.17.0.

[18] The Coq Development Team (2023b). *Custom entries*. Inria. Chapter Syntax extensions and notation scopes of [17], direct link.

[19] Zhang, Y. and Amin, N. (2022). 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.