λPSI: Exact Inference for Higher-Order Probabilistic Programs

Timon Gehr
ETH Zurich, Switzerland
timon.gehr@inf.ethz.ch

Samuel Steffen
ETH Zurich, Switzerland
samuel.steffen@inf.ethz.ch

Martin Vechev
ETH Zurich, Switzerland
martin.vechev@inf.ethz.ch

Abstract

We present λPSI, the first probabilistic programming language and system that supports higher-order exact inference for probabilistic programs with first-class functions, nested inference and discrete, continuous and mixed random variables. λPSI's solver is based on symbolic reasoning and computes the exact distribution represented by a program.

We show that λPSI is practically effective—it automatically computes exact distributions for a number of interesting applications, from rational agents to information theory, many of which could so far only be handled approximately.

CCS Concepts: • **Mathematics of computing** → **Probabilistic inference problems**; • **Software and its engineering** → *Language features*; • **Computing methodologies** → *Special-purpose algebraic systems.*

Keywords: Probabilistic Programming, Exact, Higher-order

ACM Reference Format:
Timon Gehr, Samuel Steffen, and Martin Vechev. 2020. λPSI: Exact Inference for Higher-Order Probabilistic Programs. In *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI '20), June 15–20, 2020, London, UK*. ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3385412.3386006

1 Introduction

Probabilistic programming systems (PPS) provide inference algorithms that operate on expressive models specified as probabilistic programs. Such programs are formed using standard language primitives from deterministic languages, as well as constructs for drawing random values and constructs for conditioning. The key benefit of PPS is that they typically decouple the task of specifying the (generative) model from the task of constructing inference algorithms.

The Importance of Exact Inference Because exact probabilistic inference is generally intractable and does not scale to complex models and large datasets, most inference algorithms implemented in existing PPS are approximate. However, exact inference is important for several reasons. First, it can often outperform approximate inference for smaller models, ones that otherwise have substantial structure, or on queries with low-probability evidence. Second, it naturally supports symbolic parameters, meaning it solves a possibly infinite number of inference problems at once. Third, exact inference guarantees that no precision is lost. Finally, better support for exact inference in existing PPS will enable more fruitful combinations with approximate methods.

Higher-Order PPS with Exact Inference Unfortunately, recent PPS that support exact symbolic reasoning in the presence of continuous distributions (Hakaru [14] and PSI [6]) lag behind in terms of the language features they provide. For example, PSI does not support first-class functions. While Hakaru's implementation supports first-class functions (including distributions over functions), its exact inference operators, namely normalization and disintegration (with respect to the Lebesgue measure) are external programs that manipulate Hakaru terms and are not available as first-class operators within these terms. In contrast, there are *higher-order* PPS which do support first-class inference (e.g., Church [22], WebPPL [8] and Anglican [4]), however, their exact inference algorithms only handle discrete distributions. A key challenge then is to provide support for both first-class inference *and* the ability to compute the exact posterior over discrete, continuous and mixed variables.

Our Work In this work we present λPSI, the first PPS which addresses the above challenge.

First, we introduce the statically typed higher-order probabilistic programming language (PPL) λPSI, which is based on the PSI PPL [6] but with additional support for tuples, arrays, higher-order functions, and nested inference. As demonstrated in PPLs such as Church [7], WebPPL [8], Anglican [25] and Venture [13], higher-order constructs are useful for specifying models where inference queries are nested within other inference queries. This enables, for instance, an inference to be made about agents that themselves make

PLDI '20, June 15–20, 2020, London, UK

ACM ISBN 978-1-4503-7613-6/20/06…$15.00
https://doi.org/10.1145/3385412.3386006

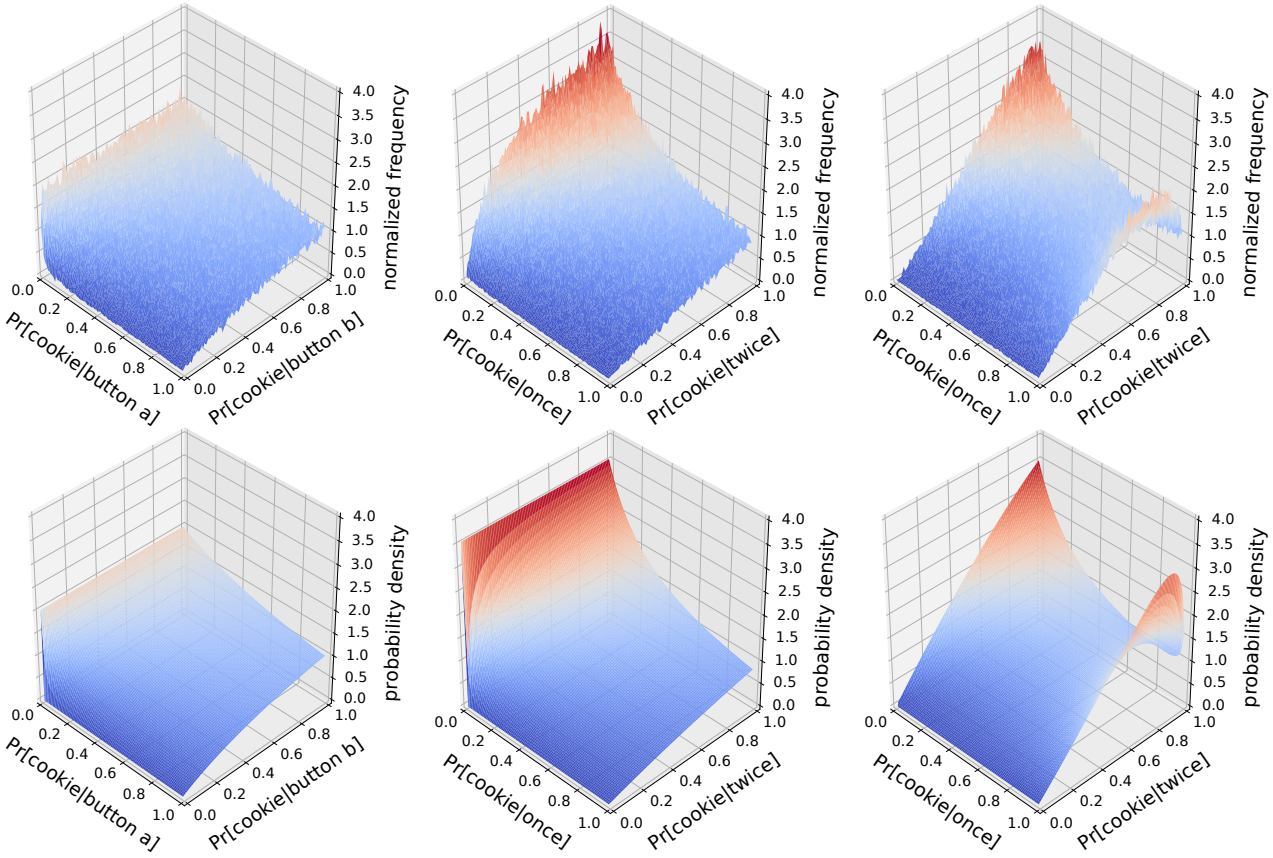

Figure 1. Inference on "Epistemic States": approximate inference in Webchurch (top) vs. exact inference in λPSI (bottom).

use of models and (incomplete) data so to infer knowledge about the state of the world. Unlike other higher-order PPLs (see above), which are dynamically typed, static typing enables easier debugging, better error messages, and avoids expensive dynamic checks during inference.

Second, we introduce an exact inference solver to handle these language features while supporting mixed, discrete, and continuous variables. λPSI's engine further explicitly computes the probability of error, while existing PPS crash stochastically at run time (e.g., randomly indexing an array may or may not cause an out-of-range error during sampling-based inference). We believe λPSI is the first to support exact inference for higher-order probabilistic programs with this level of expressiveness.

Finally, we show λPSI is powerful enough to specify a number of interesting problems ranging from information theory to rational agents, and that its solver can compute, for the first time, the exact posterior for many applications that so far could only be handled approximately.

Main Contributions Our key contributions are:

- The λPSI statically typed higher-order PPL which supports higher-order functions and nested inference (§3).
- The λPSI solver which performs exact symbolic inference and computes the posterior distribution over discrete, continuous, and mixed random variables (§4–§6).
- An extensive evaluation of higher-order exact inference with λPSI across various applications (§7).

2 Motivation and Overview

We now provide a motivating example for nested inference, followed by an overview of λPSI.

Nested Inference Example To reason about rational agent behavior, we can build probabilistic models where multiple rational agents interact and each has a model about how the other agents model the interaction. Such models can be easily built in languages where probabilistic inference is a first-class expression which is allowed to occur inside another probabilistic inference query. For instance, Goodman and Tenenbaum [9] describe a number of (Church/WebPPL) models of this kind in Chapter 15 ("Social cognition"). In order to evaluate our exact inference approach, we have specified all of those models in λPSI (see also §7).

To illustrate the results of exact inference, let's consider some examples of section "Epistemic States" of that chapter.

These examples model an observer of a rational agent operating a vending machine that probabilistically yields either a cookie or a bagel, depending on which one of two buttons a or b is pressed. In the first model (Fig. 1, left), the agent is observed to press button b. The observer assumes a uniform prior over the agent's actions and knows that the agent's goal is to obtain a cookie. The result is the posterior on the probability that the machine yields a cookie when pressing a given button. In the second and third model (Fig. 1, middle and right), the vending machine has only one button a, which may be pressed multiple times. The prior belief over the agent's actions is biased towards pressing the button fewer times, and the agent is observed to press button a twice. While the observer knows that the agent's goal is to obtain a cookie in the second model, the goal is unknown in the third model. Such models are interesting because they involve a mixture of continuous and discrete distributions as well as nested inference queries. We show the results comparing approximate vs. exact joint posteriors for these examples in Fig. 1. The plots in the top row are normalized histograms of 10^6 samples each with a resolution of 100×100, computed by the Church implementation "Webchurch". The bottom row shows plots of the exact posteriors computed by λPSI. We note that our engine evaluates all posteriors within a few seconds, while random sampling takes up to 10 minutes. To the best of our knowledge, this is the first time that those posterior distributions have been evaluated to this precision. We discuss other interesting applications in §7.

λPSI Language and Inference The λPSI program in Fig. 2 illustrates some of λPSI's core language features. Fig. 2 also visualizes the exact inference result computed by λPSI.

First, the program creates a tuple `a` of two random real numbers. One of them is drawn from a continuous uniform distribution, whereas the other is drawn from a discrete uniform distribution. In addition to tuples, λPSI supports arrays of both fixed and random length.

Next, variable `x` is initialized to a random entry of the tuple. The subsequent assignment stores the result of a nested inference query in the variable `p` of type `Distribution[ℝ]`. The `infer` expression accepts an (anonymous) function representing the query, which uses a uniform prior for the variable `y`. This variable is conditioned on the observed evidence `y <= x` to produce the nested posterior. Note that the function we pass to `infer` is itself random, as it depends on the external random variable `x`. `infer` itself is a deterministic function without side effects (in particular, the nested inference query does not influence our knowledge of `x`), but because the input is random, the returned distribution `p` is also random.

Finally, we return the expectation of `p` and a value drawn from `p`, instructing λPSI to compute a joint probability distribution for those two values. As `p` is random, so is its expectation. Therefore, the program produces a joint distribution of two dependent real random variables.

```
1  def main(){
2   a := (uniform(0,2),
3     uniformInt(1,3)/3);
4   x := a[flip(1/2)];
5   p := infer((){
6     y := uniform(0,1);
7     observe(y <= x);
8     return y;
9   });
10  return (expectation(p),
11          sample(p));
12 }
```

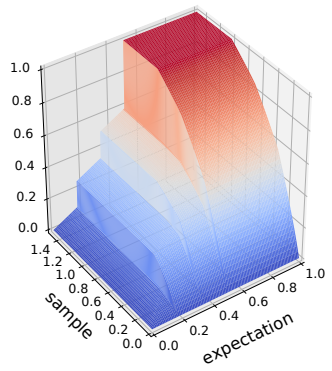

Figure 2. A λPSI program (left) and its exact joint probability distribution (right) computed by λPSI.

The system computes this distribution by combining symbolic expressions for subprograms and then simplifying them. For example, the joint distribution of variables `a` and `x` is represented by the following symbolic expression in λPSI:

$$\int \mathrm{d}v \int \mathrm{d}z \; \overbrace{\tfrac{1}{2}[0 \le v] \cdot [v \le 2] \cdot \lambda[\![v]\!]}^{\text{(a)}} \cdot \overbrace{\left(\tfrac{1}{3}\textstyle\sum_{k=1}^{3} \delta(k/3)[\![z]\!]\right)}^{\text{(b)}} \cdot \underbrace{\delta((v,z))[\![a]\!]}_{\text{(c)}} \cdot \underbrace{\tfrac{1}{2}\left(\delta(a_0)[\![x]\!] + \delta(a_1)[\![x]\!]\right)}_{\text{(d)}}.$$

The expression uses integrals to marginalize the temporary values v and z of the first, resp. second entry of `a`. Part (a) represents the uniform distribution on the interval $[0, 2]$. Here, $\lambda[\![v]\!]$ represents the Lebesgue measure, which is a continuous measure with density 1 at each real number. Part (b) uses Dirac deltas of the form $\delta(e)[\![z]\!]$, which can be interpreted as point-mass distributions on e for z, to represent the discrete uniform distribution on $\{\frac{1}{3}, \frac{2}{3}, 1\}$. Part (c) assigns the tuple (v, z) to a, and part (d) assigns the first or second entry of a to x, each with probability $\frac{1}{2}$. At this point, it is sufficient for the reader to understand the basic ideas. In §4, we provide all details required to understand this expression in depth.

The above is an example of an intermediate result computed during the symbolic analysis λPSI performs. A plot of the cumulative distribution function of the final result computed by λPSI is shown in Fig. 2. We provide the full symbolic expression for the final result in App. A.1.

3 The λPSI PPL

We next describe the higher-order probabilistic programming language λPSI, which extends the PSI language [6] by (i) higher-order functions, (ii) a first-class probabilistic inference operator, (iii) conditioning on probability-zero events, and (iv) a dependent static type system. Fig. 3 presents a simplified core syntax of λPSI, which is sufficiently expressive to highlight the key insights of this work. We provide details about the full language in App. A.2.

```
Func ::= def x (x: Type, ..., x: Type) Body
Body ::= { Stmt* } | ⇒ Ex;
Ex   ::= n | x | BuiltIn | uop Ex | Ex bop Ex |
         (Ex, ..., Ex) | Ex[Ex] |
         (x: Type, ..., x: Type) Body | Ex(Ex)

Stmt ::= x := Ex; | Ex = Ex; | return Ex; |
         observe(Ex); | cobserve(Ex,Ex); |
         if Ex { Stmt* } else { Stmt* } | for x in [n..n){ Stmt* }
Type ::= Type × ··· × Type | Type → Type | Distribution[Type] |
         ℕ | ℤ | ℝ | ···

BuiltIn ∈ {Flip, Gauss, Uniform, UniformInt, Categorical, Exponential, ...} ∪ {infer, sample, expectation} ∪ {exp, log, ...}
```

Figure 3. Core syntax of λPSI (n, x, *uop*, and *bop* denote constants, variables, unary, and binary operations, respectively).

Types Our language features a static type system. In addition to standard numeral, tuple, and function types, λPSI supports dedicated distribution types. For example, `Distribution[ℝ]` describes distributions over a real variable.

Programs and Functions A λPSI program consists of a sequence of function declarations, whose bodies can be either single expressions (e.g., the function `def succ(x:ℝ) ⇒ x+1;`) or sequences of imperative statements (e.g., the function `def succ(x:ℝ) { y:=x; y=y+1; return y; }`). The `main` function, which may take parameters, forms the entry-point of a λPSI program.

Expressions Our language supports standard unary and binary operations on boolean and numeric types. Also, it supports tuples with the usual syntax. For example, the expression `(3,4)` is a two-element tuple, whose first entry `3` can be accessed by `(3,4)[0]`.

λPSI supports multiple built-in expressions. These include constructors for built-in distributions, such as `Flip` and `Gauss`, whose lower-case variants (i.e., `flip` and `gauss`) draw a sample from the respective distribution. For example, `Flip(1/2)` is the uniform distribution on $\{0, 1\}$ (whereas `flip(1/2)` is a sample), and `Gauss(0,1)` is the standard normal distribution parameterized by mean and variance. We can sample from an expression `d` representing a distribution via the expression `sample(d)`. For example, `flip(1/2)` is equivalent to `sample(Flip(1/2))`. Similarly, `expectation(d)` computes the expected value of a random variable drawn from `d`.

Finally, λPSI supports lambda expressions denoting anonymous functions, such as `(x:ℝ){ y:=x; y=y+1; return y; }`. The syntax for function application is standard (e.g., `succ(1)`).

Statements Our language distinguishes variable declarations (e.g., `x:=3`) and assignments (e.g., `x=3`). The `observe` statement conditions the random program state on (positive-probability) observed evidence: all program states that do not satisfy the condition are discarded. The result of inference on the final program is given by renormalizing the resulting subprobability distribution at program exit. For example, the statement `observe(x>=2);` conditions the distribution on program states on the observed fact that `x` is at least `2`.

The `cobserve` ("continuous observation") statement is used to condition on a possible, but probability-zero event. In particular, `cobserve(x,y);` conditions on the probability-zero event that `x` is equal to `y`. We note that conditioning on such events is a delicate matter and hence requires its own statement in the language. See [19] for an in-depth discussion of the involved issues.

Finally, λPSI supports standard `if` statements, and `for` loops with statically-known bounds.

First-Class Inference The built-in `infer` function enables us to perform nested inference. It reifies a function `f` with return type a to a `Distribution[`a`]`, for any type a. If `f` does not execute `observe` statements, `infer` can be thought of as the inverse of sample. This is because `infer(() ⇒ sample(d))` returns the distribution `d` and `() ⇒ sample(infer(f))` yields the function `f`. Otherwise, `infer` is more interesting (see Fig. 2): it forms a context within which the observations evaluated by `f` (see Lin. 7) take effect and returns the normalized posterior of `f` given that evidence and all state outside the context. The computation of `infer` has no side effects, meaning that the observations do not affect the knowledge outside the query (e.g., about `x`).

Further Features The presented syntax is heavily simplified. The full λPSI language extends Fig. 3 by convenient features including arrays, dependent types, and polymorphic functions (see App. A.2).

4 A Symbolic Domain for Distributions

We now introduce a symbolic domain for probability distributions. When performing exact inference, λPSI simplifies representations of distributions in this domain. In §5, we will see how any λPSI program is translated to this domain.

4.1 Representation

λPSI's symbolic domain for probability distributions is shown in Fig. 4. This grammar extends the symbolic domain used in PSI [6] by representations of data structures (highlighted in second line of Fig. 4) as well as higher-order functions and distributions (highlighted in third line).

Basic Arithmetic Expressions Basic expressions (first line in Fig. 4) are inherited from PSI, including variables (x), rational constants (q), the irrational constants e and π, as well as standard arithmetic expressions including the floor ($\lfloor e \rfloor$) and ceiling ($\lceil e \rceil$) operators. The Iverson bracket $[P]$ is an indicator for the proposition P with the usual convention [12]. Like in PSI, we write divisions a/b as $a \cdot b^{-1}$.

$$e ::= \; x \mid q \mid \mathrm{e} \mid \pi \mid -e \mid e_1 + \ldots + e_n \mid e_1 \cdot \ldots \cdot e_n \mid e_1^{e_2} \mid \log(e) \mid \lfloor e \rfloor \mid \lceil e \rceil \mid [e_1 = e_2] \mid [e_1 \le e_2] \mid [e_1 \neq e_2] \mid [e_1 < e_2] \mid$$
$$(e_1, \ldots, e_k) \mid [x \mapsto e[x]](e) \mid e_1 e_2 \mid e_1 e_{2 \mapsto 3} \mid \{f_1 \mapsto e_1, \ldots, f_k \mapsto e_k\} \mid e.f \mid e_1\{f \mapsto e_2\} \mid$$
$$\textstyle\sum_{x \in \mathbb{Z}} e[x] \mid \int \mathrm{d}x\, e[x] \mid (\mathrm{d}/\mathrm{d}x)^{-1}[\mathrm{e}^{-x^2}] \mid \lambda x.\, e[x] \mid e_1(e_2) \mid \mathbb{\Lambda} x.e[x] \mid e[\![x]\!] \mid \delta(e)[\![x]\!] \mid \lambda[\![x]\!] \mid e_1 /\!/ e_2 \mid \bot \mid e_{1?}(e_2)$$

Figure 4. Symbolic domain for expressing probability distributions. We write $e[x]$ to denote that x is a free variable in e. The highlighted elements are fundamental to λPSI and new compared to PSI [6].

Data Structures Our symbolic domain can directly represent data structures of λPSI's programming language (second line in Fig. 4). In particular, it supports tuples, arrays, and records (the latter are used for program states, see §5). For example, $(1, 2)$ is a tuple and $\{\mathtt{f} \mapsto 1\}$ is a record with one field $\mathtt{f}$, which has value 1. We represent arrays as mappings from indices to values together with their lengths. For example, the identity permutation of length 5 is represented as $[x \mapsto x](5)$. The i-th value in an array or tuple a is denoted by a_i. The expression $a.f$ is the value of field f in record a. The expression $a_{i \mapsto b}$ (resp. $a\{\mathtt{f} \mapsto b\}$) represents a modification of a tuple or array (resp. record) a where the i-th value (resp. the value of field $\mathtt{f}$) is replaced by b.

Distributions and Higher-Order Functions The third line in Fig. 4 shows the most interesting expressions, which are particular to representing probability distributions and first class functions. The domain contains sums over $\mathbb{Z}$ (we write $e[x]$ to denote that x is a free variable in e) as well as integrals, which will be discussed in detail in §4.2. The domain of an integral is implicitly defined by the variable x being integrated over. The expression $(\mathrm{d}/\mathrm{d}x)^{-1}[\mathrm{e}^{-x^2}]$ denotes the antiderivative of the function e^{-x^2}, which does not have a closed-form solution but is useful to for example express the cumulative distribution function of a normal distribution.

To support higher-order functions and nested inference, our domain contains lambdas in two flavours: functions ($\lambda x.e[x]$) and distributions ($\mathbb{\Lambda} x.e[x]$). For example, $\lambda x.f(x)$ is the same as the function f, while $\mathbb{\Lambda} x.p[\![x]\!]$ is the same as the distribution p. Note that unlike for function application $e_1(e_2)$, the argument x for distribution application $e[\![x]\!]$ must be a variable. We discuss distributions in more detail in §4.2.

As we will exemplify in §4.3, all distributions in λPSI are built from two primitive distributions. The *Dirac delta* $\delta(e)[\![x]\!]$ expresses that variable x is distributed according to the point-mass distribution on e, where x cannot occur freely in e. For example, $\delta(0)[\![x]\!]$ is the point-mass distribution on 0. Dirac deltas are used to express discrete distributions. The *Lebesgue measure* $\lambda[\![x]\!]$ is used to construct a continuous distribution from a probability density function (discussed in §4.3). The operator $/\!/$ denotes *disintegration*, which we will discuss in more detail in §6.5. It can be loosely thought of as a special kind of division allowing us to eliminate Lebesgue measures by the equivalence $\lambda[\![x]\!] /\!/ \lambda[\![x]\!] = 1$.

Errors The expression $\bot$ denotes a special error value, and $\delta(\bot)[\![x]\!]$ is used to capture the probability of an error. We use $e_{1?}(e_2)$ to propagate errors upon composition, i.e. $e_{1?}(e_2)$ is equal to $e_1(e_2)$ for $e_2 \neq \bot$, while $e_{1?}(\bot)$ reduces to $\delta(\bot)$.

4.2 Interpretation

Before continuing our discussion, we provide an interpretation of the more advanced symbolic expressions.

Distributions A key concept of λPSI are *distributions*, which can be loosely thought of as unnormalized probability densities. Formally, a distribution f over a λPSI type τ (e.g., $\mathbb{R}$) is a bounded measure on τ (it is not necessarily normalized) and we write $D[\tau]$ to denote the set of all distributions over τ. We write $\mathbb{\Lambda} x.\ f[\![x]\!]$ to clarify that f is a distribution for the variable x. Consider the expression $\lambda a.\ \mathbb{\Lambda} b.\ f(a)[\![b]\!]$, which takes a as input and returns a probability distribution for b. Here, f can be thought of as taking a as input and returning a probabilistic value for b.

Integrals A distribution $\mathbb{\Lambda} x.f[\![x]\!]$ over τ can be formally interpreted as a random variable: for any $S \subseteq \tau$, it is

$$\Pr[f \in S] \propto f(S) = \textstyle\int_\tau \mathbf{1}_S \,\mathrm{d}f$$

where the integral is the Lebesgue integral (recall that f is a measure). The probability is proportional due to the missing normalization.

λPSI's symbolic domain (see Fig. 4) uses a convenient (non-standard) notation for integrals: for any types τ, τ', distribution $f \in D[\tau]$, and function $g \in \tau \to \tau'$, it is

$$\textstyle\int \mathrm{d}x\; g(x) f[\![x]\!] \quad \text{defined as} \quad \int_\tau g \,\mathrm{d}f.$$

The domain of the integral is determined by the type τ of x. Note how the Riemann-style notation ($\mathrm{d}x$) makes dependencies explicit: The "output" of f is used as an input to g. Notwithstanding the above definition, it often suffices to think about integrals in the common Riemann sense.

Integrating Higher-Order Distributions Our notation allows conveniently expressing integrals involving higher-order distributions. For $f \in D[D[\tau]]$ being a distribution over distributions over some type τ, we can e.g. write:

$$\mathbb{\Lambda} r.\ h[\![r]\!] = \mathbb{\Lambda} r. \textstyle\int \mathrm{d}x\; x[\![r]\!] \cdot f[\![x]\!]\,.$$

Here, the integration variable x and the result h are single-order distributions. The interpretation is that for any $S \subseteq \tau$: $\Pr[h \in S] \propto \int \mathrm{d}x\ x(S) \cdot f[\![x]\!]$.

Dirac Delta For any value v, the Dirac delta $\delta(v)$ is a measure capturing the point mass on v. Formally, for any type τ, $\delta\colon \tau \to D[\tau]$ is defined as:

$$\forall v \in \tau,\ S \subseteq \tau.\quad \delta(v)(S) = [v \in S].$$

The expression $\lambda x.\ \delta(v)[\![x]\!]$ denotes that x is distributed according to a point mass on v. We can loosely think of $\delta(v)[\![x]\!]$ being 0 for all $x \neq v$ and ∞ for $x = v$. The Dirac delta is normalized: $\int \mathrm{d}x\ \delta(v)[\![x]\!] = 1$.

4.3 Examples

The Dirac delta is used to represent discrete probability distributions. For example, the Bernoulli distribution with success probability $\frac{1}{3}$ (i.e., `flip(1/3)`) can be written as:

$$\lambda x.\ \mathrm{Bernoulli}(\tfrac{1}{3})[\![x]\!] := \lambda x.\ \tfrac{2}{3}\delta(0)[\![x]\!] + \tfrac{1}{3}\delta(1)[\![x]\!]\,.$$

Note that as expected, the probability of value 1 is

$$\int \mathrm{d}x\ [x = 1] \cdot \mathrm{Bernoulli}(\tfrac{1}{3})[\![x]\!] = \tfrac{1}{3}.$$

Discrete distributions with infinite support can be represented using expressions of the form $\sum_{x\in\mathbb{Z}} e[x]$. For instance, the geometric distribution with success probability $\frac{1}{4}$ (i.e., `geometric(1/4)`) can be written as:

$$\lambda x.\ \textstyle\sum_{i\in\mathbb{Z}} [i \geq 0] \cdot \left(\frac{3}{4}\right)^i \cdot \frac{1}{4} \cdot \delta(i)[\![x]\!]\,. \tag{1}$$

Intuitively, the Lebesgue measure $\lambda[\![x]\!]$ assigns uniform weight to all values. It can be used to define continuous distributions: the expression $\lambda x.\ p(x) \cdot \lambda[\![x]\!]$ denotes the distribution of a continuous random variable with probability density function p. For example, the exponential distribution with rate 2 (i.e., `exponential(2)`) can be written as:

$$\lambda x.\ [0 \leq x] \cdot 2\mathrm{e}^{-2x} \cdot \lambda[\![x]\!]\,.$$

A key property of λPSI's symbolic domain is the fact that it can represent distributions which are only partially continuous. For example, the uniform distribution over an interval $[a, b]$ (i.e., `uniform(a,b)`) is represented by:

$$\lambda a, b.\ \lambda x.\ [a < b] \cdot \tfrac{1}{b-a} \cdot [a \leq x] \cdot [x \leq b] \cdot \lambda[\![x]\!] \tag{2}$$
$$+\ [a = b] \cdot \delta(a)[\![x]\!] \tag{3}$$
$$+\ [b < a] \cdot \delta(\bot)[\![x]\!]\,. \tag{4}$$

This distribution is parametric in a and b, and it consists of three parts. For $a < b$, the part (2) defines a continuous uniform distribution between a and b. In part (3), the interval only includes a single point and we hence place a point mass on a. The case $b < a$ is treated as an error and we put all the probability mass on the error value $\bot$ in part (4).

4.4 Comparison to PSI

As a core difference to PSI [6], λPSI's symbolic domain closely follows the measure-theoretic interpretation of its terms (see §4.2). In particular, it introduces explicit Lebesgue measures ($\lambda[\![x]\!]$) for continuous distributions and explicitly specifies the random output variable of a Dirac delta. While the expression $\delta(x)$ in PSI is equivalent to $\delta(0)[\![x]\!]$ in λPSI, the formal interpretation of the PSI expression $\delta(x - y)$ is unclear. In λPSI, this is equivalent to either $\delta(x)[\![y]\!]$ or $\delta(y)[\![x]\!]$.

Note that in λPSI, the error state ($\bot$) is integrated in the symbolic domain instead of being treated separately in the program state. Also, data structures (tuples, arrays, and records) are directly modeled by λPSI's representation.

5 From Programs to Symbolic Representations

We now show how λPSI translates programs to the symbolic domain of §4. This is the first step of performing exact inference for higher-order probabilistic programs.

5.1 Translating Programs to the Symbolic Domain

A λPSI program is translated recursively. For each statement Stmt we represent the posterior distribution over values of all program variables *given* their previous values and any observations made within Stmt. More specifically, a statement Stmt is translated to an expression of the form $\lambda\sigma.\ \lambda\sigma'.\ f(\sigma)[\![\sigma']\!]$, which takes as input a *state* σ before executing Stmt, and returns the distribution over the state σ' after executing Stmt in σ. A state is a record containing values for all accessible variables. Similarly, an expression Ex is translated to a distribution of the form $\lambda\sigma.\ \lambda x.\ f(\sigma)[\![x]\!]$. This symbolic expression takes as input a state σ and returns the distribution over the value of Ex in the state σ.

Fig. 5 shows selected key rules of the translation, which is defined recursively. To reduce clutter, the presented rules ignore error handling and polymorphic types. We will discuss incorporating error states in §5.2.

Basic Expressions Variables are translated to the point mass distribution on the value of the variable according to the state (analogously for constants), see rule (5).

The rule for binary operations (6) is instantiated for addition, but works analogously for other deterministic expressions. The probability that `a+b` evaluates to a value x is computed by integrating over all possible values y and z for `a` and `b`, respectively, such that their sum $y + z$ equals x. In rule (6), this is expressed by recursively translating `a` and `b`, and introducing a Dirac delta. Note that because λPSI expressions do not have side-effects, the probabilities for the values of `a` and `b` are independent given the current state σ.

Distributions Sampling from built-in distributions (using for example `flip` or `exponential`) is directly translated to a symbolic representation as exemplified in §4.3.

Variable read	`x`	=	$\lambda\sigma.\ \lambda r.\ \delta(\sigma.\mathtt{x})[\![r]\!]$	(5)
Binary operation	`a + b`	=	$\lambda\sigma.\ \lambda x.\ \int \mathrm{d}y \int \mathrm{d}z\ \boxed{\mathtt{a}}(\sigma)[\![y]\!] \cdot \boxed{\mathtt{b}}(\sigma)[\![z]\!] \cdot \delta(y+z)[\![x]\!]$	(6)
Assignment	`x = e;`	=	$\lambda\sigma.\ \lambda\sigma'.\ \int \mathrm{d}x'\ \boxed{\mathtt{e}}(\sigma)[\![x']\!] \cdot \delta(\sigma\{\mathtt{x} \mapsto x'\})[\![\sigma']\!]$	(7)
Seq. composition	`A; B;`	=	$\lambda\sigma.\ \lambda\sigma''.\ \int \mathrm{d}\sigma'\ \boxed{\mathtt{A}}(\sigma)[\![\sigma']\!] \cdot \boxed{\mathtt{B}}(\sigma')[\![\sigma'']\!]$	(8)
Observation	`observe(e);`	=	$\lambda\sigma.\ \lambda\sigma'.\ \delta(\sigma)[\![\sigma']\!] \cdot p(\sigma) \quad \text{where } p(\sigma) := \int \mathrm{d}x\ \boxed{\mathtt{e}}(\sigma)[\![x]\!] \cdot [x \neq 0]$	(9)
Continuous obs.	`A; cobserve(b,c);`	=	$\lambda\sigma.\ \lambda\sigma'.\ \int \mathrm{d}y \left(\left(\boxed{\mathtt{A}}(\sigma)[\![\sigma']\!] \cdot \boxed{\mathtt{b}}(\sigma')[\![y]\!] \right) /\!\!/ \lambda[\![y]\!] \right) \cdot \boxed{\mathtt{c}}(\sigma')[\![y]\!]$	(10)
Control flow	`if e {A} else {B}`	=	$\lambda\sigma.\ \lambda\sigma'.\ \int \mathrm{d}x\ \boxed{\mathtt{e}}(\sigma)[\![x]\!] \cdot \left([x \neq 0] \cdot \boxed{\mathtt{\{A\}}}(\sigma)[\![\sigma']\!] \ + \ [x = 0] \cdot \boxed{\mathtt{\{B\}}}(\sigma)[\![\sigma']\!] \right)$	(11)
Scoping	`{A}`	=	$\lambda\sigma.\ \lambda\sigma'.\ \int \mathrm{d}\sigma''\ \boxed{\mathtt{A}}(\sigma')[\![\sigma'']\!] \cdot \delta(\sigma'' \setminus \{\mathtt{x} : \text{variable x introduced in A}\})[\![\sigma']\!]$	(12)
Inference	`infer(f)`	=	$\lambda\sigma.\ \lambda x.\ \delta\left(\lambda y.\ \boxed{\mathtt{f}}()[\![y]\!] \cdot Z^{-1} \right)[\![x]\!] \quad \text{where } Z := \int \mathrm{d}z\ \boxed{\mathtt{f}}()[\![z]\!]$	(13)
Sample	`sample(d)`	=	$\lambda\sigma.\ \lambda z.\ \int \mathrm{d}x\ \boxed{\mathtt{d}}(\sigma)[\![x]\!] \cdot x[\![z]\!]$	(14)
Expectation	`expectation(d)`	=	$\lambda\sigma.\ \lambda z.\ \int \mathrm{d}x\ \boxed{\mathtt{d}}(\sigma)[\![x]\!] \cdot \delta\left(\int \mathrm{d}y\ x[\![y]\!] \cdot y \right)[\![z]\!]$	(15)
Function	`(){ A; return e; }`	=	$\lambda\sigma.\ \lambda z.\ \int \mathrm{d}\sigma'\ \boxed{\mathtt{A}}(\sigma)[\![\sigma']\!] \cdot \boxed{\mathtt{e}}(\sigma')[\![z]\!]$	(16)

Figure 5. Key translation rules, ignoring error states. The rules are recursive and we write $\boxed{\mathtt{a}}$ to denote the translation of `a`.

Basic Statements For assignments `x = e`, we translate `e` to obtain the distribution over all possible right-hand sides x' in state σ. The new state σ' is equal to σ except that the value of variable `x` may be any such x' with the according probability. This is expressed using an integral over x' in (7).

The rule for sequential composition (8) is based on the standard chain rule for probabilities. In particular, the rule integrates over all possible intermediate states σ'.

Observations An observation `observe(e)` restricts the possible output states according to the boolean expression `e`. Intuitively, this amounts to setting the probabilities of all states violating `e` to zero and re-normalizing the resulting distribution. Rule (9) first computes the probability $p(\sigma)$ of `e` evaluating to a non-zero number (meaning, true) in the current state σ. If `e` is deterministic, $p(\sigma)$ is either 0 or 1. However, note that `e` may involve random choices such as in `observe(uniform(0,2) < 1)`, where $p(\sigma)$ is $\frac{1}{2}$. Next, the rule rescales the probability of the current state σ using $\delta(\sigma)[\![\sigma']\!] \cdot p(\sigma)$. The resulting distribution over σ' may not be normalized any more, but will be re-normalized later.

The effect of `observe` can be better understood under sequential composition. Consider the code in Fig. 6. After Lin. 2, `x` is uniformly distributed between 2 and 5. For Lin. 3, the probability $p(\mathtt{x})$ that `x >= 3` evaluates to true is $[\mathtt{x} \geq 3]$. According to the rule for sequential composition (8), the distribution over `x` after Lin. 3 is obtained by integrating over all intermediate values of `x` after Lin. 2. The factor $p(\mathtt{x})$ "cuts off" the distribution below 3 and we obtain the (unnormalized) uniform distribution between 3 and 5, as expected. In §6, we will see how λPSI formally derives this in a sequence of translation and simplification steps.

```
1  infer(() {
2      x := 2 + uniform(0,3);
3      observe(x >= 3);
4      return x;
5  })
```

Figure 6. Nested inference example.

Continuous Observations Rule (10) translates continuous observations `cobserve(b,c)`. For this, it incorporates all statements `A` preceding the observation in the current statement block (if there are none, `A` can be treated as an empty statement). This rule takes precedence over rule (8).

First, we recursively translate `A` and `b` to obtain a distribution for σ' and the value y of `b`. For `cobserve` to be defined, it must be possible to rewrite this distribution such that it involves a Lebesgue measure factor $\lambda[\![y]\!]$. Next, we eliminate this Lebesgue measure using disintegration ($/\!\!/$) and replace it by the distribution of `c`. This will become more clear once we discuss rules for disintegration in §6.5.

Control Flow and Scoping For if-then-else statements, we first translate both branches. A branch may introduce local variables in its scope, which must not occur in the output distribution. Hence, we use rule (12) to marginalize all variables introduced in a branch and obtain a distribution over all variables in the outer scope. Next, rule (11) translates the condition `e` and integrates over all possible values of `e`, always selecting the appropriate branch. Loops in λPSI are bounded and are unrolled during translation.

Nested Inference A key insight of λPSI is that the process of inference itself is directly expressible in λPSI's symbolic domain. The result is a distribution over the inferred distribution. In order to translate `infer(f)`, rule (13) first recursively translates the zero-argument function `f` and computes the normalization constant Z. Next, the rule normalizes the distribution represented by `f` and returns the Dirac delta at that position. Note that inference is deterministic and hence translated to a point mass.

Given a distribution `d`, the expression `sample(d)` draws a sample from `d`. Assume `d` is computed as follows (note that `Flip` is a distribution, while `flip` is a sample):

```
d := Flip(1/2);
if flip(1/4) { d = Flip(1/3); }
```

To compute the probability of a sample z from `d` we need to sum (i) the probability that `flip(1/4)` is false and z is generated by `Flip(1/2)`, and (ii) the probability that `flip(1/4)` is true and z is generated by `Flip(1/3)`. In general, we need to integrate over all possible distributions x represented by `d` and compute the probability of z according to x, see rule (14).

To translate the expression `expectation(d)`, we also integrate over all possible distributions x. For each such distribution, we compute the expectation by the standard definition (i.e., $\int \mathrm{d}y\, x[\![y]\!] \cdot y$) and construct the point mass on that value. The result is a distribution over the expected value of d.

Functions For simplicity, consider a function containing only one return statement at the end, i.e. the body has the form `A; return e;` (the general case is similar). In rule (16), we first translate `A` to obtain a distribution over the state σ', which comprises all variables in the function's scope. Then, we translate `e` to obtain a distribution over the return value in the state σ'. Finally, we integrate over all possible states σ'. Note that in general, the resulting distribution may be parameterized by the function's arguments (not shown).

Renormalization The entry point for a λPSI program is its `main` function, which may accept parameters. This function is translated just as any other function according to rule (16), but λPSI renormalizes the distribution before returning the result (similarly as in rule (13) for `infer`). Note that the normalization constant may depend on the parameters of `main`.

5.2 Accounting for Error States

Statements and expressions in λPSI may lead to errors under some states. Examples include divisions by zero and passing non-conforming parameters to distributions such as $a > b$ in `uniform(a,b)`. λPSI incorporates the probability of an error in the computed posterior distributions: the representation $\lambda\sigma.\ \mathbb{\Lambda}\sigma'.\ f(\sigma)[\![\sigma']\!]$ of a statement assigns to each *non-error* starting state σ the distribution over the output state σ', which may be the error state $\bot$ (similarly for expressions). Symbolic distributions make use of Dirac deltas $\delta(\bot)[\![x]\!]$ to capture the probability of an error (see for example Eq. (4)).

The presence of errors slightly complicates the translation rules of Fig. 5. In particular, for all integrals of the form $\int \mathrm{d}\sigma f[\![\sigma]\!]$ the integration domain also includes $\bot$ (as f may cause an error) and we hence must analyze the case $\sigma = \bot$ separately. For instance, the rule for sequential composition needs to propagate errors caused in `A` through `B` using an expression of the form $e_{1?}(e_2)$:

$$\lambda\sigma.\ \mathbb{\Lambda}\sigma''.\ \int \mathrm{d}\sigma'\ \boxed{\texttt{A}}(\sigma)[\![\sigma']\!] \cdot \boxed{\texttt{B}}_?(\sigma')[\![\sigma'']\!]\,.$$

We do not further discuss error states in this paper.

6 Inference by Symbolic Simplification

We now present how the symbolic representation of a translated program is simplified to a compact representation. This constitutes the second step of λPSI's inference procedure.

As we discuss in §6.6, the presented simplifications are an extension of the symbolic optimizations used by PSI [6]. In particular, we (i) generalize PSI's rules to λPSI's more powerful symbolic domain, and (ii) improve the former's efficiency using various (low-level) optimizations.

Basic Algebraic Simplifications λPSI applies various basic algebraic rules, such as removing multiplications by 1 and additions with 0, and simplifying terms multiplied by 0 to 0. It further leverages commutative, associative, and distributive laws where applicable. In general, integrals over sums are simplified to sums of integrals, and constant factors within integrals are moved out of the integrals.

Running Example We next describe the most important simplification rules on a running example. Concretely, we translate and simplify the λPSI expression in Fig. 6, while discussing a selection of interesting simplification steps (Fig. 7).

We start by translating and simplifying the expression `2 + uniform(0,3)` (Lin. 2 in Fig. 6). We apply the rule for binary operations (6) to obtain (17), see Fig. 7. Next, the constant `2` is translated to a point mass, and `uniform(0,3)` is translated according to (2).

6.1 Dirac Delta Substitution

Expression (18) contains an integral over y, which occurs as an "output" of a Dirac delta (see highlighted)—a common structure. Intuitively, we know that $\delta(2)[\![y]\!]$ is zero for all $y \neq 2$. Hence, we can simplify the expression by removing the integral and Dirac delta, and substituting all occurrences of y by 2 to obtain (19). In general, integrals over the output variable of a Dirac delta result in substituting the variable. This key rule is shown in (29) of Fig. 8.

6.2 Dirac Delta Linearization

The structure of (19) is similar as before, but this time the integration variable z occurs in the first argument to δ. In general, λPSI often encounters expressions of the form $\int \mathrm{d}x\ g[\![x]\!] \cdot \delta(f(x))[\![y]\!]$, which can be interpreted as y depending deterministically on x by $y = f(x)$. If g is a Dirac delta, we can

`2 + uniform(0,3)`

$$\overset{\S 5}{=} \lambda\sigma.\ \mathbb{\Lambda}x.\ \int \mathrm{d}y \int \mathrm{d}z\ \boxed{2}(\sigma)[\![y]\!] \cdot \boxed{\texttt{uniform(0,3)}}(\sigma)[\![z]\!] \cdot \delta(y+z)[\![x]\!] \tag{17}$$

$$\overset{\star}{=} \lambda\sigma.\ \mathbb{\Lambda}x.\ \int \mathrm{d}y\ \int \mathrm{d}z\ \delta(2)[\![y]\!] \cdot \tfrac{1}{3} \cdot [0 \le z] \cdot [z \le 3] \cdot \lambda[\![z]\!] \cdot \delta\Big(y + z\Big)[\![x]\!] \tag{18}$$

$$\overset{\S 6.1}{=} \lambda\sigma.\ \mathbb{\Lambda}x.\ \int \mathrm{d}z\ \tfrac{1}{3} \cdot [0 \le z] \cdot [z \le 3] \cdot \lambda[\![z]\!] \cdot \delta(2+z)[\![x]\!] \tag{19}$$

$$\overset{\S 6.2}{=} \lambda\sigma.\ \mathbb{\Lambda}x.\ \int \mathrm{d}z\ \tfrac{1}{3} \cdot [0 \le z\ ] \cdot [\ z\ \le 3] \cdot \lambda[\![x]\!] \cdot \delta(x-2)[\![z]\!] \tag{20}$$

$$\overset{\S 6.1}{=} \lambda\sigma.\ \mathbb{\Lambda}x.\ \tfrac{1}{3} \cdot [2 \le x] \cdot [x \le 5] \cdot \lambda[\![x]\!] \tag{21}$$

```
x := 2 + uniform(0,3);
observe(x >= 3);
```

$$\overset{\S 5}{=} \lambda\sigma.\ \mathbb{\Lambda}\sigma''.\ \int \mathrm{d}\sigma'\ \boxed{\texttt{x := 2 + uniform(0,3)}}(\sigma)[\![\sigma']\!] \cdot \boxed{\texttt{observe(x >= 3)}}(\sigma')[\![\sigma'']\!] \tag{22}$$

$$\overset{\star}{=} \lambda\sigma.\ \mathbb{\Lambda}\sigma''.\ \int \mathrm{d}\sigma'\ \int \mathrm{d}x'\ \tfrac{1}{3} \cdot [2 \le x'] \cdot [x' \le 5] \cdot \lambda[\![x']\!] \cdot \delta(\sigma\{\mathtt{x} \mapsto x'\})[\![\sigma']\!] \cdot \delta\Big(\sigma'\Big)[\![\sigma'']\!] \cdot [\ \sigma'\ .\mathtt{x} \ge 3] \tag{23}$$

$$\overset{\S 6.1}{=} \lambda\sigma.\ \mathbb{\Lambda}\sigma''.\ \int \mathrm{d}x'\ \tfrac{1}{3} \cdot [2 \le x'] \cdot [x' \ge 3] \cdot [x' \le 5] \cdot \lambda[\![x']\!] \cdot \delta(\sigma\{\mathtt{x} \mapsto x'\})[\![\sigma'']\!] \tag{24}$$

$$\overset{\S 6.3}{=} \lambda\sigma.\ \mathbb{\Lambda}\sigma''.\ \int \mathrm{d}x'\ \tfrac{1}{3} \cdot [3 \le x'] \cdot [x' \le 5] \cdot \lambda[\![x']\!] \cdot \delta(\sigma\{\mathtt{x} \mapsto x'\})[\![\sigma'']\!] \tag{25}$$

Fig. 6

$$\overset{\S 5}{=} \lambda\sigma.\ \mathbb{\Lambda}x.\ \delta\Big(\mathbb{\Lambda}y.\ \boxed{\text{Lin. 2–4 in Fig. 6}}()[\![y]\!] \cdot \Big(\int \mathrm{d}z\ \boxed{\text{Lin. 2–4 in Fig. 6}}()[\![z]\!]\Big)^{-1}\Big)[\![x]\!] \tag{26}$$

$$\overset{\star}{=} \lambda\sigma.\ \mathbb{\Lambda}x.\ \delta\Big(\mathbb{\Lambda}y.\ \tfrac{1}{3} \cdot [3 \le y] \cdot [y \le 5] \cdot \lambda[\![y]\!] \cdot \Big(\int \mathrm{d}z\ \tfrac{1}{3} \cdot [3 \le z] \cdot [z \le 5] \cdot \lambda[\![z]\!]\Big)^{-1}\Big)[\![x]\!] \tag{27}$$

$$\overset{\S 6.4}{=} \lambda\sigma.\ \mathbb{\Lambda}x.\ \delta\Big(\mathbb{\Lambda}y.\ \tfrac{1}{2} \cdot [3 \le y] \cdot [y \le 5] \cdot \lambda[\![y]\!]\Big)[\![x]\!] \tag{28}$$

Figure 7. Selected steps of deriving a simplified representation of the code in Fig. 6. The terms affected by substitution (§6.1), linearization (§6.2), guard simplification (§6.3), and symbolic integration (§6.4) are highlighted. Equalities annotated with ★ denote recursive translation and simplification.

Substitution

$$\int \mathrm{d}x\ f(x) \cdot \delta(v)[\![x]\!] \ =\ f(v) \tag{29}$$

Linearization

$$\delta(f(x))[\![y]\!] \cdot \lambda[\![x]\!] = [f'(x) = 0] \cdot \delta(f(x))[\![y]\!] \cdot \lambda[\![x]\!] \tag{30}$$
$$+ [f'(x) \neq 0] \cdot \overbrace{\textstyle\sum_{z:f(z)=y} \underbrace{\delta(z)[\![x]\!]\,/|f'(z)|}_{\text{part 1}} \cdot \lambda[\![y]\!]}^{\text{part 2}}$$

Disintegration

$$(e \cdot \lambda[\![x]\!]) /\!\!/ \lambda[\![x]\!] \ =\ e \tag{31}$$

Figure 8. Simplifying Dirac deltas and Lebesgue measures. Here, f' is the derivative of f.

apply (29) to substitute x in $f(x)$. Otherwise, we would like to express $\delta(f(x))[\![y]\!]$ in terms of $\delta(h(y))[\![x]\!]$ for some h such that we can later apply substitution (29). This is achieved by *linearization*, which rewrites the original Dirac delta over y as a linear combination of Dirac deltas over x.

Linearization in a Simple Example Let us have a look at our running example (19), where $g[\![z]\!] = \frac{1}{3} \cdot [0 \le z] \cdot [z \le 3]$ is the uniform distribution on $[0, 3]$. Instead of first selecting z uniformly between 0 and 3 (by g) and then setting x to $2 + z$ (by the Dirac delta), we can just as well directly select x uniformly between 2 and 5.

Intuitively, $\delta(2 + z)[\![x]\!]$ is only non-zero at locations where $x = 2 + z$, or equivalently $z = x - 2$. Hence, we can linearize this Dirac delta (see highlighted in (19)) by expressing z in terms of x and moving z to the second argument of δ, see (20). Note how thereby, $\lambda[\![z]\!]$ changes to $\lambda[\![x]\!]$. Then, we can apply substitution (29) to obtain the desired result in (21).

The General Case Rewriting $\delta(f(x))[\![y]\!]$ for general f requires more care. We now explain the general rule as presented in Fig. 8, Eq. (30).

To highlight a first issue with our previous attempt, inspect the following normalized distribution over y:

$$\mathbb{\Lambda}y.\ \int \mathrm{d}x\ [0 \le x] \cdot [x \le 1] \cdot \lambda[\![x]\!]\ \delta(2x)[\![y]\!]\ .$$

Incorrectly linearizing $\lambda[\![x]\!]\ \delta(2x)[\![y]\!]$ to $\lambda[\![y]\!]\ \delta(y/2)[\![x]\!]$ gives

$$\int \mathrm{d}x\ [0 \le x] \cdot [x \le 1] \cdot \lambda[\![y]\!]\ \delta(y/2)[\![x]\!] \overset{(29)}{=} [0 \le y] \cdot [y \le 2] \cdot \lambda[\![y]\!]$$

which is not normalized anymore. In fact, we would need to introduce a factor $\frac{1}{2}$. As can be shown by the substitution rule of Lebesgue integration, in general one needs to divide by the absolute value of the derivative f' of f (part 1 in Fig. 8).

Second, there may be more than one value x for which $f(x) = y$ (i.e., f may not be invertible). For example, for $y > 0$ the Dirac delta $\delta(x^2)[\![y]\!]$ is non-zero for both $x = \sqrt{y}$ and $x = -\sqrt{y}$, so the linearized expression is a sum of two Dirac deltas at these positions (see part 2 in Fig. 8).

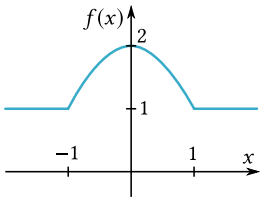

Figure 9. The function $f(x) = 1 + [-1 \le x]\cdot[x \le 1]\cdot(1-x^2)$.

Because part 2 is not defined for locations where the derivative of f is zero, we need to treat such locations separately. For this reason, (30) distinguishes $f'(x) = 0$ and $f'(x) \neq 0$.

For the first case, we can often find all solutions x of $f'(x) = 0$ and substitute these in $f(x)$. For example, consider the function f given in Fig. 9 whose derivative is zero at $x = 0$, everywhere below -1, and everywhere above 1. We can hence rewrite $[f'(x) = 0]$ to $[x \le -1] + [x \ge 1] + [x = 0]$ and distribute $\delta(f(x))[\![y]\!]$ over these three summands. Because we know that $f(0) = 2$, we can rewrite $[x = 0]\cdot\delta(f(x))[\![y]\!] = [x = 0]\cdot\delta(2)[\![y]\!]$. Further, because the derivative is zero, we know that $f(x)$ must have *the same value* (namely, 1) for all $x \le -1$. Hence, we can rewrite $[x \le -1]\cdot\delta(f(x))[\![y]\!]$ to $[x \le -1]\cdot\delta(1)[\![y]\!]$ (similarly for $x \ge 1$).

6.3 Guard Simplifications

We continue our running example by translating and simplifying Lin. 2 and Lin. 3 of Fig. 6. These lines are translated to (22) using the rule for sequential composition (8), and instantiating the simplified expressions (steps not shown in Fig. 7) gives (23). Because the integration variable σ' is the output of a Dirac delta (see highlighted), we can again apply substitution (29). Note how the access of field x in σ'.x is simplified to x', because σ' is substituted by $\sigma\{\mathsf{x} \mapsto x'\}$.

In the resulting expression (24), there are multiple Iverson bracket factors imposing constraints on x' (called *guards*). In particular, x' is bounded from below by both 2 and 3 due to the highlighted factors. As the constraint $2 \le x'$ is implied by $x' \ge 3$, we simplify the two factors to $[3 \le x']$ in (25).

In addition to eliminating redundant guards, λPSI supports many more *guard simplifications* (mostly inherited from PSI). For example, it simplifies whole terms to 0 if the therein contained guards are unsatisfiable (e.g., as in $[x = 0]\cdot[x \neq 0]$). Also, λPSI analyzes complex guard constraints (such as quadratic polynomials) to rewrite them as a combination of simpler, linear guard constraints (e.g., we rewrite $[x^2 \ge 4]$ as $[x \ge 2] + [x \le -2]$). Guard simplifications are also used to simplify $[f'(x) = 0]$ during linearization (30), see our previous example in §6.2.

6.4 Symbolic Integration

We continue translating and simplifying Fig. 6, which performs nested inference at the top level. Recall that nested inference can be directly represented in λPSI's symbolic domain: using rule (13), we translate `infer` to (26). Recursively translating and simplifying Lin. 2–4 gives (27).

The normalization constant (highlighted) is an integral to be simplified. This time, the integrand does not contain any Dirac deltas, so the simplification rules of §6.1 and §6.2 do not apply. However, the integrand is simply a constant function between 3 and 5, hence we can simplify the integral to $\frac{5-3}{3} = \frac{2}{3}$. The resulting expression (28) is fully simplified and represents the posterior distribution over the value of the expression from Fig. 6.

Simplifying Integrals λPSI extends PSI's powerful engine for symbolic integration of a wide class of functions not involving Dirac deltas. To simplify such integrals, λPSI first applies guard simplifications (§6.3) in order to determine the integration bounds. Note that a single guard constraint may be simplified to a sum of guards, hence this step may split an integral into a sum of integrals. Next, λPSI leverages antiderivatives of known function classes (e.g., polynomials and logarithms) and standard integration rules (e.g., integration by parts) to find the integrand's antiderivative. If this succeeds, the latter is evaluated at the bounds to give the final result.

Simplifying Sums λPSI applies similar techniques to simplify absolutely convergent series, which may for example occur when computing expectations of discrete distributions. For example, while simplifying `expectation(Geometric(`$\frac{1}{4}$`))` λPSI encounters the following expression (cp. (1)):

$$\int \mathrm{d}x\ \textstyle\sum_{i\in\mathbb{Z}} [i \ge 0]\cdot(\frac{3}{4})^i \cdot \frac{1}{4}\cdot\delta(i)[\![x]\!]\cdot x.$$
$$\overset{(29)}{=} \tfrac{1}{4}\cdot\textstyle\sum_{i\in\mathbb{Z}} [i \ge 0]\cdot(\tfrac{3}{4})^i\cdot i. \qquad (32)$$

λPSI identifies several convergent series with known values and heavily makes use of Abel's lemma (summation by parts) [1] to simplify such expressions. Using this, it can for instance simplify (32) to the value 3.

6.5 Symbolic Disintegration

Consider the following code snippet:

```
x := gauss(μ,ν); cobserve(2·x, y);
```

The `cobserve` statement conditions on the possible but probability zero event that $2\cdot$x equals an observed value y. Intuitively, this has two effects: (i) the current program path is reweighted by $\frac{1}{2}f(y/2;\mu,\nu)$, where f is the Gaussian density, and (ii) the value of $2\cdot$x is fixed to the observed value y. We now derive these effects from our translation and simplification rules. After translation and some simplification steps (σ and σ' omitted for brevity), the distribution of x is

$$\int \mathrm{d}z\ \big((f(x;\mu,\nu)\cdot\lambda[\![x]\!]\cdot\delta(2\cdot x)[\![z]\!])/\!/\lambda[\![z]\!]\big)\cdot\delta(y)[\![z]\!]$$
$$\overset{(30)}{=} \int \mathrm{d}z\ \big((\tfrac{1}{2}f(x;\mu,\nu)\cdot\delta(z/2)[\![x]\!]\cdot\ \lambda[\![z]\!]\ )/\!/\ \lambda[\![z]\!]\ \big)\cdot\delta(y)[\![z]\!]\,.$$

We purposefully used Dirac delta linearization to write the joint prior distribution of x and $z = 2{\cdot}x$ with an explicit factor $\lambda[\![z]\!]$. Now, we use the disintegration rule (31) to eliminate the highlighted $\lambda[\![z]\!]$, obtaining the desired weighted Dirac delta:

$$\int \mathrm{d}z\ \tfrac{1}{2} f(x;\mu,\nu)\cdot\delta\left(\tfrac{z}{2}\right)[\![x]\!]\cdot\delta(y)[\![z]\!] = \tfrac{1}{2} f\left(\tfrac{y}{2};\mu,\nu\right)\cdot\delta\left(\tfrac{y}{2}\right)[\![x]\!]$$

In general, rule (31) transforms the density of the first argument of `cobserve` to a weight for the remaining distribution. Note that due to its powerful Dirac delta linearizer, λPSI can symbolically disintegrate some programs that are not handled by Shan and Ramsey [19].

6.6 Comparison to PSI

The main differences to PSI [6] are related to adding support for the new terms of the symbolic domain (see Fig. 4) and are hence purely additive. Still, λPSI introduces major design and implementation improvements. Unfortunately, a full treatment of all simplification rules is impossible within the scope of this paper, as they have been developed over multiple years. Still, we list the most important differences to PSI's symbolic optimizations below.

While basic arithmetic simplifications and guard simplifications (§6.3) are mostly inherited from PSI, some low-level improvements were added (e.g., PSI can not simplify guards involving reciprocals of polynomials). Unlike in PSI, the Dirac delta of λPSI has an explicit "output" argument, but the rules for Dirac delta substitution (§6.1) are analogous. Linearization (§6.2) closely follows the rules already present in PSI. However, the rewrites allowed in λPSI are more restricted because Lebesgue measures are no longer implicit and must be present. While simplification rules for integrals (§6.4) are mainly inherited from PSI, λPSI introduces many non-trivial simplifications of sums (e.g., to simplify expectations). Disintegration (§6.5) is new, as PSI does not support `cobserve`.

6.7 Limitations

λPSI's simplifications are only best-effort, i.e., sound but not complete. Like virtually all existing exact inference and incomplete computer algebra systems, its limitations (what can and can not be simplified) are hard to characterize. Generally speaking, the limitations of λPSI are related to inference being intractable in general. In particular, not all programs have closed-form representations in the symbolic domain of Fig. 4, and no algorithm (efficient or not) will always be able to decide if such representations exist.

However, in §7 we show that λPSI's simplification rules work well for a set of benchmark programs. We also show an example which can not be simplified by λPSI.

6.8 Correctness

In this paper, we do not provide an explicit embedding of λPSI's symbolic representation into a system widely accepted to be consistent, such as set theory. However, we note that the correctness of λPSI is nonetheless falsifiable. For example, we can write a program that computes a known real number or real function. The expression produced by λPSI will often be interpretable as a standard mathematical expression, which can be compared to the known result. There are also less explicit ways to falsify the correctness of λPSI: For example, if it were to compute a negative probability or probability density, we would know that it was incorrect.

7 Evaluation

We implemented λPSI by extending the publicly available PSI PPL (https://psisolver.org) with the features from §3 and the exact inference capabilities from §4–§6.

We assembled a collection of 31 programs with higher-order constructs such as distributions over functions and nested inference, summarized in Tab. 1. The collection comprises examples from the literature (including the applications discussed in §2 and §7.1) and custom programs.

All our experiments were performed on a commodity laptop with 32 GB of RAM and 4 CPU cores at 2.60 GHz.

Expressiveness and Performance of λPSI We can express all programs succinctly in λPSI, as all required language features are supported as first-class citizens. For the FairSVM [23] example, there is no closed-form representation of the posterior in λPSI's symbolic domain as it depends non-trivially on properties of products of Gaussians (this is inherited from PSI [6]). A simple example that can not be simplified by λPSI for the same reason: `def main() ⇒ gauss(0,1)*gauss(0,1)<1;` For the remaining 30 examples, λPSI successfully infers a closed-form exact result (no integrals left) within at most 42 seconds. We conclude that λPSI is powerful enough to express interesting applications and that its simplification engine is effective.

7.1 Case Studies

Bayesian Regression Heunen et al. [11] motivate higher-order probabilistic programming by expressing linear regression as a prior over first-class functions f together with observed I/O examples. We use λPSI to compute the posterior density $p(y)$ of $y = f(x)$ in terms of x. We show a plot of the posterior in App. A.3.

We also encoded an example with a piecewise linear function prior [10], deriving the posterior for y at a specific x.

Conditional with Symbolic Parameters Given a probability distribution Pr, an event A and observed evidence B, we want to compute $\Pr[A \mid B]$ (shown in Fig. 10). We use Bayes' rule directly (instead of `observe`). λPSI evaluates the resulting probability for all valid values for parameters `X` and `Y` *simultaneously*; the result is shown in Fig. 10, right.

Table 1. Probabilistic programs used in evaluation (31 in total). For each program, we indicate if it involves higher-order functions (→), nested inference (⇝), first-class expectations (𝔼), continuous distributions (∧), continuous observations (↜), or symbolic parameters ([a]). SocialCognition and TotalVarDist have multiple variants. Some programs are not expressible in Hakaru (–), while others lead to errors (×), unsimplified (■) or incorrect (××) results. [a]Not directly expressible; rewritten as first-order programs without function calls, multiple manual steps. [b]For concrete instantiation of symbolic parameters.

Program(s)	Description	Features						Runtime	
		→	⇝	𝔼	∧	↜	[a]	Hakaru [19]	λPSI
SocialCognition (12) [9]	Multiple rational agent models (see §2)		●					× / ××	<5.5s
CondProb (Fig. 10)	Compute conditional probability using expectation operator	●	●	●	●		●	×[ab]	3s
Overview (Fig. 2)	Example involving multiple language features		●		●			×	0.3s
ChannelCap	Mutual information between input and output of noisy channel	●	●	●			●	-	2s
Entropy	Entropy of randomly generated sequence		●	●			●	-	3s
GenCap [3]	Generalization capacity of sorting algorithms (see Fig. 11)	●	●	●				-	16s
AIDE [5]	KL-divergence between particle filter and exact inference on HMM	●	●	●				-	42s
BivariateIndep	Verify that bivariate distribution has independent components	●	●	●				-	0.1s
SecretSanta	Five people guess secret santa in turn, based on uniform prior		●	●				×	0.5s
TotalVarDist (2) [2]	Total variation distance for random walk and Dynkin process (20 steps)		●	●				×	< 5s
MontyHall	Monty hall problem variants modeled using nested inference		●	●				×	0.1s
Variance	Compute variance of given distribution		●	●	●			-	0.5s
CDF	Compute CDF of Gaussian distribution at a point drawn from it		●	●	●			×	0.1s
GANLoss	GAN loss for simple probabilistic model against optimal discriminator	●	●	●	●		●	-	0.7s
FairSVM [23]	Infer weights for fair SVM classifier	●	●	●	●			×	■
BayesLinReg [11]	Bayesian linear regression from 5 data points with Gaussian noise	●			●	●	●	2s[ab]	0.2s
BayesPiecewiseLR [10]	Bayesian piecewise linear regression from 7 data points	●			●	●		×	33s
DisintegrateLinear [19]	Motivating example from [19], disintegrate linear function two ways		●	●	●	●		4s[a]	0.2s
DisintegrateQuadratic	Disintegrate quadratic function (involves `cobserve((x-1)^2,y)`)		●	●	●	●		■[a]	0.1s

```
def PrAgB(d: Distribution[ℝ×ℝ],
          A: ℝ×ℝ→𝔹, B: ℝ×ℝ→𝔹){
  prAB := expectation(infer((){
    x := sample(d);
    return A(x) && B(x);
  }));
  prB := expectation(infer((){
    x := sample(d);
    return B(x);
  }));
  return prAB / prB;
}
```

```
def main(X,Y){
  joint := infer((){
    x := uniform(0,1);
    y := x^2 + uniform(0,1);
    return (x,y);
  });
  A := (x,y) ⇒ x<X;
  B := (x,y) ⇒ y>Y;
  return PrAgB(joint,A,B);
}
```

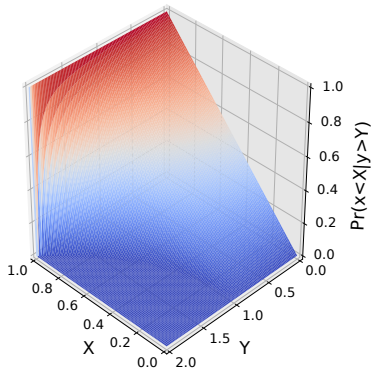

Figure 10. Conditional probability Pr[`x < X | y > Y`] depending on `X` and `Y`.

Entropy We can also naturally express information theoretical concepts such as *entropy*, *KL-divergence* and *mutual information*, shown in Fig. 11 (left). Note that there exists no (purely sampling-based) unbiased estimator for entropy or mutual information [16]. Busse et al. [3] introduce *generalization capacity*, quantifying how much algorithms depend on noise in noisy input data. Fig. 11 (right) shows a λPSI encoding of this task. We compare the generalization capacity of three sorting algorithms on sequences of length 3. AIDE [5] infers an approximate upper bound on the expected (symmetrized) KL-divergence between the results of two inference approaches. Our AIDE benchmark exactly computes the expected KL-divergence between the results of a particle filter and exact inference, and shows that the particle filter gains precision as more particles are added.

7.2 Comparison to Previous Work

We compare λPSI to Hakaru [14], the only other system we are aware of that can perform exact inference for probabilistic programs with continuous distributions. Our goal is to validate that λPSI is the first tool that can perform exact symbolic inference on *higher-order* probabilistic programs with continuous distributions. As Hakaru terms support first-class functions, this is not immediately obvious.

Hakaru provides external transformations "normalize" (for inference with positive-probability evidence), "disintegrate" (for inference with continuous evidence), and "simplify" (to transform terms produced by the other transformations into closed-form representations). In λPSI, `infer` (normalize) and `cobserve` (disintegrate) are first-class operators and can therefore be used for higher-order inference.

```
def S[a](x: a, q: Distribution[a])⇒ // surprise
    -log₂(expectation(infer(()⇒x == sample(q))));
// entropy and cross-entropy:
def H[a](p: Distribution[a])⇒
      expectation(infer(()⇒S(sample(p), p)));
def H_cross[a](p: Distribution[a], q: Distribution[a])⇒
      expectation(infer(()⇒S(sample(p), q)));
// KL-divergence and mutual information (π₁, π₂ project to marginals):
def KL[a](p: Distribution[a], q: Distribution[a])⇒H_cross(p,q) - H(p);
def I[a,b](p: Distribution[a×b])⇒H(π₁(p)) + H(π₂(p)) - H(p);
```

```
def genCap[a,b](f: a→b,p: Distribution[a],
                noise: a→a) ⇒
  I(infer(()){
    x := sample(p);
    return (f(noise(x)),f(noise(x)));
  }));
def sortCap[n:ℕ](sort: 𝔹^(n×n)→ ℕ^n){
  input := RandomComparisonMatrix(n);
  error := noise(0.1);
  return genCap(sort, input, error);
}
```

Figure 11. Information-theoretic quantities associated with discrete distributions (left) and an application (right): generalization capacity (top right); capacity of sorting algorithms (bottom right). The type parameters in square brackets enable polymorphism. For example, `a` in `S[a]` can be instantiated with any type (see App. A.2 for details).

Without Continuous Observations Hakaru's normalize transformation can in principle be expressed as a Hakaru term using the "expect" operator to compute the total weight of a measure. We use this strategy to encode most of our examples without continuous observations in Hakaru (see Tab. 1). Unfortunately, this leads to an error relating to the "expect" operator in Hakaru's simplification engine. Hakaru's "expect" operator can only be used on functions bounded between 0 and 1, hence examples including Entropy and Variance are not encodable in Hakaru. Furthermore, as Hakaru can not simplify function terms, some programs can not be directly expressed in Hakaru, particularly those involving symbolic parameters. However, we manually rewrite some programs for which inlining functions is possible. For a fully inlined version of CondProb with concretized symbolic parameters and where we use an unnormalized observation instead of Bayes' rule, simplification leads to a stack overflow in Maple (used by Hakaru). For some examples without nested evidence (SocialCognition), simplify returns the zero measure instead of the correct answer and disintegrate terminates with an error unless we inline all function definitions.

With Continuous Observations While Hakaru does not support first-class disintegration, this can sometimes be simulated by chained calls to Hakaru's disintegration, normalization and simplification engines. In cases where manual inlining of higher-order functions is easy (such as for BayesLinReg, see Tab. 1), we can use Hakaru to compute a result. Otherwise, Hakaru cannot easily be used to perform inference. For example, we cannot express BayesPiecewiseLR as an inlined Hakaru term without significant manual effort. The DisintegrateQuadratic example can be disintegrated by λPSI, but not Hakaru.

We suspect that one could automate our manual steps by directly using Hakaru's Monad within a Haskell program. Unfortunately, this mode of using Hakaru is not documented and does not seem to be encouraged. It is also important to note that this does not allow Hakaru to perform (non-trivial) nested inference, as it cannot simplify function terms.

8 Related Work

The semantics of higher-order probabilistic programs has been studied extensively [20, 21], resulting in the definition of the category of quasi-Borel spaces [11]. Ścibior et al. [18] formulate a framework for denotational verification of inference transformations, supporting higher-order probabilistic programs with continuous as well as discrete distributions. Based on this, Sato et al. [17] present a program logic.

While Hakaru [14, 19] does not currently provide exact inference support for higher-order constructs, the system has made other important advances, such as disintegrating programs with symbolic arrays [15], as well as exact reasoning about symbolic arrays to automatically and efficiently derive closed-form conditional distributions [24].

Tavares et al. [23] propose a new kind of higher-order inference operator that allows certain models with nested inference to be specified more concisely.

9 Conclusion

We presented λPSI, the first higher-order statically typed probabilistic programming language equipped with a solver that computes exact (symbolic) probability distributions of programs. We showed how to express several interesting applications (e.g., information theory, rational agents) in λPSI and demonstrated that our solver was able to compute their exact distributions.

This is the first time one is able to exactly analyze probabilistic programs at this level of expressiveness. In the future, we plan to investigate ways to further scale the exact inference algorithm as well as explore combinations with approximate inference techniques.

Acknowledgements

We thank our shepherd and the anonymous reviewers for the constructive feedback. The research leading to these results was partially supported by an ERC Starting Grant 680358.

Prog	::=	Decl^*
Decl	::=	Func | VarDecl
Func	::=	`def` x ParamList^*`(:`Ex$)^?$ Body
ParamList	::=	`(`Param^*`(,)`$^?$`)` | `[`Param^*`(,)`$^?$`]`
Param	::=	x`:` Ex
Body	::=	`{` Stmt^* `}` | $\Rightarrow$ Ex`;`
Type	::=	x | $*$ | $\mathbb{B}$ | $\mathbb{N}$ | $\mathbb{Z}$ | $\mathbb{Q}$ | $\mathbb{R}$ | $\mathbf{1}$ | Ex^{Ex} | `(`Ex $\times \cdots \times$ Ex`)` | Ex`[]` | `(`Ex $\rightarrow$ Ex`)` | `(` $\prod_{x:\text{Ex}}$ Ex$[x]$`)`
VarDecl	::=	x `:=` Ex`;`
Lambda	::=	ParamList^* LambdaBody
LambdaBody	::=	`{` Stmt^* `}` | $\Rightarrow$ Ex

Ex	::=	n | x | BuiltIn | `(`Ex`)` | `(`Ex`:` Ex`)` | `(`uop Ex`)` | `(`Ex bop Ex`)` | `()` | `(`Ex`,)` | `(`Ex`,`Ex`)`*`(,)`$^?$`)` | `[]` | `[`Ex`(,`Ex`)`*`(,)`$^?$`]` | Ex`.length` | Ex`[`Ex`]` | Ex`(`Ex`)` | Lambda | Type
AssgnLhs	::=	x | AssgnLhs`[`Ex`]` | `(`AssgnLhs`:` Ex`)` `(`AssgnLhs`, ..., `AssgnLhs`(,)`$^?$`)`
Stmt	::=	AssgnLhs `=` Ex`;` | Decl | `return` Ex`;` | `observe(`Ex`);` | `cobserve(`Ex`,`Ex`);` | `if` Ex `{` Stmt^* `}` `(else {` Stmt^* `})`$^?$ | `for` x `in` $[n..n)${ Stmt^* `}` | `assert(`Ex`);`

$n \in \{0, 1, 2, 3, \ldots\}$ $\quad$ $bop \in \{+, -, *, \%, /, \hat{}, \sim, \&\&, ||, ==, !=, <, >, <=, >=\}$

$x \in \text{Vars}$ $\quad$ $uop \in \{+, -, !\}$

$\text{BuiltIn} \in \{\texttt{Distribution}, \texttt{infer}, \texttt{sample}, \texttt{expectation}, \texttt{array}, \pi, \texttt{exp}, \texttt{log}, \texttt{floor}, \texttt{ceil}, \ldots\} \cup \text{Dist}$

$\text{Dist} \in \{\texttt{Flip}, \texttt{Gauss}, \texttt{Uniform}, \texttt{UniformInt}, \texttt{Categorical}, \texttt{Exponential}, \texttt{Dirac}, \ldots\}$

Figure 12. Full syntax of the higher-order probabilistic programming language λPSI (extension of Fig. 3).

A Appendix

A.1 Exact Posterior for Overview Example

The following expression is computed by λPSI for the posterior distribution of the code in Fig. 2 (ignoring errors):

$$\begin{aligned}&\lambda x, y.\ [0 \le y] \cdot [y \le 1] \cdot \lambda[\![y]\!] \cdot \left(\tfrac{1}{2} \cdot [y \le \tfrac{1}{3}] \cdot \delta\left(\tfrac{1}{6}\right)[\![x]\!]\right.\\ &\quad + \tfrac{1}{4} \cdot [0 \le x] \cdot [x < \tfrac{1}{2}] \cdot [y \le 2x] \cdot \tfrac{1}{x} \cdot \lambda[\![x]\!]\\ &\quad \left. + \tfrac{1}{4} \cdot [y \le \tfrac{2}{3}] \cdot \delta\left(\tfrac{1}{3}\right)[\![x]\!] + \tfrac{5}{12} \cdot \delta\left(\tfrac{1}{2}\right)[\![x]\!]\right).\end{aligned}$$

A.2 λPSI Language Details

Fig. 12 presents the full syntax of λPSI.

Expressions In addition to tuples, λPSI supports arrays. The expression `()` (resp. `[]`) is the empty tuple (resp. array) and `(Ex,)` is a single-element tuple.

Subexpressions may be annotated with their types, as in `((1:ℕ)+(2:ℕ):ℕ,[-1,2]):ℕ×ℤ[]`.

Types We support dependent types Ex^{Ex} for fixed-length arrays, which are compatible to tuple types (e.g., $\mathbb{Z}^2$ is equivalent to $\mathbb{Z} \times \mathbb{Z}$). Subtype relations are standard (e.g., $\mathbb{N} \subseteq \mathbb{Q}$ and $\mathbb{R} \to \mathbb{N} \subseteq \mathbb{Z} \to \mathbb{Q}$). We do not distinguish types and other expressions in our grammar. Types are compared modulo partial evaluation of numerical expressions.

Functions Any function in λPSI can have multiple parameter lists, which defines a curried function. For example, `def const(x:ℝ)(_:ℝ) ⇒ x` is shorthand for `def const(x:ℝ) ⇒ (_:ℝ) ⇒ x`. It is possible to optionally specify a return type: `def id(x:ℝ):ℝ ⇒ x`.

Parameter lists can also be declared with square brackets, usually used for dependent types and polymorphic functions: The function `def foo[n:ℕ](x:ℝ^n):ℝ^(2·n) ⇒ x~x;` concatenates a fixed-length array `x` with itself to yield an array of double length. Further, consider the following identity function: `def id[a:*](x:a) ⇒ x;`. The type of this function is $\prod_{x:*}(x \to x)$, which can be read as "for any type x, this provides a function from x to x". Square-bracket parameters can be provided explicitly at the call site, as for example in `id[ℝ[]]([1,2,3])`, or inferred automatically from a regular function call, which will happen twice when type-checking the example expression `id(id)([1,2,3])`.

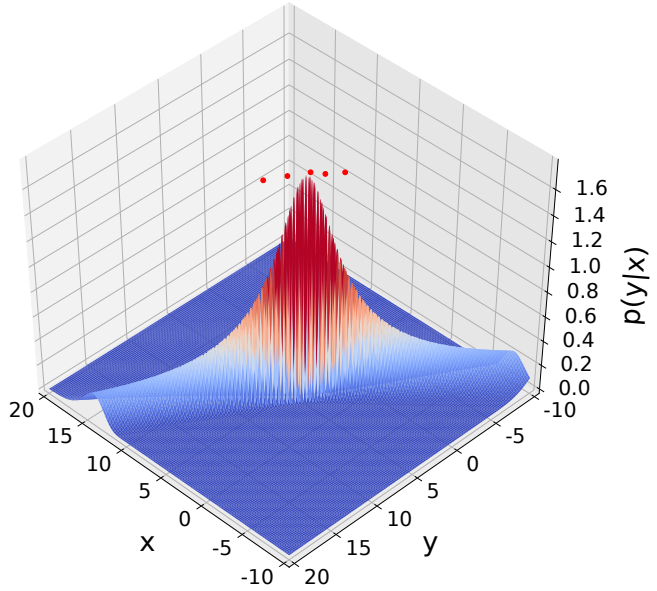

Figure 13. Posterior of $y = f(x)$ after 5 samples.

A.3 Plot for Bayesian Linear Regression

For Bayesian linear regression, we show a plot of the posterior probabilities after conditioning on 5 samples in Fig. 13.

References

[1] Niels Henrik Abel. 1826. Untersuchungen über die Reihe: 1 + (m/1)x + m·(m - 1)/(1·2)·x²+ m·(m - 1)·(m - 2)/(1·2·3)·x³+ ... In *Reine und angewandte Mathematik*. Vol. 1. 311–339.

[2] Gilles Barthe, Benjamin Grégoire, Justin Hsu, and Pierre-Yves Strub. 2017. Coupling Proofs Are Probabilistic Product Programs. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL 2017)*. ACM, New York, NY, USA, 161–174. https://doi.org/10.1145/3009837.3009896

[3] Ludwig M. Busse, Morteza Haghir Chehreghani, and Joachim M. Buhmann. 2012. The information content in sorting algorithms. In *Proceedings of the 2012 IEEE International Symposium on Information Theory, ISIT 2012, Cambridge, MA, USA, July 1-6, 2012*. 2746–2750. https://doi.org/10.1109/ISIT.2012.6284021

[4] Robert Cornish, Frank Wood, and Hongseok Yang. 2017. Efficient exact inference in discrete Anglican programs. *Workshop on Probabilitic Programming Semantics, colocated with ACM POPL'17* (2017). http://www.cs.ox.ac.uk/people/hongseok.yang/paper/pps17b.pdf

[5] Marco Cusumano-Towner and Vikash K Mansinghka. 2017. AIDE: An algorithm for measuring the accuracy of probabilistic inference algorithms. In *Advances in Neural Information Processing Systems 30*, I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett (Eds.). Curran Associates, Inc., 3000–3010. http://papers.nips.cc/paper/6893-aide-an-algorithm-for-measuring-the-accuracy-of-probabilistic-inference-algorithms.pdf

[6] Timon Gehr, Sasa Misailovic, and Martin Vechev. 2016. PSI: Exact symbolic inference for probabilistic programs. In *International Conference on Computer Aided Verification*. Springer, 62–83. https://doi.org/10.1007/978-3-319-41528-4_4

[7] Noah D. Goodman, Vikash K. Mansinghka, Daniel Roy, Keith Bonawitz, and Joshua B. Tenenbaum. 2008. Church: A Language for Generative Models. In *Proceedings of the Twenty-Fourth Conference on Uncertainty in Artificial Intelligence (UAI'08)*. AUAI Press, Arlington, Virginia, USA, 220–229.

[8] Noah D Goodman and Andreas Stuhlmüller. 2014. The Design and Implementation of Probabilistic Programming Languages. http://dippl.org. Accessed: 2017-5-15.

[9] Noah D Goodman and Joshua B. Tenenbaum. 2016. Probabilistic Models of Cognition. http://probmods.org. Accessed: 2019-11-18.

[10] Chris Heunen, Ohad Kammar, Sean Moss, Adam Ścibior, and Hongseok Yang. 2018. The semantic structure of quasi-Borel spaces. In *PPS'18*.

[11] Chris Heunen, Ohad Kammar, Sam Staton, and Hongseok Yang. 2017. A Convenient Category for Higher-order Probability Theory. In *Proceedings of the 32Nd Annual ACM/IEEE Symposium on Logic in Computer Science (LICS '17)*. IEEE Press, Piscataway, NJ, USA, Article 77, 12 pages. http://dl.acm.org/citation.cfm?id=3329995.3330072

[12] Donald E. Knuth. 1992. *Two notes on notation*. Technical Report. http://arxiv.org/abs/math/9205211

[13] V. Mansinghka, D. Selsam, and Y. Perov. 2014. Venture: a higher-order probabilistic programming platform with programmable inference. *ArXiv e-prints* (March 2014). arXiv:cs.AI/1404.0099

[14] Praveen Narayanan, Jacques Carette, Wren Romano, Chung-chieh Shan, and Robert Zinkov. 2016. Probabilistic Inference by Program Transformation in Hakaru (System Description). In *Functional and Logic Programming*. Springer International Publishing, Cham, 62–79. https://doi.org/10.1007/978-3-319-29604-3_5

[15] Praveen Narayanan and Chung-chieh Shan. 2017. Symbolic Conditioning of Arrays in Probabilistic Programs. *Proc. ACM Program. Lang.* 1, ICFP, Article 11 (Aug. 2017), 25 pages. https://doi.org/10.1145/3110255

[16] Liam Paninski. 2003. Estimation of Entropy and Mutual Information. *Neural Computation* 15, 6 (2003), 1191–1253. https://doi.org/10.1162/089976603321780272

[17] Tetsuya Sato, Alejandro Aguirre, Gilles Barthe, Marco Gaboardi, Deepak Garg, and Justin Hsu. 2019. Formal Verification of Higher-order Probabilistic Programs: Reasoning About Approximation, Convergence, Bayesian Inference, and Optimization. *Proc. ACM Program. Lang.* 3, POPL, Article 38 (Jan. 2019), 30 pages. https://doi.org/10.1145/3290351

[18] Adam Ścibior, Ohad Kammar, Matthijs Vákár, Sam Staton, Hongseok Yang, Yufei Cai, Klaus Ostermann, Sean K. Moss, Chris Heunen, and Zoubin Ghahramani. 2017. Denotational Validation of Higher-order Bayesian Inference. *Proc. ACM Program. Lang.* 2, POPL, Article 60 (Dec. 2017), 29 pages. https://doi.org/10.1145/3158148

[19] Chung-chieh Shan and Norman Ramsey. 2017. Exact Bayesian inference by symbolic disintegration. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*. 130–144. http://dl.acm.org/citation.cfm?id=3009852

[20] Sam Staton. 2017. Commutative semantics for probabilistic programming. In *European Symposium on Programming*. Springer, 855–879. https://doi.org/10.1007/978-3-662-54434-1_32

[21] Sam Staton, Hongseok Yang, Frank Wood, Chris Heunen, and Ohad Kammar. 2016. Semantics for Probabilistic Programming: Higher-order Functions, Continuous Distributions, and Soft Constraints. In *Proceedings of the 31st Annual ACM/IEEE Symposium on Logic in Computer Science (LICS '16)*. ACM, New York, NY, USA, 525–534. https://doi.org/10.1145/2933575.2935313

[22] Andreas Stuhlmüller and Noah D. Goodman. 2012. A Dynamic Programming Algorithm for Inference in Recursive Probabilistic Programs. *CoRR* abs/1206.3555 (2012). arXiv:1206.3555 http://arxiv.org/abs/1206.3555

[23] Zenna Tavares, Xin Zhang, Edgar Minaysan, Javier Burroni, Rajesh Ranganath, and Armando Solar Lezama. 2019. The Random Conditional Distribution for Higher-Order Probabilistic Inference. arXiv:cs.PL/1903.10556

[24] Rajan Walia, Praveen Narayanan, Jacques Carette, Sam Tobin-Hochstadt, and Chung-chieh Shan. 2019. From High-level Inference Algorithms to Efficient Code. *Proc. ACM Program. Lang.* 3, ICFP, Article 98 (July 2019), 30 pages. https://doi.org/10.1145/3341702

[25] Frank Wood, Jan-Willem van de Meent, and Vikash Mansinghka. 2015. A New Approach to Probabilistic Programming Inference. *CoRR* abs/1507.00996 (2015). http://arxiv.org/abs/1507.00996