# DPhil Thesis Proposal: Semantics of Probabilistic Functional Languages

Andrew Kenyon-Roberts

March 30, 2021

# 1 Introduction

The augumentation of lambda calculus with random sampling and conditioning, the fundamental constructs of probabilistic programming, can be interpreted formally in many different ways, and provides certain unique challenges in defining the language's semantics. The standard approach to defining random things, where all the random values are considered as functions from a probability space, has the problem that the random samples, which should form the basis of this probability space, are not specified in advance. There are various possible schemes of organising the probability space such that this data is available, known as trace or entropy space semantics. The conditioning construct, which modifies the posterior probability of an event, fits somewhat unnaturally in this framework, and has to be specified as an extra step that modifies the probabilities at the end.

Alternatively, the distribution of results can be specified more directly as a distribution. This omits any information on joint distributions from the semantics, but this itself can be in some ways an advantage, as a subterm can be duplicated without being correlated to itself. To define a denotational semantics for a higher-order probabilistic language, it would be natural to attempt to use the category of measurable spaces, however, that turns out to not be cartesian closed. This motivates the introduction of quasi-Borel spaces (4) as an alternative way in which a distribution can be defined, which has a more restrictive set of morphisms and does not have this issue, but is nevertheless still able to represent all the functions definable as programs.

For my thesis, I plan to extend the existing literature and my own previous work in defining the semantics of probabilistic functional languages.

# 2 Existing approaches

The various existing approaches to defining the semantics of probabilistic functional languages can generally be categorised as operational or denotational (depending on whether it works by defining the state of the program at each step, or by defining the meanings of subprograms independently and combining them together) (this distinction applies to functional languages more generally, rather than being specific to probabilistic languages), and distribution-based or trace-based (depending on whether only the distribution of outputs is defined, or the output that occurs is defined as a function of a random input).

As a running example, I will be using a simply-typed call by value language with a random sampling construct in the form of a probabilistic choice $M \oplus N$, a conditioning construct $\mathsf{score}(r)$ for non-negative real constants $r$, and a countable set $T$ of primitive values of some base type $\mathsf{T}$. Although this language is rather deficient (there is no way to use continuous distributions, for example, and no recursion), it will be sufficient to illustrate the different styles of semantics.

## 2.1 Operational Semantics

In operational semantics, the meaning of the program is defined by repeatedly transforming a term into other terms, until it reaches a value, a term which is considered finished, which is the output. Several variants of operational semantics can be seen, for example, in (2). A small-step trace-based operational semantics of the example language can be defined as follows.

First, an *environment* is defined as

$$E := \cdot \mid EM \mid VE,$$

so that an environment is a term with a single hole, where $M$ is a term and $V$ is a value. Substituting terms into environments is defined in the usual way. The single-step reduction relation can then be defined as

$$\begin{aligned}
(E[(\lambda x.M)V], t, s) &\to (M[V/x], t, s) \\
(E[M \oplus N], 0 : t, s) &\to (E[M], t, s) \\
(E[M \oplus N], 1 : t, s) &\to (E[N], t, s) \\
(E[\mathsf{score}(r)], t, s) &\to (E[(\lambda x.x)], t, rs),
\end{aligned}$$

where $t$ is a list of booleans, : is the cons operator, and $s$ is a real number. The probability of $M$ reaching a value $V$ can then be defined as

$$K \sum_{(M,t,1) \to^* (V,[],s)} 2^{-\mathrm{length}(t)} s,$$

where $K$ is a normalising constant to make the probabilities all add up to 1.

Alternatively, this could be defined in a different but closely related version of operational semantics, the big-step semantics:

$$\frac{(M, t_0, s_0) \Downarrow (\lambda x.M', t_1, s_1) \; (N, t_1, s_1) \Downarrow (N', t_2, s_2) \; (M'[N'/x], t_2, s_2) \Downarrow (O, t_3, s_3)}{(MN, t_0, s_0) \Downarrow (O, t_3, s_3)}$$

$$\frac{(M, 0 : t_0, s_0) \Downarrow (M', t_1, s_1)}{(M \oplus N, t_0, s_0) \Downarrow (M', t_1, s_1)} \qquad \frac{(N, 1 : t_0, s_0) \Downarrow (N', t_1, s_1)}{(M \oplus N, t_0, s_0) \Downarrow (N', t_1, s_1)}$$

$$\frac{}{(\mathsf{score}(r), t, s) \Downarrow (\lambda x.x, t, rs)}.$$

The notation in this case could be simplified somewhat, but that would obscure the connection to the small-step semantics. If $V$ is a value, then $(M, t_0, s_0) \to^* (V, t_1, s_1)$ iff $(M, t_0, s_0) \Downarrow (V, t_1, s_1)$.

## 2.2 Denotational Semantics

The general format of denotational semantics is a function $[\![M]\!]_\gamma$, from a term $M$ to its value, given the values of all the free variables in the context $\gamma$. The type of output depends on the type of the term, given by $[\![A]\!]$ which is defined recursively. For a non-probabilistic language, it would be something like $[\![\mathsf{T}]\!] =$

$T$, $[\![A \to B]\!] = [\![B]\!]^{[\![A]\!]}$.(1) If $M$ satisfies the typing judgement $\Gamma \vdash M : A$, and $\gamma \in [\![\Gamma]\!]$ (i.e. $\gamma$ is a function from the variables in $\Gamma$ to values such that if $x : B$ is in $\Gamma$, $\gamma(x) \in [\![B]\!]$), then $[\![M]\!]_\gamma \in [\![A]\!]$.

A key property that makes it denotational semantics (the result of the operational semantics given above could be reformatted into something like this, for example) is that the meaning of each term is defined in terms of the meanings of its subterms. (7, §3.1)

For the example language but without the probabilistic constructs (i.e. just simply-typed lambda calculus with constants), a denotational semantics could be defined as

$$\begin{aligned} [\![t]\!]_\gamma &= t \,(\text{for } t \in T) \\ [\![x]\!]_\gamma &= \gamma(x) \\ [\![MN]\!]_\gamma &= [\![M]\!]_\gamma([\![N]\!]_\gamma) \\ [\![\lambda x.M]\!]_\gamma(X) &= [\![M]\!]_{\gamma; x \mapsto X}. \end{aligned}$$

In order to define a denotational semantics for a probabilistic language, the result would have to be not just a value, but some sort of distribution of values. The rule $[\![A \to B]\!] = [\![B]\!]^{[\![A]\!]}$ also has to be changed to take into account that a function may produce outputs that differ randomly each time it is called, so something like the set of kernels from $[\![A]\!]$ to $[\![B]\!]$ (a trace-based denotational semantics would also probably be possible, but much more complicated, but as far as I am aware, such a thing has never been defined). This would require a measure structure on $[\![A]\!]$ and $[\![B]\!]$ to be defined, but in some cases it is not possible to define a measurable structure on the set of kernels between two measurable spaces such that the application map taking a kernel $A \rightsquigarrow B$ and an element of A is itself measurable, therefore the denotations of higher-order types could be undefined. This motivates the introduction of quasi-Borel spaces.(4)

### 2.2.1 Quasi-Borel Spaces

A quasi-Borel space is a pair of a base space $X$ with a set of random variables $M_X \subset X^{\mathbb{R}}$ that satisfies certain axioms. In many ways, quasi-Borel spaces are similar to measurable spaces. A $\sigma$-algebra on $X$ can be converted to a quasi-Borel space structure by taking all the measurable functions as random variables, and a quasi-Borel space can be converted to a measurable space by taking as measurable sets those whose inverse images under all random variables are measurable. There is a natural notion of QBS morphisms (functions whose compositions with random variables are random variables) and measures defined by combining measures on $\mathbb{R}$ with random variables which can be used to define integrals. The crucial difference to measurable spaces is that the set of morphisms betweem two QBSes is itself a QBS, so that the category of QBSes is cartesian closed.

Let $G(X)$ be the set of measures on a QBS $X$, equipped with its own QBS structure, with the dirac delta $\delta : X \to G(X)$ and $\eta : (X \to G(X)) \to (G(X) \to G(X))$ as the monad operations, and let $QBS(X, Y)$ be the set of morphisms

between two $QBSes$. Let $T$ be equipped with the discrete QBS structure. A denotational semantics for the example language, such that $[\![M]\!]_\gamma \in G([\![A]\!])$ if $\Gamma \vdash M : A$ and $\gamma \in [\![\Gamma]\!]$ can then be given as follows.

$$\begin{aligned}
[\![\mathsf{T}]\!] &= T \\
[\![A \to B]\!] &= [\![A]\!] \to G([\![B]\!]) \\
[\![t]\!]_\gamma &= \delta(t) \\
[\![x]\!]_\gamma &= \delta(\gamma(x)) \\
[\![MN]\!]_\gamma &= \eta(f \mapsto \eta(f)([\![N]\!]_\gamma))([\![M]\!]_\gamma) \\
[\![\lambda x.M]\!]_\gamma(X) &= [\![M]\!]_{\gamma; x \mapsto X} \\
[\![M \oplus N]\!]_\gamma &= \frac{1}{2}[\![M]\!]_\gamma + \frac{1}{2}[\![N]\!]\gamma \\
[\![\mathsf{score}(r)]\!]_\gamma &= r[\![\lambda x.x]\!]_\gamma = r\delta(x \mapsto \delta(x)).
\end{aligned}$$

Although the type of the result is different, this denotational semantics can still be shown to be equivalent to the operational semantics previously presented. For programs of type $\mathsf{T}$, both versions of the semantics produce a distribution on $T$ (one a QBS-style distribution and the other a measure-space-style distribution, but these can be converted), which are equal. The statement for function types is a bit more complicated and less direct, because the denotational semantics produces a distribution on (QBS) kernels, while the operational semantics produces a distribution on lambda terms.

Having a denotational semantics available makes certain sorts of proofs more convenient. Suppose there's a program rearrangement that can be expressed as replacing $A[M, N]$ with $B[M, N]$ (for arbitrary $M, N$), then the proof of equivalence can be quite direct, showing that the ways $[\![M]\!]$ and $[\![N]\!]$ are combined by $A$ and $B$ are equivalent (possibly subject to some conditions on certain variables not being free in $M$ and $N$, which implies that $[\![M]\!]_\gamma$ and $[\![N]\!]_\gamma$ are independent of the values of those variables in $\gamma$), whereas with operational semantics, it is much more complicated to keep track of the way the arbitrary subterms are duplicated, transformed and combined as execution progresses. For example, in (8), the equivalence of $\mathsf{let}\ x = A\ \mathsf{in}\ \mathsf{let}\ y = B\ \mathsf{in}\ C$ and $\mathsf{let}\ y = B\ \mathsf{in}\ \mathsf{let}\ x = A\ \mathsf{in}\ C$ was proven using a denotational semantics (subject to $x$ and $y$ not being free in $A$ and $B$). Similarly, the notion of contextual equivalence is not generally necessary in denotational semantics, being trivially equivalent to equality of values (though there are some cases where it could still apply, if some class of terms do not count as valid complete programs).

Another useful property of denotation semantics is that termination properties, to the extent that they exist, are much more directly visible. If the type system is strong enough to imply termination, this manifests directly as every term being given a value. If non-terminating terms are possible, it will be necessary to introduce, as part of the semantics, special cases to deal with that,(7, §5.2) otherwise the definition will simply not be consistent.

## 2.3 Trace-based semantics

A trace-based semantics for the example language was already described in Section 2.1. In trace-based semantics, the results are defined not just as distributions, but as random variables. Everything can be seen as taking place in a probability space (neglecting score which, at least if it is allowed to take values greater than 1, does not entirely fit), with the term after $n$ steps, the values of certain samples, the running time, and the final value all being random variables in the same space.

There are a few variations on how the traces work. In some versions (as is the case in Section 2.1, and (2)), each run of the program is assigned a trace of finite length, containing exactly those samples actually used in that run. If the set of results for an individual sampling operation is $S$ (in this case it's the set of 2 elements, but in a language with continuous distributions, it could be $\mathbb{R}$, or an interval, or something else), the set of traces is $\bigcup_{n\in\mathbb{N}} S^n$, and the measure defined on the set of traces is not actually a probability measure, as each set $S^n$ has measure 1. This still acts somewhat like a probability measure though, as traces of different lengths are mutually exclusive. If a trace $a$ is a prefix of a trace $b$ (with $b = a + c$), then it is not possible that $(M, a, 1) \to^* (V, [], s)$ and also $(M, b, 1) \to^* (V', [], s')$, because the evaluation with any given trace is deterministic, and $(M, b, 1) \to^* (V, c, s)$, at which point it could not proceed any further, having already reached a value. For any term, the set of traces for which it terminates therefore has measure at most 1.

Alternatively, the trace could contain infinitely many values, with any finite execution of a program leaving some of the samples unused. This is in some ways a little simpler, because the set of traces can simply have a probability distribution, and the trace does not need to depend in any way on the program (by having its length matched). An additional variant of this style of trace, used for example in (3), is to split the trace in three parts, all infinitely long, for the application case of the big-step semantics, rather than having the each step (evaluating the function, evaluating the argument, then evaluating the result of the $\beta$ reduction) use however many samples it needs from the start of the trace in turn.

## 2.4 Distribution-Based Semantics

Rather than defining the value of a program as a random variable and then defining the distribution based on that, it is possible to define the distribution of results directly. The example given above of operational semantics was distribution based (in the QBS sense of "distrbution"), but to more clearly illustrate the difference, I will give a distrbution-based operational semantics for the example language, using the same definition of environments as in the trace-based semantics.

$$
\begin{aligned}
s(E[(\lambda x.M)V]) &= \delta(M[V/x]) \\
s(E[M \oplus N]) &= \frac{1}{2}\delta(E[M]) + \frac{1}{2}\delta(E[N]) \\
s(E[\mathsf{score}(r)]) &= r\delta(E[(\lambda x.x)]) \\
s(V) &= \delta(V).
\end{aligned}
$$

The multi-step function $s^n$ can then be defined as $s^0(M) = M$, $s^{n+1}(M) = \int s^n(x)\,\mathrm{d}s(M)(x)$, then $s^\infty(M)$ can be defined as $\lim_{n\to\infty} s^n(M)$, which (after re-normalising) gives the distribution of final values. In this case, everything terminates because of the type system, therefore the limit is always well-defined. For an untyped language, or one with an unrestricted recursion operator, the definition of the final result would be a little more complicated, but the general idea is the same.

The distinction between distribution and trace-based semantics is discussed in (2), which gives both for the same language, and proves their equivalence (in the sense that the distributions derived from the trace-based semantics are the same as the distributions that the distribution-based semantics produces directly).

# 3 My Previous Research

My thesis will be building on the paper I recently submitted to LICS (in collaboration with Luke Ong), Supermartingales, Ranking Functions and Probabilistic Lambda Calculus. In it, we extend existing work on proving almost-sure termination using ranking functions (functions from the program state to non-negative real numbers which at each step have a non-positive expected change, and which satisfy some further progress condition) to the context of a functional language with continuous distributions. Because the language used is simply-typed, the only way that non-termination is possible is using the explicit recursion contruct, $\mathsf{Y}$, therefore adding a condition that the ranking function must decrease by 1 for every $\mathsf{Y}$-reduction step provides a bound on the expected number of $\mathsf{Y}$-reduction steps, which implies almost sure termination. This result is then extended in three ways (which can all be combined).

The first extension is sparse ranking functions. A sparse ranking function may be defined at a subset of the reachable states of the program, with the condition instead being that the expected value of the ranking function at the next step where it is defined is not greater than the current value (and, if there are intermediate $\mathsf{Y}$-reduction steps, there is a corresponding strict decrease). Although this does not increase the strength of the technique overall, it makes it easier to apply, because the ranking function only needs to be defined at those points in the program's execution which are actually interesting. This extension is new, having not been applicable in the context of first-order languages because the completed body of a while loop provides a natural set of checkpoints.

The second extension is antitone ranking functions. This is closely related to the progress condition given in (6), just adapted to the new functional context. A function $\epsilon : \mathbb{R}_{\geq 0} \to \mathbb{R}_{>0}$, which is required to be monotoically decreasing, is added to the definition, and rather than the ranking function having to decrease by 1 at every $\mathsf{Y}$-reduction step, it only has to decrease by $\epsilon$ of the current value of the ranking function. This allows the termination of programs which terminate much more slowly (such as the unbiased random walk) to be proven, whereas the basic result was only applicable to terms which terminated with a finite expected number of $\mathsf{Y}$-reduction steps.

The third extension, and the most interesting and relevant to my intended thesis topic, is ranking functions with respect to alternative reduction strategies. This result is proven using a variant of trace semantics which is able to satisfy a restricted confluence property. Probabilistic functional languages in general do not have the same sort of confluence properties as pure lambda calculus. For example, assuming that samples are drawn from the uniform distribution on $[0, 1]$, the term $(\lambda x.x + x)\mathsf{sample}$ evaluates either to the uniform distribution on $[0, 2]$ or to the triangle-shaped distribution with the same support, with its peak at 1, depending on whether the sample is evaluated first (in which case, one value is doubled) or the $\beta$-redex is evaluated first (in which case, two independent samples are taken and added). If, however, certain reductions are excluded (in the results I proved in this paper, it's $\beta$-reductions whose arguments aren't values, and $\mathsf{sample}$-reductions inside $\lambda$s either inside $\mathsf{Y}$s or in the argument of applications, although other criteria would also be possible), confluence can be re-gained.

This is enough to ensure that the distribution of results is (barring cases where some reduction strategies may terminate while others don't) the same, but that would be difficult to prove directly, and the result doesn't extend to the same traces giving the same results. For example, if the trace is simply a sequence of samples which are used in the order in which the $\mathsf{sample}$-reductions occur, then the term $\mathsf{sample} - \mathsf{sample}$ reduces with the trace $0, 1, ...$ to either 1 or $-1$, depending on which $\mathsf{sample}$ is reduced first. This example can be fixed by labelling the samples in the trace by positions in the term rather than the order in which they're used, but in other cases, that would still not be sufficient. Consider, for example, $(\lambda f.f0 + f0)(\lambda x.\mathsf{sample})$. This reduces (by some reduction strategies) to $\mathsf{sample} + \mathsf{sample}$, but the $\mathsf{sample}$s here don't correspond in a unique way to the $\mathsf{sample}$s in the initial term, therefore labelling samples by positions in the initial term isn't sufficient. Perhaps it would be possible to in some way label those $\mathsf{sample}$s by a combination of the position of the $\mathsf{sample}$ they're derived from and the occurrences of the variable $f$ that they were substituted in to, but the approach I ended up taking was a somewhat more brute-force solution, in that if any labelling scheme would have worked, so would this one, by design. Samples are labelled by a combination of a term reachable from the initial term (technically a reachable skeleton instead, with all the real numbers removed, for measure-theoretic reasons) and a position within that term, with an equivalence relation defined on these (reachable term, position) pairs that is the minimum required to ensure confluence.

This definition is sufficient to prove the AST result that is nominally the main focus of that paper, but also seems like something that has a lot of potential to be used for other purposes. The limits of confluence for probabilistic functional languages is not something I have seen discussed extensively elsewhere, except the fact that some reduction strategies (specifically call-by-name and call-by-value) give different results. The way that confluence is obtained with a trace-based semantics also seems likely to extend to a denotational trace semantics.

# 4 Planned research

## 4.1 Using Labelling to Combine Denotational and Trace Semantics

Trace-based semantics is, in a sense, finer than distribution-based semantics. Given a measurable function $f : \Lambda \times \mathbb{S} \to A$ that describes some property of a term $\in \Lambda$ by using a trace from some sampling space $\mathbb{S}$, which has some fixed probability measure on it $\mu_{\mathbb{S}}$, this can be converted to a kernel from $\Lambda$ to $A$ using the push-forward measure of $\mu_{\mathbb{S}}$, but some information is lost in the process: which specific trace lead to which outcome. There is no corresponding simple way to convert a distribution-based semantics to a trace-based semantics that actually results in all the correct structure expected of a trace-based semantics. In trace-based semantics, it is possible to define things like correlations between the random variables it defines, because they are all defined in the same probability space, whereas this is not possible given merely a distribution of outputs.

There are also some advantages of denotational over operational semantics. It provides meanings to sub-programs that can be reasoned about independently of their context. It can be more suitable for proving things like the effectiveness of a type system for proving termination, as the structure of the semantics more closely matches the derivations of typing judgements.

Existing approaches are either denotational and distribution-based, operational and distribution-based, or operational and trace-based. The combination of denotational and trace-based is more difficult to define, because some way of distinguishing multiple copies of the same sample statement is needed, so that they can derive their randomness from different parts of the trace. In operational semantics, this is easy enough, because the argument of a beta redex does not need to be evaluated at all before it is duplicated by the reduction, and its copies can simply be dealt with one at a time, like any other independent subterms. In denotational semantics, it is not the terms themselves which are combined in an application, but their interpretations, therefore the interpretation of the argument needs some way to retain the possibility of having multiple different results from the random process, even though there is only one trace used overall.

An extension of the labelling scheme used in the confluent trace semantics of my previous paper could provide just such a possibility. The exact labelling

scheme used there depends explicitly on the reduction sequences (and is also rather convoluted), so it is not suitable to use directly in defining a denotational semantics, but it seems highly likely that some labelling scheme that's equivalent but differently defined would be. If all of the subterms of the original term are given unique labels, and the labels form some sort of magma, so that when a reduction occurs, the labels of the argument's subterms and the function's subterms can be combined in some way to produce the labels of the reduct's subterms, then a trace that contains a value for every possible combination of labels could be used. The random samples taken in the evaluation of each subterm then depend on the transformation of the labels that occurs, which is distinct for each copy of the subterm.

## 4.2 Using labelling to perform efficient inference

Another possible application of a labelling scheme for sample is in improving the efficiency of certain inference algorithms. In lightweight metropolis-hastings, the program is first executed, picking random values for all the samples taken, with the trace of samples being remembered. Then, random variations on the trace are tested, re-running the program with the modified trace and accepting or rejecting the modification based on the difference in the likelihood of the runs. It is essential to the efficiency of this algorithm that the score of a modified run is highly correlated to the score of the original run, as otherwise the acceptance ratio becomes low. At the same time, it is necessary that the proposed changes actually change the trace by not too small an amount, so that the distribution of traces converges to the stationary distribution quickly.

The correlation between runs can be impeded if the number of samples taken varies depending on the results of earlier random choices. In the basic version of the algorithm, the trace is simply a list of the samples in the order in which they are taken. If some sample is never used, however, the rest of the samples in the list become misaligned, and end up taking different roles in the resulting execution, completely changing the score with only a small change to the trace.

This, then, is another possible application of a labelling scheme similar to the one previously mentioned. By better approximating the roles that samples play in the program execution by the labels, the problem of trace misalignment can be reduced. In (5), a similar approach is taken for variational inference with a first-order language, but in that case, they require that the labels be explicitly provided in the program (potentially being computed using the values of other variables). There is also a similar proposal in (9).

The requirements of a labelling scheme for this purpose are somewhat different from the previous cases. In the other cases, the labels needed to match across program executions that differ nondeterministically in their reduction order, but for the purposes of inference, the relevant different executions differ by the randomly selected samples. A labelling scheme adapted to the problem of inference could probably be simplified relative to the confluent trace semantics defined in my previous paper, or the more algebraic variant I intend to develop for the denotational trace semantics mentioned earlier. I am not yet sure how

similar it would end up, after this simplification, to the much simpler labelling scheme described in (9).

### 4.3 Completeness of Antitone Rankability

A term is rankable iff every term reachable from it terminates with finitely many $\mathsf{Y}$-reduction steps in expectation. This essentially puts a limit on how slowly a term can terminate while still being rankable, so any term which is not $\mathsf{Y}$-positive almost surely terminating cannot be proven AST by assigning it a ranking function. With antitone ranking functions, there is no analogous restriction: there are arbitrarily slowly terminating terms (i.e. for any probability distribution on $\mathbb{N}$, there is a term whose distribution of number of $\mathsf{Y}$-reduction steps before termination is the given distribution) which are antitone rankable. It therefore seems rather likely that any term such that every term reachable from is is AST is antitone rankable. I intend to try to prove this result, even though it does not relate very closely to my overall thesis topic, because it would make my existing work more complete.

## References


[1] Henk Barendregt, Wil Dekkers, and Richard Statman. *Lambda calculus with types*. Cambridge University Press, 2013.

[2] Johannes Borgström, Ugo Dal Lago, Andrew D Gordon, and Marcin Szymczak. A lambda-calculus foundation for universal probabilistic programming. *ACM SIGPLAN Notices*, 51(9):33–46, 2016.

[3] Ryan Culpepper and Andrew Cobb. Contextual equivalence for probabilistic programs with continuous random variables and scoring. In Hongseok Yang, editor, *Programming Languages and Systems*, pages 368–392, Berlin, Heidelberg, 2017. Springer Berlin Heidelberg. ISBN 978-3-662-54434-1.

[4] C. Heunen, O. Kammar, S. Staton, and H. Yang. A convenient category for higher-order probability theory. In *2017 32nd Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*, pages 1–12, 2017. doi: 10.1109/LICS.2017.8005137.

[5] Wonyeol Lee, Hangyeol Yu, Xavier Rival, and Hongseok Yang. Towards verified stochastic variational inference for probabilistic programs. *Proceedings of the ACM on Programming Languages*, 4(POPL):1–33, 2019.

[6] Annabelle McIver, Carroll Morgan, Benjamin Lucien Kaminski, and Joost-Pieter Katoen. A new proof rule for almost-sure termination. *Proceedings of the ACM on Programming Languages*, 2(POPL):1–28, 2017.

[7] Peter D. MOSSES. Chapter 11 - denotational semantics. In JAN VAN LEEUWEN, editor, *Formal Models and Semantics*, Handbook of Theoretical Computer Science, pages 575–631. Elsevier, Amsterdam, 1990.

ISBN 978-0-444-88074-1. doi: https://doi.org/10.1016/B978-0-444-88074-1.50016-0. URL https://www.sciencedirect.com/science/article/pii/B9780444880741500160.

[8] Sam Staton. Commutative semantics for probabilistic programming. In *European Symposium on Programming*, pages 855–879. Springer, 2017.

[9] David Wingate, Andreas Stuhlmueller, and Noah Goodman. Lightweight implementations of probabilistic programming languages via transformational compilation. In Geoffrey Gordon, David Dunson, and Miroslav Dudk, editors, *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics*, volume 15 of *Proceedings of Machine Learning Research*, pages 770–778, Fort Lauderdale, FL, USA, 11–13 Apr 2011. JMLR Workshop and Conference Proceedings. URL http://proceedings.mlr.press/v15/wingate11a.html.