Higher order programming with probabilistic effects: A model of stochastic memoization and name generation

Younesse Kaddar Sam Staton
Department of Computer Science, University of Oxford, UK

This abstract is about probabilistic programming, which is a method of programming statistical models and machine learning models. By combining higher-order functions, we can specify increasingly complex models. This abstract focuses on stochastic memoization, a higher-order method that is simple and useful in practice, but semantically elusive, particularly regarding dataflow transformations.

Deterministic versus stochastic memoization. Deterministic memoization, for a function f, is the process of storing the result when f is applied to an argument, so that we can reuse it later when the same call is made [Mic68]. Since we only need to compute results once, this leads to a speed-up of the program, but it does not change its semantics. However, in the presence of probabilistic effects, memoizing is no longer just an optimization technique. It does change the semantics [Roy+08; Sta21; Woo+09], enabling us to define infinite random sequences, which are of paramount importance in probability and statistics.

Typed stochastic memoization with monads. Suppose we have a monad for probabilistic effects, Prob. *Stochastic memoization* is a higher-order function with probabilistic effects, of type $\mathsf{mem} :: (\mathsf{a} \to \mathsf{Prob\ b}) \to \mathsf{Prob}\ (\mathsf{a} \to \mathsf{b})$. For example, suppose we start with a function which randomly picks a number in $[0,1]$ every time it is called, $\mathbf{const}\ (\mathsf{uniform}\ 0\ \ 1)\ :: \ \mathbb{R} \to \mathsf{Prob}\ \mathbb{R}$. Then the memoized function randomly assigns a number to every input number, $\mathsf{mem}\ (\mathbf{const}\ (\mathsf{uniform}\ 0\ \ 1))\ :: \ \mathsf{Prob}\ (\mathbb{R} \to \mathbb{R})$. This is sometimes called white noise. White noise is an important first example; we give a more substantial motivating example based on random graphs in Section 1. In practice, memoization is crucial in Church [GS14] and WebPPL [GS14], and in this typed form with monads in our library LazyPPL [Sta+].

Data flow properties. Stochastic memoization is easy to implement using state. (When a is enumerable, in LazyPPL we can also use laziness and tries, following [Hin].). Unlike a fully stateful language, however, stochastic memoization is still compatible with commutativity / data flow program transformations:

$$\mathsf{x} \leftarrow \mathsf{t}\ ;\ \mathsf{y} \leftarrow \mathsf{u} \quad = \quad \mathsf{y} \leftarrow \mathsf{u}\ ;\ \mathsf{x} \leftarrow \mathsf{t} \qquad \text{where } \mathsf{x} \notin \mathsf{freevars}(\mathsf{u}),\ \mathsf{y} \notin \mathsf{freevars}(\mathsf{t})$$

These transformations are very useful in program optimization and inference algorithms. On the foundational side, data flow is a fundamental concept that corresponds to monoidal categories. The challenge addressed in this work is to validate these transformations.

Challenges for probability theory. On the semantic side, these transformations are not trivial. Informally, stochastic memoization appears related to Kolmogorov's extension theorem, which relates probability measures on infinite product spaces to probability measures on the projections from the product. Arguably, then, stochastic memoization validates dataflow transformations because it is not *intrinsically* stateful, rather, it is linked to a fundamental part of pure probability theory. However, traditional probability theory does not actually support higher-order functions [Aum61], and higher-order probability is a burgeoning field [CJ19; Fri20; Heu+17; Koc11; Ste21]. Existing models of higher-order probability do not support stochastic memoization, and this is the contribution of our work: a first model of stochastic memoization with a non-enumerable type and validating the dataflow transformations.

1 Stochastic Memoization

Stochastic memoization is, on the practical side, a convenient way to implement infinite sequences of random variables, point processes, clustering and, more generally, nonparametric Bayesian models in probabilistic programming [Roy+08;

Submitted to:
HOPE 2022

Woo+09]. On the theoretical side, stochastic memoization may become of utmost importance to obtain general representation theorems for exchangeable data types, similar to de Finetti's theorem [Fin37] (for exchangeable sequences) or the Aldous-Hoover theorem [Ald81; Hoo79] (for exchangeable arrays). Consider the following example.

Exchangeable random graphs. Random arrays $(X_{i,j})_{i,j}$ of Boolean random variables (where i and j range over countable collections of potentially related objects) are a useful data analysis tool to model relational data, which are observations of binary relationships between collections of objects, *e.g.* graph social networks, world wide web, biochemical pathways, etc. When we have a single (countable) collection of objects that we wish to compare pairwise (*i.e.* i and j range over the same countable index), the array $(X_{i,j})_{i,j}$ can be seen as the adjacency matrix a random graph. Such random adjacency matrices $(X_{i,j})_{i,j}$ are said to be *exchangeable* when their joint distribution is invariant under relabeling the nodes of the corresponding graph. If the random graph is assumed to be simple (*i.e.* $(X_{i,j})_{i,j}$ is symmetric and has zero diagonal), the celebrated Aldous-Hoover theorem states that every corresponding exchangeable adjacency matrix can be parametrized by a (random) measurable function $G\colon [0,1]^2 \to [0,1]$ called a *graphon*. If u_i, u_j are iid random variables corresponding to two nodes i, j in the random simple graph, $G(u_i, u_j)$ gives the probability that there is an edge between i and j. As a result, Bayesian models modeling exchangeable simple graphs are entirely determined (in distribution) by a prior on the space of graphons.
Consider a typeclass RandomGraph, corresponding to an abstract data type whose interface allows us to draw vertices at random (with newVertex) from a random graph with a countable set $V =$ v of vertices, and inspect the presence of edges between vertices (with isEdge). In the spirit of the Aldous-Hoover representation theorem, for every graphon $G =$ graphon$\colon [0,1]^2 \to [0,1]$, we can write an implementation for this interface in our library LazyPPL [Sta+]:

```
class  RandomGraph v where
  type Graph v
  newVertex ::  Prob v
  newGraph:: Prob (Graph v)
  isEdge ::  Graph v → (v,  v)  → Bool

class  Graphon where
  graphon  ::  (ℝ,  ℝ)  → ℝ
```

```
instance Graphon => RandomGraph ℝ where
  type Graph Double = (ℝ, ℝ)  → Bool
  newVertex = uniform

  -- return  a  randomly sampled function  '(ℝ,  ℝ)  → Bool'
  newGraph = mem $ bernoulli .  graphon

  isEdge  g  (x,  y)  = g  (x,  y)
```

assuming we have a memoization function mem :: (a → Prob b) → Prob (a → b). The idea is that, once an edge between x and y has been sampled (with probability graphon (x, y)), its presence (or absence) remains unchanged in the rest of the program, hence the need to memoize the result. Staton showed that such graphon-based implementations are the only ones up to contextual equivalence, provided that they satisfy the dataflow property:

Theorem 1.1 (Staton's Aldous-Hoover representation [Sta20])**.** *An implementation for the interface RandomGraph satisfies the dataflow property (*i.e. *program lines can be reordered (commutativity) and discarded (affineness) as long as the dataflow is preserved) iff it is observationally equivalent to a graphon implementation.*

The proof involves building a categorical model based on the *Rado graph* (the countable graph that embeds every at most countable graph), paving the way for a very general approach to tackle similar representation problems.

2 Semantics

Stochastic functions are interpreted as probabilistic kernels $f\colon X \to PY$, where P is a probability monad, in a suitable cartesian closed category (to model higher-order functions), *e.g.* the category of quasi-Borel spaces ([Heu+17]). Stochastic memoization (simply referred to as 'memoization' henceforth) is then internally expressed as a morphism $\mathrm{mem}_{X,Y}\colon (PY)^X \longrightarrow P(Y^X)$ converting a probabilistic kernel (which associates, for every given input in X, a random output in Y) into a random function $X \to Y$ (randomly choosing all the outputs for all the possible inputs at once). If f is written as a lambda-abstraction $\lambda x.\, u\colon X \to PY$, $\mathrm{mem}_{X,Y}(f)$ will be denoted by $\lambda_{מ} x.\, u$.

$$\begin{array}{l} \mathsf{f} \leftarrow \lambda_{\text{מ}} x.\, u \\ \mathsf{f}\ \ \mathsf{n} \end{array} \quad \overset{\text{one sample}}{=} \quad u[\mathsf{n}/x] \qquad\qquad \begin{array}{l} \mathsf{f} \leftarrow \lambda_{\text{מ}} x.\, u \\ \mathsf{v}_1 \leftarrow \mathsf{f}\ \ \mathsf{n} \\ \mathsf{w} \leftarrow \mathsf{f}\ \ \mathsf{m} \\ \mathsf{v}_2 \leftarrow \mathsf{f}\ \ \mathsf{n} \\ \textbf{return}\ (\mathsf{v}_1,\ \mathsf{w},\ \mathsf{v}_2) \end{array} \quad \overset{\text{several samples}}{=} \quad \begin{array}{l} \mathsf{v} \leftarrow u[\mathsf{n}/x] \\ \mathsf{w} \leftarrow u[\mathsf{m}/x] \\ \textbf{return}\ (\mathsf{v},\ \mathsf{w},\ \mathsf{v}) \end{array}$$

For finite types X, memoization is a matter of simply sampling a value of $f(x)$ for all inhabitants of $x \in X$, and returning the assignment as a finite mapping. For countable types, we can sometimes successfully take advantage of the host language's laziness to circumvent the issue of manipulating potentially infinite structures (this is done in LazyPPL). But what about uncountable types X? (We cannot just range over an uncountable space for all $x \in X$ anymore.) The category of quasi-Borel spaces is unfortunately known not to support memoization in general [Sta21]. The first implementation of 'memoize' that comes to mind uses state to store previously seen values. But usually, using memory compromises the dataflow property (the state monad is not commutative). However, we conjecture that 'memoize' is a special kind of stateful operation that do preserve the dataflow property:

Proposition 2.1. *Stochastic memoization still admits the dataflow property (even if one resorts to hidden state).*

To give a semantics to stochastic memoization for Boolean-valued functions (to start small), the idea is that we may wish to have the following interface:

```
new_atom :: 𝔸 -- Atoms (randomly generated fresh  names)
new_function :: 𝔽 -- Function  labels :  type  to  be  thought  of  as  𝔸  → Bool
(@) :: (𝔽, 𝔸) → Bool -- Application  operator  making every  function  memoized: type of  a  bipartite  graph
```

where every function from a set of atoms $\mathbb{A}$ (each one of which can be randomly generated) to **Bool** would be defunctionalized and viewed as an inhabitant of a type $\mathbb{F}$ (which would then be thought of as $\mathbb{A} \to$ **Bool**). Applying a function to an argument and memoizing the result would then be made possible by an 'apply' operator (@) :: $\mathbb{F} \times \mathbb{A} \to$ **Bool**. But requiring that the results be memoized is precisely saying that (@) :: $\mathbb{F} \times \mathbb{A} \to$ **Bool** ought to be seen as the 'edge' relation of a bipartite graph (bigraph, for short) with set $\mathbb{F}$ of left nodes and $\mathbb{A}$ of right nodes – whose edges are such that their presence (or absence) remain unchanged after being sampled, exactly like isEdge in RandomGraph. Analogously to the Rado topos setting, this suggests, on the denotational semantics side, that we are looking for a topos where a random countable bigraph would play the role of the Rado graph in the Rado topos.

3 Toy language for stochastic memoization and name generation

One way to prove proposition 2.1 is to exhibit a denotational model, which we attempt in this section, in a restricted setting. We consider a small simply typed language to shed light on three features that we model semantically:

- **name generation**: we can generate fresh names – referred to as *atomic* names or *atoms*, in the sense of Pitt's nominal set theory [Pit13] – with constructs such as let $x = \mathsf{fresh}()$ in $\cdots$.
- basic **probabilistic effects**: for illustrative purposes, the only distribution we consider, as a first step, is the Bernoulli distribution with bias $p = 1/2$. Constructs like let $b = \mathsf{flip}()$ in $\cdots$ amount to flipping a fair coin and storing its result in a variable b.
- **stochastic memoization**: if a stochastic function f – memoized with the new $\lambda_{\text{מ}}$ operator – is called twice on the same argument, it should return the same result.

We have the following base types: bool (booleans), $\mathbb{A}$ (atomic names), and $\mathbb{F}$ (intended for memoized functions $\mathbb{A} \to$ bool). For the sake of simplicity, we do not have arbitrary function types.

$$A, B \ ::= \ \mathsf{bool} \mid \mathbb{A} \mid \mathbb{F} \mid A \times B$$

In fine-grained call-by-value fashion [Lev06], there are two kinds of judgments: typed values, and typed computations.

Values:

$$\frac{-}{\Gamma, x : A \vdash^{\mathsf{v}} x : A} \qquad \frac{\Gamma \vdash^{\mathsf{v}} v : A \qquad \Gamma \vdash^{\mathsf{v}} w : B}{\Gamma \vdash^{\mathsf{v}} (v, w) : A \times B} \qquad \frac{-}{\Gamma \vdash^{\mathsf{v}} \mathsf{true} : \mathsf{bool}} \qquad \frac{-}{\Gamma \vdash^{\mathsf{v}} \mathsf{false} : \mathsf{bool}}$$

Computations:

$$\frac{\Gamma \vdash^{\mathsf{v}} v : A}{\Gamma \vdash^{\mathsf{c}} \mathsf{return}(v) : A} \qquad \frac{\Gamma \vdash^{\mathsf{c}} u : A \qquad \Gamma, x : A \vdash^{\mathsf{c}} t : B}{\Gamma \vdash^{\mathsf{c}} \mathsf{let\ val}\ x \leftarrow u\ \mathsf{in}\ t : B}$$

Matching:

$$\frac{\Gamma \vdash^{\mathsf{v}} v : \mathsf{bool} \qquad \Gamma \vdash^{\mathsf{c}} u : A \quad \Gamma \vdash^{\mathsf{c}} t : A}{\Gamma \vdash^{\mathsf{c}} \mathsf{if}\ v\ \mathsf{then}\ u\ \mathsf{else}\ t : A} \qquad \frac{\Gamma \vdash^{\mathsf{v}} v : A \times B \qquad \Gamma, x : A, y : B \vdash^{\mathsf{c}} t : C}{\Gamma \vdash^{\mathsf{c}} \mathsf{match}\ v\ \mathsf{as}\ (x, y)\ \mathsf{in}\ t : C}$$

Language-specific commands:

$$\frac{-}{\Gamma \vdash^{\mathsf{c}} \mathsf{flip}() : \mathsf{bool}} \qquad \frac{-}{\Gamma \vdash^{\mathsf{c}} \mathsf{fresh}() : \mathbb{A}} \qquad \frac{\Gamma \vdash^{\mathsf{v}} v : \mathbb{A} \qquad \Gamma \vdash^{\mathsf{v}} w : \mathbb{A}}{\Gamma \vdash^{\mathsf{c}} (v = w) : \mathsf{bool}}$$

$$\frac{\Gamma \vdash^{\mathsf{v}} v : \mathbb{F} \qquad \Gamma \vdash^{\mathsf{v}} w : \mathbb{A}}{\Gamma \vdash^{\mathsf{c}} (v@w) : \mathsf{bool}} \qquad \frac{\Gamma, x : \mathbb{A} \vdash^{\mathsf{c}} u : \mathsf{bool}}{\Gamma \vdash^{\mathsf{c}} \lambda_{\text{מּ}} x.\, u : \mathbb{F}}$$

We work in the (cartesian closed) category of covariant presheaves on the category $\mathbf{BiGrph}_{\mathit{emb}}$ of finite bipartite graphs (henceforth called *bigraphs*) and embeddings (that do not add or remove edges). For a bigraph g, we denote by g_L (resp. g_R) and E^g its set of left (resp. right) nodes and its edge relation.
The denotation of basic types is given by:

$$[\![\mathbb{F}]\!] = \mathbf{BiGrph}_{\mathit{emb}}(\circ, -) \qquad [\![\mathbb{A}]\!] = \mathbf{BiGrph}_{\mathit{emb}}(\bullet, -)$$

where $\circ$ and $\bullet$ are the one-vertex left and right graphs respectively. The denotation of the type of booleans is the constant presheaf $2 \cong 1 + 1$, as usual. For a bigraph g and a presheaf $X = [\![\mathcal{X}]\!]$, $X(g)$ is thought of as the set of generative models/programs of type $\mathcal{X}$ that may use the bigraph g, in the following sense: probabilistic function (that we want to memoize) and atom labels are stored as left and right nodes respectively. The presence (resp. absence) of an edge between a given left and right node memoizes the fact that a probabilistic call of the corresponding function on the corresponding atom has resulted in true (resp. false). For every embedding $\iota\colon g \hookrightarrow g'$, the function $X\iota\colon X(g) \to X(g')$ models substitution in the programs in $X(g)$ according to ι.

4 Probabilistic local state monad

Inspired from Plotkin and Power's local state monad [PP02] (which was defined on the covariant presheaf category $[\mathbf{Inj}, \mathbf{Set}]$, where $\mathbf{Inj}$ is the category of finite sets and injections), we model probabilistic and name generation effects by a new monad that we name 'probabilistic local state monad'. In the following, $X, Y, Z\colon \mathbf{BiGrph}_{\mathit{emb}} \to \mathbf{Set}$ denote presheaves, $g = (g_L, g_R, E^g), g', h, h' \in \mathbf{BiGrph}_{\mathit{emb}}$ bigraphs, and $\iota, \iota'\colon g \hookrightarrow g'$ bigraph embeddings. We will omit subscripts when they are clear from the context.

Definition 4.1 (Probabilistic local state monad)**.** For all covariant presheaf $X\colon \mathbf{BiGrph}_{\mathit{emb}} \to \mathbf{Set}$ and bigraph $g \in \mathbf{BiGrph}_{\mathit{emb}}$:

$$T(X)(g) \;:=\; \left(P_{\mathrm{f}} \int^{g \hookrightarrow h} X(h) \times [0,1]^{(h-g)_L}\right)^{[0,1]^{g_L}}$$

where P_{f} is the finite distribution monad.

The assignment T is similar to the read-only local state monad, except that any fresh node can be initialized. Every $\lambda \in [0,1]^{g_L}$ is thought of as the probability of the corresponding function/left node being true on a new fresh atom. We will refer to such a λ as a *state of biases*. The coend takes care of garbage collection.

Notation 4.2. Equivalence classes in $\int^{g \hookrightarrow h} X(h) \times [0,1]^{(h-g)_L}$ are written $[x_h, \lambda^h]_g$. We use Dirac's bra-ket notation $\big|[x_h, \lambda^h]_g\big\rangle_h$ to denote a formal column vector of equivalence classes ranging over a finite set of h's. As such, a formal convex sum $\sum_i p_i[x_{h_i}, \lambda^{h_i}]_g \in P_{\mathrm{f}} \int^{g \hookrightarrow h} X(h) \times [0,1]^{(h-g)_L}$ will be concisely denoted by $\big\langle \overrightarrow{p} \,\big|\, [x_h, \lambda^h]_g \big\rangle_h$.

Definition 4.3 (Action of $T(X)$ on morphisms)**.**

$$T(X)(g \overset{\iota}{\hookrightarrow} g') = \begin{cases} \left(P_{\mathrm{f}} \int^{g \hookrightarrow h} X(h) \times [0,1]^{(h-g)_L}\right)^{[0,1]^{g_L}} \longrightarrow \left(P_{\mathrm{f}} \int^{g' \hookrightarrow h'} X(h') \times [0,1]^{(h'-g')_L}\right)^{[0,1]^{g'_L}} \\ \vartheta \longmapsto \lambda' \mapsto \text{let } \vartheta(\lambda' \iota_L) \;=\; \big\langle \overrightarrow{p} \,\big|\, [x_h, \lambda^h]_g \big\rangle_h \text{ in } \Big\langle \overrightarrow{p} \,\Big|\, [X(h \hookrightarrow h \textstyle\coprod_g g')(x_h), \lambda^h]_{g'} \Big\rangle_h \end{cases}$$

Theorem 4.4. *The construction T is functorial and can be endowed with the structure of a monad.*

We now use this monad to give a denotational semantics to our language.

5 Categorical semantics

In our language, the denotational interpretation of values, computations (return and let binding), and matching (elimination of bool's and product types) is standard. We interpret computation judgements $\Gamma \vdash^{\mathsf{c}} t : A$ as morphisms $[\![\Gamma]\!] \to T([\![A]\!])$, by induction on the structure of typing derivations. The context Γ is built of bool's, $\mathbb{A}$ and $\mathbb{F}$ and products. Therefore, $[\![\Gamma]\!]$ is isomorphic to an object of the form $2^k \times \mathbf{BiGrph}_{\mathit{emb}}(\circ, -)^\ell \times \mathbf{BiGrph}_{\mathit{emb}}(\bullet, -)^m$.

Definition 5.1. For every bigraph g, we denote by R_g (resp. L_g) the set of bigraphs $h \in g/\mathbf{BiGrph}_{\mathit{emb}}$ having one more right (resp. left) node than g, and that are the same otherwise.

$$\begin{aligned} R_g &:= \{\, h \in \mathbf{BiGrph}_{\mathit{emb}} \mid h_L = g_L,\, g_R \subsetneq h_R \text{ and } |h_R| = |g_R| + 1 \,\} \\ L_g &:= \{\, h \in \mathbf{BiGrph}_{\mathit{emb}} \mid h_R = g_R,\, g_L \subsetneq h_L \text{ and } |h_L| = |g_L| + 1 \,\} \end{aligned}$$

Denotation of $\Gamma \vdash^{\mathsf{c}} \mathsf{fresh}() : \mathbb{A}$

The map $[\![\mathsf{fresh}]\!]_g : [\![\Gamma]\!](g) \to T([\![\mathbb{A}]\!])(g)$ randomly chooses connections to each left node according to the state of biases, and makes a fresh right node with those connections.

$$[\![\mathsf{fresh}]\!]_g \colon \begin{cases} 2^k \times \mathbf{BiGrph}_{\mathit{emb}}(\circ, g)^\ell \times \mathbf{BiGrph}_{\mathit{emb}}(\bullet, g)^m \longrightarrow P_{\mathrm{f}}(g_R + 2^{g_L})^{[0,1]^{g_L}} \\ \_, \_, \_ \mapsto \lambda \mapsto \left\langle \dfrac{1}{Z} \displaystyle\prod_{\mathfrak{f} \in g_L} \lambda(\mathfrak{f})^{E^h(\mathfrak{f}, a_h(\bullet))} (1 - \lambda(\mathfrak{f}))^{1 - E^h(\mathfrak{f}, a_h(\bullet))} \,\middle|\, [\underbrace{\bullet}_{\cong (h-g)_R} \overset{a_h}{\hookrightarrow} h,\, !]_g \right\rangle_{h \in R_g} \end{cases}$$

where Z is a normalization constant.

Denotation of $\Gamma \vdash^{\mathsf{c}} \lambda_{\text{מ}} x.\, u : \mathbb{F}$

As $\lambda_{\text{מ}}$-abstractions are formed based on computation judgements of the form $\Gamma, x : \mathbb{A} \vdash^{\mathsf{c}} u : \mathsf{bool}$, we first note that

$$T([\![\mathsf{bool}]\!])g \;\cong\; P_{\mathrm{f}}(2)^{[0,1]^{g_L}} \;\cong\; [0,1]^{[0,1]^{g_L}}$$

Also, we can decompose the extra variable x in the environment $\Gamma,\, x : \mathbb{A}$, the denotation of which is of the form $[\![\Gamma,\, x : \mathbb{A}]\!](g) \;=\; 2^k \times \mathbf{BiGrph}_{\mathit{emb}}(\circ, g)^\ell \times \mathbf{BiGrph}_{\mathit{emb}}(\bullet, g)^m \;\times\; \mathbf{BiGrph}_{\mathit{emb}}(\bullet,\, g)$ for a bigraph $g \in \mathbf{BiGrph}_{\mathit{emb}}$.

Now, the extra part x is a right node, and its valuation will either be a node already in the graph described in the rest of the environment, or a new one with particular edges to the rest of the environment. The argument u can test (if it wants) what kind of node x is, before returning a probability.

As a result, for $[\![u]\!]_g \colon 2^k \times \mathbf{BiGrph}_{emb}(\circ, g)^\ell \times \mathbf{BiGrph}_{emb}(\bullet, g)^m \times \mathbf{BiGrph}_{emb}(\bullet, g) \longrightarrow [0,1]^{[0,1]^{g_L}}$, the denotation $[\![u]\!]$ gives us the edge probability of the left node that we need to generate, both to the existing right nodes, and to any future right nodes (which needs to be remembered). This can be formalized into a natural transformation $[\![\lambda_{\text{מ}} x.\, u]\!] \colon [\![\Gamma]\!] \to T([\![\mathbb{F}]\!])$.

$$[\![\lambda_{\text{מ}} x.\, u]\!]_g \colon \begin{cases} 2^k \times \mathbf{BiGrph}_{emb}(\circ, g)^\ell \times \mathbf{BiGrph}_{emb}(\bullet, g)^m \longrightarrow P_{\mathrm{f}}(g_L + 2^{g_R} \times [0,1])^{[0,1]^{g_L}} \\ b^k, \begin{matrix}(\circ \overset{\kappa_i}{\hookrightarrow} g)_i, \\ (\bullet \overset{\tau_j}{\hookrightarrow} g)_j\end{matrix} \mapsto \lambda \mapsto \left\langle \frac{1}{Z} \prod_{a \in g_R} p_a^{E^h(\mathfrak{f}_h(\circ), a)} (1 - p_a)^{1 - E^h(\mathfrak{f}_h(\circ), a)} \middle| \big[\underbrace{\circ}_{\cong (h-g)_L} \overset{\mathfrak{f}_h}{\hookrightarrow} h,\ \_ \mapsto \tilde{p}\big]_g \right\rangle_{h \in L_g} \end{cases}$$

where Z is a normalization constant, and

- for every $a \in g_R, \quad p_a \ := \ [\![u]\!]_g\big(b^k,\, (\circ \overset{\kappa_i}{\hookrightarrow} g)_i,\, (\bullet \overset{\tau_j}{\hookrightarrow} g)_j,\, \bullet \overset{a}{\hookrightarrow} g,\, \lambda\big)$
- $\tilde{p} \ := \ [\![u]\!]_g\big(b^k,\, (\circ \overset{\kappa_i}{\hookrightarrow} g \overset{\iota_1}{\hookrightarrow} g + \bullet)_i,\, (\bullet \overset{\tau_j}{\hookrightarrow} g \overset{\iota_1}{\hookrightarrow} g + \bullet)_j,\, \bullet \overset{\iota_2}{\hookrightarrow} g + \bullet,\, \lambda\big)$ where ι_1, ι_2 are the coprojections.

Finally, we show that the monad T is commutative and affine, which yields an abstract model of probability (as defined in [Koc11]) and implies that the language enjoys the dataflow property:

Theorem 5.2. *The probabilistic local state monad T is strong commutative and affine.*

References

[Ald81] David J. Aldous. “Representations for Partially Exchangeable Arrays of Random Variables”. In: *Journal of Multivariate Analysis* 11.4 (Dec. 1, 1981), pages 581–598. ISSN: 0047-259X. DOI: 10.1016/0047-259x(81)90099-3. URL: http://www.sciencedirect.com/science/article/pii/0047259X81900993 (visited on 11/25/2020).

[Aum61] R. Aumann. “Borel Structures for Function Spaces”. In: (1961). DOI: 10.1215/IJM/1255631584.

[CJ19] Kenta Cho and Bart Jacobs. “Disintegration and Bayesian Inversion via String Diagrams”. In: *Mathematical Structures in Computer Science* 29.7 (Aug. 2019), pages 938–971. ISSN: 0960-1295, 1469-8072. DOI: 10.1017/s0960129518000488. arXiv: 1709.00322. URL: http://arxiv.org/abs/1709.00322 (visited on 01/25/2021).

[Fin37] Bruno De Finetti. “La prévision : ses lois logiques, ses sources subjectives”. In: *Annales de l’institut Henri Poincaré* 7 (1937), page 69. URL: http://www.numdam.org/item/?id=AIHP_1937__7_1_1_0.

[Fri20] Tobias Fritz. “A Synthetic Approach to Markov Kernels, Conditional Independence and Theorems on Sufficient Statistics”. In: *Advances in Mathematics* 370 (Aug. 2020), page 107239. ISSN: 00018708. DOI: 10.1016/j.aim.2020.107239. arXiv: 1908.07021. URL: http://arxiv.org/abs/1908.07021 (visited on 01/31/2021).

[GS14] Noah D Goodman and Andreas Stuhlmüller. *The Design and Implementation of Probabilistic Programming Languages*. 2014. URL: http://dippl.org/ (visited on 08/17/2021).

[Heu+17] Chris Heunen et al. “A Convenient Category for Higher-Order Probability Theory”. Apr. 18, 2017. arXiv: 1701.02547 [cs, math]. URL: http://arxiv.org/abs/1701.02547 (visited on 02/11/2020).

[Hin] Ralf Hinze. *Generalizing Generalized Tries*.

[Hoo79] Douglas N. Hoover. *Relations on Probability Spaces and Arrays of Random Variables.* Institute for Advanced Study, Princeton, 1979. URL: http://www.stat.berkeley.edu/~aldous/Research/hoover.pdf.

[Koc11] Anders Kock. "Commutative Monads as a Theory of Distributions". Aug. 30, 2011. arXiv: 1108.5952 [math]. URL: http://arxiv.org/abs/1108.5952 (visited on 02/07/2021).

[Lev06] Paul Blain Levy. "Call-by-Push-Value: Decomposing Call-by-Value and Call-by-Name". In: *Higher-Order and Symbolic Computation* 19.4 (Dec. 2006), pages 377–414. ISSN: 1388-3690, 1573-0557. DOI: 10/fwb7vh. URL: http://link.springer.com/10.1007/s10990-006-0480-6 (visited on 11/12/2021).

[Mic68] Donald Michie. ""Memo" Functions and Machine Learning". In: (1968), page 4.

[Pit13] A. M. Pitts. *Nominal Sets: Names and Symmetry in Computer Science.* Cambridge Tracts in Theoretical Computer Science 57. Cambridge ; New York: Cambridge University Press, 2013. 276 pages. ISBN: 978-1-107-01778-8.

[PP02] Gordon Plotkin and John Power. "Notions of Computation Determine Monads". In: *Foundations of Software Science and Computation Structures.* Edited by Mogens Nielsen and Uffe Engberg. Redacted by Gerhard Goos, Juris Hartmanis and Jan van Leeuwen. Volume 2303. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pages 342–356. ISBN: 978-3-540-43366-8 978-3-540-45931-6. DOI: 10.1007/3-540-45931-6_24. URL: http://link.springer.com/10.1007/3-540-45931-6_24 (visited on 05/24/2021).

[Roy+08] D. Roy et al. "A Stochastic Programming Perspective on Nonparametric Bayes". In: 2008. URL: https://www.semanticscholar.org/paper/A-stochastic-programming-perspective-on-Bayes-Roy-Mansinghka/946ed04f705a2a28539a8af1f5e9ccdedd7fabc2 (visited on 01/21/2022).

[Sta20] Sam Staton. "Categorical Models of Probability with Symmetries". Categorical Probability and Statistics. June 2020. URL: http://perimeterinstitute.ca/personal/tfritz/2019/cps_workshop/slides/staton.pdf.

[Sta21] Sam Staton. "Some Formal Structures in Probability". 2021. URL: http://www.cs.ox.ac.uk/people/samuel.staton/2021fscd.pdf.

[Sta+] Sam Staton et al. *LazyPPL.* URL: https://lazyppl.bitbucket.io/ (visited on 04/04/2022).

[Ste21] Dario Maximilian Stein. "Structural Foundations for Probabilistic Programming Languages". University of Oxford, 2021. 221 pages.

[Woo+09] Frank Wood et al. "A Stochastic Memoizer for Sequence Data". In: *Proceedings of the 26th Annual International Conference on Machine Learning - ICML '09.* The 26th Annual International Conference. Montreal, Quebec, Canada: ACM Press, 2009, pages 1–8. ISBN: 978-1-60558-516-1. DOI: 10/fg8z4q. URL: http://portal.acm.org/citation.cfm?doid=1553374.1553518 (visited on 01/21/2022).