A model of stochastic memoization and name generation in probabilistic programming: categorical semantics via monads on presheaf categories

Younesse Kaddar and Sam Staton

Department of Computer Science, University of Oxford, UK

Abstract

Stochastic memoization is a higher-order construct of probabilistic programming languages that is key in Bayesian nonparametrics, a modular approach that allows us to extend models beyond their parametric limitations and compose them in an elegant and principled manner. Stochastic memoization is simple and useful in practice, but semantically elusive, particularly regarding dataflow transformations. As the naive implementation resorts to the state monad, which is not commutative, it is not clear if stochastic memoization preserves the dataflow property – *i.e.* whether we can reorder the lines of a program without changing its semantics, provided the dataflow graph is preserved. In this paper, we give an operational and categorical semantics to stochastic memoization and name generation in the context of a minimal probabilistic programming language, for a restricted class of functions. Our contribution is a first model of stochastic memoization of constant Bernoulli functions with a non-enumerable type, which validates data flow transformations, bridging the gap between traditional probability theory and higher-order probability models. Our model uses a presheaf category and novel probability monad on it.

Keywords: probabilistic programming, quasi-Borel spaces, synthetic measure theory, stochastic memoization, name generation, categorical semantics, commutative monads, nominal sets.

1 Introduction

Bayesian nonparametric models are a powerful approach to statistical learning. Unlike parametric models, which have a fixed number of parameters, nonparametric models can have an unbounded number of parameters that grows as needed to fit complex data. This flexibility allows them to capture subtle patterns in data that parametric models may miss, and it makes them more composable, because they are not arbitrarily truncated.

Prominent examples of nonparametric models include Dirichlet process models for clustering similar data points, and the Infinite Relational Model for automatically discovering latent groups and features, amongst others. These infinite-dimensional models can accommodate an unbounded number of components, clusters, or other features in order to fit observed data as accurately as possible.

Probabilistic programming is a powerful method for programming nonparametric models. *Stochastic memoization* [47, 57] has been identified as a particularly useful technique in this. This paper is about semantic foundations for stochastic memoization.

In deterministic memoization [38], the idea is to compute a function the first time it is called with a particular argument, and store the result in a memo-table. When the function is called again with the same argument, the memo-table is used, resulting in performance improvement but no semantic difference. Stochastic memoization is this memoization applied to functions that involve random choices, and so a memoized function is semantically different from a non-memoized one, because the random choices will only be made once for each argument.

We illustrate this with a simple example; this is informal and we consider a precise language and semantics in

Section 3. Consider a function f that returns a random number $[0,1]$ for each argument. It might be written $f(x) =$ uniform. One run of the program might call f with various arguments, and example runs are as follows:

Calls to f in a particular run of a program:	$f(0)$	$f(1)$	$f(0)$	$f(2)$	$f(1)$	$f(3)$	…
Results of calls in a run without memoization:	0.43	0.01	0.72	0.26	0.48	0.16	…
Results of calls in a run with memoization:	0.43	0.01	**0.43**	0.26	**0.01**	0.16	…

Thus in the memoized version, when the function is called again with the same value, the previous result is recalled, and the random choices are not made again. (Note that although this is called 'stochastic memoization', the terminology is perhaps confusing: the memoization always happens, and it is not 'randomly deciding whether or not to memoize'.)

From a semantic perspective, the role of stochastic memoization is clear when we use a monad-based interpretation with a probability monad Prob. This might be thought of as the Giry monad [15] or a probabilistic powerdomain [20, 25], or a Haskell monad (e.g. [10]).
A distribution on a type b with parameters from a has type a → Prob(b). On the other hand, a random function is a probability distribution on the type of deterministic functions, having type Prob(a → b). Whereas parameterized distributions are a key idea in parametric statistics, random functions are a key idea in nonparametric statistics. And stochastic memoization is a higher-order function with probabilistic effects, of type

```
mem :: (a → Prob b) → Prob (a → b)
```

that converts parameterized distributions into random functions, by making the random choice once for each argument. This mem combinator plays a crucial role in Church [17] and WebPPL [19], and appears with this type in our Haskell library LazyPPL [52]. Stochastic memoization also plays a role in Blog [39], Hansei [29], and many other languages (e.g. [5, 11]). It is not difficult to implement stochastic memoization, by using a memo-table. Nonetheless, its semantic properties remain elusive and developers have noted bugs and complications (e.g. [16, 30]). Moreover, the existing semantic models of probability (such as [20, 21, 25]) only support mem for very restricted domain types a (see §1). In particular our own Haskell library [52] supports stochastic memoization but the recent semantic analysis [10] only explains it at certain domain types. The point of this paper is to extend this semantic analysis of stochastic memoization to a broader class of domains.

First example: White noise in a non-parametric clustering model.
One common first example of stochastic memoization is as follows. Suppose we have a finite set of individuals, and we want to group them into an unknown number of clusters, and then assign attributes to the clusters. For example, we may want to form clusters and consider attributes on the clusters such as 'Brexit-supporters', 'mean geographic latitude/longitude', 'geographic variance', 'mean salary', and so on. A popular route is the 'Dirichlet process with memoization', as follows, for which a generative model has the following pseudocode (see e.g. [18, 19, 47] [14]):

(i) We randomly decide which proportion of individuals are in each cluster. We assign a unique identifier to each cluster, from some space $\mathbb{A}$ of identifiers. One might use the Dirichlet process with a diffuse base measure on $\mathbb{A}$, for example the normal distribution on the real numbers.

(ii) Assign attributes to the cluster identifiers. For example, depending on whether that cluster supports Brexit, assign either true or false to the identifier. This particular assignment is a sample from a random function in $(\mathbb{A} \to 2)$. This distribution might come from memoizing a constant Bernoulli distribution, assigning 'true' to any cluster identifier with probability 0.5.

(iii) Steps (i)-(ii) are generative, and we could run them to get some synthetic data. The idea of Bayesian clustering is to start with steps (i)–(ii) as a reasonable *prior* distribution, in generative form, and to combine this with actual data to arrive at a *posterior* distribution. In this example the actual data might come from a telephone survey, and we use conditional probability (aka Bayesian inversion) to arrive at a posterior distribution on the cluster proportions and their attributes. We can then use this to make predictions. The constant Bernoulli memoization is a reasonable prior for Brexit support, but the posterior will typically be much more complicated, with various correlations, etc.

In this paper, we focus on step (ii), stochastic memoization: steps (i) and (iii) are studied extensively elsewhere (e.g. see [14] in the statistics literature, or [2, 9, 51] in the semantics literature, and references therein).

This simple example of a memoized constant Bernoulli function is easy to implement using a memo-table, but already semantically complicated. If we put $\mathbb{A} = \mathbb{R}$, the real numbers, for the base measure, as is common in statistical modelling, then the memoized constant Bernoulli distribution on $(\mathbb{A} \to 2)$ is 1-dimensional white noise: intuitively, for every $x \in \mathbb{R}$ we toss a coin to pick true or false, making an uncountable number of independent random choices. (As an aside, we note that we could combine steps (i) and (ii), using a complicated base measure for the Dirichlet process that includes all the attributes. This model would not be compositional, and in any case, some kind of memoization would still be needed to implement the Dirichlet process.)

Challenge.
In this paper, we address the challenge of showing that the following items are consistent:

(1) a type $\mathbb{A}$ with a diffuse probability distribution (Def 2.2);
(2) a type bool of Booleans with Bernoulli probability distributions (i.e. tossing coins, including biased coins);
(3) a type of functions $[\mathbb{A} \to \mathsf{bool}]$, with function application (4);
(4) stochastic memoization of the constant Bernoulli functions (3);
(5) the language supports the dataflow property (Def. 2.3).

These items are together inconsistent with traditional measure theory, as we discuss in Section 2.3, where we also make the criteria precise. Nonetheless (1)-(4) are together easy to implement in a probabilistic programming language, and useful for Bayesian modelling. Item (5) is a very useful property for program reasoning and program optimization. Item (5) is also a fundamental conceptual aspect of axiomatic probability theory, since in the measure-theoretic setting it amounts to Fubini's theorem [32] and the fact that probability measures have mass 1, and in the categorical abstraction of Markov categories [13] it amounts to the interchange law of affine monoidal categories.
There *are* measure-theoretic models where some of these items are relaxed (§2.1–2.3). For example, if we drop the requirement of a diffuse distribution, then there are models using Kolmogorov extension (§2.2).
A grand challenge is to further generalize these items, for example to allow memoization of functions $A \to B$ for yet more general A and B, and to allow memoization of all definable expressions. Since the above five items already represent a significant challenge, and our semantic model is already quite complicated, we chose to focus on a 'minimal working example' for this paper.
To keep things simple and minimal, in this paper we side-step measure-theoretic issues by noticing that the equations satisfied by a diffuse probability distribution are exactly the equations satisfied by name generation (e.g. [50, §VB]). Because of this, we can use categorical models for name generation (following e.g. [41, §4.1.4], [49, §3.5]) instead of traditional measure theory. Name generation can certainly be implemented using randomness, and there are no clashes of fresh names if and only if the names come from a diffuse distribution (see also e.g. [48]). On the other hand, if we keep things simple by regarding the generated names as *pure names* [40], we avoid any other aspects of measure theory, such as complicated manipulations of the real numbers.

Contributions.
To address the challenge of the consistency of items (1)–(5) above, our main contributions are then as follows.

(i) We first provide an operational semantics for a minimal toy probabilistic programming language that supports stochastic memoization and name generation (§4).
(ii) We then (§5) construct a cartesian closed (for function spaces) categorical model of this language endowed with an affine commutative monad (Theorem 5.5). In common with other work on local state (e.g. [28, 44]), we use a functor category semantics, indexing sets by possible worlds. In this paper, those worlds are finite fragments of a memo-table.
(iii) We prove that our denotational semantics is sound with respect to the operational semantics, ensuring the correctness of our approach and validating that lines can be reordered in the operational semantics (Theorem 5.11). The class of functions that can be memoized includes constant Bernoulli functions. We call these functions *freshness-invariant* (Definition 5.8).
The soundness theorem (5.11) is not trivial because the timing of the random choices differs between the operational and denotational semantics. In the operational semantics, the memo-table is partial, and populated lazily as needed, when functions are called with arguments. This is what happens in all implementations.

However, this timing is intensional, and so by contrast, in the denotational semantics, the memo-table is always totally populated as soon as the current world is extended with any functions or arguments.

(iv) Finally, we present a practical Haskell implementation [26] which compares the small-step, big-step operational, and denotational semantics, demonstrating the applicability of our results (§6).

2 Stochastic memoization by example

This section discusses the law of stochastic memoization and provides examples in finite, countable, and non-enumerable domain settings. We then address the challenges posed by the naive use of the state monad, and we clarify our objective: finding a model of probability that supports stochastic memoization over non-enumerable domains, satisfying the dataflow property, and that has function spaces.

In what follows, we use two calculi: (a) The internal metalanguage of a cartesian closed category with a strong monad Prob, for which we use Haskell notation, but which is roughly Moggi's monadic metalanguage [42, §2.2]. (b) An ML-like programming language which is more useful for practical programming, but which would translate into language (a); this is roughly Moggi's 'simple programming language' [42, §2.3]. We assume passing familiarity with probability and monadic programming in this section, but the informal discussion here sets the context, and we move to more formal arguments in Section 3.

(Recall some Haskell notation: we write `\x → t` for lambda abstraction; `>>=` for monadic bind, i.e. Kleisli composition; **return** for the unit; a **do** block allows a sequence of monadic bound instructions. We write **const** `x` for the constant `x` function, **const** `x = \y → x`.)

Memoization law.

Definition 2.1 A strong monad *supports stochastic memoization of type* `a→ b` if it is equipped with a morphism `mem :: (a → Prob b) → Prob (a → b)` that satisfies the following equation in the metalanguage, for every x_0 `::` `a` and `f :: a → Prob b`:

$$\texttt{mem f = f } x_0 \texttt{ >>= (\y}_0 \texttt{ → mem f >>= (\fMem → return (\x → if x == } x_0 \texttt{ then } y_0 \texttt{ else fMem x)))} \tag{1}$$

As noted at the beginning of this section, we will pass between an internal metalanguage for strong monads, and an ML-like programming language that would be interpreted using strong monads. In Section 3 we introduce this programming language precisely, but for now we note that it has a special syntax $\lambda_{\text{מ}} x.\, u$, meaning mem $(\backslash x \to u)$, since this is a common idiom [1] . The law of Definition 2.1 requires equations such as:

$$\begin{array}{l}\texttt{let val } f \leftarrow \lambda_{\text{מ}} x.\, u \\ \texttt{in } f@n\end{array} \overset{\text{1 sample}}{=} u[n/x] \qquad \begin{array}{l}\texttt{let val } f \leftarrow \lambda_{\text{מ}} x.\, u \texttt{ in} \\ \texttt{let val } v_1 \leftarrow f@n \texttt{ in} \\ \texttt{let val } v_2 \leftarrow f@n \texttt{ in} \\ \texttt{return}(v_1, v_2)\end{array} \overset{\text{several samples}}{=} \begin{array}{l}\texttt{let val } v \leftarrow u[n/x] \texttt{ in} \\ \texttt{return}(v, v)\end{array} \tag{2}$$

The examples in the introduction use memoization of a constant Bernoulli function, i.e.

$$\texttt{mem (\x → bernoulli p) = mem (const (bernoulli p)) :: Prob(a → Bool)} \tag{3}$$

i.e. $\lambda_{\text{מ}} x.$ `bernoulli p`, where `bernoulli p :: Prob Bool` is a Bernoulli probability distribution (biased coin toss) with bias `p`. An intuition is that this is a binary white noise; every point in `a` has an independently chosen random Boolean value.

Notice that for the laws we have also needed function application

$$@ \quad :: ((a \to b)\ ,\ a) \to b \tag{4}$$

In summary, memoized constant Bernoulli functions (3), and function application (4), are a bare minimum to discuss semantic issues around stochastic memoization.

We now consider interpretations where the domain `a` is finite (§2.1), countable (§2.2), and uncountable (§2.3).

[1] borrowing Melliès' use of the Hebrew letter מ ("mem") [37]

2.1 *Memoization with finite domain*

For finite domains a, memoization is straightforward. It involves simply sampling a value of $f(x)$ for all inhabitants of $x \in X$ and returning the assignment as a finite mapping. For example, when a = bool, we can implement memoization in Haskell as follows:

```
mem f = do { fT ← f True ; fF ← f False ; return (\b → if b then fT else fF )}
```

Semantic interpretation with finite domain.
Memoization with finite domains is supported by a denotational semantics using any strong monad. For example, the category of sets and the monad of finitely supported probability distributions (e.g. [23]). For a = bool, this is nothing but the double-strength:

$$(\text{bool} \to \text{Prob b}) \cong (\text{Prob b}, \text{Prob b}) \xrightarrow{\text{double-strength}} \text{Prob (b,b)}.$$

For other finite a, it is defined using the double-strength by induction.

2.2 *Memoization with countable/enumerable domain*

When a is enumerable, such as a=**Int**, memoization is useful for defining point processes. Memoization can be regarded as providing an infinite stream of random choices, since the streams over b are isomorphic with the functions a → b.
Infinite streams of random choices are crucial examples of statistical processes [14]. For an example of an application, recall the one-dimensional Poisson point process. This is a random sequence of real numbers in which the gaps between consecutive numbers are exponentially distributed. Assuming an exponential distribution with a given rate, exponential rate :: Prob RealNum, we can sample a sequence of these exponential gaps from mem (**const** (exponential rate)) :: Prob (**Int** → RealNum). To get the corresponding list of points of the Poisson point process (with exponential interoccurence times), we simply keep a cumulative sum total of the points, starting from the lower point:

```
poissonPP :: Double → Double → Prob [Double]
poissonPP lower rate = do { gaps ← mem (const (exponential rate )) ; return (scanl1 (+) lower (map gaps [1 .. ])) }
```

We implement memoization with enumerable a in the Haskell LazyPPL library [10] without using state, instead using Haskell's laziness and tries, following [22] (see [10]). We use the Poisson process extensively in the demonstrations for LazyPPL [52].

Semantic interpretation with enumerable domains.
Memoization with enumerable domains is supported by a denotational semantics using the category of measurable spaces and the Giry monad [15]. Although the category is not Cartesian closed, the function space $B^{\mathbb{N}}$ *does* exist for all standard Borel B, and is given by the countable product of B with itself, $\prod_{\mathbb{N}} B$. Memoization amounts to using Kolmogorov's extension theorem to define a map $(G\,B)^{\mathbb{N}} \to G(B^{\mathbb{N}})$ (see [45, §4.8] and [9, Thm. 2.5]).

2.3 *Memoization with non-enumerable/diffuse domain*

We now move beyond enumerable domains, to formalize the challenge from Section 1. In Section 1 we illustrated this with a clustering model. See [52] for the full implementation in our Haskell library, LazyPPL, along with other models that also use memoization, including a feature extraction model that uses the Indian Buffet Process, and relational inference with the infinite relational model (following [18]).
Rather than axiomatizing uncountability, we consider diffuse distributions.

Definition 2.2 [Diffuse distribution] Let a be an object with an equality predicate ((a,a)→ bool). A *diffuse distribu-*

tion [2] is a term p such that

$$\textbf{do}\ \{x \leftarrow p\ ;\ y \leftarrow p\ ;\ \textbf{return}\ (x == y)\} \qquad \text{is semantically equal to} \qquad \textbf{return}\ (\text{false}).$$

For example, in a probabilistic programming language over the real numbers, we can let a be the type of real numbers and let p be a uniform distribution on $[0, 1]$, or a normal distribution, or an exponential distribution. These are all diffuse in the above sense. The Bernoulli distribution on the booleans is not diffuse, because there is always a chance that we may get the same result twice in succession.

For the reader familiar with traditional measure theory, we recall that if p is diffuse then a is necessarily an uncountable space. For any probability distribution on a countable discrete space must give non-zero measure to at least one singleton set.

The implementation trick using tries from Section 2.2 will not work for diffuse measures, because we cannot enumerate the domain of a diffuse distribution. It is still possible to implement memoization using state and a memo-table (e.g. [52]). Unlike a fully stateful effect, however, in this paper we argue that stochastic memoization is still compatible with commutativity/dataflow program transformations:

Definition 2.3 [Dataflow property] A programming language is said to have the *dataflow property* if program lines can be reordered (commutativity) and discarded (discardability, or affineness) provided that the dataflow is preserved. In other words, the language satisfies the following commutativity and discardability equations:

$$\textbf{do}\ \{x1 \leftarrow t1\ ;\ x2 \leftarrow t2\ ;\ u\} = \textbf{do}\ \{x2 \leftarrow t2\ ;\ x1 \leftarrow t1\ ;\ u\} \tag{5}$$

$$\textbf{do}\ \{x1 \leftarrow t1\ ;\ t2\} = t2 \qquad \text{where } x1 \notin \mathsf{fv}(t2) \text{ and } x2 \notin \mathsf{fv}(t1). \tag{6}$$

The dataflow property expresses the fact that, to give a meaning to programs, the only thing that matters is the topology of dataflow diagrams. These transformations are very useful for inference algorithms and program optimization. But above all, on the foundational side, dataflow is a fundamental concept that corresponds to monoidal categories and is crucial to have a model of probability. As for monoidal categories, a strong monad is commutative (5) if and only if its Kleisli category is monoidal (commutativity is the monoidal interchange law), and affine (6) if the monoidal unit is terminal. In synthetic probability theory, dataflow is regarded by various authors as a fundamental aspect of the abstract axiomatization of probability: Kock [31] argues that any monad that is strong commutative and affine can be abstractly viewed as a probability monad, and affine monoidal categories are used as a basic setting for synthetic probability by several authors [7, 13, 55, 56]. The reader familiar with measure-theoretic probability will recall that the proof that the Giry monad satisfies (5) amounts to Fubini's theorem for reordering integrals (e.g. [51]).

Semantic interpretations for diffuse domains

The point of this paper is to provide the first semantic interpretation for memoization of the constant Bernoulli functions (3) with diffuse domain (Def. 2.2). We emphasize that although other models can support some aspects of this, there is no prior work that supports everything.

- With countable domain, there is a model in measurable spaces, as discussed in Section 2.2. But there can be no diffuse distribution on a countable space.
- In measurable spaces, we can form the uncountable product space $\prod_{\mathbb{R}} 2$ of $\mathbb{R}$-many copies of 2. We can then define a white noise probability measure on $\prod_{\mathbb{R}} 2$ via Kolmogorov extension (e.g. [45, 4.9(31)]). Moreover, there are diffuse distributions on $\mathbb{R}$, such as the uniform distribution on $[0, 1]$. However, it is known that there is no measurable evaluation map $\mathbb{R} \times (\prod_{\mathbb{R}} 2) \to 2$ (see [1]), and so we cannot interpret function application (4).
- In quasi-Borel spaces [21], there is a quasi-Borel space $[\mathbb{R} \to 2]$ of measurable functions, and a measurable evaluation map $\mathbb{R} \times ([\mathbb{R} \to 2) \to 2$, but there is no white noise probability measure on $[\mathbb{R} \to 2]$. The intuitive reason is that, in quasi-Borel spaces, a probability measure on $[\mathbb{R} \to 2]$ is given by a random element, i.e. a morphism $\Omega \to [\mathbb{R} \to 2]$, which curries to a measurable function $\Omega \times \mathbb{R} \to 2$. But there is no such measurable function representing white noise (e.g. [27, Ex 1.2.5]).

[2] Diffuse measures are often called 'atomless' in probability theory. We will also want to regard names in name generation as atomic, so we avoid this clash of terminology.

- There are domain-theoretic treatments of probability theory that support Kolmogorov extension, uniform distributions on $\mathbb{R}$, and function spaces [20, 25]. However, these treatments regard the real numbers $\mathbb{R}$ as constructive, and hence there are no non-trivial continuous morphisms $\mathbb{R} \to 2$, and there is no equality test on $\mathbb{R}$, so that we cannot regard $\mathbb{R}$ with a diffuse distribution as formalized equationally in Definition 2.2. The same concern seems to apply to recent approaches using metric monads [36].
- The semantic model of beta-bernoulli in [53] is a combinatorial model that includes aspects of the beta distribution, which is diffuse in measure theory. That model does not support stochastic memoization, but as a presheaf-based model it is a starting point for the model in this paper.
- There is a straightforward implementation of stochastic memoization that uses local state, as long as the domain supports equality testing [52]. The informal idea is to make the random choices as they are needed, and remember them in a memo-table, and keep this memo-table in a local state associated with the function. Therefore one could use a semantic treatment of local state to analyze memoization. For example, one could build a state monad in quasi-Borel spaces. However, state effects in general do not support the dataflow property (Def. 2.3), since we cannot reorder memory assignments in general. Ideally, one could use a program logic to prove that this particular use of state does support the dataflow property. Although there are powerful program logics for local state and probability (e.g. [3]), we have not been able to use them to prove this.

There are other models of higher-order probability (e.g. [6, 8, 12]). These do not necessarily fit into the monad-based paradigm, but there may be other ways to use them to address the core challenge in Section 1.

3 A language for stochastic memoization and name generation

In this section, we define the syntax of our minimal probabilistic programming language. The syntax only has a handful of constructs, to focus on the following key features:

- **name generation**: we can generate fresh names (referred to as *atomic* names or *atoms*, in the sense of Pitts' nominal set theory [43]) with constructs such as $\mathrm{let}\ x = \mathsf{fresh}()\ \mathrm{in}\ \cdots$. In the terminology of Def. 2.2, this is like a generic diffuse probability measure, since fresh names are distinct.
- basic **probabilistic effects**: for illustrative purposes, the only distribution we consider, as a first step, is the Bernoulli distribution (but it can easily be extended to other discrete distributions). Constructs like $\mathrm{let}\ b = \mathsf{flip}(\theta)\ \mathrm{in}\ \cdots$ amount to flipping a coin with bias θ and storing its result in a variable b.
- **stochastic memoization**: if a probabilistic function f – defined with the new $\lambda_{\text{ᴍ}}$ operator – is called twice on the same argument, it should return the same result (eq. (2)).

We have the following base types: bool (booleans), $\mathbb{A}$ (atomic names), and $\mathbb{F}$ (which can be thought of as the type of memoized functions $\mathbb{A} \to \mathsf{bool}$). For the sake of simplicity, we do not have arbitrary function types. In fine-grained call-by-value fashion [33], there are two kinds of judgments: typed values, and typed computations. The grammar and typing rules of our language are given in Figure 1. The typing rules are standard, except for the $\lambda_{\text{ᴍ}}$ operator, which is the key novelty of our language. The typing rule for $\lambda_{\text{ᴍ}}$ is given in Figure 1 and is explained in the next section. (Also, equality $v = w$ and memoized function application $v@w$ are pure computations, *i.e.* in the categorical semantics (section 5.3), they will be composed by the unit of the monad.)

Table 1: Grammar and typing rules of the language

Types
$A, B \quad ::= \quad \mathsf{bool} \mid \mathbb{A} \mid \mathbb{F} \mid A \times B$
Expressions
Values:
$v, w \quad ::= \quad \mathsf{true} \mid \mathsf{false} \mid x \mid (v, w)$
Computations:
Continued on next page

$$u, t \quad ::= \quad \mathsf{return}(v) \mid \mathsf{let\ val}\ x \leftarrow u\ \mathsf{in}\ t \mid \mathsf{if}\ v\ \mathsf{then}\ u\ \mathsf{else}\ t \mid \mathsf{match}\ v\ \mathsf{as}\ (x, y)\ \mathsf{in}\ t \mid \mathsf{flip}(\theta) \mid \mathsf{fresh}() \mid v = w \mid \lambda_{\text{מ}} x.\, u \mid v@w$$

Typing judgements

Typed values:

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

Typed computations:

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

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

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

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

4 Operational Semantics

We now present a small-step operational semantics for our language. The operational semantics defines the rules for reducing program expressions, which form the basis for understanding the behavior of programs written in the language. Henceforth, we fix a countable set of variables $x, y, z, \ldots \in \mathsf{Var}$, and consider the terms up to α-equivalence for the $\lambda_{\text{מ}}$ operator. Since we focus on functions with boolean codomain, our partial memo-tables are represented as partial bigraphs (bipartite graphs).

Definition 4.1 [Partial bigraph] A partial bigraph $\mathfrak{g} \stackrel{\text{def}}{=} (\mathfrak{g}_L, \mathfrak{g}_R, E)$ is a finite bipartite graph where the edge relation $E\colon \mathfrak{g}_L \times \mathfrak{g}_R \to \{\mathsf{true}, \mathsf{false}, \bot\}$ is either true, false or undefined ($\bot$) on each pair of left and right nodes $(\mathsf{f}, a) \in \mathfrak{g}_L \times \mathfrak{g}_R$. In the following, left nodes will be thought of as function labels and right nodes as atom labels. By abuse of notation, syntactic truth values will be conflated with semantic ones. For a partial graph $\mathfrak{g}$, $E(\mathsf{f}, a) = \beta \in \{\mathsf{true}, \mathsf{false}, \bot\}$ will be written $\mathsf{f} \xrightarrow{\beta} a$ when $\mathfrak{g}$ is clear from the context.

4.1 Extended expressions

We introduce extended expressions e, by extending the grammar of computations (1) with an extra construct $\{\!\{u\}\!\}^{\mathsf{f},a}_{\gamma}$, where u is a computation,(f, a) is a pair of function and atom labels to memoize, and γ is the environment to restore after the result of f at a has been computed and stored. Intuitively, the decoration $\{\!\{-\}\!\}^{\mathsf{f},a}_{\gamma}$ is thought of as a memoization context, indicating expressions where memoization should happen: $\{\!\{u\}\!\}^{\mathsf{f},a}_{\gamma}$ is a computation that memoizes the result of u, and then restores the environment to the state it was in before the computation u was evaluated. In the following, $\Delta \in \bigcup_{n \geq 0} (\mathfrak{g}_L \times \mathfrak{g}_R)^n$ is a finite stack of function–atom label pairs, indicating that we are in the process of computing the result of these functions at these atoms for the first time. Each newly introduced function–atom label pair is assumed not to already belong to the memoization stack.

Table 2: Extended expression typing rules.

Extended expression typing judgements. Here, $(\mathfrak{f}, a) \notin \Delta \cup \Delta_1 \cup \Delta_2$.

$$\frac{\Gamma \vdash^{\mathsf{c}} u : A}{\Gamma \mid \emptyset \vdash^{\mathsf{c}} u : A} \qquad \frac{\Gamma \mid \Delta \vdash^{\mathsf{c}} u : A}{\Gamma \mid (\mathfrak{f}, a), \Delta \vdash^{\mathsf{c}} \{\!\{u\}\!\}^{\mathfrak{f},a}_{\gamma} : A} \qquad \frac{\Gamma \mid \Delta_1 \vdash^{\mathsf{c}} u : A \quad \Gamma, x : A \mid \Delta_2 \vdash^{\mathsf{c}} t : B}{\Gamma \mid \Delta_1, \Delta_2 \vdash^{\mathsf{c}} \mathsf{let\ val}\ x \leftarrow u\ \mathsf{in}\ t : B}$$

$$\frac{\Gamma \mid \Delta_1 \vdash^{\mathsf{c}} u : A \quad \Gamma, x : A \mid \Delta_2 \vdash^{\mathsf{c}} t : B}{\Gamma \mid (\mathfrak{f}, a), \Delta_1, \Delta_2 \vdash^{\mathsf{c}} \mathsf{let\ val}\ x \leftarrow \{\!\{u\}\!\}^{\mathfrak{f},a}_{\gamma}\ \mathsf{in}\ t : B} \qquad \frac{\Gamma \mid \Delta_1 \vdash^{\mathsf{c}} u : A \quad \Gamma, x : A \mid \Delta_2 \vdash^{\mathsf{c}} t : B}{\Gamma \mid (\mathfrak{f}, a), \Delta_1, \Delta_2 \vdash^{\mathsf{c}} \mathsf{let\ val}\ x \leftarrow u\ \mathsf{in}\ \{\!\{t\}\!\}^{\mathfrak{f},a}_{\gamma} : B}$$

$$\frac{\Gamma \vdash^{\mathsf{v}} v : \mathsf{bool} \quad \Gamma \mid \Delta_1 \vdash^{\mathsf{c}} u : A \quad \Gamma \mid \Delta_2 \vdash^{\mathsf{c}} t : A}{\Gamma \mid \Delta_1, \Delta_2 \vdash^{\mathsf{c}} \mathsf{if}\ v\ \mathsf{then}\ u\ \mathsf{else}\ t : A}$$

$$\frac{\Gamma \vdash^{\mathsf{v}} v : \mathsf{bool} \quad \Gamma \mid \Delta_1 \vdash^{\mathsf{c}} u : A \quad \Gamma \mid \Delta_2 \vdash^{\mathsf{c}} t : A}{\Gamma \mid (\mathfrak{f}, a), \Delta_1, \Delta_2 \vdash^{\mathsf{c}} \mathsf{if}\ v\ \mathsf{then}\ \{\!\{u\}\!\}^{\mathfrak{f},a}_{\gamma}\ \mathsf{else}\ t : A} \qquad \frac{\Gamma \vdash^{\mathsf{v}} v : \mathsf{bool} \quad \Gamma \mid \Delta_1 \vdash^{\mathsf{c}} u : A \quad \Gamma \mid \Delta_2 \vdash^{\mathsf{c}} t : A}{\Gamma \mid (\mathfrak{f}, a), \Delta_1, \Delta_2 \vdash^{\mathsf{c}} \mathsf{if}\ v\ \mathsf{then}\ u\ \mathsf{else}\ \{\!\{t\}\!\}^{\mathfrak{f},a}_{\gamma} : A}$$

$$\frac{\Gamma \vdash^{\mathsf{v}} v : A \times B \quad \Gamma, x : A, y : B \mid \Delta \vdash^{\mathsf{c}} t : C}{\Gamma \mid \Delta \vdash^{\mathsf{c}} \mathsf{match}\ v\ \mathsf{as}\ (x, y)\ \mathsf{in}\ t : C} \qquad \frac{\Gamma \vdash^{\mathsf{v}} v : A \times B \quad \Gamma, x : A, y : B \mid \Delta \vdash^{\mathsf{c}} t : C}{\Gamma \mid (\mathfrak{f}, a), \Delta \vdash^{\mathsf{c}} \mathsf{match}\ v\ \mathsf{as}\ (x, y)\ \mathsf{in}\ \{\!\{t\}\!\}^{\mathfrak{f},a}_{\gamma} : C}$$

4.2 Configurations

We now define the set-theoretic interpretation of contexts. Context values are built by combining booleans, atomic names and functions using pairing. Thus a context value is a tree, where the branches are understood as pairing.

Definition 4.2 If S is a finite set, $\mathsf{Tree}(S) \cong \biguplus_{n \geq 0} C_n\, S^{n+1}$ (where C_n is the n-th Catalan number, and $C_n\, S^{n+1}$ is a coproduct of n copies of S^{n+1}, one for each possible bracketing) denotes the set of all possible non-empty trees with internal nodes the cartesian product and leaf nodes taken in S.

Example 4.3 If $S \stackrel{\text{def}}{=} \{s_1, s_2\}$, then $s_1 \in \mathsf{Tree}(S), (s_2, s_1) \in \mathsf{Tree}(S), (s_1, (s_1, s_2)) \in \mathsf{Tree}(S), \ldots$

Definition 4.4 [Set-theoretic denotation of contexts.] Let $\mathfrak{g}$ be a partial bigraph. The set-theoretic denotation $(\!|-|\!)$ of a context Γ is defined as $(\!|\mathsf{bool}|\!) \stackrel{\text{def}}{=} 2 \cong \{\mathsf{true}, \mathsf{false}\}$, $(\!|\mathbb{F}|\!) \stackrel{\text{def}}{=} \mathfrak{g}_L$, $(\!|\mathbb{A}|\!) \stackrel{\text{def}}{=} \mathfrak{g}_R$ and $(\!|-|\!)$ is readily extended to every context Γ. Moreover, in the following, $\gamma \in (\!|\Gamma|\!) \subseteq \mathsf{Tree}(2 + \mathfrak{g}_L + \mathfrak{g}_R)^{\mathsf{Var}}$ denotes a context value.

Example 4.5 If $\Gamma \stackrel{\text{def}}{=} (x : \mathsf{bool}, y : \mathbb{F}, z : ((\mathbb{F} \times 2) \times \mathbb{A}))$, then $(\!|\Gamma|\!) \stackrel{\text{def}}{=} \{x \mapsto 2, y \mapsto \mathfrak{g}_L, z \mapsto ((\mathfrak{g}_L \times 2) \times \mathfrak{g}_R)$ and an example of a context value is $\gamma \stackrel{\text{def}}{=} \{x \mapsto \mathsf{true}, y \mapsto \mathfrak{f}_0, z \mapsto ((\mathfrak{f}_1, \mathsf{true}), a_0)\}$.

We now have everything to introduce terminal computations, redexes, reduction contexts, and configurations (table 3). Configurations provide a complete description of the state of the computation, including the context value, extended expression, partial graph, and a mapping from the partial graph to closures. These components help keep track of different parts of the program as the computation proceeds.

Table 3: Terminal computations, redexes, reduction contexts, and configurations.

Terminal computations r, Redexes ρ, and Reduction contexts $\mathcal{C}[-]$

r	::=	$\mathsf{return}(v) \mid \lambda_{\text{ɱ}}\, x.\, u \mid \mathsf{fresh}()$	
ρ	::=	$\mathsf{let\ val}\ x \leftarrow r\ \mathsf{in}\ u \mid \{\!\{\mathsf{return}(v)\}\!\}^{\mathfrak{f},a}_{\gamma}$	where $\mathfrak{f} \in \mathfrak{g}_L,\ a \in \mathfrak{g}_R,\ \gamma \in \mathsf{Tree}(2 + \mathfrak{g}_L + \mathfrak{g}_R)^{\mathsf{Var}}$
		$\mid \mathsf{match}\ v\ \mathsf{as}\ (x, y)\ \mathsf{in}\ t \mid \mathsf{if}\ v\ \mathsf{then}\ t\ \mathsf{else}\ u \mid \mathsf{flip}(\theta) \mid (v = w) \mid (v @ w)$	

Continued on next page

$$\mathcal{C}[-] \quad ::= \quad [-] \;\mid\; \mathsf{let\ val}\ x \leftarrow \mathcal{C}[-]\ \mathsf{in}\ u \;\mid\; \{\!\{\mathcal{C}[-]\}\!\}_{\gamma}^{\mathsf{f},a}$$

Configurations $(\gamma, u, \mathfrak{g}, \lambda)$

- $\gamma \in \mathsf{Tree}(2 + \mathfrak{g}_L + \mathfrak{g}_R)^{\mathsf{Var}}$ is a context value.
- u is an extended expression $\Gamma \mid \Delta \vdash^{\mathsf{c}} u : A$.
- $\mathfrak{g} \stackrel{\text{def}}{=} (\mathfrak{g}_L, \mathfrak{g}_R, E)$ is a partial graph.
- $\lambda \colon \mathfrak{g}_L \to \mathsf{Closures}$, where $\mathsf{Closures} \stackrel{\text{def}}{=} \left\{(\lambda_{\text{ם}}\, x.\, u, \gamma) \;\mid\; \Gamma \vdash^{\mathsf{c}} \lambda_{\text{ם}}\, x.\, u : \mathbb{F} \text{ and } \gamma \in (\!|\Gamma|\!)\right\}$

4.3 Reduction rules

Let $(\!|-|\!)_\gamma$ be the function evaluating an expression value in a context value γ (*e.g.* $(\!|x|\!)_\gamma = \gamma(x)$, $(\!|\mathsf{true}|\!)_\gamma = \mathsf{true}$, etc). We can define the operational semantics of the language using reduction rules. They provide a step-by-step description of how expressions are evaluated and transformed during execution, following a left-most outer-most strategy, with lexical binding. Given a configuration $(\gamma, u, \mathfrak{g}, \lambda)$ (note that if u is of the form $\{\!\{u'\}\!\}_{\gamma}^{(\mathsf{f},a)}$, then it is assumed that the function–atom label pair $(\mathsf{f}, a) \in \mathfrak{g}_L \times \mathfrak{g}_R$), we will apply the following reduction rules:

Table 4: Reduction rules.

Reduction Rules

$$(\gamma, \mathsf{let\ val}\ x \leftarrow \mathsf{return}(v)\ \mathsf{in}\ u, \mathfrak{g}, \lambda) \to (\gamma \sqcup \{x \mapsto (\!|v|\!)_\gamma\}, u, \mathfrak{g}, \lambda)$$

$$(\gamma, \{\!\{\mathsf{return}(v)\}\!\}_{\gamma'}^{(\mathsf{f},a)}, \mathfrak{g}, \lambda) \to \begin{cases} (\gamma', \mathsf{return}((\!|v|\!)_\gamma), (\mathfrak{g}_L, \mathfrak{g}_R, E \cup \{\mathsf{f} \xrightarrow{(\!|v|\!)_\gamma} a\}), \lambda) \\ \qquad\qquad \text{if } (\!|v|\!)_\gamma \in \{\mathsf{true}, \mathsf{false}\} \\ \text{else, failure (cannot memoize a non-boolean function)} \end{cases}$$

$$(\gamma, \mathsf{let\ val}\ x \leftarrow \lambda_{\text{ם}}\, y.\, u\ \mathsf{in}\ t, \mathfrak{g}, \lambda) \to \big(\gamma \sqcup \{x \mapsto \mathsf{f}\}, t, (\mathfrak{g}_L \sqcup \{\mathsf{f}\}, \mathfrak{g}_R, E \sqcup \{f \xrightarrow{\perp} a\}_{a \in \mathfrak{g}_R}), \lambda \sqcup \{\mathsf{f} \mapsto (\lambda_{\text{ם}}\, y.\, u, \gamma)\}\big)$$

$$(\gamma, \mathsf{let\ val}\ x \leftarrow \mathsf{fresh}()\ \mathsf{in}\ t, \mathfrak{g}, \lambda) \to \big(\gamma \sqcup \{x \mapsto a\}, t, (\mathfrak{g}_L, \mathfrak{g}_R \sqcup \{a\}, \mathfrak{g}_R, E \sqcup \{\mathsf{f} \xrightarrow{\perp} a\}_{\mathsf{f} \in \mathfrak{g}_L}), \lambda\big)$$

$$(\gamma, (v@w), \mathfrak{g}, \lambda) \to \begin{cases} (\gamma, \mathsf{return}(\beta), \mathfrak{g}, \lambda) & \text{if } \beta \stackrel{\text{def}}{=} E((\!|v|\!)_\gamma, (\!|w|\!)_\gamma) \neq \perp \\ (\gamma_0 \sqcup \{y \mapsto (\!|w|\!)_\gamma\}, \{\!\{u\}\!\}_{\gamma}^{\mathsf{f},a}, \mathfrak{g}, \lambda) & \text{else,} \\ \qquad \text{where } \lambda((\!|v|\!)_\gamma) \stackrel{\text{def}}{=} (\lambda_{\text{ם}}\, y.\, u, \gamma_0) & \end{cases}$$

$$(\gamma, v = w, \mathfrak{g}, \lambda) \to (\gamma, \mathsf{return}(\beta), \mathfrak{g}, \lambda) \text{ where } \beta \stackrel{\text{def}}{=} ((\!|v|\!)_\gamma = (\!|w|\!)_\gamma)$$

$$(\gamma, \mathsf{flip}(\theta), \mathfrak{g}, \lambda) \xrightarrow{\text{with proba. } \theta} (\gamma, \mathsf{return}(\beta), \mathfrak{g}, \lambda) \quad \text{where } \beta \in \{\mathsf{true}, \mathsf{false}\}$$

$$(\gamma, \mathsf{match}\ v\ \mathsf{as}\ (x, y)\ \mathsf{in}\ t, \mathfrak{g}, \lambda) \to (\gamma \sqcup \{x \mapsto (\!|v|\!)_\gamma, y \mapsto (\!|w|\!)_\gamma\}, t, \mathfrak{g}, \lambda)$$

$$(\gamma, \mathsf{if}\ v\ \mathsf{then}\ t\ \mathsf{else}\ u) \to \begin{cases} (\gamma, t, \mathfrak{g}, \lambda) & \text{if } v = \mathsf{true} \\ (\gamma, u, \mathfrak{g}, \lambda) & \text{else, if } v = \mathsf{false} \end{cases}$$

$$\frac{(\gamma, e, \mathfrak{g}, \lambda) \to (\gamma', e', \mathfrak{g}', \lambda')}{(\gamma, \mathcal{C}[e], \mathfrak{g}, \lambda) \to (\gamma', \mathcal{C}[e'], \mathfrak{g}', \lambda')}$$

Example 4.6 We now give an example showcasing how these reduction rules apply on a program combining name generation, a coin flip, function abstraction, and stochastic memoization. An atom x_0 is generated and used as an argument for a function f_1, which performs a coin flip if the argument matches x_0. The outcome is then

memoized and the result is returned in the second application. There are two execution traces, depending on the outcome of the coin flip ($\beta \in \mathsf{true}, \mathsf{false}$).

$$
\begin{aligned}
&\Big(\emptyset,\quad \mathsf{let\ val}\ x_0 \leftarrow \mathsf{fresh}()\ \mathsf{in} \\
&\qquad \mathsf{let\ val}\ f_1 \leftarrow \lambda_{\text{ם}}\, x.\ (\mathsf{let\ val}\ b \leftarrow (x = x_0)\ \mathsf{in} \\
&\qquad\qquad \mathsf{if}\ b\ \mathsf{then}\ \mathsf{flip}(\tfrac{1}{2})\ \mathsf{else}\ \mathsf{false})\ \mathsf{in} \\
&\qquad \mathsf{let\ val}\ f_2 \leftarrow \lambda_{\text{ם}}\, y.\ f_1@y\ \mathsf{in}\ f_2@x_0, \\
&\quad (\emptyset,\emptyset,\emptyset),\ \emptyset\Big)
\end{aligned}
$$

$$
\begin{aligned}
&\Big(\{x_0 \mapsto a_0\}, \\
&\quad \mathsf{let\ val}\ f_1 \leftarrow \lambda_{\text{ם}}\, x.\ (\mathsf{let\ val}\ b \leftarrow (x = x_0)\ \mathsf{in} \\
&\qquad\qquad \mathsf{if}\ b\ \mathsf{then}\ \mathsf{flip}(\tfrac{1}{2})\ \mathsf{else}\ \mathsf{false})\ \mathsf{in} \\
&\quad \mathsf{let\ val}\ f_2 \leftarrow \lambda_{\text{ם}}\, y.\ f_1@y\ \mathsf{in}\ f_2@x_0, \\
&\quad (\emptyset,\{a_0\},\emptyset),\ \emptyset\Big)
\end{aligned}
$$

$$
\begin{aligned}
&\to^2 \Big(\overbrace{\{x_0 \mapsto a_0,\ f_1 \mapsto \mathsf{f}_1,\ f_2 \mapsto \mathsf{f}_2\}}^{\stackrel{\text{def}}{=}\gamma_0},\quad f_2@x_0, \\
&\qquad (\{\mathsf{f}_1,\mathsf{f}_2\},\ \{a_0\},\{\mathsf{f}_1 \xrightarrow{\perp} a_0,\ \mathsf{f}_2 \xrightarrow{\perp} a_0\}), \\
&\qquad \big\{\mathsf{f}_1 \mapsto (\lambda_{\text{ם}}\, x.\ \mathsf{let\ val}\ b \leftarrow (x = x_0)\ \mathsf{in} \\
&\qquad\qquad \mathsf{if}\ b\ \mathsf{then}\ \mathsf{flip}(\tfrac{1}{2})\ \mathsf{else}\ \mathsf{false}), \{x_0 \mapsto a_0\}), \\
&\qquad \mathsf{f}_2 \mapsto (\lambda_{\text{ם}}\, y.\ f_1@y, \{x_0 \mapsto a_0, f_1 \mapsto \mathsf{f}_1\})\big\}\Big) \\
&\to \Big(\{x_0 \mapsto a_0,\ x \mapsto a_0\}, \\
&\qquad \Big\{\!\Big\{ \{\!\{\mathsf{let\ val}\ b \leftarrow (x = x_0)\ \mathsf{in} \\
&\qquad\qquad \mathsf{if}\ b\ \mathsf{then}\ \mathsf{flip}(\tfrac{1}{2})\ \mathsf{else}\ \mathsf{false}\}\!\}^{\mathsf{f}_1,a_0}_{\gamma_1} \Big\}\!\Big\}^{\mathsf{f}_2,a_0}_{\gamma_0}, \\
&\qquad (\{\mathsf{f}_1,\mathsf{f}_2\},\ \{a_0\},\{\mathsf{f}_1 \xrightarrow{\perp} a_0,\ \mathsf{f}_2 \xrightarrow{\perp} a_0\}), \\
&\qquad \big\{\mathsf{f}_1 \mapsto (\lambda_{\text{ם}}\, x.\ \mathsf{let\ val}\ b \leftarrow (x = x_0)\ \mathsf{in} \\
&\qquad\qquad \mathsf{if}\ b\ \mathsf{then}\ \mathsf{flip}(\tfrac{1}{2})\ \mathsf{else}\ \mathsf{false}), \{x_0 \mapsto a_0\}), \\
&\qquad \mathsf{f}_2 \mapsto (\lambda_{\text{ם}}\, y.\ f_1@y, \{x_0 \mapsto a_0, f_1 \mapsto \mathsf{f}_1\})\big\}\Big) \\
&\to \Big(\{x_0 \mapsto a_0,\ x \mapsto a_0,\ b \mapsto \mathsf{true}\}, \\
&\qquad \Big\{\!\Big\{ \{\!\{\mathsf{flip}(\tfrac{1}{2})\}\!\}^{\mathsf{f}_1,a_0}_{\gamma_1} \Big\}\!\Big\}^{\mathsf{f}_2,a_0}_{\gamma_0}, \\
&\qquad (\{\mathsf{f}_1,\mathsf{f}_2\},\ \{a_0\},\{\mathsf{f}_1 \xrightarrow{\perp} a_0,\ \mathsf{f}_2 \xrightarrow{\perp} a_0\}), \\
&\qquad \big\{\mathsf{f}_1 \mapsto (\lambda_{\text{ם}}\, x.\ \mathsf{let\ val}\ b \leftarrow (x = x_0)\ \mathsf{in} \\
&\qquad\qquad \mathsf{if}\ b\ \mathsf{then}\ \mathsf{flip}(\tfrac{1}{2})\ \mathsf{else}\ \mathsf{false}), \{x_0 \mapsto a_0\}), \\
&\qquad \mathsf{f}_2 \mapsto (\lambda_{\text{ם}}\, y.\ f_1@y, \{x_0 \mapsto a_0, f_1 \mapsto \mathsf{f}_1\})\big\}\Big) \\
&\to^2 \Big(\overbrace{\{x_0 \mapsto a_0,\ f_1 \mapsto \mathsf{f}_1,\ f_2 \mapsto \mathsf{f}_2\}}^{\stackrel{\text{def}}{=}\gamma_0},\quad \mathsf{return}(\beta),\quad (\{\mathsf{f}_1,\mathsf{f}_2\},\ \{a_0\},\{\mathsf{f}_1 \xrightarrow{\beta} a_0,\ \mathsf{f}_2 \xrightarrow{\beta} a_0\}), \\
&\qquad \big\{\mathsf{f}_1 \mapsto (\lambda_{\text{ם}}\, x.\ \mathsf{let\ val}\ b \leftarrow (x = x_0)\ \mathsf{in\ if}\ b\ \mathsf{then}\ \mathsf{flip}(\tfrac{1}{2})\ \mathsf{else}\ \mathsf{false}), \{x_0 \mapsto a_0\}), \\
&\qquad \mathsf{f}_2 \mapsto (\lambda_{\text{ם}}\, y.\ f_1@y, \{x_0 \mapsto a_0, f_1 \mapsto \mathsf{f}_1\})\big\}\Big)
\end{aligned}
$$

$$
\begin{aligned}
&\to \Big(\{x_0 \mapsto a_0,\ x \mapsto a_0,\ b \mapsto \mathsf{true}\}, \\
&\qquad \Big\{\!\Big\{ \{\!\{\mathsf{if}\ b\ \mathsf{then}\ \mathsf{flip}(\tfrac{1}{2})\ \mathsf{else}\ \mathsf{false}\}\!\}^{\mathsf{f}_1,a_0}_{\gamma_1} \Big\}\!\Big\}^{\mathsf{f}_2,a_0}_{\gamma_0}, \\
&\qquad (\{\mathsf{f}_1,\mathsf{f}_2\},\ \{a_0\},\{\mathsf{f}_1 \xrightarrow{\perp} a_0,\ \mathsf{f}_2 \xrightarrow{\perp} a_0\}), \\
&\qquad \big\{\mathsf{f}_1 \mapsto (\lambda_{\text{ם}}\, x.\ \mathsf{let\ val}\ b \leftarrow (x = x_0)\ \mathsf{in} \\
&\qquad\qquad \mathsf{if}\ b\ \mathsf{then}\ \mathsf{flip}(\tfrac{1}{2})\ \mathsf{else}\ \mathsf{false}), \{x_0 \mapsto a_0\}), \\
&\qquad \mathsf{f}_2 \mapsto (\lambda_{\text{ם}}\, y.\ f_1@y, \{x_0 \mapsto a_0, f_1 \mapsto \mathsf{f}_1\})\big\}\Big) \\
&\to \Big(\{x_0 \mapsto a_0,\ x \mapsto a_0,\ b \mapsto \mathsf{true}\}, \\
&\qquad \Big\{\!\Big\{ \{\!\{\mathsf{return}(\beta)\}\!\}^{\mathsf{f}_1,a_0}_{\gamma_1} \Big\}\!\Big\}^{\mathsf{f}_2,a_0}_{\gamma_0}, \\
&\qquad (\{\mathsf{f}_1,\mathsf{f}_2\},\ \{a_0\},\{\mathsf{f}_1 \xrightarrow{\perp} a_0,\ \mathsf{f}_2 \xrightarrow{\perp} a_0\}), \\
&\qquad \big\{\mathsf{f}_1 \mapsto (\lambda_{\text{ם}}\, x.\ \mathsf{let\ val}\ b \leftarrow (x = x_0)\ \mathsf{in} \\
&\qquad\qquad \mathsf{if}\ b\ \mathsf{then}\ \mathsf{flip}(\tfrac{1}{2})\ \mathsf{else}\ \mathsf{false}), \{x_0 \mapsto a_0\}), \\
&\qquad \mathsf{f}_2 \mapsto (\lambda_{\text{ם}}\, y.\ f_1@y, \{x_0 \mapsto a_0, f_1 \mapsto \mathsf{f}_1\})\big\}\Big)
\end{aligned}
$$

$$
\begin{aligned}
&\to^2 \Big(\overbrace{\{x_0 \mapsto a_0,\ f_1 \mapsto \mathsf{f}_1,\ f_2 \mapsto \mathsf{f}_2\}}^{\stackrel{\text{def}}{=}\gamma_1},\quad \{\!\{f_1@y\}\!\}^{\mathsf{f}_2,a_0}_{\gamma_0}, \\
&\qquad (\{\mathsf{f}_1,\mathsf{f}_2\},\ \{a_0\},\{\mathsf{f}_1 \xrightarrow{\perp} a_0,\ \mathsf{f}_2 \xrightarrow{\perp} a_0\}), \\
&\qquad \big\{\mathsf{f}_1 \mapsto (\lambda_{\text{ם}}\, x.\ \mathsf{let\ val}\ b \leftarrow (x = x_0)\ \mathsf{in} \\
&\qquad\qquad \mathsf{if}\ b\ \mathsf{then}\ \mathsf{flip}(\tfrac{1}{2})\ \mathsf{else}\ \mathsf{false}), \{x_0 \mapsto a_0\}), \\
&\qquad \mathsf{f}_2 \mapsto (\lambda_{\text{ם}}\, y.\ f_1@y, \{x_0 \mapsto a_0, f_1 \mapsto \mathsf{f}_1\})\big\}\Big)
\end{aligned}
$$

Configuration judgements.
We now show that the operational semantics satisfies the memoization equations eq. (2). Initial configurations are of the form $(\emptyset, e, \emptyset, \emptyset)$, where e is a non-extended expression. One can associate a *configuration judgment* $\mathrm{J}(\emptyset, e, \emptyset, \emptyset) \stackrel{\text{def}}{=} \emptyset \mid \emptyset \vdash^{\varsigma} e : A$ to every initial configuration. A configuration $(\gamma, e, \mathfrak{g}, \lambda)$ is said to be *accessible* (from $(\emptyset, e_0, \emptyset, \emptyset)$) if there exists a reduction trace $s = (\emptyset, e_0, \emptyset, \emptyset) \to^* (\gamma, e, \mathfrak{g}, \lambda)$ with probability > 0. The big-step

operational semantics of an initial configuration is defined in a standard way as the resulting probability distribution over the set of configurations accessible from it. We can then prove that a configuration is accessible only if it has a corresponding configuration judgment that is derivable:

Lemma 4.7 *If a configuration $(\gamma, e, \mathfrak{g}, \lambda)$ is accessible, there exists a corresponding* configuration judgement $\mathrm{J}(\gamma, e, \mathfrak{g}, \lambda) \stackrel{def}{=} \Gamma \mid \Delta \vdash^{\complement} e : A$ *where $\gamma \in (\!|\Gamma|\!)$ and such that $\mathrm{J}(\gamma, e, \mathfrak{g}, \lambda)$ is derivable (with tables 1 and 2).*

Due to the fact that we have at most one redex per (extended) expression and we do not have recursion (so the dataflow graph does not have self-loops and is acyclic), we can prove that:

Lemma 4.8 *If a configuration of the form $(\gamma, \mathcal{C}[v@w], \mathfrak{g}, \lambda)$ is accessible and $E((\!|v|\!)_\gamma, (\!|w|\!)_\gamma) = \bot$, then $J(\gamma, \mathcal{C}[v@w], \mathfrak{g}, \lambda) \stackrel{def}{=} \Gamma \mid \Delta \vdash^{\complement} \mathcal{C}[v@w] : A$ is such that the memoization stack Δ does not contain a function–atom label pair with $(\!|v|\!)_\gamma$ as first component.*

As a corollary, we can then prove that a configuration is accessible only if its memoization stack has no duplicates:

Lemma 4.9 *If a configuration $(\gamma, e, \mathfrak{g}, \lambda)$ is accessible and $\mathrm{J}(\gamma, e, \mathfrak{g}, \lambda) \stackrel{def}{=} \Gamma \mid \Delta \vdash^{\complement} e : A$ is its corresponding configuration judgment, there is no duplicate in Δ.*

This in turn enables us to ensure that the operational semantics satisfies the memoization equations:

Proposition 4.10 *If e_1 and e_2 are programs of the form*

$$e_1 \stackrel{def}{=} \mathsf{let\ val}\ x \leftarrow \mathsf{fresh}()\ \mathsf{in\ let\ val}\ f \leftarrow \lambda_{\text{M}} y.\, e\ \mathsf{in\ let\ val}\ v_1 \leftarrow f@x\ \mathsf{in\ let\ val}\ v_2 \leftarrow f@x\ \mathsf{in\ return}(v_1, v_2)$$
$$e_2 \stackrel{def}{=} \mathsf{let\ val}\ x \leftarrow \mathsf{fresh}()\ \mathsf{in\ let\ val}\ f \leftarrow \lambda_{\text{M}} y.\, e\ \mathsf{in\ let\ val}\ v_1 \leftarrow f@x\ \mathsf{in\ return}(v_1, v_1)$$

the configurations $(\emptyset, e_1, \emptyset, \emptyset)$ and $(\emptyset, e_2, \emptyset, \emptyset)$ have the same big-step operational semantics.

5 Denotational Semantics

In this section we propose a denotational model that verifies the dataflow property (Def. 2.3, Theorem 5.5) and which supports memoization of constant Bernoulli functions (Theorem 5.9) and is sound with respect to the operational semantics of Section 4 (Theorem 5.11). Thus we show that criteria (1)–(5) of Section 1 are consistent.
The memo-tables in memoization are a kind of hidden or local state, and our semantic domain is similar to other models of local state [28, 37, 44, 46] in that it uses a possible worlds semantics in the guise of a functor category.

Definition 5.1 A *total bigraph* is a partial bigraph (Def. 4.1) that does not have any undefined ($\bot$) elements. This represents a fully populated memo-table. We notate this $g = (g_L, g_R, E^g)$, omitting the superscript when it is clear. An *embedding* between total bigraphs $\iota \colon g \to g'$ is a pair of injections $(\iota_L : g_L \to g'_L, \iota_R : g_R \to g'_R)$ that do not add or remove edges ($E^g(\mathrm{f}, a) = E^{g'}(\iota_L(\mathrm{f}), \iota_R(a))$). These can be thought of as conservative extensions of the memo-table. We let $\mathbf{BiGrph}_{emb}$ be the category where the objects are total finite bigraphs and graph embeddings.

We will interpret our types as covariant presheaves, i.e. functors in $[\mathbf{BiGrph}_{emb}, \mathbf{Set}]$, and programs will be interpreted as natural transformations. We discuss this category in Section 5.1, before defining a monad (§5.2) and giving a denotational semantics (§5.3) and proving a soundness theorem (§5.4).

5.1 Base category

We work in the category $[\mathbf{BiGrph}_{emb}, \mathbf{Set}]$ of covariant presheaves on the category $\mathbf{BiGrph}_{emb}$ of finite bigraphs. The types of the language A are interpreted as presheaves $[\![A]\!]$. The idea is that once some functions and atomic names are fixed, and a memo-table g for them is given, then we can say what the values or expressions are, $[\![A]\!](g)$. The values can be renamed by permuting functions and atomic names, and are monotonic in that they remain unchanged when we conservatively extend the memo-table. This is the functorial action, $[\![A]\!]\iota : [\![A]\!](g) \to [\![A]\!](g')$. Programs will be interpreted as natural transformations: the naturality ensures that they are invariant under permuting the functions and atomic names, or extending the memo-table.

We write $\circ$ and $\bullet$ for the one-vertex left and right graphs respectively. The denotation of basic types is given by:

$$[\![\mathbb{F}]\!] \stackrel{\text{def}}{=} \mathbf{BiGrph}_{emb}(\circ, -) \quad [\![\mathbb{A}]\!] \stackrel{\text{def}}{=} \mathbf{BiGrph}_{emb}(\bullet, -) \quad \text{so that } [\![\mathbb{F}]\!](g) \cong g_L,\ [\![\mathbb{A}]\!](g) \cong g_R.$$

The presheaf category $[\mathbf{BiGrph}_{emb}, \mathbf{Set}]$ has products and coproducts, given pointwise [35]. In particular, the denotation of the type of booleans is the constant presheaf $2 \cong 1 + 1$.
The edge relations collect to form a natural transformation $\mathcal{E} : [\![\mathbb{F}]\!] \times [\![\mathbb{A}]\!] \to 2$ given by $\mathcal{E}_g(\mathrm{f}, a) = E^g(\mathrm{f}, a)$.
The category $[\mathbf{BiGrph}_{emb}, \mathbf{Set}]$ is cartesian closed, as is any presheaf category. By currying $\mathcal{E}$, we have an embedding of $[\![\mathbb{F}]\!]$ in the function space $2^{[\![\mathbb{A}]\!]}$, i.e. $[\![\mathbb{F}]\!] \to 2^{[\![\mathbb{A}]\!]}$. In fact, in this development to keep things simpler, we will focus on $[\![\mathbb{F}]\!]$ rather than the full function space $2^{[\![\mathbb{A}]\!]}$.

5.2 *Probabilistic local state monad*

In the following, $X, Y, Z : \mathbf{BiGrph}_{emb} \to \mathbf{Set}$ denote presheaves, $g = (g_L, g_R, E^g), g', h, h' \in \mathbf{BiGrph}_{emb}$ bigraphs, and $\iota, \iota' : g \hookrightarrow g'$ bigraph embeddings. We will omit subscripts when they are clear from the context.
Let P_{f} be the finite distribution monad: $P_{\mathrm{f}}(X)(g) = \left\{p : X(g) \to [0,1] \;\middle|\; \mathsf{supp}(p) \text{ finite and } \sum_x p(x) = 1\right\}$. By considering the following 'node-generation' monad $N(X)(g) \stackrel{\text{def}}{=} \mathrm{colim}_{g \hookrightarrow h} X(h)$ on $[\mathbf{BiGrph}_{emb}, \mathbf{Set}]$, one could be tempted to think that modeling name generation and stochastic memoization is a matter of composing these two monads. But this is not quite enough. We also need to remember, in the monadic computations, the probability of a function returning true for a fresh, unseen atom. To do so, inspired from Plotkin and Power's local state monad [44] (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 the following monad, defined using a coend [35], that we name 'probabilistic local state monad':

Definition 5.2 [Probabilistic local state monad] For all covariant presheaves $X : \mathbf{BiGrph}_{emb} \to \mathbf{Set}$ and bigraphs $g \in \mathbf{BiGrph}_{emb}$:

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

The monad 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 yielding true on a new fresh atom. We will refer to such a λ as a *state of biases*. The coend 'glues together' the extensions of the memo-table that are compatible with the constraints imposed by the current computation. The monad allows manipulating probability distributions over such extensions, while keeping track of the probability of new nodes.
Equivalence classes in $\int^{g \hookrightarrow h} X(h) \times [0,1]^{(h-g)_L}$ are written $[x_h, \lambda^h]_g$. In the coend, the quotient can be thought of as taking care of garbage collection: nodes that are not used in the bigraph environment can be discarded. We use Dirac's bra-ket notation [3] $\left|[x_h, \lambda^h]_g\right\rangle_h$ 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 $\left\langle \boldsymbol{p} \;\middle|\; [x_h, \lambda^h]_g \right\rangle_h$.

Definition 5.3 [Action of $T(X)$ on morphisms]

$$T(X)(g \stackrel{\iota}{\hookrightarrow} g') : \begin{cases} \left(P_{\mathrm{f}} \int^{g \hookrightarrow h} X(h) \times [0,1]^{(h-g)_L}\right)^{[0,1]^{g_L}} \to \left(P_{\mathrm{f}} \int^{g' \hookrightarrow h'} X(h') \times [0,1]^{(h'-g')_L}\right)^{[0,1]^{g'_L}} \\ \vartheta \mapsto [0,1]^{g'_L} \xrightarrow{-\circ \iota_L} [0,1]^{g_L} \xrightarrow{\vartheta} P_{\mathrm{f}} \int^{g \hookrightarrow h} X(h) \times [0,1]^{(h-g)_L} \\ \qquad\qquad \xrightarrow{P_{\mathrm{f}} \psi_{g,g'}} P_{\mathrm{f}} \int^{g' \hookrightarrow h'} X(h') \times [0,1]^{(h'-g')_L} \end{cases}$$

where

[3] popularized by Bart Jacobs for finite probability distributions [24]

- $\iota_L : g_L \hookrightarrow g'_L$ is the embedding restricted to left nodes, the maps $\psi_{g,g'}$ are given by:

$$\frac{\begin{cases} X(h) \times [0,1]^{(h-g)_L} \to X(h \coprod_g g') \times [0,1]^{(h \coprod_g g' - g')_L} \to \int^{g' \hookrightarrow h'} X(h') \times [0,1]^{(h'-g')_L} \\ (x_h,\, \lambda^h) \mapsto (X(h \hookrightarrow h \coprod_g g')(x_h),\, \lambda^h) \mapsto [X(h \hookrightarrow h \coprod_g g')(x_h),\, \lambda^h]_{g'} \end{cases}}{\int^{g \hookrightarrow h} X(h) \times [0,1]^{(h-g)_L} \xrightarrow{\psi_{g,g'}} \int^{g' \hookrightarrow h'} X(h') \times [0,1]^{(h'-g')_L}} \text{ extranatural in } h$$

- and $h \coprod_g g'$ is the pushout in the category of graphs regarded as an object of $\mathbf{BiGrph}_{emb}$.

More concretely, with Dirac's bra-ket notation, $T(X)(g \overset{\iota}{\hookrightarrow} g')$ can be written as:

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

T can be endowed with the structure of a $[\mathbf{BiGrph}_{emb}, \mathbf{Set}]$-enriched monad, that is, since $[\mathbf{BiGrph}_{emb}, \mathbf{Set}]$ is a (cartesian) monoidal closed category, a strong monad. Its enriched unit $\eta_X : 1 \to TX^X$ and bind $(-)^* : TY^X \to TY^{TX}$ are as follows [4].

$$\eta_X(g) : \begin{cases} \{*\} \to [X \times \text{ɟ}(g), TX] \\ * \mapsto \mathfrak{y}g' : \begin{cases} X(g') \times \text{ɟ}(g)(g') \to TX(g') \\ x_{g'},\, \_ \mapsto \left([0,1]^{g'_L} \ni \lambda \mapsto 1 \cdot \left|[x_{g'}, !]_{g'}\right\rangle\right) \end{cases} \end{cases} \qquad (-)^* : \begin{cases} TY^X(g) \to \left[TX \times \text{ɟ}(g), TY\right] \\ \phi \mapsto \phi^* \end{cases}$$

where

$$\phi^*_{g'} : \begin{cases} \left(P_{\mathrm{f}} \displaystyle\int^{g' \hookrightarrow h} X(h) \times [0,1]^{(h-g')_L}\right)^{[0,1]^{g'_L}} \times \text{ɟ}(g)(g') \xrightarrow{\phi^*_{g'}} \left(P_{\mathrm{f}} \displaystyle\int^{g' \hookrightarrow h'} Y(h') \times [0,1]^{(h'-g')_L}\right)^{[0,1]^{g'_L}} \\ (\vartheta,\; g \overset{\iota}{\hookrightarrow} g') \overset{\phi^*_{g'}}{\longmapsto} \lambda' \mapsto \quad \text{let } \vartheta(\lambda') \;=\; \left\langle \boldsymbol{p} \,\middle|\, [x_h, \lambda^h]_{g'} \right\rangle_{h \in H_{g'}} \text{ in} \\ \qquad\qquad\qquad\quad \text{for each } h \in H_{g'}, \\ \qquad\qquad\qquad\quad\; \text{let } \phi_h(x_h, g \overset{\iota}{\hookrightarrow} g' \hookrightarrow h)(\lambda^h \sqcup \lambda') \;=\; \left\langle \boldsymbol{q_h} \,\middle|\, [y'_h, \gamma^{h'}]_h \right\rangle_{h' \in H_h} \text{ in} \\ \qquad\qquad\qquad\quad \left\langle \boldsymbol{p} \,\middle|\, Q \,\middle|\, [y_{h'}, \gamma^{h'} \sqcup \lambda^h]_{g'} \right\rangle_{h' \in \bigcup\limits_{h \in H_{g'}} H_h} \end{cases}$$

and $Q \overset{\text{def}}{=} \begin{pmatrix} \boldsymbol{q_{h_1}} \\ \vdots \\ \boldsymbol{q_{h_n}} \end{pmatrix}_{h_i \in H_{g'}} = \left(\begin{array}{c|c|c} q_{h_1,h'_1} & \cdots & q_{h_1,h'_m} \\ \vdots & & \vdots \\ q_{h_n,h'_1} & \cdots & q_{h_n,h'_m} \end{array}\right)_{\substack{h_i \in H_{g'} \\ h'_j \in \bigcup\limits_{h \in H_{g'}} H_h}}$ (where each $\boldsymbol{q_h}$ has been 0-padded accordingly)

As argued before, to construct an abstract model of probability, we show that the monad is commutative. Affineness straightforwardly stems from the following lemma:

Lemma 5.4 *Let X be a constant presheaf on the coslice category $g/\mathbf{BiGrph}_{emb}$, i.e. there exists a set S_0 such that $X(g \overset{\iota}{\hookrightarrow} h) = S_0 \overset{\mathrm{id}}{\to} S_0$ for every $g \overset{\iota}{\hookrightarrow} h \in g/\mathbf{BiGrph}_{emb}$. Then $T(X)(g) \cong P_{\mathrm{f}}(S_0)^{[0,1]^{g_L}}$.*

We have the desired dataflow property, meaning that T is an abstract model of probability [32]:

Theorem 5.5 *The monad T satisfies the dataflow property (2.3): it is strong commutative and affine.*

[4] following Fosco Loregiàn [34], ɟ : $\mathbf{BiGrph}_{emb} \to [\mathbf{BiGrph}_{emb}, \mathbf{Set}]^{\mathrm{op}}$ denotes the (contravariant) Yoneda embedding.

Proof (Sketch) In the presheaf category, let $Z^Y \times Y^X \xrightarrow{\circ} Z^X$ and $Z^Y \times Y \xrightarrow{\mathrm{ev}} Z$ denote the internal composition and evaluation, and $f^* \stackrel{\mathrm{def}}{=} 1 \xrightarrow{f} TY^X \xrightarrow{(-)^*} TY^{TX}$ the internal Kleisli lifting of a global element f. To prove that T is strong, we show, internally, the associativity $((\Psi_g^* \times \Phi_g^*)\,;\circ = ((\Psi^* \times \Phi)\,;\circ)^*)$ of the bind, the left unit law ($\eta^* = \lambda_{TX}.\mathrm{id}_{TX}$), and the right unit law $((\Phi^* \times \eta)\,;\circ = \Phi)$, for all $\Phi\colon 1 \to TY^X$, $\Psi\colon 1 \to TZ^Y$. Finally, affineness stems from lemma 5.4, and commutativity is the equation $a \gg\!= \lambda x.\, b \gg\!= \lambda y.\, \eta(x,y) \;=\; b \gg\!= \lambda y.\, a \gg\!= \lambda x.\, \eta(x,y)$ internally, for all $a\colon 1 \to TA, b\colon 1 \to TB$, which amounts to showing:

$$\Big(\Big(\lambda_A.\Big(((\lambda_B.\eta)^* \times b)\;;\;\mathrm{ev}\Big)\Big)^* \times a\Big)\;;\mathrm{ev} = \Big(\Big(\lambda_B.\Big(((\lambda_A.\eta)^* \times a)\;;\;\mathrm{ev}\Big)\Big)^* \times b\Big)\;;\mathrm{ev}$$

□

5.3 *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, $\mathbb{F}$, $\mathbb{A}$ and products. Therefore, $[\![\Gamma]\!]$ is isomorphic to a presheaf of the form $2^k \times \mathbf{BiGrph}_{emb}(\circ,-)^\ell \times \mathbf{BiGrph}_{emb}(\bullet,-)^m$, where k,ℓ,m are the numbers of booleans, functions and atoms in Γ, and X^n is is the n-fold finite product in the category of presheaves. Computations of type $\mathbb{A}$ and $\mathbb{F}$ then have an intuitive interpretation:

Proposition 5.6 *A computation of type*

- $\mathbb{A}$ *returns the label of an already existing atom or a fresh one with its connections to the already existing functions:*

$$T([\![\mathbb{A}]\!])(g) \;\cong\; P_{\mathrm{f}}(g_R + 2^{g_L})^{[0,1]^{g_L}}$$

- $\mathbb{F}$ *returns the label of an already existing function or create a new function with its connections to already existing atoms and a fixed probabilistic bias:*

$$T([\![\mathbb{F}]\!])(g) \;\cong\; P_{\mathrm{f}}(g_L + 2^{g_R} \times [0,1])^{[0,1]^{g_L}}$$

Definition 5.7 For every bigraph g, we denote by R_g (resp. L_g) the set of bigraphs $h \in g/\mathbf{BiGrph}_{emb}$ having one more right (resp. left) node than g, and that are the same otherwise. For every $e \in 2^{g_L}$ (resp. $e \in 2^{g_R}$), we denote by $g +_e \bullet \in R_g$ (resp. $g +_e \circ \in L_g$) the bigraph obtained by adding a new right (resp. left) node to g with connectivity e to the right (resp. left) nodes in g.

We now give the denotational semantics of various constructs in our language. Henceforth, we will denote normalization constants (that can easily be inferred from the context) by Z.

Denotations of $\Gamma \vdash^{\mathsf{c}} \mathsf{flip}(\theta) : \mathsf{bool}$**,** $\Gamma, v : \mathbb{F}, w : \mathbb{A} \vdash^{\mathsf{c}} v@w : \mathsf{bool}$**, and** $\Gamma, v : \mathbb{A}, w : \mathbb{A} \vdash^{\mathsf{c}} v = w : \mathsf{bool}$
First, by Lemma 5.4, we note that $T([\![\mathsf{bool}]\!])g \;\cong\; P_{\mathrm{f}}(2)^{[0,1]^{g_L}} \;\cong\; [0,1]^{[0,1]^{g_L}}$. So naturally, the map $[\![\mathsf{flip}(\theta)]\!]_g$ is the constant function returning the bias θ.

Denotations of $\Gamma, v : \mathbb{F}, w : \mathbb{A} \vdash^{\mathsf{c}} v@w : \mathsf{bool}$**, and** $\Gamma, v : \mathbb{A}, w : \mathbb{A} \vdash^{\mathsf{c}} v = w : \mathsf{bool}$
The map $[\![v@w]\!]_g : [\![\Gamma, v : \mathbb{F}, w : \mathbb{A}]\!](g) \to [0,1]^{[0,1]^{g_L}}$ returns 1 if the left node corresponding to v is connected to the one of w in g, 0 otherwise. Using the internal edge relation $\mathcal{E}$, it is the internal composition:

$$[\![v@w]\!] \stackrel{\mathrm{def}}{=} 1 \times ([\![\Gamma]\!] \times [\![\mathbb{F}]\!] \times [\![\mathbb{A}]\!]) \xrightarrow{\eta \times (! \times \mathcal{E})} T([\![\mathsf{bool}]\!])^{[\![\mathsf{bool}]\!]} \times [\![\mathsf{bool}]\!] \xrightarrow{\mathrm{ev}} T([\![\mathsf{bool}]\!])$$

And similarly, the map $[\![v = w]\!]_g : [\![\Gamma, v : \mathbb{A}, w : \mathbb{A}]\!](g) \to [0,1]^{[0,1]^{g_L}}$ is given by:

$$[\![v = w]\!] \stackrel{\mathrm{def}}{=} [\![\Gamma]\!] \times [\![\mathbb{A}]\!]^2 \cong 1 \times [\![\Gamma]\!] \times \Big(\text{よ}(\bullet) + \text{よ}(\bullet + \bullet)\Big) \xrightarrow{\eta \times ! \times \left[!\,;\,\iota_{\mathsf{true}},\,!\,;\,\iota_{\mathsf{false}}\right]} T([\![\mathsf{bool}]\!])^{[\![\mathsf{bool}]\!]} \times [\![\mathsf{bool}]\!] \xrightarrow{\mathrm{ev}} T([\![\mathsf{bool}]\!])$$

where $[-,\,-]$ is the copairing and $\iota_{\mathsf{true}}, \iota_{\mathsf{false}} \colon 1 \to [\![\mathsf{bool}]\!] \cong 2$ are the coprojections.

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.

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

It suffices to consider the bigraphs that belong to R_g only, by garbage collection of the coend.

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 can decompose the extra variable x in the environment $\Gamma,\, x : \mathbb{A}$, the denotation of which is of the form $[\![\Gamma,\, x : \mathbb{A}]\!](g) = 2^k \times \mathbf{BiGrph}_{emb}(\circ, g)^\ell \times \mathbf{BiGrph}_{emb}(\bullet, g)^m \times \mathbf{BiGrph}_{emb}(\bullet,\, g)$ for a bigraph $g \in \mathbf{BiGrph}_{emb}$. 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, the denotation $[\![u]\!]_g \colon 2^k \times \mathbf{BiGrph}_{emb}(\circ, g)^\ell \times \mathbf{BiGrph}_{emb}(\bullet, g)^m \times \mathbf{BiGrph}_{emb}(\bullet, g) \to [0,1]^{[0,1]^{g_L}}$ gives us the edge probability of the left node (atom) that we need to generate, both to the existing right nodes (functions), and to any future right node (which needs to be remembered). This can be formalized into a natural transformation $[\![\lambda_{\text{מ}}\, x.\, u]\!] \colon [\![\Gamma]\!] \to T([\![\mathbb{F}]\!])$, provided that u satisfies the following property:

Definition 5.8 [Freshness-invariant functions] A function $\lambda_{\text{מ}}\, x.\, u$ is *freshness-invariant* if, for every $g, b^k \in 2^k$, $\kappa_i \colon \circ \hookrightarrow g$, $\tau_j \colon \bullet \hookrightarrow g$ and $\lambda \in [0,1]^{g_L}$, we have (where $\iota_1,\, \iota_2$ are the coprojections):

$$\forall e \in 2^{g_L},\ [\![u]\!]_g\big(b^k,\ (\circ \overset{\kappa_i}{\hookrightarrow} g \overset{\iota_1}{\hookrightarrow} g +_e \bullet)_i,\ (\bullet \overset{\tau_j}{\hookrightarrow} g \overset{\iota_1}{\hookrightarrow} g +_e \bullet)_j,\ \bullet \overset{\iota_2}{\hookrightarrow} g +_e \bullet,\ \lambda\big) \text{ is a constant } \tilde{p}_u$$

A sufficient condition to ensure that a function of the form $\lambda_{\text{מ}}\, x.\, u$ be freshness-invariant is that it has no subexpression of the form $f@y$, where $y \notin \mathsf{fv}(\lambda_{\text{מ}}\, x.u)$. An example thereof is $\lambda_{\text{מ}}\, x.\mathsf{let\ val}\ b \leftarrow f@x_0\ \mathsf{in\ if}\ b\ \mathsf{then\ true\ else}\ (x = x_0)$. Non examples are $\lambda_{\text{מ}}\, x.\ \mathsf{let\ val}\ y \leftarrow \mathsf{fresh}()\ \mathsf{in}\ f@y$ and $\lambda_{\text{מ}}\, x.\ \mathsf{if}\ f@x\ \mathsf{then\ false\ else\ true}$ (negation of f). We can interpret freshness-invariant functions as follows:

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

where $p_a \overset{\text{def}}{=} [\![u]\!]_g\big(b^k,\ (\circ \overset{\kappa_i}{\hookrightarrow} g)_i,\ (\bullet \overset{\tau_j}{\hookrightarrow} g)_j,\ \bullet \overset{a}{\hookrightarrow} g,\ \lambda\big)$ for every $a \in g_R$, and $\tilde{p}_u$ is as in Def. 5.8. As a result, the probabilistic local state monad validates (2):

Theorem 5.9 *The monad T supports stochastic memoization (Def. 2.1) for freshness-invariant functions (Def. 5.8), which include any function $\lambda_{\text{מ}}\, x.\, u$ that does not contain a subexpression of the form $f@y$, where $y \notin \mathsf{fv}(\lambda_{\text{מ}}\, x.\, u)$ (so, in particular, constant Bernoulli functions).*

Proof (Sketch) The denotation of $\lambda_{\text{מ}}$-abstractions enables us to define a map $T([\![\mathsf{bool}]\!])^{[\![\mathbb{A}]\!]} \to T(\mathbb{F})$, which can in turn be postcomposed by $T(\mathbb{F}) \xrightarrow{\phi} T\big([\![\mathsf{bool}]\!]^{[\![\mathbb{A}]\!]}\big)$, where

$$\phi_g \colon \begin{cases} T(\mathbb{F})(g) \cong P_{\mathrm{f}}(g_R + 2^{g_L})^{[0,1]^{g_L}} \to [0,1]^{[0,1]^g \times (g_R + 2^{g_L})} \cong T\big([\![\mathsf{bool}]\!]^{[\![\mathbb{A}]\!]}\big)(g) \\ \vartheta \mapsto (\lambda, a) \in [0,1]^g \times (g_R + 2^{g_L}) \mapsto \text{let } \vartheta(\lambda) \;=\; \sum_{a' \in g_R + 2^{g_L}} p_{a'} \, |a'\rangle \text{ in } p_a \end{cases}$$

to obtain mem: $T([\![\mathsf{bool}]\!])^{[\![\mathbb{A}]\!]} \to T([\![\mathsf{bool}]\!]^{[\![\mathbb{A}]\!]})$, and then we show eq. (1) in the presheaf topos. □

Example 5.10 The denotation of $\mathsf{let\ val}\ x \leftarrow \mathsf{fresh}()\ \mathsf{in\ let\ val}\ f \leftarrow \lambda_{\text{ᵯ}}\, y.\ \mathsf{flip}(\theta)\ \mathsf{in}\ f@x$ is the map

$$1 \times 1 \xrightarrow{\left(\lambda_{[\![\mathbb{A}]\!]}.\left(\left((\lambda_{[\![\mathbb{F}]\!]}.f@x)^* \times (\lambda_{\text{ᵯ}}\, y.\ \mathsf{flip}(\theta))\right);\mathrm{ev}\right)\right)^* \times \mathsf{fresh}()} T([\![\mathsf{bool}]\!])^{T[\![\mathbb{A}]\!]} \times T([\![\mathbb{A}]\!]) \xrightarrow{\mathrm{ev}} T([\![\mathsf{bool}]\!])$$

given by $*, * \mapsto \lambda \mapsto \theta\,|\mathsf{true}\rangle + (1-\theta)\,|\mathsf{false}\rangle$, as desired.

5.4 *Soundness*

Configurations of the form $(\gamma, e, \mathfrak{g}, \lambda)$, where e is of type A, can be denotationally interpreted as

$$[\![(\gamma, e, \mathfrak{g}, \lambda)]\!] \overset{\text{def}}{=} \sum_{\tilde{e} \in 2^{U_{\mathfrak{g}}}} \prod_{(\mathfrak{f},a) \in U_{\mathfrak{g}}} \lambda(\mathfrak{f})^{\tilde{e}(\mathfrak{f},a)} \big(1 - \lambda(\mathfrak{f})\big)^{1-\tilde{e}(\mathfrak{f},a)} [\![u]\!]_{\mathfrak{g}_{\tilde{e}}}(\gamma, \lambda) \in T(A)_{\mathfrak{g}_{\tilde{e}}}(\gamma)(\lambda)$$

where $U_{\mathfrak{g}} \overset{\text{def}}{=} \big\{(\mathfrak{f}, a) \mid E(\mathfrak{f}, a) = \bot\big\} \subseteq \mathfrak{g}_L \times \mathfrak{g}_R$ and $\mathfrak{g}_{\tilde{e}}$ extends $\mathfrak{g}$ according to $\tilde{e}$: $E(\mathfrak{f}, a) = \tilde{e}(\mathfrak{f}, a)$ for all $(\mathfrak{f}, a) \in U_{\mathfrak{g}}$. We can then prove that the denotational semantics is sound with respect to the operational semantics:

Theorem 5.11 (Soundness)

$$[\![(\gamma, e, \mathfrak{g}, \lambda)]\!] \cong \sum_{\substack{(\gamma,e,\mathfrak{g},\lambda) \to (\gamma',e',\mathfrak{g}',\lambda') \\ \textit{with proba. } p}} p \cdot [\![(\gamma', e', \mathfrak{g}', \lambda')]\!]$$

Proof (Sketch) As an intermediate step, we build a big-step semantics, and show that this is sound, i.e. making a small step of the operational semantics (§4) does not change the distributions in the final big-step semantics. Next, we show that the big step semantics of a configuration corresponds to the denotational semantics, for which the main thing to check is that the equivalence classes of the coend are respected. □

6 Haskell Implementation

We have a practical Haskell implementation comparing the small-step, big-step operational, and denotational semantics to showcase the soundness theorem with QuickCheck, in a setting analogous (albeit slightly different [5], to better suit the specificities of Haskell) to the theoretical one we presented. The artefact is openly available [26].

7 Summary

In conclusion, we have successfully tackled the open problem of finding a semantic interpretation of stochastic memoization for a class of functions with diffuse domain that includes the constant Bernoulli functions. Our contributions pave the way for further exploration and development of probabilistic programming and the sound application of stochastic memoization in Bayesian nonparametrics.

[5] Unlike our mathematical framework, where we can memoize all freshness-invariant functions (5.8), our implementation only memoizes constant Bernoulli functions. Another key difference is that we could not implement coends in Haskell, so we used a global state monad transformer to manage the memoization bigraph, keeping track of edges between left nodes (function labels) and right nodes (atom labels) that have been sampled.

8 Acknowledgements

We are grateful to Nate Ackerman, Cameron Freer, Dan Roy and Hongseok Yang for various conversations over many years, relating to [54], name generation, stochastic memoization and subsequent developments. The presheaf category here is related to the Rado topos [4] that we have been exploring in ongoing work, with Jacek Karwowski and Sean Moss and the above four coauthors. Thanks to Dario Stein for discussions about name generation and for pointing out [27]. Thanks too to Swaraj Dash, Mathieu Huot, Ohad Kammar, Oleg Kiselyov, Alex Lew, and all in the Oxford group for many discussions about this topic. Finally, thank you to our reviewers for detailed feedback.

References

[1] Aumann, R. J., *Borel structures for function spaces*, Illinois Journal of Mathematics **5** (1961).

[2] Barthe, G., J.-P. Katoen and A. Silva, editors, "Foundations of probabilistic programming," CUP, 2021.

[3] Bizjak, A. and L. Birkedal, *Step-indexed logical relations for probability*, in: *Proc. FOSSACS 2015*, 2015.

[4] Caramello, O., *Fraïssé's construction from a topos-theoretic perspective*, Logica Universalis **8** (2014), pp. 261–281.

[5] Cassel, J., *Probabilistic programming with stochastic memoization*, The Mathematica Journal **16** (2014).

[6] Castellan, S. and H. Paquet, *Probabilistic programming inference via intensional semantics*, in: *Proc. ESOP 2019*, 2019.

[7] Cho, K. and B. Jacobs, *Disintegration and Bayesian inversion via string diagrams*, Math. Struct. Comput. Sci. **29** (2019), pp. 938–971.

[8] Crubillé, R., *Probabilistic stable functions on discrete cones are power series*, in: *Proc. LICS 2018*.

[9] Danos, V. and I. Garnier, *Dirichlet is natural*, in: *Proc. MPFS 2015*, 2015.

[10] Dash, S., Y. Kaddar, H. Paquet and S. Staton, *Affine monads and lazy structures for Bayesian programming*, Proc. ACM Program. Lang. **7** (2023).

[11] De Raedt, L. and A. Kimming, *Probabilistic (logic) programming concepts*, Mach. Learn. **100** (2015).

[12] Ehrhard, T., M. Pagani and C. Tasson, *Measurable cones and stable, measurable functions: a model for probabilistic higher-order programming*, in: *Proc. POPL 2018*, 2018.

[13] Fritz, T., *A synthetic approach to Markov kernels, conditional independence and theorems on sufficient statistics*, Adv. Math. **370** (2020).

[14] Ghosal, S. and A. van der Vaart, "Fundamentals of non-parametric Bayesian inference," CUP, 2017.

[15] Giry, M., *A categorical approach to probability theory*, Categorical Aspects of Topology and Analysis. Lecture Notes in Mathematics (1982).

[16] Goodman, N., *Semantics of mem is questionable*, Open Github issue for WebPPL (2018), https://github.com/probmods/webppl/issues/896.

[17] Goodman, N. D., V. K. Mansinghka, D. Roy, K. Bonawitz and J. B. Tenenbaum, *Church: A language for generative models*, in: *Proceedings of the Twenty-Fourth Conference on Uncertainty in Artificial Intelligence*, UAI'08 (2008), pp. 220–229.

[18] Goodman, N. D., T. J. O'Donnell and J. B. Tenenbaum, *Probabilistic models of cognition (ProbMods): Non-parametric models*, http://v1.probmods.org/non-parametric-models.html (2016).

[19] Goodman, N. D. and A. Stuhlmüller, *The Design and Implementation of Probabilistic Programming Languages*, http://dippl.org (2014), accessed: 2020-10-15.

[20] Goubault-Larrecq, J., X. Jia and C. Théron, *A domain-theoretic approach to statistical programming languages*, arxiv:2106.16190 (2021).

[21] Heunen, C., O. Kammar, S. Staton and H. Yang, *A convenient category for higher-order probability theory*, in: *Proc. LICS 2017*, 2017.

[22] Hinze, R., *Generalizing Generalized Tries*, Journal of Functional Programming **10** (2000), pp. 327–351.

[23] Jacobs, B., *Probabilities, distribution monads, and convex categories*, Theoret. Comput. Sci. **412** (2011).

[24] Jacobs, B., *New directions in categorical logic, for classical, probabilistic and quantum logic*, Log. Methods Comput. Sci. **11** (2015).

[25] Jia, X., B. Lindenhovius, M. W. Mislove and V. Zamdzhiev, *Commutative monads for probabilistic programming languages*, in: *Proc. LICS 2021*, 2021.

[26] Kaddar, Y. and S. Staton, *Stochastic memoization implementation* (2023), available at Github, https://github.com/youqad/stochastic-memoization-implementation.

[27] Kallianpur, G., "Stochastic Filtering Theory," Springer, 1980.

[28] Kammar, O., P. B. Levy, S. K. Moss and S. Staton, *A monad for full ground reference cells*, in: *Proc. LICS 2017*, 2017.

[29] Kiselyov, O., *Logic Programming in HANSEI* (2010).
URL https://okmij.org/ftp/kakuritu/logic-programming.html

[30] Kiselyov, O. and C. Shan, *Nested inference and lazy variables*, Code example for Hansei (2009), https://okmij.org/ftp/kakuritu/nested.ml.

[31] Kock, A., *Monads on symmetric monoidal closed categories*, Arch. Math. **21** (1970).

[32] Kock, A., *Commutative monads as a theory of distributions*, Theory and Applications of Categories **26** (2012).

[33] Levy, P. B., "Call-By-Push-Value: A Functional/Imperative Synthesis," Springer, Dordrecht, 2003.

[34] Loregian, F., "(Co)end Calculus," London Mathematical Society Lecture Note Series, Cambridge University Press, 2021.

[35] Mac Lane, S., "Categories for the Working Mathematician," Springer, 1971.

[36] Mardare, R., P. Panangaden and G. D. Plotkin, *Free complete Wasserstein algebras*, Log. Methods Comput. Sci. **14** (2018).

[37] Melliès, P.-A., *Local states in string diagrams*, in: *Proc. TLCA 2014*, 2014.

[38] Michie, D., *"Memo" functions and machine learning*, Nature **218** (1968), pp. 306–306.

[39] Milch, B., B. Marthi, S. Russell, D. Sontag, D. L. Ong and A. Kolobov, *BLOG: Probabilistic models with unknown objects*, in: *Introduction to Statistical Relational Learning*, 2007 .

[40] Milner, R., *What's in a name?*, in: A. Herbert and K. S. Jones, editors, *Computer Systems: theory, technology and applications*, Springer, 2004 .

[41] Moggi, E., *An abstract view of programming languages*, Technical report, Edinburgh University (1989), http://www.lfcs.inf.ed.ac.uk/reports/90/ECS-LFCS-90-113/.

[42] Moggi, E., *Notions of computation and monads*, Information and Computation (1991).

[43] Pitts, A. M., "Nominal Sets: Names and Symmetry in Computer Science," Number 57 in Cambridge Tracts in Theoretical Computer Science, Cambridge University Press, 2013.

[44] Plotkin, G. and J. Power, *Notions of computation determine monads*, in: *Proc. FOSSACS 2002*, 2002.

[45] Pollard, D., "A user's guide to measure-theoretic probability," CUP, 2002.

[46] Pym, D. J., P. W. O'Hearn and H. Yang, *Possible worlds and resources: the semantics of BI*, Theoret. Comput. Sci. (2004).

[47] Roy, D., V. K. Mansinghka, N. D. Goodman and J. Tenenbaum, *A stochastic programming perspective on nonparametric Bayes*, in: *ICML Workshop on Nonparametric Bayes*, 2008.

[48] Sabok, M., S. Staton, D. Stein and M. Wolman, *Probabilistic programming semantics for name generation*, in: *Proc. POPL 2021*, 2021.

[49] Stark, I., "Names and Higher-Order Functions," Ph.D. thesis, University of Cambridge (1994).

[50] Staton, S., *Instances of computational effects:an algebraic perspective*, in: *Proc. LICS 2013*, 2013.

[51] Staton, S., *Commutative semantics for probabilistic programming*, in: *Proc. ESOP 2017*, 2017.

[52] Staton, S., H. Paquet, S. Dash and Y. Kaddar, *LazyPPL: A Haskell library for probabilistic programming.* (2022).
URL https://lazyppl-team.github.io/

[53] Staton, S., D. Stein, H. Yang, L. Ackerman, C. E. Freer and D. M. Roy, *The Beta-Bernoulli process and algebraic effects*, 2018.

[54] Staton, S., H. Yang, N. L. Ackerman, C. Freer and D. Roy, *Exchangeable random process and data abstraction*, in: *PPS 2017*, 2017.

[55] Stein, D., "Structural Foundations for Probabilistic Programming Languages," Ph.D. thesis, University of Oxford (2021).

[56] Stein, D. and S. Staton, *Compositional semantics for probabilistic programs with exact conditioning*, in: *Proc. LICS 2021*, 2021.

[57] Wood, F. D., C. Archambeau, J. Gasthaus, L. James and Y. W. Teh, *A stochastic memoizer for sequence data*, in: *Proc. ICML 2009*, 2009.