Monads for measurable queries in probabilistic databases

Swaraj Dash and Sam Staton[1]

Department of Computer Science
University of Oxford
Oxford, United Kingdom

Abstract

We consider a bag (multiset) monad on the category of standard Borel spaces, and show that it gives a free measurable commutative monoid. Firstly, we show that a recent measurability result for probabilistic database queries (Grohe and Lindner, ICDT 2020) follows quickly from the fact that queries can be expressed in monad-based terms. We also extend this measurability result to a fuller query language. Secondly, we discuss a distributive law between probability and bag monads, and we illustrate that this is useful for generating probabilistic databases.

Keywords: Monads, probabilistic programming, probabilistic databases.

1 Introduction

Probabilistic databases cater for uncertainty in data. There may be uncertainty about whether rows should be in a database, or uncertainty about what values certain attributes should have.

For example, consider a database of movies. We might have a table that assigns the gross amount to each movie, which may be quite uncertain for older movies. We might have a table that records which actors appeared in which movies, and there may be uncertainty about whether a particular actor appeared in a given movie. The uncertainty might come from incorrect text processing, for example if the information was scraped off internet forums, or just noise in measurement, e.g. if the gross amount is difficult to calculate precisely. This is a simple example, but probabilistic databases have applications in other areas of information extraction as well as in scientific data management, medical records, and in data cleaning. See the textbook [27] for further examples.

In this paper, we argue that the semantics of probabilistic databases lies in combining a probability monad, P, with a bag monad B (aka multiset). This builds on the long-established tradition of using monads to structure computational effects in functional programming [30,3].

A good semantic analysis is important in view of the recent work of Grohe and Lindner [11,12] which builds on [17,5]. This new line of work breaks with the traditional approach of having a fixed finite support for the probabilistic database, and argues that the support should be infinite, possibly uncountable. For example, it may be that the gross takings from a movie are approximated as a real number taken from a normal distribution, and it may be that the number of actors appearing in a movie is unknown and unbounded. This leads to semantic complications and introduces issues of measurability.

1.1 Two monads

We argue that probabilistic databases are best understood as inhabitants of a set, or space,

$$P(B(X))$$

[1] Email: {swaraj.dash,sam.staton}@cs.ox.ac.uk

This paper will be published
in the proceedings of MFPS XXXVII
URL: https://www.coalg.org/calco-mfps2021/mfps/

where

- X is a space of all records (aka rows, tuples) that are allowed according to the schema. For example, in the movie database above, we put $X = \mathsf{MovieFact}$ where

$$\mathsf{MovieFact} \;=\; \Big(\mathsf{cast}\colon (\mathsf{Actor} \times \mathsf{Movie}) \;\uplus\; \mathsf{gross}\colon (\mathsf{Movie} \times \mathbb{R})\Big)$$

  since we can either record that an actor appeared in a movie, or that a movie had a certain gross. (Here we are using a standard notation for tagged disjoint unions.)
- B is a monad of bags (aka multisets). So $B(X)$ is the space of bags over X, and these are the *deterministic* databases for the given schema.
- P is a probability monad. So $P(B(X))$ is the space of probability distributions (or measures) over the space of deterministic databases, and these are the probabilistic databases for the given schema.

In the traditional case, studied in [27], the probability distributions have finite support. In the general setting proposed by [12], the support of a distribution is uncountable. This is formalized using measure theory, by placing a σ-algebra on X, by deriving a σ-algebra on $B(X)$ and $P(B(X))$. We can regard this as moving from the category of sets to a category of measurable spaces. As we will show (Theorem 3.9), the bag monad B extends to a monad on the category of measurable spaces. We can then regard P as the Giry monad on the category of measurable spaces [8].

We clarify a subtle point. The support of the distributions in $P(B(X))$ might be infinite, and this means that the set of records that have a chance of appearing in the database can be infinite. But this is a different issue from the sizes of the bags under consideration, which will always be finite. For example, there are infinite possibilities as to what the gross from a movie is, but the number of movies will always be finite. The number of actors in a movie is unbounded, but there is never an infinite cast list for a particular movie.

1.2 Measurable queries

In the deterministic setting, a query (aka view) translates a database from one schema to another. For example, we might ask,

"Which actors appeared in films that grossed at least $200m?" (1)

This is a function $q : B(\mathsf{MovieFact}) \to B(\mathsf{Actor})$. For probabilistic databases, the usual approach is to consider queries on deterministic databases, and then lift them to probabilistic databases. Semantically, this can regarded as the functorial action of the monad P, which gives a translation between probabilistic databases:

$$P(q) \;:\; P(B(\mathsf{MovieFact})) \to P(B(\mathsf{Actor}))$$

Notice that if there is uncertainty about whether an actor appeared in a movie, or about what the gross of the movie was, then this will lead to uncertainty about whether that actor should appear in this view.

This functorial action $P(q)$ amounts to pushing forward the probability measure. But this is only legitimate if the query q is measurable. In Theorem 4.1, we show that all queries are measurable provided they are definable in the standard BALG query language for bags [13].

Our proof of measurability is straightforward, because most of the BALG query operations are directly definable from the monad structure of B (Theorem 3.9). The remaining operations are easily definable from an fold construction (Theorem 3.6), which is connected to the fact that $B(X)$ is the free commutative monoid on X.

Measurability of a fragment of BALG is perhaps the main technical result of [12]. That work was groundbreaking, but here we have two additional contributions:

(i) we show that the full language BALG is measurable, which allows us to also treat aggregation queries within the same framework, and

(ii) we demonstrate that the proof of measurability is almost immediate from the categorical properties of the monad B.

We give the full details of BALG in Section 4. But for now we note that another way to see that the particular query (1) is measurable is that it can be written in the monad comprehension syntax as

$$q(b) \;=\; \{\!\!\{\; a \mid \mathsf{cast}(a, m) \leftarrow b, \mathsf{gross}(m', r) \leftarrow b, m = m', r > 200\,000\,000 \;\}\!\!\}$$

This comprehension syntax works for any strong monad (Section 2.2.1 and [30]), indeed it is merely a convenient

shorthand for

$$b \gg\!= \lambda x.\, b \gg\!= \lambda y. \begin{cases} \mathsf{return}(a) & \text{if } x = \mathsf{cast}(a, m),\ y = \mathsf{gross}(m, r) \text{ and } r > 200\,000\,000 \\ \varnothing & \text{otherwise} \end{cases}$$

where $\gg\!=$ is the monad bind (Kleisli composition) and return is the monad unit. The predicates ($>$, $=$) are well-known to be measurable on the domains where they are used here, and so the query must be measurable.

As an aside, we remark that much work in the database literature is on computing the results of queries efficiently. In the probabilistic setting, this is even more of a problem. But in this paper (as in [12]) we are focusing on the semantic aspects.

1.3 Generating probabilistic databases

Having established the measurability of the query language, in Section 5 we turn to investigate languages for generating probabilistic databases. For this we turn to the composite of the monads, $P \circ B$, which we have already shown to be a monad in [6] (see also [15,18]). As we demonstrate, the language for the monad $P \circ B$ appears to be ideal for generating probabilistic databases, at least as an intermediate language.

The paradigm for using infinite support probabilistic databases is still under debate, but typically one would begin from a deterministic database, and then add some randomness. Very simple kinds of randomness include

- adding noise to certain attributes, such as the movie gross, or blood pressure in a medical database;
- adding or deleting records at random, if there was uncertainty in the accuracy of those records.

We demonstrate how this can be done easily in the monad $P \circ B$. We also investigate a more elaborate model based on a GDatalog program, which translates very cleanly into the language of the $P \circ B$ monad.

1.4 Connection with other work on programming semantics

Our work discusses probabilistic databases in the context of monads and functional programming, and so we bring the general ideas of probabilistic databases to the language of functional probabilistic programming languages. We have already prototyped our examples simply by implementing a bag monad in Haskell and using a standard Haskell library for probabilistic inference. The idea of applying ideas from probabilistic programming to databases already has some momentum on the practical side, through languages such as BayesDB [26] and PClean [19]. Slightly further afield are probabilistic logic languages such as Blog [31] and ProbLog [7].

Probabilistic programming is a general approach for statistics. Within statistics, inhabitants of $P(B(X))$ are well-known and important, and called 'point processes'.

Further over to the semantic side, we note that the relevance of bags for probability has recently been emphasised by Jacobs [15,16]. Bags are a form of non-determinism, and the problem for combining non-determinism and probability is notoriously subtle, although there has been plenty of recent progress [9,18,21,22,28,29]. The particular combination we use here is trouble-free.

1.5 Summary

In this paper we show the following.

- The Bag monad extends to a strong monad on standard Borel spaces (Thm. 3.9).
- The Bag monad gives a free commutative monoid, and has a 'fold' construction (Thm. 3.8, Thm. 3.6).
- The BALG language for database queries always yields functions that are measurable (Thm. 4.1).
- The composite monad $P \circ B$ combines probability and bags and is useful for generating probabilistic databases.

Acknowledgements.

We are grateful to Peter Lindner for discussions. It has also been helpful to discuss this work with Martin Grohe, Bart Jacobs, Sean Moss, and Philip Saville. We acknowledge funding from Royal Society University Research Fellowship, the ERC BLAST grant, and the Air Force Office of Scientific Research under award number FA9550-21-1-0038. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the United States Air Force.

2 Mathematical preliminaries

2.1 Measure theory

Definition 2.1 The *Borel sets* form the least collection $\Sigma_{\mathbb{R}}$ of subsets of $\mathbb{R}$ containing intervals $(a, b) \subseteq \mathbb{R}$ which is closed under complementation and countable unions.

Definition 2.2 A *σ-algebra* on a set X is a nonempty family Σ_X of subsets of X that is closed under complements and countable unions. The pair (X, Σ_X) is called a *measurable space* (we just write X when Σ_X can be inferred from context).

Given (X, Σ_X), a *measure* is a function $\nu : \Sigma_X \to \mathbb{R}^{\infty}_{+}$ such that for all countable collections of disjoint sets $A_i \in \Sigma_X$, $\nu\left(\bigcup_i A_i\right) = \sum_i \nu(A_i)$. In particular, $\nu(\varnothing) = 0$. It is a *probability measure* if $\nu(X) = 1$.

Definition 2.3 Let (X, Σ_X) and (Y, Σ_Y) be measurable spaces. A measurable function $f : X \to Y$ is a function such that $f^{-1}(U) \in \Sigma_X$ when $U \in \Sigma_Y$.

Definition 2.4 A measurable space (X, Σ_X) is a *standard Borel space* if it is either measurably isomorphic to $(\mathbb{R}, \Sigma_{\mathbb{R}})$ or it is countable and discrete.

(This is equivalent to the usual definition of standard Borel spaces, which involves Polish spaces.)

Standard Borel spaces include the measurable spaces of real numbers and the integers, as well as all finite discrete spaces such as the booleans. So all the measurable spaces that arise in probabilistic databases are standard Borel, and indeed the restriction to standard Borel spaces is also made in Grohe and Lindner (see [12, Section 3.1]). Standard Borel spaces are closed under countable products and countable coproducts. Moreover, the equality predicate $X \times X \to \mathsf{Bool}$ is measurable when X is a standard Borel space.

2.2 Monads

Definition 2.5 A monad on a category is given by an object TX for each object X, a morphism $X \to TX$ for each object X, and for objects X and Y and morphism $f : X \to TY$ a morphism $f =\!\ll : TX \to TY$ is given, satisfying identity and associativity laws (see e.g. [23]).

A *strong* monad on a category with products is equipped with a morphism $X \times TY \to T(X \times Y)$ that respects the structure (see e.g. [23]).

The construction $=\!\ll$ is sometimes called bind or Kleisli composition.

2.2.1 Monad comprehension notation

For any strong monad we can use a comprehension notation. Given $f_1 : A \to TX_1$, $f_2 : A \times X_1 \to TX_2$, $f_3 : A \times X_1 \times X_2 \to TX_3$, $f_n : A \times X_1 \times \cdots \times X_{n-1} \to TX_n$, and given $g : A \times X_1 \times X_2 \times \cdots \times X_n \to Y$, we write

$$a \;\mapsto\; \{\!|\, g(a, x_1, \ldots x_n) \mid x_1 \leftarrow f_1(a), x_2 \leftarrow f_2(a, x_1), \ldots, x_n \leftarrow f_n(a, x_1, \ldots, x_{n-1}) \,|\!\}$$

for the composite morphism

$$A \xrightarrow{\bar{f}_1} T(A \times X_1) \xrightarrow{\bar{f}_2 =\!\ll} T(A \times X_1 \times X_2) \to \ldots \xrightarrow{\bar{f}_n =\!\ll} T(A \times X_1 \times X_2 \times \cdots \times X_n) \xrightarrow{T(g)} T(Y)$$

where

$$\bar{f}_i = A \times X_1 \times \ldots X_{i-1} \xrightarrow{(\mathrm{id}, f_i)} A \times X_1 \times \ldots X_{i-1} \times TX_i \xrightarrow{\mathrm{str}} T(A \times X_1 \times \cdots \times X_i).$$

When T is the powerset monad on the category of sets, this is exactly set comprehension. But it makes sense for any monad, and is often used with the list monad [30].

2.3 The Giry monad

The Giry monad [8] is a first key monad on measurable spaces. It also restricts to standard Borel spaces. If X is a measurable space, then $P(X)$ is the set of probability measures on X equipped with the σ-algebra generated by $A^U_r = \{p \in P(X) \mid p(U) \leq r\}$. The unit is given by the Dirac measures ($\eta(x)(U) = 1$ if $x \in U$, otherwise 0). The bind is given by Lebesgue integration: if $f : X \to P(Y)$ then $(f =\!\ll p)(U) = \int f(x)(U)\, p(\mathrm{d}x)$. The strength $s : X \times PY \to P(X \times Y)$ is given by $s(x, p)(U) = p(\{y \mid (x, y) \in U\})$.

3 The bag monoid and monad on measurable spaces

Let X be a set. A bag, aka multiset, is a finite unordered list of elements of X, or more formally an equivalence class of lists under permutation. Equivalently, a bag is a function $b : X \to \mathbb{N}$ such that $\{x \mid b(x) \neq 0\}$ is finite, or more formally it is an integer valued finite measure.

In this section we will focus on bags in the category of standard Borel spaces. We will show that the bags form a free commutative monoid, and support a 'fold' operation. We will also show that the bag construction forms a strong monad.

We begin by defining the measurable space of bags on some measurable space.

Definition 3.1 Let X be a measurable space. Let BX be the set of bags on the set underlying X. Equip BX with the least σ-algebra Σ_{BX} containing the generating sets $A_k^U = \{b \in BX \mid b \text{ contains exactly } k \text{ elements in } U\}$.

$$\Sigma_{BX} = \sigma(\{A_k^U \mid U \in \Sigma_X, k \in \mathbb{N}\})$$

Then (BX, Σ_{BX}) is the measurable space of bags of X.

Some observations about the space of bags BX are helpful in what follows. First we note that for any measurable space X we can decompose the set BX of bags of X into a disjoint union of the set of bags B_nX of size n for all $n \in \mathbb{N}$. We can equip each B_nX with the sub-σ-algebra. Then,

$$BX = \biguplus_{n \in \mathbb{N}} B_nX.$$

Second we record the following lemma.

Lemma 3.2 *Let X be a non-empty standard Borel space. The quotient function $X^n \to B_nX$, which takes a tuple $(x_1, \ldots, x_n)$ to the bag $\{\!| x_1 \ldots x_n |\!\}$, is measurable, and has a measurable section $B_nX \to X^n$.*

Proof. The idea is that we can regard B_nX as a space of sorted lists in X^n, as is common in practice in databases. Any standard Borel space is either isomorphic to the reals or countable and discrete. All of these spaces have a measurable total order $(<) \subseteq X \times X$. There may or may not be a canonical choice for a particular X, but it doesn't matter for the sake of this proof.

We can then use this within the language of measurable functions to write a measurable sorting function $i : X^n \to X^n$ that takes a list and returns the sorted version of it. (For example, if $n = 2$, let $i(x, y) = (x, y)$ if $x < y$ and otherwise (y, x).)

As a set-theoretic function, this sorting function $i : X^n \to X^n$ factors through the quotient map,

$$i \;=\; X^n \xrightarrow{q} B_nX \xrightarrow{s} X^n.$$

It remains to show that these two functions are measurable. That q is measurable is well-known, and in fact the σ-algebra on B_nX can be characterized as $\Sigma_{B_nX} = \{U \mid q^{-1}(U) \in \Sigma_{X^n}\}$ [20,24]. Finally, to see that s is measurable, suppose $U \in \Sigma_{X^n}$, then we must show that $s^{-1}(U) \in \Sigma_{B_nX}$, i.e. that $q^{-1}(s^{-1}(U)) \in \Sigma_{X^n}$. Since $sq = i$, and i is measurable, we are done. □

Proposition 3.3 *The space (BX, Σ_{BX}) is standard Borel when X is standard Borel.*

Proof. If X is standard Borel, then so is X^n, since any countable product of standard Borel spaces is standard Borel. So each $B_n(X)$ is standard Borel, since any retract of a standard Borel space is standard Borel. So $B(X)$ is a countable union of standard Borel spaces, hence also standard Borel. □

Note. Since all the spaces involved in probabilistic databases are standard Borel, in the remainder of this paper we only consider standard Borel spaces.

3.1 Measurable structural recursion on bags

All the computations we were interested in relied on a form of structural recursion over bags, which we now introduce. This is reminiscent of the fold construction from functional programming. For example, given a list of integers, it is possible to compute the sum of its elements by extracting elements starting at the head and calculating a running sum until we reach the tail. In this way the function sum can be defined as $\mathsf{fold}(\mathsf{plus}, 0)$ where $\mathsf{plus} : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ plays the role of the accumulating function and 0 is the initial argument provided to plus along with the head of the list. If the list being considered is empty, the result of the fold is simply the initial argument provided, which in this case is 0. The same approach works for bags too, provided that our

accumulating function will need to be such that the order in which it receives its arguments does not matter. This leads us to the definition of commutative functions. In the rest of this Section we define what it means to *measurably* fold a bag.

Definition 3.4 A function $f : X \times Y \to Y$ is *commutative* if

$$\forall x_1, x_2 \in X, y \in Y.\ f(x_1, f(x_2, y)) = f(x_2, f(x_1, y)).$$

Definition 3.5 Let $f : X \times Y \to Y$ be a measurable commutative function. Then define $\mathsf{fold}_f : Y \times BX \to Y$ to be the function which applies the accumulating function f with initial value $y \in Y$ to each element of $b \in BX$ one-by-one. Note that the order of selection of elements does not matter as f is commutative. We first define fold^n_f for bags of size n. When $n = 0$, $\mathsf{fold}^0_f(y, \varnothing) = y$. For non-zero n,

$$\begin{aligned}&\mathsf{fold}^n_f : Y \times B_n X \to Y\\ &\mathsf{fold}^n_f(y, \{\!|\, x_1, \ldots, x_n \,|\!\}) = f(x_1, f(x_2, \ldots f(x_n, y) \ldots)).\end{aligned}$$

From this, we obtain fold as the unique function coming out of the coproduct of each of the fold^n_f's above, giving us

$$\mathsf{fold}_f : Y \times BX \to Y.$$

Theorem 3.6 *fold_f is measurable for commutative measurable $f : X \times Y \to Y$.*

Proof. We use Lemma 3.2. First we define fold on lists:

$$\begin{aligned}&\mathsf{ofold}^n_f : Y \times X^n \to Y\\ &\mathsf{ofold}^n_f(y, (x_1, \ldots, x_n)) = f(x_1, f(x_2, \ldots f(x_n, y) \ldots)).\end{aligned}$$

This is clearly measurable, because it is just built from composition of measurable functions and product operations. Next we note that for any section $B_n X \to X^n$ of the quotient map,

$$\mathsf{fold}^n_f \;=\; Y \times B_n X \xrightarrow{Y \times s} Y \times X^n \xrightarrow{\mathsf{ofold}^n_f} Y$$

The commutativity of f means that the choice of section s does not matter. This again is a composition of measurable functions and so fold^n_f is measurable. The full function $\mathsf{fold}_f : Y \times BX \to Y$ is a copairing of measurable functions, and so it is measurable too. □

3.2 The space of bags as the free commutative monoid

In order to define a monoidal structure on the space of bags we first consider the function $\mathsf{add} : X \times BX \to BX$ which adds a single element to a bag, incrementing its multiplicity by one. It is clear that add is commutative.

Proposition 3.7 *$\mathsf{add} : X \times BX \to BX$ is measurable.*

Proof. Consider a measurable set $A^U_k \in \Sigma_{BX}$. This is the set of bags with exactly k elements belonging to U. Then $\mathsf{add}^{-1}(A^U_k)$ is the set of pairs (x, b) such that $\mathsf{add}(x, b) \in A^U_k$. In other words, each bag in A^U_k is decomposed into a set of pairs consisting of an element from the bag and the remaining bag. We consider the cases when $k = 0$ and when $k > 0$. In both these cases the inverse image map is in $\Sigma_{X \times BX}$.

- $k = 0$: Here we consider the set of bags such that no element belongs to U. Then it is guaranteed that any element removed from the bag will be in $\overline{U}$ and the remaining bag still in A^U_0, resulting in the inverse map being $\overline{U} \times A^U_0$, which is in $\Sigma_{X \times BX}$.
- $k > 0$: Here each bag in the set has a non-zero number of elements in U and as a result we have two further cases depending on whether or not the element extracted from the bag is in U. If it is not in U, the pair consisting of the element and the remaining bag belongs to $\overline{U} \times A^U_k$. If it belongs to U, the pair is an element of $U \times A^U_{k-1}$. And so the inverse image of A^U_k under add is the union of these two sets. Each of these sets is an element of $\Sigma_{X \times BX}$; consequently so too is their union.

□

With add as an accumulating function we can define the disjoint union of two bags as a measurable function by considering the fold of add where one bag provides all the new elements to be added to the other bag, which acts as the base case.

$$\uplus : BX \times BX \to BX$$
$$\uplus\,(b_1, b_2) = \mathsf{fold}_{\mathsf{add}}(b_1, b_2)$$

Theorem 3.8 *For any standard Borel space X, $(BX, \uplus, \varnothing)$ is a free commutative monoid.*

Proof. First note that $(BX, \uplus, \varnothing)$ is a commutative monoid. Given any commutative monoid $(Y, +_Y, e_Y)$ and a map $f : X \to Y$ we can define $g = \mathsf{fold}_{\mathsf{monAcc}} : Y \times BX \to Y$ where monAcc is the composite

$$\mathsf{monAcc} = X \times Y \xrightarrow{f \times Y} Y \times Y \xrightarrow{+_Y} Y.$$

From this we obtain the unique commutative monoid homomorphism $f^*(b) = g(e_Y, b) : BX \to Y$. □

As an aside, we remark that a fold-like operation is sometimes regarded as immediate from the free (commutative) monoid property. For example, in a cartesian closed category with list objects X^*, the space $Y \to Y$ is a monoid (under composition), and hence any map $X \to (Y \to Y)$ induces a canonical monoid homomorphism $X^* \to (Y \to Y)$, which is a curried form of fold. However, the category of measurable spaces is *not* cartesian closed [1,14], and so we have recorded the existence of fold as a separate fact to the free commutative monoid property.

3.3 The bag monad

We now use this universal property to describe the structure of the bag monad on standard Borel spaces.

- The unit $\eta : X \to B(X)$ is given by the singleton bag: $\eta(x) = \{\!|\, x \,|\!\}$. This is measurable because $\eta^{-1}(A_k^U)$ is $\overline{U}$ if $k = 0$, U if $k = 1$, and $\varnothing$ otherwise.
- The bind is given as follows. Informally, for $f : X \to B(Y)$, let $f =\!\ll : B(X) \to B(Y)$,

$$f =\!\ll \{\!|\, x_1 \dots x_n \,|\!\} = \biguplus_{i=1}^{n} f(x_i)$$

  Formally, we apply fold to the composite measurable function

$$X \times B(Y) \xrightarrow{f \times B(Y)} B(Y) \times B(Y) \xrightarrow{\uplus} B(Y)$$

  to get a measurable function $B(X) \times B(Y) \to B(Y)$, and then pass in the empty set as the initial argument. Equivalently, the monad multiplication can be given by applying fold to the function

$$\uplus : B(X) \times B(X) \to B(X)$$

  to get a measurable function $\mu : B(B(X)) \to B(X)$, by passing in $\varnothing$ as the initial argument.
- The strength $X \times B(Y) \to B(X \times Y)$ is given by applying fold to the function

$$(\mathsf{add}, \pi_2) : B(X \times Y) \times X \times Y \to B(X \times Y) \times X$$

  to get a measurable function $B(X \times Y) \times X \times B(Y) \to B(X \times Y) \times X$, and passing in the empty set as the initial argument and projecting the first result.

As an aside we note that in the statistics literature, it is quite common to regard $B(X)$ as a space of integer valued measures on X. With this perspective, regarding B as a monad of measures, the strong monad structure on B is entirely analogous to the monad structure of the Giry monad P.

Theorem 3.9 *(B, η, μ) is a strong monad on the category of standard Borel spaces.*

4 Measurable query operations on bags

In the standard theory of database modelling, relations are assumed to be sets, disallowing the existence of duplicates. Most database software, however, relax this restriction, often to save the cost of duplicate elimination. BALG ("bag algebra"), an algebra for manipulating bags, was first introduced in [13]. In that paper BALG was presented as an extension of the nested relation algebra (RALG), with a focus on the study of its expressive power and relative complexity to RALG. The authors showed that BALG as a query language was more expressive than RALG.

In this Section we will consider the entire BALG query language and show that it extends to measurable functions on bags.

For now we briefly review the query language BALG; we discuss these queries and their semantics in more detail later in this Section. The SINGLETON operation returns a singleton bag consisting of the input. Restructuring rows of tables is possible using the MAP_f query, which applies the function f to every row in the table. The queries PRODUCT, DUNION, DIFFERENCE, UNION, and INTERSECT compute the product, disjoint union, difference, union, and intersection of the input tables respectively. The PROJECT query projects out user-specified columns. FLATTEN transforms a bag of bags to a bag consisting of the disjoint unions of all the internal bags. Duplicate elimination, or deduplication, is possible using the DEDUP query. Finally, we can compute the bag of sub-bags of any bag using POWERBAG, and the bag of subsets of any bag using POWERSET.

Given the expressiveness of the BALG query language it comes as no surprise that many operations can be defined in terms of each other. For example, the powerset of a bag is simply the deduplicated version of the powerbag. It is also known that the union and intersection of bags can be defined using the disjoint union and difference operators. To this end we will only consider the following minimal subset of BALG queries, in terms of which all other queries can be defined: {SINGLETON, FLATTEN, MAP, PRODUCT, PROJECT, SELECT, DUNION, DIFFERENCE, POWERBAG, DEDUP}.

Previous work: In their work, Grohe and Lindner [11] considered BALG^1 [1], a subset of BALG restricted to bags of nesting level 1. That is, the queries of BALG^1 are defined on bags of type BX where X cannot have another type BY in its definition. The minimal set of queries for BALG^1 is the same set we consider here minus FLATTEN and POWERBAG since they operate on bags of bags. In their work Grohe and Lindner showed that BALG^1 queries extend to measurable functions on bags. We generalise their results and show, using our monadic and monoidal structure on bags, that all of BALG extends to measurable functions on bags, and give a clearer picture of how it comes together. Furthermore, we discuss the actions of grouping and aggregation as measurable queries in BALG.

4.1 Measurability of BALG queries

We provide a semantics to BALG queries by mapping each query to a measurable function on bags. The measurability of the semantics of the SINGLETON, FLATTEN, MAP, PRODUCT, PROJECT, and SELECT queries is guaranteed by defining their semantics as monad comprehensions. The measurability of the semantics of the remaining queries, DUNION, DIFFERENCE, POWERBAG, and DEDUP, is obtained by defining their semantics using our fold construction introduced in Section 3. Note that commutativity holds for all the measurable accumulating functions in the fold-based definitions to follow. The condition of commutativity is easy to check.

Bagging and flattening

The semantics for the SINGLETON and FLATTEN queries are given by the unit η^B and multiplication μ^B maps for the bag monad B. The measurability of these maps is proved in Theorem 3.9.

$$[\![\text{SINGLETON}]\!] : X \to BX \qquad [\![\text{FLATTEN}]\!] : B^2X \to BX$$
$$[\![\text{SINGLETON}]\!](x) = \eta^B_X(x) = \{\!|\ x\ |\!\} \qquad [\![\text{FLATTEN}]\!](b) = \mu^B_X(b)$$

Restructuring

The MAP_f query comes parametrized with a measurable function $f : X \to Y$ and its semantics is simply the functorial action of B on f, which yields yet another measurable map. This can equivalently be written using monad comprehension notation.

$$[\![\text{MAP}_f]\!] : BX \to BY \qquad [\![\text{MAP}_f]\!](b) = B(f)(b)$$

[1] It is called BALG^1, with superscript 1.

This can be neatly written using monad comprehension notation: $[\![\textsc{map}_f]\!](b) = \{\!|\ f(x) \mid x \leftarrow b\ |\!\}$.

Product and projection

Monad comprehensions make it straightforward to define the PRODUCT of two bags where the arity of the resultant schema is the sum of the arities of the input schemas.

$$[\![\textsc{product}]\!] : B(X_1 \times \cdots \times X_m) \times B(Y_1 \times \cdots \times Y_n) \to B(X_1 \times \cdots \times X_m \times Y_1 \times \cdots \times Y_n)$$
$$[\![\textsc{product}]\!](b_1, b_2) = \{\!|\ (x_1, \ldots, x_m, y_1, \ldots, y_n) \mid (x_1, \ldots, x_m) \leftarrow b_1,\ (y_1, \ldots, y_n) \leftarrow b_2\ |\!\}$$

A similar treatment can be given to the $\textsc{project}_{i_1,\ldots,i_k}$ query which projects out the $i_1, \ldots, i_k$ indices of the input schema.

$$[\![\textsc{project}_{i_1,\ldots,i_k}]\!] : B(X_1 \times \cdots \times X_n) \to B(X_{i_1} \times \cdots \times X_{i_k})$$
$$[\![\textsc{project}_{i_1,\ldots,i_k}]\!](b) = \{\!|\ (x_{i_1}, \ldots, x_{i_k}) \mid (x_1, \ldots, x_n) \leftarrow b\ |\!\}$$

Selection

$\textsc{select}_\psi$ is parametrized by a measurable Boolean predicate $\psi : X \to \mathsf{Bool}$ and filters out the rows in the input table satisfying ψ. We can lift ψ to the measurable function $\hat{\psi} : X \to B1$, where $1 = \{\star\}$ is the singleton space with a unique element. $\hat{\psi}$ evaluates to $\{\!|\ \star\ |\!\}$ (resp. $\varnothing$) when ψ evaluates to True (resp. False). This construction enables us to define the semantics of $\textsc{select}_\psi$ as a monad comprehension where rows not satisfying ψ do not get included due to $\hat{\psi}$ evaluating to the empty bag. (Define filter_ψ to be this map.)

$$[\![\textsc{select}_\psi]\!] : BX \to BX \qquad [\![\textsc{select}_\psi]\!](b) = \mathsf{filter}_\psi(b) = \{\!|\ x \mid x \leftarrow b,\ \star \leftarrow \hat{\psi}(x)\ |\!\}$$

In monad comprehension syntax, for a monad with a given zero element (e.g. $\varnothing \in BX$), a shorthand notation $\{\!|\ x \mid x \leftarrow b,\ \psi(x)\ |\!\}$ is often used.

Disjoint union

DUNION simply computes the disjoint union of its arguments. In Section 3.2 we defined the measurable disjoint union $\uplus : BX \times BX \to BX$ as $\mathsf{fold}_{\mathsf{add}}$.

$$[\![\textsc{dunion}]\!] : BX \times BX \to BX \qquad [\![\textsc{dunion}]\!](b_1, b_2) = b_1 \uplus b_2 = \mathsf{fold}_{\mathsf{add}}(b_1, b_2)$$

Bag difference

Bag difference is the first operation we consider that is not quite immediate from the monad and monoid structure. The idea is that, for instance, $[\![\textsc{difference}]\!](\{\!|\ 1, 1, 2\ |\!\}, \{\!|\ 1, 2, 3\ |\!\}) = \{\!|\ 1\ |\!\}$. Defining bag difference as a fold requires a little care. To this end, we first define the measurable function remove which takes an element and a bag as input and returns either the same bag if that element did not belong to the bag, or a modified bag with one fewer instance of the given element. remove is defined as a fold over the accumulating function remAcc, which maintains a triple as an accumulator:

- the value $x_{\mathsf{rem}} \in X$ to be removed,
- a Boolean value indicating whether or not x_{rem} has already been removed (in order to prevent us from removing x_{rem} more than once),
- and the resulting bag (which is initially empty) to which elements are added.

Each of the three cases has been defined by combining tupling and add, both of which are measurable functions. From this it follows that remAcc is measurable.

$$\mathsf{remAcc} : X \times ((X \times \mathsf{Bool}) \times BX) \to (X \times \mathsf{Bool}) \times BX$$
$$\mathsf{remAcc}(x, ((x_{\mathsf{rem}}, c), b)) = \begin{cases} ((x_{\mathsf{rem}}, \mathsf{True}), \mathsf{add}(x, b)) & \text{if } c = \mathsf{True} \\ ((x_{\mathsf{rem}}, \mathsf{True}), b) & \text{if } x = x_{\mathsf{rem}} \\ ((x_{\mathsf{rem}}, \mathsf{False}), \mathsf{add}(x, b)) & \text{otherwise} \end{cases}$$

To get the final bag after removal we project out the second element of the pair returned by remAcc.

$$\mathsf{remove} : X \times BX \to BX \qquad \mathsf{remove}(x_{\mathsf{rem}}, b) = \pi_2\left(\mathsf{fold}_{\mathsf{remAcc}}(((x_{\mathsf{rem}}, \mathsf{False}), \varnothing), b)\right)$$

Using remove we define the bag difference of b_1 and b_2 by letting b_1 be the initial input and from it removeing each element in b_2 one-by-one. The measurability of bag difference follows from the commutativity and measurability of remove.

$$[\![\textsc{difference}]\!] : BX \times BX \to BX \qquad [\![\textsc{difference}]\!](b_1, b_2) = \mathsf{fold}_{\mathsf{remove}}(b_1, b_2)$$

Powerbag

For example, the powerbag of the bag $\{\!\{ 1, 1 \}\!\}$ is $\{\!\{ \varnothing, \{\!\{ 1 \}\!\}, \{\!\{ 1 \}\!\}, \{\!\{ 1, 1 \}\!\} \}\!\}$. The powerbag of a bag is defined by folding over the accumulating function powerAcc where for every new element x added to the accumulating bag of bags b_0 we first add x to every bag in b_0 and then take the disjoint union with the initial b_0. We define add_x to be the x-section of add (that is, $\mathsf{add}_x(b) = \mathsf{add}(x, b)$). Sections of measurable functions are measurable.

$$\mathsf{powerAcc} : X \times B^2X \to B^2X \qquad [\![\textsc{powerbag}]\!] : BX \to B^2X$$
$$\mathsf{powerAcc}(x, b_0) = b_0 \uplus B(\mathsf{add}_x)(b_0) \qquad [\![\textsc{powerbag}]\!](b) = \mathsf{fold}_{\mathsf{powerAcc}}(\{\!\{ \varnothing \}\!\}, b)$$

Deduplication

In order to deduplicate a bag we recurse over its elements and add them to an accumulating bag one-by-one. We avoid multiplicities greater than one in the final bag by first filtering out the value we want to add from the accumulating bag before adding it, which is reflected in the definition of dedupAcc.

$$\mathsf{dedupAcc} : X \times BX \to BX \qquad [\![\textsc{dedup}]\!] : BX \to BX$$
$$\mathsf{dedupAcc}(x, b) = \mathsf{add}(x, \mathsf{filter}_{\neq x}(b)) \qquad [\![\textsc{dedup}]\!](b) = \mathsf{fold}_{\mathsf{dedupAcc}}(\varnothing, b)$$

The function $\neq x : X \to \mathsf{Bool}$ is measurable since $\neq x^{-1}(\{\mathsf{True}\}) = X \setminus \{x\}$ and $\neq x^{-1}(\{\mathsf{False}\}) = \{x\}$, both of which are measurable sets (due to X being standard Borel).

Theorem 4.1 *BALG queries yield measurable functions on bags.*

4.2 Grouping and aggregation

Consider, for example, the table $\mathsf{cast} : \mathsf{Actor} \times \mathsf{Movie}$ from the database MovieFact introduced at the start of this paper. A natural query that one may want to compute is, *"How many movies has each actor appeared in?"* In order to calculate the answer to this we first need to be able to group actors with the bag of all the movies they appeared in. To this resultant table we can map a size function to the second column to get the numbers we need. Here we introduce a group query to BALG and show that it is a measurable operation on bags.

Definition 4.2 The $\textsc{group}_{p_1,p_2}$ query acts on tables of schema $X_1 \times \cdots \times X_k$ and is parametrized by two projection functions $p_1 : X_1 \times \cdots \times X_k \to X_{i_1} \times \cdots \times X_{i_m}$ and $p_2 : X_1 \times \cdots \times X_k \to X_{j_1} \times \cdots \times X_{j_n}$. The result of this query is a table with schema $(X_{i_1} \times \ldots X_{i_m}) \times B(X_{j_1} \times \cdots \times X_{j_n})$ where the elements of the first column are paired with the bag of elements they were related to in the input table. In other words, we group the rows of the table by the elements in the p_1-projection of the table.

The measurable bag-semantics for $\textsc{group}_{p_1,p_2}$ is given by

$$\begin{aligned}
&[\![\textsc{group}_{p_1,p_2}]\!] : B(X_1 \times \cdots \times X_k) \to B((X_{i_1} \times \ldots X_{i_m}) \times B(X_{j_1} \times \cdots \times X_{j_n})) \\
&[\![\textsc{group}_{p_1,p_2}]\!](b) = \{\!\{ (i, B(p_2)(\mathsf{filter}_{\lambda x.\, p_1(x) \equiv i}(b))) \mid i \leftarrow [\![\textsc{dedup}]\!](B(p_1)(b)) \}\!\} \\
&\qquad\qquad = \{\!\{ (i, \{\!\{ p_2(x) \mid x \leftarrow b, p_1(x) = i \}\!\}) \mid i \leftarrow [\![\textsc{dedup}]\!](\{\!\{ p_1(x) \mid x \leftarrow b \}\!\}) \}\!\}
\end{aligned}$$

In the monad comprehension we first project out the columns of interest using p_1 and deduplicate the resultant bag. From this bag we extract out the rows indices by which we index the rows of the input bag.

For each index i we return the pair consisting of i along with the p_2-projection of the input where the p_1-projection of the rows is equal to i. We can conclude that this query is measurable by defining it as a monad comprehension composed of other measurable functions.

Recall the actor grouping example suggested earlier. Given an input bag from $B(\mathsf{Actor} \times \mathsf{Movie})$ we can apply $\textsc{group}_{\pi_1,\pi_2}$ to create a table of rows relating actors to the bag of movies they appeared in. To this table we can apply $\textsc{map}_{\lambda(x,y).(x,\mathsf{size}(y))}$ to arrive at the final result. The function $\mathsf{size} : B\,\mathsf{Movie} \to \mathbb{N}$ can be measurably defined as a fold, for example.

A second option for defining an group/aggregation query comes from an extension to monad comprehensions in Haskell where the syntax has been extended with the keywords `group by` [25]. This extension works for any strong monad, but the user needs to provide a grouping function which, in our case, needs to map a bag on X to a bag of bags on X where each sub-bag contains the same element. This can be written in BALG as

$$\textsc{group'}(b) = \textsc{map}_{\lambda i.\textsc{select}_{\lambda x.\, x \equiv i}(b)}(\textsc{dedup}(b))$$
$$[\![\textsc{group'}(b)]\!] = \{\!|\, \{\!|\, x \mid x \leftarrow b, x = i \,|\!\} \mid i \leftarrow \textsc{dedup}(b) \,|\!\}$$

So $[\![\textsc{group'}]\!] : BX \to B^2X$. The entire actor query can now be concisely written in the notation of [25] as

$$\{\!|\, (\mathsf{the}\ a, \mathsf{size}\ m) \mid (a, m) \leftarrow \mathsf{actorMovieTable},\ \texttt{group by}\ a \,|\!\}.$$

This modified comprehension syntax of [25] works by implicitly changing the types of a and m from Actor and Movie in the right half of the comprehension to $a \in B\,\mathsf{Actor}$ and $m \in B\,\mathsf{Movie}$ on the left half. This allows us to apply the measurable aggregation function $\mathsf{size} : BX \to \mathbb{N}$ to m. The aggregation function used on a is $\mathsf{the} : B\,\mathsf{Actor} \to \mathsf{Actor}$, which is some measurable function such that $\mathsf{the}(b) = x$ when b is a bag that only contains copies of x. (For example, we could sort b and return the first element.)

5 Generating probabilistic databases

The main focus of this paper has been measurable query languages (Theorem 4.1). We now turn to the question of where probabilistic databases come from in the first place, particularly in the setting where they have infinite support. A good language for generating infinite probabilistic databases remains a topic of active research, but we now illustrate that the monads for probability P and bags B could be the basis of a good intermediate language.

5.1 A distributive law

We recall the distributive law distr between the monads P and B that we provided in an earlier paper [6] (see also [15,18,32]):

$$\mathsf{distr} : B(P(X)) \to P(B(X))$$

Using our fold technology (Thm. 3.6) we can define the distributive law as $\mathsf{distr} = \mathsf{fold}_{\mathsf{distrAcc}}$, where

$$\mathsf{distrAcc} : P(X) \times P(B(X)) \to P(B(X))$$
$$\mathsf{distrAcc} \;=\; P(X) \times P(B(X)) \xrightarrow{\text{double strength}} P(X \times B(X)) \xrightarrow{P(\mathrm{add})} P(B(X))$$

As usual, this distributive law determines a monad structure on $P \circ B$ [2].

5.2 Randomizing attributes

For a first example of a probabilistic database, suppose we are given a deterministic database of (movie,gross) pairs. We may then decide that the gross figure is inaccurate and should be subject to a noise from a normal distribution, yielding a probabilistic database. This can be done categorically by the following map:

$$B(\mathsf{Movie} \times \mathbb{R}) \xrightarrow{B(\mathsf{Movie} \times \mathit{normal})} B(\mathsf{Movie} \times P(\mathbb{R})) \xrightarrow{\text{strength}} B(P(\mathsf{Movie} \times \mathbb{R})) \xrightarrow{\text{distr}} P(B(\mathsf{Movie} \times \mathbb{R}))$$

Since PB is a monad, we can use comprehension notation for it, and equivalently write the above generation method $\mathit{addNoise} : B(\mathsf{Movie} \times \mathbb{R}) \to P(B(\mathsf{Movie} \times \mathbb{R}))$ as

$$\mathit{addNoise}(b) = \{\!|\, (m, r') \mid (m, r) \leftarrow b, r' \leftarrow \mathit{normal}(r, 100000) \,|\!\}$$

Here we are implictly casting $b \in B(X)$ to $b \in P(B(X))$ and $normal(r, 100000) \in P(\mathbb{R})$ to $P(B(\mathbb{R}))$, implicitly using the units $\eta_P B : B(X) \to P(B(X))$ and $P\eta_B : P(X) \to P(B(X))$.

Another use-case for random attributes, studied in [12], is to deal with null attributes by drawing them randomly from a vague prior distribution. This would also be easy to express using the PB monad.

5.3 Adding random records

We can also add and remove random records straightforwardly. We note a few helpful facts.

- The disjoint union operation $\uplus : B(X) \times B(X) \to B(X)$ lifts to $\uplus : P(B(X)) \times P(B(X)) \to P(B(X))$ by composing with the strength of P. In this way the composite monad PB has a commutative monoid structure.
- Since $B(1) \cong \mathbb{N}$, we can regard the Poisson distribution as a map in $\mathbb{R} \to P(B(1))$, parameterized by rate.

Now supposing we also have reasonably uniform distributions $r\text{-}actor : P(\mathsf{Actor})$ and $r\text{-}movie : P(\mathsf{Movie})$, we can delete some credits and generate random additional actors for movies, modelling the fact that some actors are unlisted:

$$\begin{aligned} addRemove(b) = \quad & \{\!| \ (a,m) \mid (a,m) \leftarrow b, 1 \leftarrow bernoulli(0.9) \ |\!\} \\ \uplus \ & \{\!| \ (a,m) \mid n \leftarrow \text{poisson}(3), a \leftarrow r\text{-}actor, m \leftarrow r\text{-}movie \ |\!\} \end{aligned}$$

The first line deletes random rows with probability 0.1, and the second line adds in some extra actors (on average, 3 extra actors). Of course, a more sophisticated model could take into account other prior information such as relationships and ages between actors.

5.4 Towards GDatalog

The GDatalog language has recently been proposed as a generative language for probabilistic databases [4,10]. The language combines datalog-style features with continuous probability distributions.

In general, GDatalog is recursive. We have not treated recursion in this paper, so we focus on the non-recursive fragment. This can easily be translated into the PB monad. For example, consider the following GDatalog program taken from [10]. The idea is to simulate possibly faulty burglar alarms which either go off because of a burglary or because of an earthquake.

$$\begin{aligned} &\mathsf{earthquake}(c, bernoulli(0.1)) \leftarrow \mathsf{crimechance}(c, r) \\ &\mathsf{burglary}(x, bernoulli(r)) \leftarrow \mathsf{address}(x, c), \mathsf{crimechance}(c, r) \\ &\mathsf{trigger}(x, bernoulli(0.6)) \leftarrow \mathsf{address}(x, c), \mathsf{earthquake}(c, 1) \\ &\mathsf{trigger}(x, bernoulli(0.9)) \leftarrow \mathsf{burglary}(x, 1) \\ &\mathsf{alarm}(x) \leftarrow \mathsf{trigger}(x, 1) \end{aligned}$$

We will regard this as transforming a deterministic database into a probabilistic one, $B(X) \to P(B(X))$ where

$$\begin{aligned} X \ = \quad & (\mathsf{earthquake} : \mathsf{City} \times 2) \uplus (\mathsf{crimechance} : \mathsf{City} \times [0,1]) \uplus (\mathsf{address} : \mathsf{House} \times \mathsf{City}) \\ \uplus \ & (\mathsf{burglary} : \mathsf{House} \times 2) \uplus (\mathsf{trigger} : \mathsf{House} \times 2) \uplus (\mathsf{alarm} : \mathsf{House}) \end{aligned}$$

The GDatalog program can be translated almost verbatim as a sequence of definitions of a probabilistic database in $PB(X)$ using monad comprehensions, starting from $b \in B(X)$, as follows.

$$\begin{aligned} b_1 &= b \ \uplus \{\!| \ \mathsf{earthquake}(c,z) \mid \mathsf{crimechance}(c,r) \leftarrow b, z \leftarrow bernoulli(0.1) \ |\!\} \\ b_2 &= b_1 \uplus \{\!| \ \mathsf{burglary}(x,z) \mid \mathsf{crimechance}(c,r) \leftarrow b_1, \mathsf{address}(x,c') \leftarrow b_1, c = c', z \leftarrow bernoulli(r) \ |\!\} \\ b_3 &= b_2 \uplus \{\!| \ \mathsf{trigger}(x,z) \mid \mathsf{address}(x,c) \leftarrow b_2, \mathsf{earthquake}(c',1) \leftarrow b_2, c = c', z \leftarrow bernoulli(0.6) \ |\!\} \\ b_4 &= b_3 \uplus \{\!| \ \mathsf{trigger}(x,z) \mid \mathsf{burglary}(x,1) \leftarrow b_3, z \leftarrow bernoulli(0.9) \ |\!\} \\ b_5 &= b_4 \uplus \{\!| \ \mathsf{alarm}(x) \mid \mathsf{trigger}(x,1) \leftarrow b_4 \ |\!\} \end{aligned}$$

One of the main results of [10] is that GDatalog programs yield proper probabilistic databases, that is, that all the constructions are measurable. For examples such as this, in the non-recursive fragment, this measurability is immediate from the fact that we are programming in the PB monad, where everything is measurable.

6 Summary and outlook

We have shown that the bag monad on standard Borel spaces is strong (Thm. 3.9) and supports a fold operation (Thm. 3.6) which is connected to its characterization as the free commutative monoid. We have used this to show straightforwardly that all the bag query operations of BALG yield measurable queries (Thm. 4.1), and so they are all safe to use in defining queries of probabilistic databases. This affirms the results in [11], generalizing them to the full BALG language and clarifying the measurability arguments by factoring them through the measurability of the bag monad. Finally, in Section 5 we have argued by illustrations that the combination of the bag and probability monads gives a powerful intermediate language for processes that generate probabilistic databases.

References

[1] R. J. Aumann. Borel structures for function spaces. *Illinois Journal of Mathematics*, 5:614–630, 1961.

[2] J. Beck. Distributive laws. In B. Eckmann, editor, *Seminar on Triples and Categorical Homology Theory*, pages 119–140, Berlin, Heidelberg, 1969. Springer Berlin Heidelberg.

[3] P. Buneman, L. Libkin, D. Suciu, V. Tannen, and L. Wong. Comprehension syntax. *SIGMOD Rec.*, 23(1):87–96, 1994.

[4] V. Bárány, B. ten Cate, B. Kimelfeld, D. Olteanu, and Z. Vagena. Declarative probabilistic programming with Datalog. *ACM Transactions on Database Systems (TODS)*, 42(4), 2017.

[5] I. I. Ceylan, A. Darwiche, and G. Van den Broeck. Open-world probabilistic databases. In *Proc. KR 2016*, 2016.

[6] S. Dash and S. Staton. A monad for probabilistic point processes. In *Proc. ACT 2020*, 2020.

[7] L. De Raedt and A. Kimmig. Probabilistic (logic) programming concepts. *Machine Learning*, 100(1):5–47, 2015.

[8] M. Giry. A categorical approach to probability theory. In *Categorical aspects of topology and analysis (Ottawa, Ont., 1980)*, volume 915 of *Lecture Notes in Mathematics*, pages 68–85. Springer, Berlin, 1982.

[9] A. Goy and D. Petrisan. Combining probabilistic and non-deterministic choice via weak distributive laws. In *Proc. LICS 2020*, 2020.

[10] M. Grohe, B. L. Kaminski, J.-P. Katoen, and P. Lindner. Generative datalog with continuous distributions. In *Proc. PODS 2020*, pages 347–360.

[11] M. Grohe and P. Lindner. Probabilistic databases with an infinite open-world assumption. In *Proc. PODS 2019*, pages 17–31, 2019.

[12] M. Grohe and P. Lindner. Infinite probabilistic databases. In *Proc. ICDT 2020*, 2020.

[13] S. Grumbach, L. Libkin, T. Milo, and L. Wong. Query languages for bags: expressive power and complexity. *SIGACT News (Database Theory Column)*, pages 30–37, June 1996.

[14] C. Heunen, O. Kammar, S. Staton, and H. Yang. A convenient category for higher-order probability theory. In *Proc. LICS 2017*. IEEE Press, 2017.

[15] B. Jacobs. From multisets over distributions to distributions over multisets. In *Proc. LICS 2021*, 2021.

[16] B. Jacobs. Multinomial and hypergeometric distributionsin markov categories. In *Proc. MFPS 2021*, 2021.

[17] R. Jampani, F. Xu, M. Wu, L. L. Perez, C. Jermaine, and P. J. Haas. MCDB: A Monte Carlo approach to managing uncertain data. In *Sigmod 2008*, 2008.

[18] K. Keimel and G. Plotkin. Mixed powerdomains for probability and nondeterminism. arXiv:1612.01005.

[19] A. K. Lew, M. Agrawal, D. Sontag, and V. K. Mansinghka. PClean: Bayesian data cleaning at scale with domain-specific probabilistic programming. In *Proc. AISTATS 2021*, 2021.

[20] O. Macchi. The coincidence approach to stochastic point processes. *Advances in AppliedProbability*, 7:83–122, 1975.

[21] M. Mio and V. Vignudelli. Monads and quantitative equational theories for nondeterminism and probabilities. In *Proc. CONCUR 2020*, 2020.

[22] M. W. Mislove, J. Ouaknine, and J. Worrell. Axioms for probability and nondeterminism. In *Proc. EXPRESS 2003*, 2003.

[23] E. Moggi. Notions of computation and monads. *Inf. Comput.*, 93(1):55–92, July 1991.

[24] J. E. Moyal. The general theory of stochastic population processes. *Acta Mathematica*, 108, 1962.

[25] S. Peyton Jones and P. Wadler. Comprehensive comprehensions. In G. Keller, editor, *Proceedings of the ACM SIGPLAN Workshop on Haskell, Haskell 2007, Freiburg, Germany, September 30, 2007*, pages 61–72. ACM, 2007.

[26] F. Saad and V. Mansinghka. A probabilistic programming approach to probabilistic data analysis. In *NeurIPS*, 2016.

[27] D. Suciu, D. Olteanu, C. Ré, and C. Koch. *Probabilistic Databases*. Morgan and Claypool, 2011.

[28] G. van Heerdt, J. Hsu, J. Ouaknine, and A. Silva. Convex language semantics for nondeterministic probabilistic automata. In *Proc. ICTAC 2018*, 2018.

[29] D. Varacca and G. Winskel. Distributing probability over non-determinism. *Mathematical structures in computer science*, 16:87–113, 2006.

[30] P. Wadler. Comprehending monads. *Mathematical Structures in Computer Science*, 2:461–493, 1992.

[31] Y. Wu, S. Srivastava, N. Hay, S. Du, and S. J. Russell. Discrete-continuous mixtures in probabilistic programming: Generalized semantics and inference algorithms. In *Proc. ICML 2018*, pages 5339–5348, 2018.

[32] M. Zwart and D. Marsden. No-go theorems for distributive laws. In *Proc. LICS 2019*, 2019.