Exchangeable Random Processes and Data Abstraction

Extended Abstract

Sam Staton, Univ. of Oxford
Hongseok Yang, Univ. of Oxford
Nathanael Ackerman, Harvard Univ.
Cameron Freer, Gamalon and Borelian
Daniel M. Roy, Univ. of Toronto

1. Introduction

Several Turing-complete probabilistic programming languages such as Church, Venture and Anglican support so-called exchangeable random processes (XRPs) as built-in abstract data types. Instances of these data types denote infinite sequences of random values. Every such object satisfies the so-called exchangeability property, which says that the probability distribution of the object does not change under permutations of the sequence. In fact, exchangeability is one of the core concepts in Bayesian statistics, from which these infinite sequences originate. (Extensions of the notion of exchangeability for sequences apply to other infinite random objects such as graphs, which we consider in Section 3.)

We have been working on semantic foundations and reasoning principles for data types for exchangeable random processes, with special emphasis on the client programs of these data types. One slogan that summarises our efforts well is this: a data type for a random process is exchangeable if its inclusion in a probabilistic programming language does not break the following commutativity and discardability of the language, i.e.,

$$(\textbf{let } x{=}M \textbf{ in } y{=}M' \textbf{ in } N) \;=\; (\textbf{let } y{=}M' \textbf{ in } x{=}M \textbf{ in } N)$$
$$(\textbf{let } x{=}M \textbf{ in } M') \;=\; M'$$

where $x \notin \mathrm{FV}(M')$ and $y \notin \mathrm{FV}(M)$. (See also informal discussions in [1, 5], [10, §2.2.1].) The specifics of a particular data type for an exchangeable random process are then captured by additional program equations, which in turn give rise to a monad for interpreting client programs of the data type. This equational view may suggest how the data-type developers should use an abstraction mechanism of a programming language (e.g., abstract type) so as to hide implementation details and to get the desired program equations. Also, it may reveal how exchangeability from statistics is related to algebraic effects studied in programming languages. Finally, it may lead to the generalisation of important results on exchangeable random processes in Bayesian statistics, such as de Finetti's theorem and the Aldous–Hoover theorem. Realising these potentials has been the aim of our research.

In this extended abstract, we describe our preliminary results. We present a set of program equations for certain exchangeable random processes, and explain data types that implement these processes and validate these equations. Also, we describe a variant of de Finetti's theorem for a probabilistic programming language extended with these data types. Our results are interleaved with open questions that have been puzzling us.

2. Exchangeable sequences

Consider the following abstract type in an ML-like language.

```
module type PROCESS = sig
  type process
  val new : int * int → process
  val get : process → bool
end
```

The idea is that process is an abstract type of exchangeable random processes that produce booleans. A client program creates a process by calling new, and uses it to generate a sequence of random booleans by calling get. The exchangeability means that permuting the order of a generated infinite sequence does not change the probability of the sequence. It does not come for free by the type signature, but should be ensured by an implementation of the signature. We will focus on the Beta–Bernoulli process. We will use an ML-like language with primitives for state:

t ref	a type of references (pointers) to things of type t.
ref(x)	allocate a new memory cell initialized with x.
!r	read the contents of reference cell r.
r:=x	write x to reference cell r.

and probability:

sample(d)	draw a sample from the distribution d.

We call this language Probabilistic ML. Here is an implementation based on Pólya's urn.

```
module Polya = (struct
  type urn = (int * int) ref
  type process = urn
  let new(a,b) = ref (a,b)
  let get p =
    let (a,b) = !p in
    if sample( bernoulli (a/(a+b))) then p := (a+1,b) ; true
    else p := (a,b+1) ; false
end : PROCESS)
```

An urn is a hidden state which contains a-many balls marked true and b-many balls marked false. To sample, we draw a ball from the urn at random; before we return what we drew, we put back the ball we drew as well as an identical copy of it.

Programs that use memory do not satisfy commutativity and discardability in general. For instance,

$$\textbf{let } x{=}(!r) \textbf{ in } y{=}(r{:=}3) \textbf{ in } x \;\neq\; \textbf{let } y{=}(r{:=}3) \textbf{ in } x{=}(!r) \textbf{ in } x.$$

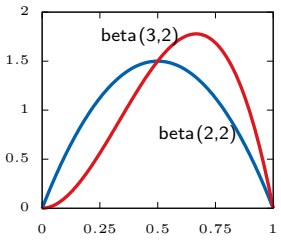
But our module Polya *does* satisfy commutativity and discardability, for instance,

$$\textbf{let } x{=}(\text{get } p) \textbf{ in } y{=}(\text{get } p) \textbf{ in } (x,y) \;=\; \textbf{let } y{=}(\text{get } p) \textbf{ in } x{=}(\text{get } p) \textbf{ in } (x,y).$$

As we will explain, this property is related to the following observation. It is legitimate to use this module within a probabilistic program because it is indistinguishable from the following module, which only uses probabilistic primitives:

```
module Beta_Bernoulli = (struct
  type process = real
  let new(a,b) = sample(beta(a,b))
  let get p = sample(bernoulli(p))
end : PROCESS)
```

Here, the Beta distribution beta(a,b) is the probability measure on the unit in-



terval $[0,1]$, which, as illustrated, measures the probable bias of a random source from which true has been observed $(\mathsf{a}-1)$ times and false has been observed $(\mathsf{b}-1)$ times.

2.1 General analysis

We now state a general formulation of the above phenomenon. Notice that although Probabilistic ML is not generally commutative, because it allows memory access, it *is* commutative if we restrict the use of memory to the calls in Polya. We make this general by considering, alongside Probabilistic ML, a simple first-order programming language which is the first-order fragment of Moggi's monadic metalanguage [11] extended with a type constant $\mathsf{process}$ and two special typing rules:

$$\frac{\Gamma \vdash \mathsf{M} : \mathsf{int} \quad \Gamma \vdash \mathsf{N} : \mathsf{int}}{\Gamma \vdash \mathsf{new(M,N)} : \mathsf{process}} \qquad \frac{\Gamma \vdash \mathsf{P} : \mathsf{process}}{\Gamma \vdash \mathsf{get(P)} : \mathsf{bool}}$$

The idea is that this simple first-order language has the chance to fully satisfy commutativity and discardability, because it has no explicit memory. Different notions of 'process' will endow the simple language with a different semantics and hence a different equality theory. In particular, from any Probabilistic ML module $\mathcal{I}$ that implements the signature PROCESS, we can derive a notion of contextual equivalence for the simple first-order language, as follows. A closed program of type bool determines a probability distribution on $\{\mathsf{true}, \mathsf{false}\}$, found by running it and building a histogram of the results. In general for $\Gamma \vdash \mathsf{M}, \mathsf{N} : \mathsf{t}$ we write $\mathsf{M} \approx_{\mathcal{I}} \mathsf{N}$ if for all contexts $\mathcal{C}$ such that $\vdash \mathcal{C}[\mathsf{M}], \mathcal{C}[\mathsf{N}] : \mathsf{bool}$, the closed programs $\mathcal{C}[\mathsf{M}]$ and $\mathcal{C}[\mathsf{N}]$ describe the same distribution.

The modules Polya and Beta_Bernoulli induce the same notion of contextual equivalence for the simple first-order language. This is an instance of the following more general result.

We say that a Probabilistic ML module $\mathcal{I}$ for the signature PROCESS is *exchangeable* if the induced contextual equivalence $\approx_{\mathcal{I}}$ on the simple first-order language is commutative and discardable. We can now state a de Finetti-type theorem (see also [3]).

Theorem 2.1. *For any exchangeable module there is an implementation using only probabilistic primitives and no local state that induces the same notion of contextual equivalence.*

2.2 Particular analysis of Beta–Bernoulli

Contextual equivalence is difficult to reason about because of the quantification over all contexts. For a fixed module $\mathcal{I}$, we can try to axiomatise contextual equivalence, following [15]. We now do this for the Beta–Bernoulli process. First, we define $\mathsf{random(m{:}n)}{:}\mathsf{bool}$ to be $\mathbf{let}\ \mathsf{p{=}new(m,n)}\ \mathbf{in}\ \mathsf{get(p)}$. Informally, $(\mathsf{m{:}n})$ are the odds that $\mathsf{random(m{:}n)}$ returns true. We propose the following program equation:

$$\begin{bmatrix}\mathbf{let}\ \mathsf{p{=}new(m,n)}\\ \mathbf{in}\ \mathsf{(get(p),p)}\end{bmatrix} = \begin{bmatrix}\mathbf{if}\ \mathsf{random(m{:}n)}\\ \mathbf{then}\ \mathsf{(true\,,new(m{+}1,n))}\\ \mathbf{else}\ \mathsf{(\ false\ ,new(m,n{+}1))}\end{bmatrix} \qquad (1)$$

in addition to commutativity and discardability and the laws of probability [20]:

$$\mathsf{random(m{:}n)} = \mathsf{not(random(n{:}m))}$$

$$\mathsf{random((m*k){:}(n*k))} = \mathsf{random(n{:}m)}$$

$$\mathsf{random(1{:}0)} = \mathsf{true}$$

$$\begin{bmatrix}\mathbf{let}\ \mathsf{a{=}random(f(true\,,true){+}f(true\,,\ false\ ){:}}\\ \qquad\qquad \mathsf{f(\ false\ ,true){+}f(\ false\ ,\ false\ ))}\ \mathbf{in}\\ \mathbf{let}\ \mathsf{b{=}random(f(a,true){:}f(a\,,\ false\ ))}\ \mathbf{in}\ \mathsf{(a,b)}\end{bmatrix} = \begin{bmatrix}\mathbf{let}\ \mathsf{b{=}random(f(true\,,true){+}f(\ false\ ,true\,){:}}\\ \qquad\qquad \mathsf{f(true\,,\ false\ ){+}f(\ false\ ,\ false\ ))}\ \mathbf{in}\\ \mathbf{let}\ \mathsf{a{=}random(f(true,b){:}f(\ false\ ,b))}\ \mathbf{in}\ \mathsf{(a,b)}\end{bmatrix}$$

where f ranges over functions $\mathsf{bool{*}bool}\to\mathsf{int}$. (This is perhaps not the usual formulation of barycentric algebra [8] but it is equivalent, and it is actually the axiomatization proposed by Stone [20].)

Conjecture 2.2. *These equations completely describe the contextual equivalence induced by the module Beta_Bernoulli (equivalently, Polya).*

We say a few words about the soundness of these equations. In the urn-based implementation, Polya, commutativity involves some basic arithmetic. The key equation (1) is essentially the source code of the implementation.

For the purely probabilistic implementation, Beta_Bernoulli, in the standard semantics for probabilistic languages [16, 19], programs $\Gamma \vdash \mathsf{M} : \mathsf{t}$ are understood as probability kernels

$$[\![\Gamma]\!] \times \Sigma([\![\mathsf{t}]\!]) \to [0,1]$$

and sequencing $\mathbf{let}\ \mathsf{x{=}M}\ \mathbf{in}\ \mathsf{N}$ means integration:

$$[\![\mathbf{let}\ \mathsf{x{=}M}\ \mathbf{in}\ \mathsf{N}]\!] = \int [\![\mathsf{N}]\!]\,\mathrm{d}[\![\mathsf{M}]\!].$$

Thus commutativity is Fubini's theorem for interchanging the order of integration. The key equation (1) is the conjugate prior relationship between the Bernoulli distribution and the Beta distribution.

2.3 Remarks on affine monads

In earlier work [18], apart from probabilistic programming, we have shown how program equations such as the ones for the Beta–Bernoulli process correspond bijectively with monads on the category of functors $[\mathbf{FinSet}, \mathbf{Set}]$. The idea of using functor categories to build monads with name generation goes back to Moggi's initial work [11], and is common throughrout work on names and local memory [2, 12, 14, 15, 17]. From the point of view of monads, commutativity and discardability correspond to a monad being *affine* (e.g., [6]); see also [4, 7]. In future we hope to find explicit characterizations of the monads that arise from exchangeable processes.

3. Other kinds of exchangeability

We now consider some other exchangeability issues and show that they can also be expressed in terms of commutativity and discardability.

3.1 Chinese Restaurant Process

The Chinese Restaurant Process (CRP) describes a restaurant with the following protocol: as new people enter the restaurant, they pick a table; they are more likely to pick a busy table (see, e.g., [13, Ex. II.4]). It is often said that the sequence of tables allocated by CRP is an exchangeable sequence, but what does this mean? What is a table? If a table is an integer, then what is the number of the first table? If the first table is always 1, then statement of exchangeability becomes complicated. This is unfortunate because we would have to attach this bespoke statement of exchangeability to our CRP module. In fact, we can resolve this by saying that the type of tables is an abstract type, and the statement of exchangeability then becomes commutativity and discardability, so a bespoke statement of exchangeability is not needed. We replace the signature PROCESS with the following:

```
module type ABSTRACT_PROCESS = sig
  type process
  type result
  val new : unit → process
  val get : process → result
  val equal : result * result → bool  end
```

Here is a simple implementation, where a table is implemented as a pointer to an integer that records how many people are currently sitting at it:

```
module Crp = (struct
  type table = int ref
  type restaurant = (table list) ref
  type process = restaurant
  type result = table
  let new () = ref []
  let get r =
    let brand_new_table = ref 0 in
    let table_list =
      (1, brand_new_table) :: (map (fun t → (!t,t)) !r) in
    let chosen_table = sample(categorical( table_list )) in
    if chosen_table == brand_new_table
    then r := brand_new_table :: !r ;
    chosen_table := !chosen_table + 1 ; chosen_table
  let equal(x,y) = (x==y)
end : ABSTRACT_PROCESS)
```

Here categorical denotes a categorical distribution. For instance, sample(categorical [(11,'' hi '');(12,'' bye '');(27,'' pps '')]) produces hi with probability $0.22 = \frac{11}{(11+12+27)}$, bye with probability 0.24, and pps with probability 0.54.

We can now reformulate contextual equivalence in this setting.

Proposition 3.1. *The module for the Chinese Restaurant Process (Crp) is exchangeable: we have commutativity and discardability up to contextual equivalence.*

3.2 Exchangeable random graphs via commutativity and discardability

The Chinese Restaurant Process can also be seen as a random graph. We give the terms of the signature different labels:

```
module type RANDOM_GRAPH = sig
  type graph
  type node
  val new : unit → graph
  val fresh_node : graph → node
  val is_edge : node * node → bool
end
```

The idea is that there is an underlying distribution over infinite graphs. The command new samples a new infinite graph from the distribution, and fresh_node retrieves the name of a fresh, unseen node. We can explore the graph by asking whether there are edges between nodes. In the CRP, the graph is actually an equivalence relation, but one can consider random graphs more generally.

A probability distribution over infinite graphs is said to be *exchangeable* if permuting the nodes of the graph does not affect the distribution. In fact, this notion of exchangeability coincides with commutativity and discardability of calls to fresh_node.

Given nodes a_1 ,..., a_n:node, we can find the adjacency matrix $\mathsf{adjacency}_n(\mathsf{a}_1,...,\mathsf{a}_n)$:$\mathsf{bool}^{n\times n}$. For example, $\mathsf{adjacency}_2(\mathsf{a},\mathsf{b})$ is

$$(\mathsf{isedge}(\mathsf{a},\mathsf{a}), \mathsf{isedge}(\mathsf{a},\mathsf{b}), \mathsf{isedge}(\mathsf{b},\mathsf{a}), \mathsf{isedge}(\mathsf{b},\mathsf{b})) : \mathsf{bool}^{2\times 2}.$$

Now, commutativity and discardability of calls to fresh_node amounts to exchanging the row/column indices in the adjacency matrix. For instance, for any permutation π of n,

$$\begin{bmatrix} \textbf{let } g{=}\mathsf{new}() \textbf{ in let } \mathsf{a}_1{=}\mathsf{fresh\_node\ g} \textbf{ in } \ldots \\ \qquad\qquad\qquad \textbf{let } \mathsf{a}_n{=}\mathsf{fresh\_node\ g} \textbf{ in} \\ \qquad\qquad\qquad \mathsf{adjacency}_n(\mathsf{a}_1,...,\mathsf{a}_n) \end{bmatrix}$$

$$= \begin{bmatrix} \textbf{let } g{=}\mathsf{new}() \textbf{ in let } \mathsf{a}_{\pi^{-1}(1)}{=}\mathsf{fresh\_node\ g} \textbf{ in } \ldots \\ \qquad\qquad\qquad \textbf{let } \mathsf{a}_{\pi^{-1}(n)}{=}\mathsf{fresh\_node\ g} \textbf{ in} \\ \qquad\qquad\qquad \mathsf{adjacency}_n(\mathsf{a}_1,...,\mathsf{a}_n) \end{bmatrix}$$

$$= \begin{bmatrix} \textbf{let } g{=}\mathsf{new}() \textbf{ in let } \mathsf{a}_1{=}\mathsf{fresh\_node\ g} \textbf{ in } \ldots \\ \qquad\qquad\qquad \textbf{let } \mathsf{a}_n{=}\mathsf{fresh\_node\ g} \textbf{ in} \\ \qquad\qquad\qquad \mathsf{adjacency}_n(\mathsf{a}_{\pi(1)},...,\mathsf{a}_{\pi(n)}) \end{bmatrix}$$

The first step is by commuting calls to fresh_node, and the second step is just renaming variables.

Open Question 3.2. *For every exchangeable implementation of the signature RANDOM_GRAPH, is there an implementation that only uses probabilistic primitives, not those for states, and induces the same notion of contextual equivalence?*

At first sight, one might expect the answer to be "yes" by some generalisation of Theorem 2.1. However, we do not know whether there is a generalisation. We know that sampling from a Gaussian distribution is very much like name generation, in that it almost surely never gives the same result twice, and exchangeable random graphs have canonical representations via the Aldous–Hoover theorem, which generalises de Finetti's theorem (see, e.g., [13]). But we do not know whether it is possible to combine these ideas. If not, it would be interesting to find a minimal extension of probability theory that admits an answer of "yes".

4. Conclusion and other directions

In this abstract we have focussed on the semantics of sampling from exchangeable processes. We have argued that the concepts of exchangeability from statistics can be studied in terms of the commutativity and discardability program equations, by carefully using abstract types and fresh name generation.

There are other aspects to probabilistic programming, besides sampling, such as conditioning, non-termination, and simulation, which we have not covered here. For instance, efficient simulation for stateful exchangeable modules would require thread-local state [9] or 'unsampling' [21].

Selected References

[1] N. Ackerman, C. Freer, and D. Roy. Exchangeable random primitives. In *Proc. PPS 2016*, 2016.

[2] M. P. Fiore, G. D. Plotkin, and D. Turi. Abstract syntax and variable binding. In *Proc. LICS 1999*, 1999.

[3] C. E. Freer and D. M. Roy. Computable de Finetti measures. *Ann. Pure Appl. Logic*, 163(5):530–546, 2012.

[4] C. Führmann. Varieties of effects. In *Proc. FOSSACS 2002*, 2002.

[5] N. Goodman, V. Mansinghka, D. M. Roy, K. Bonawitz, and J. B. Tenenbaum. Church: a language for generative models. In *UAI*, 2008.

[6] B. Jacobs. Affine monads and side-effect-freeness. In *CMCS 2016*.

[7] O. Kammar and G. D. Plotkin. Algebraic foundations for effect-dependent optimisations. In *Proc. POPL 2012*, 2012.

[8] K. Keimel and G. D. Plotkin. Mixed powerdomains for probability and nondeterminism. 2015.

[9] O. Kiselyov and C.-C. Shan. Probabilistic programming using first-class stores and first-class continuations. In *Proc. 2010 ACM SIGPLAN Workshop on ML*, 2010.

[10] V. K. Mansinghka. *Natively Probabilistic Computing*. PhD thesis, MIT, 2009.

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

[12] P. W. O'Hearn and R. D. Tennent. Parametricity and local variables. *J ACM*, 42(3):658–709, 1995.

[13] P. Orbanz and D. Roy. Bayesian models of graphs, arrays and other exchangeable random structures. *IEEE Trans. Pattern Anal. Mach. Intelligence (PAMI)*, 2014.

[14] A. M. Pitts. *Nominal sets: Names and Symmetry in Computer Science*. CUP, 2013.

[15] G. D. Plotkin and J. Power. Notions of computation determine monads. In *Proc. FOSSACS 2002*, 2002.

[16] N. Ramsey and A. Pfeffer. Stochastic lambda calculus and monads of probability distributions. In *Proc. POPL 2002*, 2002.

[17] I. Stark. Categorical models for local names. *Lisp and Symbolic Computation*, 1996.

[18] S. Staton. Instances of computational effects. In *Proc. LICS 2013*.

[19] S. Staton, H. Yang, C. Heunen, O. Kammar, and F. Wood. Semantics for probabilistic programming: higher-order functions, continuous distributions, and soft constraints. In *Proc. LICS 2016*, 2016.

[20] M. H. Stone. Postulates for the barycentric calculus. *Annali di Matematica Pura ed Applicata*, 29:25–30, 1949.

[21] J. Wu. Reduced traces and JITing in church. Master's thesis, MIT, 2013.