Probabilistic Programming

Mads Buch

Aarhus, Denmark

Abstract

Probabilistic programming is pervasive in all computations considering incomplete data, and this is a large domain. As such, strong abstractions and formalizations are necessary to program efficiently. In this article, we will look into fundamental techniques for probabilistic programming. In particular, we will take offset in the monadic structure of probabilities.

We will apply these developed techniques in two areas: Machine Learning and Security. In machine learning the goal task is given some amounts of data, to make a function that fits that data. We will take offset in a simple application considering dice rolls.

As for security, we will look into quantitative information flow and cryptography. Here we are respectively interested in limits on how much information is observable from some entity and to what extent two distributions are equivalent. For the information flow perspective on the entropy of a probability distribution while we take a look at some logics that allow propositional reasoning about probability distributions to look into the relation between two distributions.

Keywords: Probabilistic Programming Languages, Machine learning, quantitative information flow, Cryptography, Programming Languages

Part I
Overview

Probabilistic reasoning is ubiquitous in computer science. It is in all places where we do not have complete information. Here we need to approximate, to guess or to predict some value. The applications for probabilistic reasoning are wide, but even so, we try to delve into some general notions of probabilistic reasoning.

For machine learning, which is the first application we delve into, the key goal is to predict. The idea is that, given some data, we may be able to predict the outcome of a future event. It should be possible to do that. After all, it is what we constantly do as human beings.

Next, after machine learning, we will look at the applications of probabilistic reasoning in security. Concretely we look at two areas that rely on probabilistic techniques: Quantitative information flow and cryptography.

The motivation for quantitative information is to figure out how much information flows between entities in a system. These entities can be timing modeled formally.

In Cryptography we seek to support the strength of our cryptographic protocols. We develop these proofs through games where an adversary has to guess better than picking at random. Here probabilistic reasoning is at the core when defining bounds on security.

This technical report we will provide an overview of the current research directions within probabilistic programming. The first section will provide an overview of the area with a strong offset in a monadic formulation of probabilities.

After that, we will look into select areas of probabilistic programming such as sampling, entropy, and propositional reasoning using probabilities.

The two last parts of the report is a dive into machine learning and how we use probabilistic programming in that setting. This part is followed up by one on security elaborating on quantitative information flow and cryptography.

1. Monadic Structure

Monads make out a good choice to encapsulate probability distributions. They provide a natural way to lift values to a distribution and to compose distributions from existing ones.

In this section, we seek to investigate probabilities using monads. The core structure is identical for all of them, but other than that, there is no consensus on these frameworks. We will look at some of the ideas and try to compare them.

The ideas are from following papers [1, 2, 3, 4]. [1, 2] both use a purely monadic style implemented in Haskell, while [3] implemented it in Coq. Both of these choices make good sense taking their goals into consideration. In [4] they develop a language which structure resembles the monad.

The core functions implemented for monads are *bind* and *return*. They are the same in both Ramsey's, and Scibior's implementations, namely that *bind* resembles a composition and return is the Dirac distribution. Park implemented bind through sampling with all expressions lifted through the use of the *prob* construction.

The differences lay in what auxiliary functions their frameworks provide.

1.1. Operations

After having defined the core monadic structure, we implement some operations that let us perform some useful tasks with monads. They are different depending on who you ask, their application, and what constraints they impose.

For learning tasks of large domains, we want to be able to sample because it is infeasible to consider the full domain. In information flow, we want to be able to derive the entropy of a distribution. For this, we need a *support* query and an *expectation* query. For sufficiently large domains we will also use the Monte Carlo methods for expectation and support, but more on that in the sampling section.

1.2. Implementations

Having looked into the monad approach to probability distributions, we will now elaborate on the implementations. We will take a look at their design motivations and how they compare.

Fundamentally all the monads have the same core functions. That is, the *return* operator is the Dirac distribution that forms a distribution of a single value with an expectation of 1. The *bind* operator amounts to draw from one distribution to constitute another distribution.

Now, these two constructions do not introduce random choices in any ways. Their purpose is primarily there for modeling reasons.

The following implementations are interesting in the way they tackle the task of introducing randomness, and what features they emphasize.

Ramsey's monad [1] augments the standard monad constructors by a choice operator. The operator takes a probability, p. This probability is a number between zero and one. It then returns one of two other expressions with probability respectively p and $p - 1$.

The *choose* operator is simple in the sense that it is how we think about stochastic lambda calculus. It makes it possible to implement the *support* query, and the *expectation* query using exact implementations. This is desirable for any application using the entropy of a distribution or otherwise in need of these queries. They are, however, slow for large distributions which should not come as a surprise, as the *choose* operator introduces probability distributions with exponential fan-out.

Furthermore, it is possible to implement the *sample* query widely used in any applications that need approximate queries.

```
data P a where
    Return :: a -> P a
    Bind   :: P a -> (a -> P b) -> P b -- The reason for GADTs
    Choose :: Probability -> P a -> P a -> P a
```

Figure 1: The algebraic definition of the probability monad from [1].

Figure 1 shows the implementation using monadic terms. We could also have implemented it using measures. This way would, however, pose a limit on implementing the *support* query. Such an implementation would include enumerating the entire domain of the probability distribution. for each element we would need to calculate the expectation and exclude it if the expectation is 0.

Lastly, we mention that this implementation only works for discrete distributions, again, due to the *choose* operator. That choice gives rise to some applications, e.g., propositional reasoning, but makes other applications harder.

Next, we look at Scibiors implementation [2].

This implementation extends the monad with a *primitive* operator and a *conditional* operator. This implementation is from the paper titled *Practical Probabilistic Programming with Monads* and has a heavy focus on the machine learning aspect of probabilistic programming. As such, none of these

added operators provide a good fit for exact queries. They do, however, make it much easier to do sampling and conditioning. It is even possible to encode continuous distributions in this implementation.

```
data Dist a where
    Return      :: a -> Dist a
    Bind        :: Dist b -> (b -> Dist a) -> Dist a
    Primitive   :: Sampleable d => d a -> Dist a
    Conditional :: (a -> Prob) -> Dist a -> Dist a
```

Figure 2: The algebraic definition of the probability monad from [2].

The code in Figure 2 is the algebraic definition of the code from the paper. The two first cases are not particularly interesting. The *Primitive* constructor takes anything that implements the *Sampleable* type class and lifts it into the monad. This lifting includes external functions and is the reason why we can make arbitrary distributions.

The *Conditional* constructor is used to condition a probability distribution. To do this, we need a function calculating the likelihood of an element and a prior probability distribution. We then get a conditioned distribution. It is not possible to sample from that. First, we need to *infer* the posterior distribution.

Next, we have an implementation with a goal of supporting propositional reasoning. Aside from the usual constructors, we consider the *Rnd* and the *Repeat* constructors. The biggest difference from this to the others is, that it makes randomness completely explicit when we ask for a vector of random bits.

```
Inductive Comp : Set -> Type :=
| Ret : forall {A : Set}{H: EqDec A}, A -> Comp A
| Bind : forall {A B : Set}, Comp B -> (B -> Comp A) -> Comp A
| Rnd : forall n, Comp (Bvector n)
| Repeat : forall {A : Set}, Comp A -> (A -> bool) -> Comp A.
```

Figure 3: The algebraic definition of the probability monad in [3].

Figure 3 shows the implementation used in their paper. The *Ret* and *Bind* are the usual monad constructors for the probability monad. *Rnd* is

the constructor that exposes randomness. It returns an n-sized vector of random bits, which then has to be utilized by the application routines. The last constructor, *Repeat*, is used to do a kind of conditioning on a distribution. This conditioning could aid sampling in some interval on a distribution over natural numbers.

The last implementation is from [4]. They went with a complete formulation of a language rather than implementing it as a DSL in an existing language. They base the language on the idea of lifting entire terms into a *circle*-type on which we can do sampling.

Other operations they implemented in the language are the *expectation* query and the *Bayes* operation, which is essentially inference. They are both implemented through sampling and need an argument on how much data to sample to approximate the queries. We may embed both of the queries in a probabilistic term, e.g. we can do an expectation query within a distribution.

Furthermore, the sampling based algorithms use a global randomness pool. This global approach results in computations with side effects which are notoriously hard to reason about.

While this might seem like a good idea, it makes it hard to reason about the distribution as a whole and certainly breaks compositionality. Therefore this approach is mostly here for completeness and reference.

Final remarks: All above implementations are different. From here it is hard to say whether one is better than another as they all suit different goals. The one by Ramsey emphasizes a general purpose implementation that is conceptually simple. However, they did not emphasize inference.

Scibior made a monad with the particular aim of being specialized to inference based on sampling. They indeed succeeded in that, but this implementation would not be useful in the security context where rigorous introspection or entropy of large distributions is the question. Hence he made a good implementation for Machine learning.

Lastly, Petcher made a monad specialized towards doing formal reasoning. Among other things, the way he handles randomness is very suited this goal. The monad provides randomness as a vector of discrete values. This approach makes it well suited for the purpose. However, it is not natural to express random computations.

2. Semantics

After the above formulations of probability distributions, a natural question is to what extent we can express the probability distributions we usually do. To answer this question, we look into the semantics of the probability distributions

In [1], they develop an interpretation of the probability through stochastic lambda calculus. This translation shows the generality of the probability monad. In his monad, we can implement the usual stochastic lambda calculus. This interpretation should not come as a surprise as he augments his monad with the *choose* operator which tightly resembles a stochastic choice.

Both Ramsey and Scibior interpret the monad into measure terms. This interpretation yields an argument of the soundness of their monadic approaches.

We will not look further into the formal definitions other than referring to [1, 2].

3. Sampling

In this section, we look a bit further into sampling. Sampling is necessary because it is the only method that seems tractable for arbitrary distributions. Hence we wish to be able to express the other queries through sampling. This idea is fundamental for [4, 5, 2]

The issue on sampling also tabs into the difference between *discriminative* and *generative* models. The approach described in this text takes offset in generative models. A complete comparison of the two approaches is out of scope for the report. However, the go-to paper on this is [6]. The conclusion of the article is that for performance oriented applications we should use discriminative models. Discriminative models solely model conditional distributions and therefore are not general enough to capture all uses of probabilities.

The motivation is the tractability of doing exact calculations of certain properties about probability distributions. Given a normally distributed five character string (in the interval 'aaaaa' .. 'zzzzz'), I can sample at a rate of 22MB (3MSamples) per second while it takes 3.348 seconds to calculate the exact expectation of an element in the distribution.

The Main two points we emphasize is that it takes linear time to sample (in a graphical model) where it takes exponential time to do exact queries

(traversing trees). This complexity difference, naturally, comes with a trade-off; We can only do approximate queries from sampling.

For machine learning applications sampling is usually fine. We need approximate queries. They just have to answer well within range. On the other hand, doing formal reasoning makes it strictly harder to use sampling. The conclusion is that sampling is useful in empiric domains but it is not ideal in strictly formal domains.

4. Entropy

Sometimes we are interested in how random a distribution is. We cover this question under the term entropy. The notion entropy often defaults to what was defined by Shannon, i.e. what is the lower bound of the number of bits we need to do a transmission. However, other notions exist, defined on some other metrics being more useful in some applications.

For this report, we will look at two types of entropy: Shannon entropy and min-entropy. They differ in how they fundamentally approach the question of entropy in information science and their applications.

The machine learning community widely uses the Shannon entropy where we seek to decrease the entropy as much as possible to a concise description of some data. Its origins as a mean to discuss compression of data, and as such it is used in compression based models.

On the other hand, min-entropy is proposed to be the notion used in information flow as it takes a more conservative approach to the concept of entropy.

Following to this is a presentation on the implementations showing how the entropy discussion fits into the Ramsey's probability monad. It will be evident that it is indeed possible to implement it and use it in real applications for sufficiently small distributions.

4.1. Implementations

We used Ramsey's probability monad to calculate entropies. This was because their monad supports the *support* and the *expectation* query. These can also be implemented in an exact manner in Petchers monad but will be hard to realize in Scibiors. In Scibiors monad we would approximate entropy through sampling. This approximation becomes infeasible when the domain of the distribution gets bigger.

```haskell
shannon :: (Eq a, Ord a) => P a -> Double
shannon px =
  let
      exp x = expectThat px x
   in
      sum ( map (\x -> (exp x) *
                       (logBase 2 $ (1 / exp x))) $ supp px )
```

Figure 4: The implementation of the Shannon entropy as written in [7].

The code in Figure 4 shows the Haskell implementation of the Shannon entropy. It uses some functions not explicitly defined though it should be immediate from their names what they do. From the last line, it is apparent that it maps over the full domain of the distribution and sums the result. Though this problem is embarrassingly parallel, its complexity still outperforms computing power for interesting distributions.

Above implementation was proposed bad for the application of quantitative information flow as it classified. Instead, Smith proposes to use the min-entropy notion for evaluating distributions and their information leakage [7].

```haskell
condVuln :: (Ord x, Ord y) => (P y -> P x) -> P y -> Double
condVuln px py = sum [(expectThat py y) *
                      (vuln $ px $ return y) | y <- supp py]

condMinEntropy px py = logBase 2 (1/(condVuln pl ph))
```

Figure 5: The implementation of the the entropy entropy as written in [7].

The implementation in Figure 5 shows how we implemented min-entropy through Ramsey's probability monad. Again it completely resembles the implementation from [7] and it has the same complexity concerns as the other implementation.

5. Propositional Reasoning

Probability is a general term that many contexts use. Inferring from samples yield a learning perspective while calculating the entropy of a prob-

ability distribution is interesting in an information-theoretic setting. In this section, we look into frameworks for propositional reasoning. In particular, we look into the methodology developed for reasoning about cryptographic proofs from the FCF paper[3].

The key thing we reason about is the equivalence of distributions. Being able to do this allows us to drag a pair of probability distributions into the world of propositions. With such a notion, we can further develop the theory to incorporate logical connectors.

The core monadic approach used is the same as the formulation in Ramsey's paper. Concretely they use the formulation based on measure terms here, i.e. the continuation monad that returns a probability for a given input. In this context, they formulate it in Gallina (Coq), as opposed to Haskell, which also makes sense, as we get many things out of the box.

Finally, as this is propositional reasoning, we need to have terminating computations to have a sound system. For this, we require discrete probability distributions with a finite support. This requirement also yields a value of 1 when summing the expectations of the whole support.

5.1. Relating Distributions

As mentioned we want a notion of equivalence for the distributions we consider. For this, two cases have been derived as theorems, one where the distributions are identical and one where they have a mutual bound.

For directly equal probability distributions we simply treat the distributions as pure functions. In this context, it yields their probability mass functions. For this property to be satisfied, we need that, for each input the PMFs yield the same mass. Notice that this is not a bound, and as such, the masses have to be identical.

This property is the Theorem 4 from the FCF paper.

Furthermore, we want to reason about bounds of probabilities. To do that in this framework they have introduced the notion of bad events that might occur in a probability distribution. This approach is in the spirit of the identical until bad games from [8].

5.2. Program Logic

To express richer logical statements about a distribution they develop a Hoare logic. They design the logic in two stages. The Probabilistic Relational Postcondition Logic (PRPL) allows to reason about pot conditions of

a probability distribution. The idea for this is to be able to set up a formula Φ to relate two computations under:

$$p \sim q\{\Phi\} \tag{1}$$

Given PRPL it is straightforward to define the Probabilistic Relational Hoare Logic (PRHL). By that, we can reason relationally about probabilities in the following form:

$$\{\Psi\}p \sim q\{\Phi\} \tag{2}$$

This section forms the foundation to build cryptographic game proofs.

6. Research Questions

This area is still emerging, and as such, much of the material is rather new. Following is a select list of research questions from the literature. It should provide an idea of the current state of this field.

- **Operations based on sampling:** Evidently it is not tractable to implement exact methods of certain operations. Hence we need to approximate them. A neat foundation for this would be sampling as it is cheap and does not require any introspection on the distribution.
- **Parallelization:** This is an extension of above. The process should be fast enough to achieve good results in feasible time. Parallelization is a hot topic, and plenty of specialized hardware for the purpose exist. Hence efficient parallel implementation of sampling based operations would push this field into a useful state.
- **Formal semantics:** The body of work is quite small in this area of providing a formal semantics for these constructions. A formal semantics would increase the reliability on the constructions and provide deeper insights into their applications.

Many more questions exist. A good starting point is [2].

Part II
Machine Learning

> Don't calculate probabilities;
> Sample good guesses.
>
> *Vikash Kumar [5]*

One of the tasks using probabilistic reasoning is machine learning. Machine learning is a very broad field collecting research from many different areas. This foundation makes it a rather daunting area to study within as many ideas, and approaches are in play.

The general goal, however, is to come up with a function resembling some real world process generating some data. The literature proposes several ways for deriving these functions though following are, what I find to be, two central ways to achieve this.

- **Inference:** The central idea here is to have a proposal distribution. For each observation, we condition the distribution. The task is to infer a posterior distribution from the prior distribution and all its conditioning. In [2], the authors used this approach.
- **Optimization:** Again, we have a proposal distribution. This distribution contains potentially several variables in the range $]0; 1]$ each representing a random choice. The task is then to optimize these variables to yield a distribution that matches the proposal data.

The Hakaru project is one of the promising, relevant, techniques [9]. They model the problem as a joint distribution and then turn it into a conditional distribution. This approach is general, and model developers can use to develop generative models, which is also those we focus on in this report.

Traditionally the field has a great focus on speed over expressibility. This goal leads to an algorithmic centric view which has no coherent foundation. Furthermore, most algorithms are developed using discriminative models as these are faster but more restricted [6]. To mitigate that, a research industry of applying probabilistic reasoning to learning tasks is emerging.

In this chapter we will look at a simple example building a distribution from a list of elements. It is simple, yet illustrative. The central point is

```
observe :: a -> (Int, P a) -> (Int, P a)
observe e (t, prior) = let t' = t+1 in
        (t'+1, choose (1/(fromIntegral t')) (return e) prior)
```

Figure 6: A function that takes an observation and adjust the distribution.

that it is very hard to talk about machine learning in general terms. The material is in plenty. Hence a project is necessary to guide through the ocean of knowledge.

7. Dice

To have an illustrative example of a learning task, we will attempt to learn the distribution of a dice based on a list of dice rolls. We use frequencies as our main approach. This method is useful when considering large amounts of data.

In 6 the function *observe* takes an element and a distribution of the same type as the element. The input distribution is augmented by a count of elements already observed. Now we choose that element by $\frac{1}{N}$. Doing this yields a uniformly distributed distribution over all elements it has observed, given that all elements are unique. If multiple elements are the same, we sum their expectations.

We wrapped the probability distributions in a tuple of the distribution and the total count of elements it has observed. We need this total count to calculate how much impact the new element should have. In this approach we let the impact of the new element be inversely proportional to the number of elements we already saw. As we also noted, this yields an inductive uniform distribution.

Part III
Security

Probabilistic reasoning is a fundamental technique for assessing security risks. In this part, we will look into two security relevant subjects: Quantitative information flow and cryptography.

In quantitative information flow, we are interested in how much information flows from one entity to another in a system. To uncover covert channels we consider models of the computations.

In cryptography, we are interested in assessing the difficulty in retrieving the clear text of some encrypted string. Here we are interested in doing proofs that reduce the hardness of a protocol to something that we consider hard. For this, we use the notion of game-based security proofs. These can be formalized using the monadic approach presented in Part 1.

8. Quantitative Information Flow

In quantitative information flow, we are interested in how much data flows from one entity to another. These entities can be variables in a programming language, processes in a computer, etc.

In the setting of security, we are particularly interested in how much information flows from secret data source to public data source of a system. These variables can represent information in both a direct or indirect manner. An example of an indirect data source is the response time from another process to derive information about some private data in the service.

In this section, we will first take a look at some clean examples to establish the vocabulary. After that, we will look into a password checker, where we try to derive how much information we get about the users in the database simply by looking at timing.

8.1. Motivation and Entropy

In Smiths paper, [7], he presents two expressions. They both leak information from a variable h (denoting high security and should be secure) to another variable l (denoting low security and is publicly available).

```
if h mod 8 == 0
    then l = h
    else l = 1
```

Figure 7: Example of leakage from [7].

```
l = h & 0^(7k1)1^(k+1)
```

Figure 8: Example of leakage from [7].

The first expression, Figure 7, leaks all data with a probability of $\frac{1}{8}$ where the other expression, Figure 8 always leaks a fraction of the high variable.

Using the Shannon entropy, the two programs leak equally much information indifferent of k. The leak is worse 7 which motivates to find another notion of entropy for this setting.

Smith proposes the notion of min-entropy. This notion makes sense as it is a more conservative measure of entropy.

8.2. Password

In this section, we build a model of a password checker. The scenario is as follows: We check user credentials in a database. We do this by scanning through the database linearly. We have detected that the interface responds faster when the user exists in the database because we do not need to scan through all entries.

In this scenario, an attacker can derive a username by asking the database and see how fast it answers. As developers of the interface, we want to figure out how much data the timing channel leaks.

The amount of data leaked is the conditional entropy of the private variable, the user name, and the public variable, the timing. To investigate how this can be used we build a **model** of a password checker making the timing explicit. For simplicity reasons we decided to use an increasing counter on the list of user/password-pairs, we are traversing.

The code in Figure 9 shows the implementation of a password checker. It enumerates a list until it finds a hit. Here it checks the password not explicitly reporting whether the user is in the table. If no hit is found it returns False.

```
checkPword :: (Uname, Pword) -> [(Uname, Pword)] -> (Bool, Time)
checkPword e es = doCheckPword e es 0
  where
    doCheckPword _ [] t = (False, t)
    doCheckPword e (x : xs) t
            | (fst x) == (fst e) =  if snd e == snd x
                                    then (True, t)
                                    else (False, t)
            | otherwise          =  doCheckPword e xs (t+1)
```

Figure 9: An implementation of a password checker that deliberately leaks time

The public variable in our model is the time. We have made the time explicit as an increasing number depending on where the element is in the list. We build a probabilistic model of the private and public variables. For this, it is evident that the public variable is dependent on the private one.

```
-- Public
pTime pUname = do
    pword <- pStringNormal -- We don't care about the pword
    uname <- pUname
    return (snd ( checkPword (uname, pword) pWords) )

-- Private
pUname = pStringNormal
```

Figure 10: Probabilistic modeling of the private and public data

In Figure 10 we have the implementation of the model. As visible we directly use the implementation of the password checker, and as such, we are always sure that our probabilistic model corresponds to our actual model of a given situation.

After the modeling, we can attempt to derive the entropy. To make it feasible we decided to reduce to the domain of the strings to two characters. Already at three characters, the queries took too long to be feasible on our computers with a sequential implementation.

The derived Shannon entropy was roughly **0.09618** while the min-entropy

yielded approximately **0.99112**. This difference indicates, as expected that the min-entropy reports a more conservative measure than the Shannon entropy.

A remark on modeling and sampling: When implementing this example I had to implement uniform distributions over strings. Here we have a couple of ways to do it. The naive way would be something like following.

```
pThreeLetterStringOld = uniform [[x1, x2, x3] |
    x1 <- ['a' .. 'z'],
    x2 <- ['a' .. 'z'],
    x3 <- ['a' .. 'z']]
```

Figure 11: An inefficient implementation of a uniform distribution over a 3 character string

The idea is to make a list of all three-letter strings and then make it to a uniform distribution. This construction is not efficient for sampling as we get a list of 17576 elements we potentially have to run through.

```
pStringNormal = do
    x1 <- uniform ['a' .. 'z']
    x2 <- uniform ['a' .. 'z']
    x3 <- uniform ['a' .. 'z']
    return [x1, x2, x3]
```

Figure 12: An efficient implementation of a uniform distribution over a 3 character string

An alternative implementation in this setting would be the one in Figure 12. This method is strictly faster to sample from, as we have three lists of 26 elements summing to a maximum of 78 elements.

The speed increase does have a cost as we use three times more randomness.

Another thing we also have to note is that exact expectation querying takes the same time in both implementations. However, if we do keep the recommendations about sampling in mind, the second implementation is preferred.

9. Cryptography

We have already discussed propositional reasoning using probabilities. These techniques are applied to numerous proofs within cryptography. For a comprehensive walk through on the examples, we refer to [3].

References

[1] N. Ramsey, A. Pfeffer, Stochastic lambda calculus and monads of probability distributions, SIGPLAN Not. 37 (2002) 154–165.

[2] A. Ścibior, Z. Ghahramani, A. D. Gordon, Practical probabilistic programming with monads, in: Proceedings of the 2015 ACM SIGPLAN Symposium on Haskell, Haskell '15, ACM, New York, NY, USA, 2015, pp. 165–176.

[3] A. Petcher, G. Morrisett, The foundational cryptography framework, in: Proceedings of the 4th International Conference on Principles of Security and Trust - Volume 9036, Springer-Verlag New York, Inc., New York, NY, USA, 2015, pp. 53–72.

[4] S. Park, F. Pfenning, S. Thrun, A probabilistic language based on sampling functions, ACM Trans. Program. Lang. Syst. 31 (2008) 4:1–4:46.

[5] V. K. Mansinghka, Natively Probabilistic Computation, Ph.D. thesis, Cambridge, MA, USA, 2009. AAI0821830.

[6] A. Y. Ng, M. I. Jordan, On discriminative vs. generative classifiers: A comparison of logistic regression and naive bayes, in: T. G. Dietterich, S. Becker, Z. Ghahramani (Eds.), Advances in Neural Information Processing Systems 14, MIT Press, 2002, pp. 841–848.

[7] G. Smith, On the foundations of quantitative information flow, in: Proceedings of the 12th International Conference on Foundations of Software Science and Computational Structures: Held As Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, FOSSACS '09, Springer-Verlag, Berlin, Heidelberg, 2009, pp. 288–302.

[8] M. Bellare, P. Rogaway, The security of triple encryption and a framework for code-based game-playing proofs, in: Proceedings of the 24th Annual

International Conference on The Theory and Applications of Cryptographic Techniques, EUROCRYPT'06, Springer-Verlag, Berlin, Heidelberg, 2006, pp. 409–426.

[9] P. Narayanan, J. Carette, W. Romano, C. Shan, R. Zinkov, Probabilistic inference by program transformation in hakaru (system description), Functional and Logic Programming - 13th International Symposium, FLOPS 2016, Kochi, Japan, March 4-6, 2016, Proceedings (2016) 62–79.