# Probabilistic Programming Language and its Incremental Evaluation

Oleg Kiselyov

Tohoku University, Japan
oleg@okmij.org

**Abstract.** This system description paper introduces the probabilistic programming language Hakaru10, for expressing, and performing inference on (general) graphical models. The language supports discrete and continuous distributions, mixture distributions and conditioning. Hakaru10 is a DSL embedded in Haskell and supports Monte-Carlo Markov Chain (MCMC) inference.
Hakaru10 is designed to address two main challenges of probabilistic programming: performance and correctness. It implements the incremental Metropolis-Hastings method, avoiding all redundant computations. In the presence of conditional branches, efficiently maintaining dependencies and correctly computing the acceptance ratio are non-trivial problems, solved in Hakaru10. The implementation is unique in being explicitly designed to satisfy the common equational laws of probabilistic programs. Hakaru10 is typed; specifically, its type system statically prevents meaningless conditioning, enforcing that the values to condition upon must indeed come from outside the model.

## 1 Introduction

Broadly speaking, probabilistic programming languages are to express computations with degrees of uncertainty, which comes from the imprecision in input data, lack of the complete knowledge or is inherent in the domain. More precisely, the goal of probabilistic programming languages is to represent and automate reasoning about probabilistic models [4, 16], which describe uncertain quantities – random variables – and relationships among them. The canonical example is the grass model, with three random variables representing the events of rain, of a switched-on sprinkler and wet grass. The (a priori) probabilities of the first two events are judged to be 30% and 50% correspondingly. Probabilities are real numbers from 0 to 1 that may be regarded as weights on non-deterministic choices. Rain almost certainly (90%) wets the grass. The sprinkler also makes

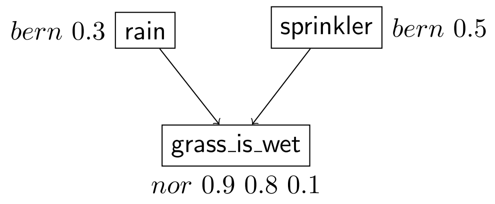


**Fig. 1.** Grass model

the grass wet, in 80% of the cases. The grass may also be wet for some other reason. The modeler gives such an unaccounted event 10% of a chance. This model is often depicted as a directed acyclic graph (DAG) – so-called Bayesian, or belief network [17] (Fig.1) – with nodes representing random variables and edges conditional dependencies. Associated with each node is a distribution (such as Bernoulli distribution bern: the flip of a biased coin), or a function that computes a distribution from the node's inputs (such as the noisy disjunction nor to be described below).

The sort of reasoning we wish to perform on the model is finding out the probability distribution of some of its random variables. For example, we can work out from Fig.1 that the probability of the grass being wet is 60.6%. Such reasoning is called probabilistic *inference*. Often we are interested in the distribution conditioned on the fact that some random variables have been observed to hold a particular value. In our example, having observed in the morning that the grass is wet, we want to find out the chance it was raining overnight. We are thus estimating a hidden parameter – inferring the likelihood of an unseen or unobservable cause – from observations. For background on the statistical modeling and inference, the domain area of the present paper, the reader is referred to Pearl's classic [17] and Getoor and Taskar's collection [4].

In the probabilistic language Hakaru10 to be presented in this paper, the conditional model just described in English can be written as follows:

```
grass = do
  rain          ← dist bern 0.3
  sprinkler     ← dist bern 0.5
  grass_is_wet  ← dist (True `condition` nor 0.9 0.8 0.1) rain sprinkler
  return rain

-- noisy-or function
nor strengthX strengthY noise = \x y →
  bern $ 1 − nnot (1−strengthX) x ∗ nnot (1−strengthY) y ∗ (1−noise)

-- noisy not function
type Prob = Double
nnot :: Prob → Bool → Prob
nnot p True  = p
nnot p False = 1
```

Even though some words like dist are not yet defined, one can already see the correspondence with Fig.1. The grass code compactly and, mainly, unambiguously represents the model – for the domain experts and also for the Hakaru10 system. The latter relieves us from working out probabilities by hand, performing the requested inference. Thus probabilistic programming languages let us separate the description of the model from computations on them, making the models declarative and accessible, for domain experts, to discuss and modify.

Probabilistic programming languages are easier to use and develop when they are embedded DSLs, implemented as a library or a macro on top of the existing general-purpose programming language. Hakaru10 is in fact such a DSL, embedded in Haskell. The grass model is the ordinary Haskell function, making use of the library functions bern, dist and condition. The full power of Haskell and all of its libraries is available for expressing deterministic parts of the model (such as the noisy-or probability computation). Embedded DSL let us also take the full advantage of the abstraction facilities of the host language such as functions, module systems, etc. For example, we have defined `nor` as a particular parameterized Bernoulli distribution conditioned on two inputs `x` and `y`. This function can later be re-used in other models. We may hence compose models from simpler components. Starting from the pioneering work of Sato [19], many probabilistic DSL have been proposed [5, 8], with host languages been logical [2, 19], functional [6, 12, 20, 22], object-functional [18], etc. Wingate et al. [21] deserves special mention for proposing a technique of adding probabilistic programming facilities to just about any language. The authors demonstrated this on the examples of Scheme and Matlab. Wingate's et al. approach has become well-spread and employed in many more probabilistic systems such as [9, 22].

There are so many probabilistic languages and we are still writing papers about (more of) them – driven by two main challenges. One is obvious, the other may come as a surprise. The obvious challenge is performance. An expressive, pleasant to use, well abstracted probabilistic programming language may be, it is all for naught if doing inference with realistic models takes unreasonable time or runs out of memory. For example, the probability monad – which adds weights to the well-known List monad for non-determinism – is the straightforward and the easiest to understand example of probabilistic programming in Haskell. It is Haskell folklore, well described in [3]. It is also disastrously inefficient, failing even for toy problems. Therefore, it is all too common in Machine Learning/AI communities to tailor the model to a specific inference method, and tune the inference code for a specific model. However prominent are the drawbacks of such tight coupling, often it is the only way to handle problems of realistic size.

That correctness is still a challenge may be surprising. Given the long history of probabilistic programming, one may think that the basics of the implementation are beyond doubts. Yet we keep finding problems in the published work [11]. The well-known and widely used systems such as STAN [9] turn out to give plain wrong answers even in simple cases, as Hur et al. [10] have clearly demonstrated.

*Contributions* Hakaru10 was developed to address both challenges, performance and correctness. It started as a project to improve the implementation of the (original) Hakaru [24] on two points: avoid redundant re-computations and to strengthen the typing discipline. It was discovered [11] along the way that the implementation principles, taken from [21] were flawed. Hakaru10 is the complete re-write, on new principles, to be described in the present paper. Specifically, the paper makes the following contributions.

1. It presents the probabilistic programming language Hakaru10 embedded as a DSL in Haskell.
2. It describes the design of Hakaru10, specifically, its type system, which ensures not only that a model is well-typed, but also that it is well-conditioned. That is, the values used for conditioning really come from the sources external to the model, rather than being produced from random sources and

computations within the model. In [20] that semantic well-conditioning constraint is a mere coding convention, whose violation leads to a run-time exception. We encode the constraint in types, without losing the benefits of the **do**-notation. Although the original Hakaru [24] enforced well-conditioning statically, it had to give up on the ordinary monads and made the conditioning difficult and error-prone to use: the observed quantities had to be referred to by De Bruijn-like indices. The type system of Hakaru10 improves not only on the static guarantees but also on the 'syntax' of the language: its Haskell embedding.

3. We describe the implementation of the Metropolis-Hastings (MH) probabilistic inference method (one of the Markov Chain Monte Carlo (MCMC) methods, see §4 for a reminder) that ensures semantic-preserving model transformations such as introduction and elimination of dirac random variables. The implementation is thus guaranteed to obey theoretically justified equational laws of probabilistic programs. The correct MH implementation, in the presence of branching, is quite non-trivial [10].
4. We present the method to improve the efficiency of MH by avoiding redundant re-computations. Although the idea is simple – upon resampling recompute only those parts of the model that depend on the changed value – the challenge is to minimize the overhead of determining the dependencies and their order. The challenge is acute in the presence of branching: if-then-else statements.

Hakaru10 thus fixes the three problems of the popular Wingate et al. approach [21] that have been pointed out in [11]. First, the implementation is designed to respect the unit law of the Dirac distribution. Second, Hakaru10 by design avoids the accidental sharing of random primitives. In Wingate et al. such sharing, however unjustified theoretically, was justified practically as increasing the performance. Hakaru10 improved the performance by avoiding unnecessary recomputations. The third problem, not able to use conditioning other than 'at the top level' has also been dealt with. This problem is so involved and important that is out of scope here, to be discussed in a separate paper.

We start in §2 with a Hakaru10 tutorial. §3 briefly evaluates the expressiveness and performance of the system. §4 describes the implementation in detail. We then review the related work and conclude. The complete code for Hakaru10 with many tests and examples is available at `http://okmij.org/ftp/kakuritu/Hakaru10/`.

## 2 Hakaru10 by Example

This section introduces Hakaru10 on a series of many small examples, showing off the features of the language. Incidentally, these simple models are useful regression tests for any probabilistic system. The section also demonstrates equational laws, or valid transformations of Hakaru10 programs.

Hakaru10 is designed for models that are described by a finite directed acyclic graph like Fig. 1, whose nodes represent random variables and edges indicate (typically causal) dependencies. Unlike graphical models in the strict sense [16], we allow dependencies with arbitrary pure computations.

## 2.1 Model Compositions and their Laws

The first, elementary program is

```
pbern = dist bern 0.4
```

whose inferred type is Model Bool. The program represents the model consisting of a single Boolean ('Bernoulli') random variable, whose distribution is True with the probability 40% and False with the probability 60%. One may think of the function bern :: Double → DistK Bool as creating a distribution given its parameter, and dist :: (a → DistK b) → a → Model b as sampling from it[1]. The types DistK and Model are abstract; the latter has the additional structure to be shown shortly.

The function mcmC, the interpreter of probabilistic programs,

```
mcmC :: Integer → Model a → [a]
pbern_run = mcmC 10 pbern
```

performs the MH inference on a model and returns the list of samples from the model's distribution. In particular, mcmC 10 pbern produces a list of 10 booleans, which indeed contains 4 True and 6 False values. Hakaru supports not only discrete like bern but also continuous distributions:

```
pnorm = dist normal 10 0.5
```

Here, pnorm, of the inferred type Model Double is a model with the single normally-distributed random variable, with the mean 10 and the standard deviation 0.5. Hakaru10 offers other primitive distributions, such as categorical, uniform, gamma and beta.

Almost all models are more complex, with more random variables, and mainly, with dependencies between random variables. (Hakaru10 intentionally does not support cyclic dependencies, as the semantics of such models is problematic.) The following model has two random variables, normal- and Dirac-distributed[2]. The parameter of the latter depends on the value of the former, for which we re-use the earlier written pnorm:

```
-- pdep1 :: Model Double
pdep1 = do
  x ← pnorm
  diracN (x+1)
```

To build complex models we use the Haskell **do** notation. In the example above, we 'name' the submodel (random variable) pnorm as x, which we later use in the expression to compute the parameter of the Dirac distribution. Haskell embedding truly brings modularity: we can name Hakaru10 models (binding them to Haskell variables) and (re)use constructed models as parts of other models, as we have just done with pnorm. Since the Dirac distribution is frequent and

[1] These signatures are somewhat simplified. We will later see that dist is overloaded.
[2] Dirac distribution, or Dirac delta, is taken as density of a discrete random variable with the single value.

special, as we are about to see, there is a shorter syntax for it, just diracN. As expected, the inferred distribution of pdep1 is exactly the same as that of dist normal 11 0.5.

Likewise, the model

```
pdepR = do
  x ← pnorm
  diracN x
```

is equivalent to just pnorm. This is the general property, not limited to normal distributions: for any model p

```
do {x ← p; diracN x}
```

has the distribution identical to that of p. Even the sequences of samples for the two programs are identical. Likewise, for any e,

```
do {x ← diracN e; p}
```

is equivalent to **let** x=e **in** p. Hence diracN acts as the left and the right unit of the model composition. Not surprisingly, diracN has the alias return. At this point a Haskell programmer may think that Model is a monad. This is not quite true, as we shall soon see.

The joint distribution of pbern and pnorm models is described by

```
pjoin = do
  x ← pbern
  y ← pnorm
  return (pair x y)
```

Then mcmC 100 pjoin produces a set of (Bool,Double) pairs sampled from the joint distribution, which in this case is the product of pbern and pnorm distributions. That the distributions of x and y are independent can be seen *syntactically*, from the fact that the model named y, namely, pnorm, has no mentioning of x. We can even integrate (that is, 'marginalize') over x:

```
pmarg = do
  x ← pbern
  y ← pnorm
  return y
```

Since x does not appear further in the program, this random variable is irrelevant and is essentially marginalized. Hence the distribution of pmarg is the same as that of pnorm (although the sequences of samples certainly differ). This property is again general, for all models (not using conditioning, see below).

To demonstrate once more the advantage of the Haskell embedding, we borrow the example of a simple hierarchical model from [10, Fig. 3]. In the standalone, C-like imperative probabilistic language of that paper, the example looks as follows:

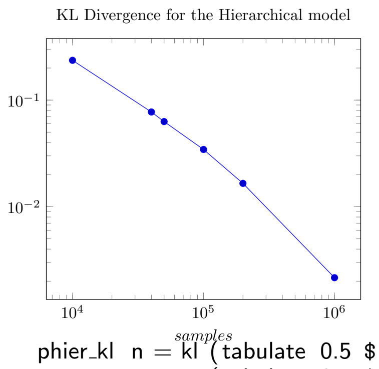
```
double x;
int i = 0;
x ~ Gaussian(0, 1);
while (i < 10) do {
  x ~ Gaussian(x, 3);
  i = i+1;
}
return x;
```

The graph of this model with 11 variables looks like the straight line. In this C-like code, like in C, x denotes a sample from the model rather than a model.

In Hakaru10, we write the hierarchical model as

```
phier = (iterate (\m → do {x←m; dist normal x 3}) $ dist normal 0 1) !! 10
```

taking the advantage of Haskell's standard library: iterate f x produces a list whose i-th element is the i-th iterate of f over x. The significance of the example is that the popular probabilistic programming systems like STAN infer a wrong distribution for it, as demonstrated in [10]. Hakaru10, like the system of [10], infers the expected normal distribution with the center 0 and the standard deviation $\sqrt{91}$. We can verify the fact by computing the (estimate of the) Kullback-Leibler (KL) divergence, a common metric of dissimilarity between distributions:



```
phier_kl n = kl (tabulate 0.5 $ mcmC n phier)
                (tabulate 0.5 $ mcmC n $ dist normal 0 (sqrt 91))
```

Here tabulate computes the histogram with the specified bin size. The above figure plots phier_kl n for the different number of samples n. For the correct sampler, KL is expected to decay as $O(n^{-k})$ for some constant $k$. Hence, the plot of KL vs. n on the log-log scale should look like a straight line.

## 2.2 Branching Models

Hakaru10 can express not only 'straight-line' but also branching models. An example is a simple mixture model

```
mixng = do
  x ← dist normal 0 1
  if_ ((>0) <$> x)
         (dist normal 10 2)
         (dist gamma 3 (1/3))
```

whose distribution is the mixture of normal and gamma distributions, with the random variable x determining the proportion of the mixture. One can also read this program as sampling either from the normal or the gamma distributions, depending on the sign of x. The mixng program is the first betraying Model being not quite a monad. The variable x is not actually of the type Double, as one might have thought. We have maintained the illusion so far because numbers in Haskell are overloaded. Sadly, booleans are not. As should be apparent from the use of Applicative operator <$> , which is just fmap, Model a is something like M (A a) where M is a monad but A is an applicative. We discuss the representation of Model in §4.

Just by looking at the mixng code one can tell that what matters for the final distribution is the event of x being positive – which, for the standard normal distribution happens 50% of the time. Therefore, mixng should be equivalent to the following mixture

```
mixng' = do
  x ← dist bern 0.5
  if_ x
       (dist normal 10 2)
       (dist gamma 3 (1/3))
```

That is, mcmC 5000 mixng should be roughly the same sequence of samples as mcmC 5000 mixng' – and also as interleave (mcmC 2500 (dist normal 10 2)) (mcmC 2500 (dist gamma (1/3))). Again, we verify the similarity using the KL divergence, which is under 2.3e−2. This example is also borrowed from [10, Fig 2]. Despite its simplicity, several widely known and used probabilistic programming systems (e.g., STAN) infer wrong distributions for it.

## 2.3 Conditioning

Finally, Hakaru10 supports conditioning, that is, inferring the conditional distribution of a model where some of its variables have been observed to hold particular values. Conditioning is extraordinarily tricky, especially in case of continuous distributions. The interested reader may look up the Borel paradox. The syntax and the type system of Hakaru10 are specifically designed to steer the programmer (far) away from the pitfalls.

We take as an example the experiment of estimating the bias of a coin, that is, its inherent probability b of coming up as head (that is, True). We toss the coin twice and observe the results as c1 and c2. The following is the model of the experiment, taking the observed values as parameters.

```
biased_coin c1 c2 = do
  b ← dist beta 1 1
  dist (c1 `condition` bern) b
  dist (c2 `condition` bern) b
  return b
```

We do not know the true bias b, but assume a priori that it is distributed as beta 1 1. This is a popular assumption (not in small part due to the fact we can compute the posterior analytically). We toss the coin twice and 'observe the results as c1 and c2': specify that the first toss came in reality as c1 and the second as c2. Then biased_coin c1 c2 gives us the (posterior) distribution of b, letting us estimate the coin's bias. If in the experiment the coin came up first head and then tail, the posterior analytically is beta 2 2, with the average 0.5 and the variance 0.05. Running mcmC 10000 (biased_coin True False) gives the list of 10000 samples, from which we estimate the average as 0.499.

As biased_coin code demonstrates, in Hakaru10 conditioning may only be applied to distributions. One may think that (c1 `condition` bern) creates a new distribution out of bern, with the singular value c1. Conditioning on arbitrary boolean formulas is fraught with peril, both theoretical and practical, degenerating MCMC algorithm into the inefficient rejection sampling. Since repeated conditioning does not make sense, (c1 `condition` (c2 `condition` bern)) is a type error. Hakaru10 is indeed typed, although we have not paid much attention to types, which were all inferred. Types do prevent silly errors like

```
biased_coin_ill_typed  c1 c2 = do
 b ← dist bern 0.5
 dist (c1 `condition` bern) b
 dist (c2 `condition` bern) b
 return b
```

since the parameter of bern, the probability, cannot be a Boolean. Types also prevent less obvious errors like

```
biased_coin_ill_typed_too  c1 c2 = do
 b ← dist bern 0.5
 dist (b  `condition` bern) 0.4
 return b
```

Here, we attempted to condition bern on the random choice within the model. This is not allowed: observations must be external to the model. In [20], this semantic condition was a merely a coding convention, whose violation manifested as a run-time exception. In Hakaru10, the violation is a type error.

Previously we have seen that random variables that do not contribute to the result in any way are effectively marginalized. Conditioning changes that. The two condition lines in the biased_coin model are the random variables that do not seem to contribute to the model (therefore, we did not even give them names). However, they had effect. The following model makes that fact clear:

```
post_bias c = do
  coin ← dist bern 0.5
  if_ coin (dist (c `condition` normal) 0 1)
           (dist (c `condition` normal) 100 1)
  return coin
```

The result of the model does not overtly depend on the result of the if_ statement. However, it changes the coin's posterior distribution: running mcmC 100 (post_bias 1) gives all True samples.

We conclude the tutorial by looking back at the canonical grass model example described in §1, repeated below for reference.

```
grass = do
  rain          ← dist bern 0.3
  sprinkler     ← dist bern 0.5
  grass_is_wet  ← dist (True `condition` nor 0.9 0.8 0.1) rain  sprinkler
  return rain
```

This code hopefully has become more understandable. Evaluating mcmC 20000 grass and counting the number of True gives the posterior estimate of rain having observed that the grass is wet: it comes out to 0.468, which matches the analytically determined result.

## 3 Evaluation

The Hakaru10 tutorial might have given an impression that Hakaru10 tries so hard to preclude problematic behavior, by restricting conditioning and models, that one cannot do much interesting in it. In this section we briefly evaluate the expressiveness of the language on two realistic models.

*Bayesian Linear Regression* The first model comes from the small problems collection of challenge problems[3] assembled in the course of DARPA's Probabilistic Programming for Advancing Machine Learning (PPAML) program[4]. It is Problem 4.1, Bayesian linear regression: Given the set of training points $(x_{ij}, y_i), i = 1..N, j = 1..k$ and the generative model $y_i = \sum_j x_{ij} * w_j + noise_i$ find the posterior distribution on $w_j$. In the conventional linear regression language, we are given $N$ observations of $y$ assumed to be a linear combination of the controlled quantities $x$ with the parameters $w$; we have to estimate $w$. The generative model is expressed in Hakaru10 as

```
type RegrDatum = ([Double],Double)   -- Xs and Y
model :: [RegrDatum] → Model [Double]
model xsy = do
  let mu   =  replicate  dimK 0
  w_mean   ← normals mu 2
  w        ← normals w_mean 1
  noise_sd ← (1/) <$>  dist gamma 0.5 0.5
  let make_cond (xs,y) = dist  (y `condition` normal) (dot xs w) noise_sd
  mapM_ make_cond xsy
  return $  collect  w
```

Its argument is the list of the training points $(x_{ij}, y_i)$. The model starts by defining the prior for the parameters w and the standard deviation noise_sd for the noise (as specified in the problem description). We then create a random variable for each noisy observation point and condition it to $y_i$. We are interested in the distribution of the parameter vector $w$ given the conditioning. The Hakaru10 model rather closely matches the problem description (and the RSTAN code given in the problem description document). The model is straightforward but not small: its graph has 511 vertices (there are five hundred observations and five parameters).

The model is naturally expressed in terms of vectors; Hakaru10 however does not provide out of the box any distributions over vectors. Nevertheless, we can express them through Hakaru10 primitives in our host language, Haskell. For example, we can write normals, which produces a list of independently distributed normal random variables of the same standard deviation std whose means are given by the list means:

```
normals means std = mapM (\m → dist normal m std) means
```

Likewise we can write collect (to convert a list of random variables into a random list) in pure Haskell – to say nothing of the dot-product dot.

*Population Estimation* We also evaluate Hakaru10 by implementing a realistic model of population estimation, taken from [14, Ex 1.1]: "An urn contains an

[3] `http://ppaml.galois.com/wiki/wiki/CP4SmallProblemsCollection`

[4] `http://www.darpa.mil/program/probabilistic-programming-for-advancing-machine-Learning`

unknown number of balls – say, a number chosen from a Poisson or a uniform distributions. Balls are equally likely to be blue or green. We draw some balls from the urn, observing the color of each and replacing it. We cannot tell two identically colored balls apart; furthermore, observed colors are wrong with probability 0.2. How many balls are in the urn? Was the same ball drawn twice?" This example is hard to implement in many probabilistic programming languages. That is why it was used to motivate the language BLOG.

First we define ball colors

```
data Color = Blue | Green deriving (Eq, Show)
 opposite_color :: Color → Color
 opposite_color Blue  = Green
 opposite_color Green = Blue
```

and introduce the distribution for the observed ball color accounting for the observation error:

```
observed_color color = categorical [( color , 0.8), ( opposite_color color , 0.2)]
```

Although the exact number of balls in unknown, we can reasonably impose an upper bound. We create that many instances of uniformly color-distributed random variables, for each ball. We populate the IntMap data structure, mapping ball's index to the corresponding random variable, for easy retrieval.

```
maxBalls = 8
balls_prior n = do
  balls ← sequence ∘ replicate n $ dist uniformly (pure [Blue,Green])
  return $ M.fromList $ zip [1..] balls
```

Some of these random variables will be unused since the number of balls in the urn is often less than the upper bound. The unused variables will be marginalized[5].

The model is conceptually simple: it takes a list of observations obs as an argument, generates random variables for all possible balls, draws the number of balls from the prior and dispatches to the instance of the model with that number of balls.

```
cballs_model [obs1,obs2 ,...] = do
  balls  ← balls_prior maxBalls
  nballs ← dist uniformly (pure [1.. maxBalls])
  if_ ((== 1) <$> nballs) (cballs_model_with_Nballs balls obs 1) $
   if_ ((== 2) <$> nballs) (cballs_model_with_Nballs balls obs 2) $
   ...
   if_ ((== 8) <$> nballs) (cballs_model_with_Nballs balls obs 8) $
     return ()
  return nballs
```

When the number of balls is fixed, the experiment is easy to model: pick one ball b and check its true color balls ! b against the color of the first observed ball; repeat for the second observed ball, etc.

[5] Imposing the upper bound on the number of balls may be undesirable, especially for the Poisson distribution. In principle, Hakaru10 could instantiate conditional branches of a model lazily; in which case balls could be an infinite list. We are investigating this possibility.

```
cballs_model_with_Nballs  balls  [obs1,obs2 ,...]  nballs  = do
  b ← dist uniformly (pure [1.. nballs ])
   if_ ((== 1) <$> b) (dist (obs1 `condition` observed_color ) ( balls ! 1)) $
    if_ ((== 2) <$> b) (dist (obs1 `condition` observed_color ) ( balls ! 2)) $
          ...
  b ← dist uniformly (pure [1.. nballs ])
   if_ ((== 1) <$> b) (dist (obs2 `condition` observed_color ) ( balls ! 1)) $
    if_ ((== 2) <$> b) (dist (obs2 `condition` observed_color ) ( balls ! 2)) $
          ...
   ...
  return ()
```

The result is a rather large Bayesian network with deeply nested conditional branches with conditioning in the leaves. The fact that the same balls variable is shared among all branches of the complex if-statement corresponds to the intuition that the same ball can be drawn twice since we return the drawn balls into the urn. A ball keeps its true color, no matter how many times it is drawn.

The code outline just shown is not proper Hakaru10 (and is not proper Haskell) because of many ellipses. It is clear however that the code has the regular structure, which can be programmed in Haskell. For example, the (proper, this time, with ellipses filled in) code for cballs_model is as follows:

```
cballs_model :: [Color] → Model Int
cballs_model obs = do
  balls  ← balls_prior maxBalls
  nballs ← dist uniformly (pure [1.. maxBalls])
  let obs_number i = if_ ((== i) <$> nballs) $ cballs_model_with_Nballs balls obs i
  foldr obs_number (return (pure())) [1.. maxBalls]
  return nballs
```

One may think that the huge size of the model makes the inference difficult. However, only small part of the large nest of conditional branches is evaluated on each MH step. Therefore, performance is rather good: on 1.8GHz Intel Core i3, Hakaru10 running within the GHCi interpreter (bytecode, no optimizations) takes 16 seconds to do 10,000 samples and reproduce the results of this model programmed in Hansei [12] (running in OCaml bytecode, using importance sampling, taking 5000 samples within 13.4 seconds) and the results reported in [14, Fig. 1.7], which took 35 seconds on 3.2GHz Pentium 4 to obtain.

## 3.1 Performance

The motivation for Hakaru10 was to improve the performance of the original Hakaru [24], which was the straightforward implementation of the Wingate et al. [21] algorithm. The previous section has already touched upon the Hakaru10 performance. This section evaluates performance directly, against the original Hakaru, on the phier model from §2. Recall, its expected distribution is normal with the average 0 and variance 91. The table below reports the estimates of the average and variance, as well as the CPU time taken to obtain 1 million samples from the model. The table compares Hakaru10 with the original Hakaru. The platform is Intel Core i3 1.8GHz; the systems were compiled with GHC 7.8.3 with the −O2 flag.

| | Average | Variance | CPU time (sec) |
|---|---|---|---|
| Hakaru original | 0.39 | 87 | 72 sec |
| Hakaru10 | 0.22 | 93 | 20 sec |

Hakaru10 indeed significantly improves performance.

## 4 Implementation

Hakaru10 programs represent directed graphical models. Although we can use state and other effects (e.g., reading various parameters from a file) to build the graph, models themselves are declarative, describing connections between random variables, or their distributions. Having built the model, we want to determine the distribution of some of its random variables, either marginalizing or conditioning on the others. Usually we determine the desired distribution as a sequence of samples form it. If the model is encoded as a program that does the sampling of random variables respecting the dependencies, determining the distribution amounts to repeatedly running the program and collecting its results. Taken literally, this process is rather inefficient however.

Conditioning, especially in the case of continuous distributions, poses a problem. Consider the model

```
do
 tempr ← dist uniform (−20) 50
 (25 `condition` normal) tempr 0.1
 return tempr
```

which represents a simple measurement (of tempr, the air temperature). The measurement has random noise, which is believed to be Gaussian with the standard deviation 0.1. We are interested in the distribution of the true temperature given the observed value 25 (degrees Celsius). The naive procedure will uniformly sample tempr from [−20,50], then sample from normal tempr 0.1, and, if the latter result differs from 25, reject the tempr sample and repeat. Alas, we will be rejecting almost all samples and produce nothing: mathematically speaking, the event that a value drawn even from the normal distribution centered at 25, is exactly 25 has the zero probability. A more useful question therefore to ask is how likely 25 may come as a sample from the distribution normal tempr 0.1. It becomes clear why Hakaru10 insists the conditioning be applied only to distributions: we have to know what distribution the observation comes from, so we can tell how likely it is. Sampling thus becomes an optimization problem, maximizing the livelihood. One of the multi-dimensional optimization methods is Markov-Chain Monte-Carlo (MCMC).

Hakaru10 supports one of the MCMC methods: Metropolis-Hastings (MH) method of sampling from a model distribution. We remind the algorithm on the following example:

```
mhex = do
  x ← dist normal 0 1
  y ← dist normal x 2
  if_ ((>0.5) <$> x)
```

```
(return x)
(do {z ← dist beta 1 1; return (y + z)})
```

The algorithm constructs the sequence of Doubles, drawn from the distribution of mhex, that is, the distribution of the values returned by the last statement of the program. To start with, MH "runs" the program, sampling from the distributions of its random variables. For example, x gets a sample from the standard normal distribution, say, 0.1. Then y is sampled from normal 0.1 2, say, as −0.3, and z is sampled as 0.5. The result of the whole program is then 0.2. Along with the samples, MH remembers their probability in the distribution. It is more convenient to work with the logarithms, that is, log likelihoods (LL). For example, the LL of the initial x sample is −0.616. The collection of samples along with their LLs is called the trace of the program. LL of the trace is the sum of the LLs of its samples.

The just constructed trace becomes the initial element in the Markov chain. The next element is obtained by attempting to 'disturb' the current trace. This is the key to the efficiency of Markov Chain Monte Carlo (MCMC) as contrasted to the naive resampling (simple Monte Carlo): rather than resample all of the random variables, we attempt to resample only one/a few at a time. The algorithm picks a subset of random variables and proposes to change them, according to some proposal distribution. Commonly, and currently implemented in Hakaru10, the algorithm selects one random variable and resamples it from its distribution. Suppose we pick x and find another sample from its standard normal distribution. Suppose the result is 0.6. The program is then re-run, while keeping the values of the other random variables. In other words, we re-compute the trace to account for the new value of x. The change in x switches to the first branch of the **if** statement, and the program result becomes 0.6. Since y was not affected by the change proposal, its old sample, −0.3, is kept. However, it is now drawn from the different distribution, normal 0.6 2, an hence has the different LL. Thus even if a random variable does not contribute to the result of a trace, it may contribute to its LL. From the LLs of the original and the updated trace, MH computes the acceptance ratio (a number between 0 and 1) and accepts the updated trace with that probability. If the trace is accepted, it becomes the new element of the Markov chain. Otherwise, the original trace is retained as current. Running the trace re-computation many times constructs the sequence of samples from the distribution of the trace – or, retaining only the trace result, the sequence of samples from the program distribution.

## 4.1 Design Overview

We now describe the Hakaru10 implementation in more detail. Hakaru10 represents the trace – random variables of the model and their dependencies – as a directed acyclic graph (DAG). Each node (vertex) in the graph stands for one random variable (whose value can be sampled and resampled) or an observed variable, whose value cannot be resampled. There are also computational nodes, representing 'samples' from the dirac distribution. They cannot be resampled either. §4.3 describes the reasons for the special treatment of the dirac distribution. One node in a graph, with no outgoing vertices, is designated as the 'result' node. The graph with the result type a has the type SExp a[6]. For the grass model the trace graph has three nodes and looks exactly like the graphical representation of the model, Fig. 1.

A Hakaru10 model has the type

```
type Model a = MCMCM (SExp a)
```

It is a computation that constructs the trace graph. MCMCM is a monad, but SExp is not. It is an applicative [13]: it lets us construct new graph nodes, without looking at their values. The function

```
mcmC :: Integer → Model a → [a]
```

first builds the trace graph and then repeatedly runs the trace update algorithm the specified number of times. The fact that SExp is not a monad is significant: the current node values cannot influence the graph construction. Therefore, after the graph is built, its structure does not change. All dependencies among nodes can be computed once and for all.

The type system not only makes the implementation more efficient. It also enforces semantic constraints. Let's recall the code that attempts to condition on the value computed within the model, which is invalid semantically.

```
biased_coin_ill_typed_too   c1 c2 = do
 b ← dist bern 0.5
 dist (b  `condition`  bern) 0.4
 return b
```

This code does not type-check since the first argument of condition should be Bool, since bern is the distribution over booleans. However, b is not Bool, it is of the type SExp Bool. In the original Hakaru [24], the semantic constraint was enforced via a parameterized monad, which made its syntax (the Haskell embedding) cumbersome. Worse, the values to condition upon could only be referred to indirectly, via De Bruijn-like indices, which were very easy to confuse. The type system of the embedded language thus has significant influence on its 'syntax', its embedding.

## 4.2 Incremental Recomputation

One of the main features of Hakaru10 is its incremental recomputation algorithm, which avoids the redundant computations during the trace update. Only those nodes that (transitively) depend on the resampled random variable are recomputed. The following example should clarify the meaning of the dependency:

```
pdep = do
    x ← dist normal 0 1
    y ← dist normal x 1
    z ← dist normal y 1
    return (x+y+z)
```

[6] The actual implementation has one more level of indirection, but the description given here is a good approximation.

Suppose MH proposes to resample x (and only x). Although the y sample keeps its old value, its distribution parameters, x specifically, changed. Therefore, the LL of the old y sample in the new distribution has to be recomputed. The variable z is not affected by the resampling proposal; also, the parameters of its distribution remain the same. Therefore, no update to the z node is needed. The last, Dirac, node of the trace, corresponding to return (x+y+z), is also updated, to account for the new x: the special treatment of Dirac nodes is explained in the next subsection.

Since the type system ensures that the structure of the graph is preserved, the dependency graph can be computed and topologically sorted once and for all. The fact that Hakaru10 models are acyclic and Hakaru10 programs are declarative (with no mutations) lets us avoid the topological sort and rely on the 'creation times' of trace nodes. A node can only depend on those constructed before it.

The update procedure has the obvious correctness requirement: a node is updated only after all the nodes it depends on have been updated. If we consider the trace update as a graph traversal, the correctness property amounts to maintaining the invariant that a visited node has the creation time earlier than any other node in the update queue. This invariant has guided writing the code and remains in the code in the form of assertions (mostly for documentation).

## 4.3 Special Treatment of the Dirac Distribution

Hakaru10 by design enforces the laws that

```
do { x ← p; diracN x }    ≡ p
do { x ← diracN v; p }    ≡ let x = v in p
```

for all programs p.

The law is tricky to enforce. Moreover, an MH implementation that does not pay attention to it (such as [21], for example) exhibits incorrect behavior, as pointed out in [11]. We recall the latter's argument here.

Let us consider the following program

```
p2 = do {x ← dist uniform 0 1; diracN x}
```

Suppose in the initial trace x is sampled to 0.5 and the MH algorithm now proposes to change it to 0.7. When updating the trace, the values of other random variables are kept as their are; only their LL may change. Thus the value of the dirac node will be kept at 0.5; the update procedure will then try to find its LL within the changed distribution dirac 0.7. Clearly the LL of the old sample in the new distribution is $-\inf$. Therefore, the proposal to resample x will be rejected. *Every* proposal to modify x will likewise be rejected and so the Markov chain of p2 will contain the identical samples.

Mathematically, composing with Dirac is the identity transformation, so p2 should be equivalent to just uniform 0 1, whose Markov chain is anything but constant. Without taking precautions, the MH algorithm converges to the wrong distribution for p2.

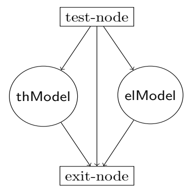
One may be tempted to dismiss the problem: the chain fails to mix (all proposals are rejected) because of the single-variable update proposals. However, more general proposals require the interface for the user to tell the system how to make correlated multi-variable proposals. Moreover, the user has to know how to make good a proposal, which is a non-trivial skill. Asking the end user for non-trivial extra hints seems especially bothersome for such a simple problem. Once we know which equational laws we have to satisfy, it is quite easy to account for it and make the problems involving dirac go away.

Therefore, Hakaru10 implementation treats dirac nodes specially. They are considered as pure computation nodes, and their value is always updated whenever their dependencies change. In effect, any proposal to change one random variable is automatically extended to the proposal to change all dependent dirac random variables, in a way that the latters' LL stays zero. Hakaru10 thus employs multi-variable correlated proposals, for all dirac variables. Therefore, Hakaru10 satisfies the Dirac laws by construction.

### 4.4 Branching

Branching models, with if-expressions, bring in quite a bit of complexity. A change in the branch condition effectively causes one part of the model vanish and a new submodel, from the other branch, to appear. Maintaining node dependencies and correctly computing the acceptance ratio in such a dynamic environment is non-trivial.

Hakaru10 avoids any modifications to the graph structure during the trace update. It compiles both if-branches when constructing the initial graph. That is, the model if_ test thModel elModel corresponds to the following DAG:



The entry node corresponds to the test condition. The exit node is the result node of the entire if-statement; it holds the value of the thModel or the elModel, depending on the value of the test-node.

Updating the nodes in the inactive branch, whose values will be ultimately ignored, is not good for performance. Therefore, we mark the nodes in the non-current branch as inactive, and delay their updates until they are activated. Thus each node, along with its current value, LL and the distribution also keeps the inactivation count. (The inactivity mark is not a simple boolean because of the nested conditionals and because the same node may appear in several conditionals).

## 5 Related Work

There are many probabilistic programming languages, with a variety of implementations [7, 16]. Closely related to Hakaru10 in its design is Figaro [18], which is also an embedded DSL, in Scala. Like Hakaru10, a Figaro program produces a trace graph, which is then repeatedly updated by the MH algorithm. Figaro does not appear to use the incremental evaluation. Infer.net [15] is also an embedded DSL, for the .NET platform, that first constructs a graph; instead of MH it relies mostly on Expectation Propagation and its variants for inference.

The system of Ścibior et al. [20] is related in its use of Haskell and MCMC. The similarities end here, however. Ścibior et al. use a monad to express models. Therefore, the model construction is "too dynamic" and the semantic constraints on conditioning cannot be statically enforced. On the other hand, Ścibior et al. can express cyclic models (whose semantics, however, may be difficult to determine.) Ścibior et al. implement different MCMC algorithms; none of the implementations are incremental at present.

Although the MH algorithm is the old staple – the original MCMC algorithm – it is not the only one. More advanced MCMC algorithms have been proposed, and recently have been incorporated into probabilistic programming, most prominently in Anglican [22]. It is an interesting challenge to turn these algorithms incremental and implement within Hakaru10, while preserving Hakaru10's features such as conditioning within conditional branches.

Yang et al. [23] propose an incremental evaluation algorithm, using in effect staging, or program generation. They analyze the initial trace and then generate code for the efficient MH update.

Hur et al. [10], whose paper we often used for references and examples, propose the provably correct MH algorithm for the imperative, C-like probabilistic language. It has to deal with the problem of multiple assignments to random variables. The problem does not exist in the declarative Hakaru10, where a random variable, as the name of a model, is immutable.

## 6 Conclusions and Future Work

We have presented the probabilistic programming language Hakaru10 embedded as a DSL in Haskell. The language features the type system to prevent silly and more subtle mistakes with probabilistic conditioning. We have described the incremental MH evaluation of Hakaru10 programs.

The immediate future work is using the language for more, interesting models. We should consider adding Dirichlet processes. An interesting design challenge is the interface to let the user specify proposals, including proposals to change several random variables in concert.

Although the minimalism of Hakaru10 simplifies the implementation and checking of its correctness, it makes writing interesting models, such as those in §3, cumbersome. Hakaru10 may hence be viewed as an intermediate language. Fortunately, Haskell proved quite powerful 'macro' language to improve convenience.

*Acknowledgments* I am indebted Rob Zinkov and Chung-chieh Shan for many helpful discussions. Comments and suggestions by anonymous reviewers are gratefully acknowledged. The work on Hakaru10 was supported by DARPA grant FA8750-14-2-0007.

## References

[1] *AISTATS 2014*, number 33, Cambridge, 2014. MIT Press.

[2] Luc De Raedt, Angelika Kimmig, and Hannu Toivonen. ProbLog: A probabilistic Prolog and its application in link discovery. In Manuela M. Veloso, editor, *Proceedings of the 20th International Joint Conference on Artificial Intelligence*, 6–12 January 2007.

[3] Martin Erwig and Steve Kollmansberger. Probabilistic functional programming in Haskell. *Journal of Functional Programming*, 16(1):21–34, January 2006.

[4] Lise Getoor and Ben Taskar, editors. *Introduction to Statistical Relational Learning*. MIT Press, Cambridge, November 2007.

[5] Noah D. Goodman. The principles and practice of probabilistic programming. In *POPL '13: Conference Record of the Annual ACM Symposium on Principles of Programming Languages*, pages 399–402, New York, January 2013. ACM Press.

[6] Noah D. Goodman, Vikash K. Mansinghka, Daniel Roy, Keith Bonawitz, and Joshua B. Tenenbaum. Church: A language for generative models. In David Allen McAllester and Petri Myllymäki, editors, *Proceedings of the 24th Conference on Uncertainty in Artificial Intelligence*, pages 220–229, Corvallis, Oregon, 9–12 July 2008. AUAI Press.

[7] Noah D. Goodman and Andreas Stuhlmüller. The design and implementation of probabilistic programming languages. `http://dippl.org`, 2014.

[8] Andrew D. Gordon, Thomas A. Henzinger, Aditya V. Nori, and Sriram K. Rajamani. Probabilistic programming. In *FOSE*, pages 167–181. ACM, 2014.

[9] Matthew D. Hoffman and Andrew Gelman. The No-U-Turn Sampler: Adaptively setting path lengths in Hamiltonian Monte Carlo. e-Print 1111.4246, arXiv.org, 2011.

[10] Chung-Kil Hur, Aditya Nori, Sriram Rajamani, and Selva Samuel. A provably correct sampler for probabilistic programs. In *FSTTCS 2015*, 2015.

[11] Oleg Kiselyov. Problems of the lightweight implementation of probabilistic programming. In *Proc. Workshop on probabilistic programming semantics*, 2016.

[12] Oleg Kiselyov and Chung-chieh Shan. Monolingual probabilistic programming using generalized coroutines. In *Proceedings of the 25th Conference on Uncertainty in Artificial Intelligence*, pages 285–292, Corvallis, Oregon, 19–21 June 2009. AUAI Press.

[13] Conor McBride and Ross Paterson. Applicative programming with effects. *Journal of Functional Programming*, 18(1):1–13, January 2008.

[14] Brian Milch, Bhaskara Marthi, Stuart Russell, David Sontag, Daniel L. Ong, and Andrey Kolobov. BLOG: Probabilistic models with unknown objects. In Getoor and Taskar [4], chapter 13, pages 373–398.

[15] Tom Minka, John M. Winn, John P. Guiver, and Anitha Kannan. Infer.NET 2.2. Microsoft Research Cambridge. `http://research.microsoft.com/infernet`, 2009.

[16] Kevin Murphy. Software for graphical models: A review. *International Society for Bayesian Analysis Bulletin*, 14(4):13–15, December 2007.

[17] Judea Pearl. *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*. Morgan Kaufmann, San Francisco, CA, 1988. Revised 2nd printing, 1998.

[18] Avi Pfeffer. Figaro: An object-oriented probabilistic programming language. Technical Report 137, Charles River Analytics, 2009.

[19] Taisuke Sato. A glimpse of symbolic-statistical modeling by PRISM. *Journal of Intelligent Information Systems*, 31(2):161–176, October 2008.

[20] Adam Ścibior, Zoubin Ghahramani, and Andrew D. Gordon. Practical probabilistic programming with monads. In *Proceedings of the 8th ACM SIGPLAN Symposium on Haskell*, pages 165–176, New York, 2015. ACM Press.

[21] David Wingate, Andreas Stuhlmüller, and Noah D. Goodman. Lightweight implementations of probabilistic programming languages via transformational compilation. In *AISTATS 2011*, number 15, pages 770–778, Cambridge, 2011. MIT Press. Revision 3. February 8, 2014.

[22] Frank Wood, Jan Willem van de Meent, and Vikash Mansinghka. A new approach to probabilistic programming inference. In *AISTATS 2014* [1], pages 1024–1032.

[23] Lingfeng Yang, Pat Hanrahan, and Noah D. Goodman. Generating efficient MCMC kernels from probabilistic programs. In *AISTATS 2014* [1], pages 1068–1076.

[24] Rob Zinkov and Chung-chieh Shan. Probabilistic programming language Hakaru. v1. DARPA PPAML Report, 2014.