Problems of the Lightweight Implementation of Probabilistic Programming

Oleg Kiselyov
Tohoku University, Japan
oleg@okmij.org

Abstract

We identify two problems and an open research question with Wingate et al. lightweight implementation technique for probabilistic programming. Simple examples demonstrate that common, what should be semantic-preserving program transformations drastically alter the program behavior. We briefly describe an alternative technique that does respect program refactoring. There remains a question of how to be really, formally sure that the MCMC acceptance ratio is computed correctly, especially for models with conditioning and branching.

1. Summary

Recently Wingate et al. [3] proposed a general method of turning any programming language into a probabilistic programming system with an MCMC inference. The key idea is the naming of each random choice made by a program, and replacing actual nondeterministic choices with the deterministic lookup in a database of choices: the table of random numbers, so to speak.

Unfortunately, the technique was developed without considering equational laws, which define valid optimizations and program transformations. Therefore, optimizing and refactoring a program, moving some pieces of code to separate functions and libraries, or inlining – all may change the program behavior in startling and drastic ways.

This paper demonstrates on simple examples the problems of [3], which we have verified on its implementation [5]. We briefly mention an alternative technique, which implements incremental MCMC and which does validate the expected equational laws and program optimizations (see §6).

We advocate designing algorithms that are mindful and respectful of program transformations and equational laws. We also wish to prove that the key parts of our algorithm, specifically, computing the MCMC acceptance ratio, are correct. We would like to pose a question how to go about such a proof, especially in the case of conditioning within conditional branches (see §5.)

2. Lightweight Implementation of Probabilistic Programming

This section recalls the gist of the Wingate et al. technique [3]. We will be using the Haskell notation, which is essentially the notation of [2].

A probabilistic program expresses a stochastic model. For example, the program:

```
p1 = do {x ← uniform 0 1; bern x}
```

describes the composition of the uniform and Bernoulli distributions. One may also read it operationally: `uniform 0 1 :: Prob Float` produces a floating-point number `x` uniformly chosen within the interval [0,1]; `bern x` then returns `True` or `False` with the probability `x`. Here `Prob` is some probabilistic monad.

The insight of Wingate et al. is to think of the stochastic program `p1` as a *deterministic* program

```
p1Det = do let n1 = 1
           x ← lookup n1 uniform [0,1]
           let  n2 = 2
           lookup n2 bern x
```

where each stochastic primitive (elementary random primitives, or ERPs, in [3] terminology) is implemented as a lookup in the global database `D`. Then `Prob` is just the state monad `State D`. (Likewise, a pseudo-random generator may be thought of as a lookup in the table of random numbers). The variables `n1` and `n2` in `p1Det` are the *names*, or lookup keys, identifying each instance of the stochastic operation. The lookup operation uses the name to find and return the corresponding sample from the database. If the database has no record for the name `n`, it is created by sampling from the ERP's distribution and recording the sample, the sampling parameters such as the range [0,1] for `uniform`, and the sample's log likelihood (LL). Re-running `p1Det` with the same `D` produces the same results.

The second key idea of Wingate et al. is to use MCMC to evolve the values recorded in the database. For example, for the (single-site) Metropolis-Hastings (MH) MCMC algorithm, we pick up one recorded choice, modify it, re-run the program computing the new result and its LL, and accept or reject the modification. Repeating the process many times effectively turns the deterministic `p1Det` into the stochastic `p1`: the sequence of results produced by re-running `p1Det` is the sequence of samples from `p1`'s distribution.

When MH picks and modifies the record for `n1` in `p1Det` (in other words, resamples `x` from `xold` to `xnew`), the rest of the database stays the same. Therefore, `lookup n2` returns the same value it did on the previous run of the program. However, that old value is now drawn from the different distribution, `bern xnew`, and hence has the different LL. The difference in LL determines whether to accept or reject the proposal to resample `x`.

3. Unit Transformation

We now demonstrate the first problem with the technique. It may seem trivial and easy to explain (away). It is a simple illustration of more serious things to come.

Let us consider the program `p2`, quite similar to `p1`

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

If we take probabilistic programs to represent (extended) graphical models, then the trivial program `return e :: Prob t` corresponds to the model with the degenerate discrete random variable, that is, Dirac delta. The original Hakaru implementation [5] indeed represented `return a` in this way. Although `dirac` and `return` are synonymous, we will write `dirac` to highlight that it is also an ERP.

Suppose on the first run x is sampled to 0.5. The program will return 0.5 with the database

```
[(1,  {sample= 0.5, distr= uniform, parm= [0,1], ll= −0.5}),
 (2,  {sample= 0.5, distr= dirac,  parm = 0.5,  ll = 0})]
```

Assume that on the second run, MH proposes to resample x to 0.7. It modifies the first record in the database, to have the different sample and ll, and reruns the program. Now, uniform 0 1 returns 0.7, as recorded in the new database. However, dirac 0.7 still returns 0.5 since its database record stays the same. It comes, however, from the different distribution, dirac 0.7. Clearly, dirac 0.7 can never yield 0.5 and hence 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 MCMC chain of p2 repeats one value over and over.

Mathematically, composing with the Dirac distribution is the identity, so p2 should be equivalent to just uniform 0 1, whose Markov chain is anything but constant. The technique of [3] thus fails the Dirac composition law.

One may be tempted to dismiss the problem: the chain fails to mix (that is, all proposals are rejected) because the original [3] algorithm was simplistic: it considered only proposals that update only one record in the database. If we entertain more general proposals, of modifying several records in the database in a correlated way, the problem disappears.

However, more general proposals require the interface for the user to tell the system how to make correlated multi-record proposals. Moreover, the end user has to know how to make a good proposal, which is a non-trivial skill. Pestering the end user for non-trivial hints is bothersome for such a simple problem. Once we know which equational laws we have to satisfy, it is quite easy to account for them and make the problems involving dirac to go away. Hakaru10 implementation described in §6 does exactly that. It satisfies the law of composing with Dirac (that is, the unit law of the Prob monad) by construction.

4. Conditional Branches

The more serious problem with [3], which infects the key technical contribution of that paper, shows up in models with conditional branches. Consider

```
p3 = do
  c ← bern 0.5
  if  c then uniform 0 1
      else  uniform 10 20
  return  c
```

The Markov chain of that program turns out to be the constant stream, of all True or all False. This is surprising given that the program merely returns the sample from bern 0.5, the coin flip.

To see the problem we show the corresponding transformed, deterministic program, with the name assignment.

```
p3Det = do
  let  n1 = 1
  c ← lookup n1 bern 0.5
  let  n2 = 2
  if  c then lookup n2 uniform [0,1]
      else  lookup n2 uniform [10,20]
  return  c
```

The two uniform choices in the branches of **if** receive the same name. This is the key, intentional property of both imperative and functional name assignment algorithms of [3] (Fig. 2 and 3 in the paper, resp.)

Suppose on the first run x is sampled at True. The program will return True and its database will contain

```
[(1,  {sample= True, distr= bern, parm= 0.5, ll= −0.5}),
 (2,  {sample= 0.5, distr= uniform, parm= [0,1], ll= −0.5})]
```

Assume MH proposes to resample x to False and changes the database accordingly. When we re-run the program on the updated database, lookup n2 uniform [10,20] finds the existing record for node 2 and verifies it corresponds to the uniform distribution. Thus the lookup operation returns the old sample, 0.5, but rescores it to the new parameters, uniform [10,20]. Clearly that old sample has the zero probability of coming from the interval [10,20] and hence the new LL will be $-\inf$ and the proposal is rejected. The chain will have only True samples.

The addition of an (irrelevant!) dirac cures the problem

```
p4 = do c ← bern 0.5
        if  c then do {dirac  1;  uniform  0  1}
             else  uniform 10 20
        return  c
```

The Markov chain of this program has the uniform mixture of True and False, as expected of bern 0.5. Such a drastic change in behavior upon the addition of an irrelevant code is the consequence of the fact that refactoring, however innocuous, may change the name assignment. Wingate et al algorithm is very sensitive to the choice of node names: it tries, for the sake of performance, to explore sharing as much as possible, even if the sharing is accidental and unjustified.

Thus the technique of [3] is sensitive to what should be irrelevant details such as code layout and code organization. Simple refactoring could drastically change the program behavior.

In the presence of conditional branches, the program may make different sequences of choices across different runs. Wingate et al. hence introduce the concepts of "fresh" and "stale" database records to describe the newly introduced choices and those now hidden in inactive conditional branches. The acceptance ratio calculation takes into account only the difference in the number of fresh vs. stale choices. This is correct, albeit given in [3] with no explanation. The likelihoods of fresh and stale records are ignored: if we look carefully at the acceptance ratio formula [3, p.3], we see that the "fresh randomness" occurs explicitly in the denominator – and also in the numerator of the ratio, as part of the likelihood $p(x')$ of the new program run. The same holds for the "stale randomness". This is again correct (and again, given with no explanation) – but only until conditioning enters the scene.

5. Branching and Conditioning

We come to the thorny, still open problem of conditioning within conditional branches. This issue per se is not the problem of Wingate et al. [3] since that paper, although mentioning conditioning in text, does not give an algorithm that includes conditioning. When using the Wingate et al. method, the implementors have to decide for themselves how to handle conditioning. The most 'natural' way is to follow the hints in the Wingate et al. text and treat random variables whose values have been observed quite like the ordinary ERP – which, however, cannot be resampled. That was the approach implemented in the original Hakaru [5]. We now show that it is problematic.

Consider the following program

```
pcond = do
  x ← categorical  [(1,0.5),  (2,0.5)]
  if  x== 1 then return x
          else  do
                (True `conditioned` bern)  0.5
                categorical  [(20,0.5),  (21,0.5)]
```

In the sampling semantics, we draw 1 or 2 with equal probability. If 1 is drawn, it is returned. Otherwise, we flip a coin and assert that it lands head. Then we draw 20 or 21 with equal probability and return the result. Although this model may be regarded as exotic, it is just a detailed, re-written version of the more natural, easier to understand model:

```
pcond' = do
  x      ← categorical  [(1,0.5),  (2,0.5)]
```

```
(y,z) ← if  x== 1
            then return  (True,x)
             else  do
                 y ← bern 0.5
                 z ← categorical   [(20,0.5),   (21,0.5)]
                 return  (y,z)
observe  (y == True)
return  z
```

That is, we draw x, sample from the joint distribution (y,z) and return z conditioned on y being the fixed value True. The joint distribution (y,z) is clearly (True,1) with the probability $\frac{1}{2}$, and (True,20), (True,21), (False,20) and (False,21) – each with the probability $\frac{1}{8}$. Conditioning on the first component of the pair being True gives the distribution of the second component as 1 with the probability $\frac{2}{3}$, and 20 and 21 with the probability $\frac{1}{6}$.

Implementing the pcond model as the program in the original Hakaru

```
pcond1 = do
  x ← categorical   [(1,0.5),   (2,0.5)]
  if  x== 1 then do
     (True `conditioned` dirac ) True   -- Why we need this?
     return  x
  else  do
     dirac  True                        -- Why we need this?
     (True `conditioned` bern) 0.5
     categorical   [(20,0.5),   (21,0.5)]
```

and obtaining 100,000 samples gives the estimate of the model distribution as [(1,0.57),(20,0.21),(21,0.22)], which differs from the expected. It is an open research problem how to perform MH for this model.

The reader may be wondering about the superfluous statements in pcond1 such as dirac True. The reader is encouraged to guess what happens if we remove them. (For a hint, see §4.)

6. Incremental Hakaru

Hakaru v10 (hereafter, Hakaru10) [2] is a probabilistic programming language embedded in Haskell. It lets the user specify a variety of models using discrete or continuous distributions and conditioning. The models may contain branches ('if'-statements). It is the complete re-write of [5]. Like the original Hakaru and Church, Hakaru10 relies on MH for inference.

The main feature of Hakaru10 is the incrementality of the inference algorithm: upon resampling, only those computations are re-done that (transitively) depend on the resampled value.

To this end, the incremental MCMC maintains the DAG of dependencies and stores the intermediate results so they do not have to be recomputed. Despite this, the incremental MCMC seems to use less memory compared to the original Hakaru. The DAG of dependencies is static: it is produced at the first program run and not modified afterwards, even if the model has conditional branches.

Hakaru10 guarantees by construction that the following LHS and RHS models have the identical behavior (the identical sequences of samples):

```
do x ← dirac  c          do let  x = c
   e                        e

do x ← e
   dirac  x              e
```

(where e is an arbitrary submodel and c is an arbitrary constant.) Hakaru10 ensures that inlining and replacing a submodel with a primitive preserve the program behavior (chains). Also unlike [3], it supports sharing of submodels. Hakaru10 therefore uses its own formula for computing the acceptance ratio, derived from the common sense considerations and verified experimentally (on test models whose behavior is known analytically). We would like to be able to derive the acceptance ratio formally or to rigorously prove its correctness. We pose this as an open problem and are looking for suggestions and examples.

In related work, [1] demonstrate the problem of the algorithm in [3] on different examples, exhibiting convergence to an incorrect distribution. The paper [1] as well as [4] propose different sampling algorithms to avoid the problems.

Acknowledgments

I thank Rob Zinkov and Chung-chieh Shan for many discussions. Comments and suggestions by Daniel E. Huang and anonymous reviewers are gratefully acknowledged. The work on Hakaru10 was supported by DARPA grant FA8750-14-2-0007.

References

[1] C.-K. Hur, A. Nori, S. Rajamani, and S. Samuel. A provably correct sampler for probabilistic programs. In *FSTTCS 2015*, 2015.

[2] O. Kiselyov. Embedded probabilistic programming language with the incremental MCMC. Proc. Workshop on Programming and Programming Languages (PPL), Mar. 2016.

[3] D. Wingate, A. Stuhlmüller, and N. 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.

[4] F. Wood, J. W. van de Meent, and V. Mansinghka. A new approach to probabilistic programming inference. In *AISTATS 2014*, number 33, pages 1024–1032, Cambridge, 2014. MIT Press.

[5] R. Zinkov and C.-c. Shan. Probabilistic programming language Hakaru. v1. DARPA PPAML Report, 2014.