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, which implements incremental 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 does validate the expected functional laws and program optimizations (see §6).

We advocate designing algorithms that are mindful and respectful of program transformations and functional 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:

\[
p1 = \text{do } \{ x \leftarrow \text{uniform } [0,1] ; \text{ bern } x \}
\]

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

The insight of Wingate et al. is to think of the stochastic program \( p1 \) as a deterministic program

\[
p1\text{Det} = \text{do let } n1 = 1
x \leftarrow \text{lookup } n1 \text{ uniform } [0,1]
\text{let } n2 = \text{lookup } n2 \text{ bern } x
\]

where each stochastic primitive (elementary random primitives, or ERPs, in [3] terminology) is implemented as a lookup in the global database \( D \). Then \( \text{Prob} \) just the state monad \( \text{State} D \). (Likewise, a pseudo-random generator may be thought of as a lookup in the table of random numbers.) The variables \( n1 \) and \( n2 \) in \( p1\text{Det} \) 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 \( p1\text{Det} \) 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 \( p1\text{Det} \) into the stochastic \( p1 \): the sequence of results produced by re-running \( p1\text{Det} \) is the sequence of samples from \( p1 \)’s distribution.

When MH picks and modifies the record for \( n1 \) in \( p1\text{Det} \) (in other words, resamples \( x \) from \( \text{from } x \) to \( \text{new} \)), the rest of the database stays the same. Therefore, \( \text{lookup } n2 \) returns the same value it did on the previous run of the program. However, that old value is now drawn from the different distribution, \( \text{bern } x \) \( \text{new} \), 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 \( p1 \)

\[
p2 = \text{do } \{ x \leftarrow \text{uniform } [0,1] ; \text{ dirac } x \}
\]

If we take probabilistic programs to represent (extended) graphical models, then the trivial program \( \text{return } e \cdot \text{Prob } t \) 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 \( \text{dirac} \) and return are synonymous, we will write \( \text{dirac} \) 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

\[
\begin{align*}
(1, \text{sample}=0.5, \text{distr}=\text{uniform}, \text{parm}=[0.1], \text{ll}=-0.5), \\
(2, \text{sample}=0.5, \text{distr}=\text{dirac}, \text{parm}=0.5, \text{ll}=0)
\end{align*}
\]

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 \( \text{ll} \), and re-runs the program. Now, \( \text{uniform} 0 1 \) returns 0.7, as recorded in the new database. However, \( \text{dirac} 0.7 \) still returns 0.5 since its database record stays the same. It comes, however, from the different distribution, \( \text{dirac} 0.7 \). Clearly, \( \text{dirac} 0.7 \) can never yield 0.5 and hence the \( \text{LL} \) 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 \( \text{uniform} 0 1 \), 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 what 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. Persuading 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 non-trivial hints is bothersome for such a simple problem. When using the Wingate et al. method, the implementors have to do it 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].

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 by 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

\[
p_{\text{cond}} = \begin{cases} 
\text{do} & c \sim \text{bern} 0.5 \\
\text{if} \ c \text{ then } \text{do} & \{ \text{dirac} 1; \text{uniform} 0 1 \} \\
\text{else } \text{uniform} 10 20 & \text{return} \ c
\end{cases}
\]

The Markov chain of this program has the uniform mixture of \( \text{True} \) and \( \text{False} \), as expected of \( \text{bern} 0.5 \). 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.
True gives the distribution of the second component as $z \sim \text{dirac } (0,0.5)$, $z \sim \text{categorical } [(20,0.5), (21,0.5)]$.

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 $(\text{True},1)$ with the probability $\frac{1}{2}$, and $(\text{True},20)$, $(\text{True},21)$, $(\text{False},20)$ and $(\text{False},21)$ – each with the probability $\frac{1}{4}$. Conditioning on the first component of the pair being True gives the distribution of the second component as 1 with the probability $\frac{1}{2}$, and 20 and 21 with the probability $\frac{1}{4}$.

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
        do let e = bern 0.5
            c ← categorical [(20,0.5), (21,0.5)]
            do (y, z) ← (bern 0.5, bern 0.5)
                observe (y == True)
                return z
    end
end
```

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 (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.


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