Automatic Discovery of Static Structures in Probabilistic Programs

Daniel Lundén[*], David Broman[*], Fredrik Ronquist[†], Lawrence M. Murray[‡]
[*]KTH Royal Institute of Technology, Sweden
[†]Swedish Museum of Natural History, Sweden
[‡]Uppsala University, Sweden

1 Introduction

Probabilistic programming is an approach to probabilistic modeling where one motivation is greater *expressive power* than traditional graphical models. This increase in expressive power comes from two properties of probabilistic programming: *stochastic branching* and *recursion.* There has been a considerable amount of work on *inference* algorithms for graphical models, but most of this work is not directly transferable to probabilistic programs. By now, there are nevertheless quite a few approaches to performing inference in probabilistic programs [1, 2], many being extensions of their graphical model inference counterparts.

Naturally, many of these algorithms do not perform well on all probabilistic programs. One example is the family of inference algorithms known as *sequential Monte Carlo* (SMC) methods. In general, SMC operates by performing parallel forward sampling on a model, where each sample is known as a particle. In graphical models, this system of particles is *resampled* according to particle likelihoods whenever an *observed* variable is encountered during forward sampling. This means that less likely samples under the observation are thrown out and replaced by more likely samples. In probabilistic programming languages such as Anglican [2] and WebPPL [1], the approach for SMC is to adapt the same algorithm as for graphical models—resample whenever an observed[1] random variable is encountered. The problem with SMC in probabilistic programs is that not all particles might align at the same resampling point simultaneously. This can be problematic, as is shown in Figure 1. Here, approximately half of all particles will align at the term `weight(-1000)` (line 2), and half at `weight(-1)` (line 5) . Since `weight(-1)` has a much higher weight, all the `weight(-1000)` particles will be thrown out. As we can see however, both branches are, in the end, equally weighted because of the `weight(999)` term (line 3) in the first branch. We should therefore have an approximately equal amount of samples from both branches. This shows that the resampling step in the algorithm can be detrimental to performance, and perhaps also incorrect (this requires further investigation).

```
1   if flip() then {
2     weight(-1000)
3     weight(999)
4   } else
5     weight(-1)
```

Figure 1. A toy example illustrating when resampling can be problematic. Written in our own functional, higher-order, probabilistic programming language (under development). The function `flip` represents a coin flip.

To solve this problem, we propose a new approach for SMC inference in probabilistic programming. Our contributions are: (1) A static analysis algorithm for discovering static structures in probabilistic programs. An illustrative example of this is given in Section 2, and an outline of the formalization of the analysis is given in Section 3. (2) An application of the algorithm to automatically select correct resampling locations for SMC inference in probabilistic programming. This is illustrated in Section 2, and briefly mentioned in Section 3 relating it to the formalization.

It is also possible that the static structures can be used by other types of inference algorithms. This is left for future work.

2 Discovery of static structures: an illustrative example

Consider the probabilistic program[2] in Figure 2. The `sim` function (lines 1–7) has both a stochastic branch (lines 3–6) and recursion (line 5). Because of this, the `sim` function can produce a random number of random variables, each of which is given by `t` (line 2). This corresponds not to a single, but many different possible graphical models. Hence, this part of the model *cannot* be expressed with a graphical model. The static structure given by statically analyzing the program is shown in Figure 2. We can always perform such an analysis on any probabilistic program—a trivial solution to any program is a static structure with a single node, encapsulating the entire program.

[1]The programming language constructs available for this differs between languages. In Anglican, an `observe` construct is used. WebPPL uses a `factor` construct. In our language (Figures 1 and 2) we use a `weight` construct, similar to `factor`. This construct simply adds a weight to the sample.

This project is financially supported by the Swedish Foundation for Strategic Research (ASSEMBLE RIT15-0012).

[2]This small example model is a highly simplified version of a phylogenetic birth-death model.

```
1   function sim(stop, lambda) {
2     t ~ exponential(lambda)
3     if t <= stop then {
4       weight(2.0)
5       sim(stop-t, lambda+0.1)
6     } else t
7   }
8
9   function model() {
10    lambda ~ gamma(1.0, 1.0)
11    stop ~ gamma(10.0, 3.0)
12    sim = sim(stop, lambda)
13    weight(sim+lambda)
14    lambda
15  }
```

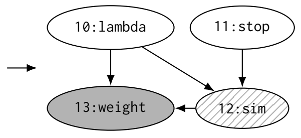

Figure 2. Example of static structure discovery. The program on the left is written in the same language as in Figure 1. On the right is the discovered static structure (note the similarities to a graphical model). The shaded node is a common notation for observed random variables. The striped node represents the part of the program that cannot be decomposed further due to stochastic branching and recursion.

$$\mathbf{t} ::= \texttt{x} \mid \texttt{c} \mid \lambda\texttt{x}.\mathbf{t} \mid \mathbf{t}\ \mathbf{t} \mid \texttt{if}\ \mathbf{t}\ \texttt{then}\ \mathbf{t}\ \texttt{else}\ \mathbf{t} \mid \texttt{sample}$$

$\texttt{x} \in \mathbb{X}$ (Variable names)

$\texttt{c}$ is a constant

Figure 3. The minimal target language for our algorithm.

Given the discovered static structure in Figure 2, it is possible to apply SMC to the program by only resampling at calls to `weight` *outside of* any striped nodes. Doing this ensures that, when resampling, all particles are aligned at the same `weight`. In Figure 2, this means that resampling will only be performed once, at line 13. The other calls to weight (line 4) are located within the `sim` node, and here no resampling occurs. Instead, the weight of the sample simply accumulates. This method for selecting resampling locations can be generalized to all higher-order probabilistic programs.

3 Formalization

We will now briefly outline the algorithm for discovering the static structures in probabilistic programs. The approach consists of *flagging* the *dynamic* parts of programs. A dynamic part of a program is a part that can be reached from within a stochastic branch. Intuitively, in the program in Figure 2, the entire `sim` function (lines 1–7) should be flagged. This is because of the recursive call to `sim` (line 5) inside the stochastic branch (lines 3–6). For the formalization, we use a small subset of our language based on the untyped lambda calculus with all necessary properties, shown in Figure 3. Note in particular the `sample` term, which, when supplied as the condition of an `if` term, makes a random choice in which branch to take. This is the minimal construct needed to introduce stochastic branching.

Given a program $\mathbf{t}$, the algorithm proceeds as follows: (1) Conservatively annotate each term in the program with its *stochasticness*. A term is stochastic if it can vary due to randomness (caused by the `sample` construct). This step transforms all terms $\mathbf{t}$ in our program to terms $\langle\mathbf{t}\rangle^s$, where $s \in \{\text{true}, \text{false}\}$. (2) By using these stochasticness annotations to identify stochastic branches, transform all terms $\langle\mathbf{t}\rangle^s$ to terms $\langle\mathbf{t}\rangle^d$, indicating whether they are static or dynamic ($d \in \{\text{true}, \text{false}\}$). We have formalized the above algorithm as syntax-directed inference rules with one caveat—they require explicit *annotations* on lambda parameters, one of which is the parameter's stochasticness. Such annotations should in the end be inferred automatically, and is left for future work. The inference rules are omitted due to space constraints.

To illustrate the challenges in the formalization, consider the following program

$$(\lambda\texttt{x}.\texttt{if sample then (x c) else c})\ (\lambda\texttt{x}.\texttt{x}) \tag{1}$$

which the final algorithm should transform to

$$(\lambda\texttt{x}.\texttt{if sample then}\ \underline{\texttt{(x c)}}\ \texttt{else}\ \underline{\texttt{c}})\ \underline{(\lambda\texttt{x}.\underline{\texttt{x}})} \tag{2}$$

where the underlining shows which parts of the program have been flagged as dynamic. The function $(\lambda\texttt{x}.\texttt{x})$ is flagged as dynamic since it is called from within a stochastic branch in the LHS of the application. Hence, the flagging of the RHS of an application depends on the LHS. It can also go in the other direction, as can be seen in the following example:

$$\begin{aligned}&(\lambda\texttt{x}.(\lambda\texttt{y}.\texttt{x y})\ (\lambda\texttt{x}.\texttt{x}))\\ &\qquad(\lambda\texttt{x}.\texttt{if sample then (x c) else c})\end{aligned} \tag{3}$$

which should transform to

$$\begin{aligned}&(\lambda\texttt{x}.(\lambda\texttt{y}.\texttt{x y})\ \underline{(\lambda\texttt{x}.\underline{\texttt{x}})})\\ &\qquad(\lambda\texttt{x}.\texttt{if sample then}\ \underline{\texttt{(x c)}}\ \texttt{else}\ \underline{\texttt{c}})\end{aligned} \tag{4}$$

To relate the above to SMC as described in the previous section, we simply note that resampling should only occur for `weight`s with $d = \text{false}$ (a `weight` term can easily be added to the language in Figure 3).

4 Conclusion and future work

We have presented a new approach to SMC in probabilistic programming. Future work includes adding inference of the now explicit annotations in the formalization, reasoning about the approach's correctness, evaluating the algorithm for SMC on a full-scale model, and making the approach less conservative.

References

[1] Noah D Goodman and Andreas Stuhlmüller. 2014. The Design and Implementation of Probabilistic Programming Languages. http://dippl.org. Accessed: 2018-7-17.

[2] Frank Wood, Jan Willem van de Meent, and Vikash Mansinghka. 2014. A New Approach to Probabilistic Programming Inference. In *Proceedings of the 17th International conference on Artificial Intelligence and Statistics.*