Checking Bounded-Memory Execution for Delayed Sampling on Probabilistic Streams

ERIC ATKINSON, MIT, USA
GUILLAUME BAUDART, INRIA, École normale supérieure – PSL University, France
LOUIS MANDEL, MIT-IBM Watson AI Lab, IBM Research, USA
CHARLES YUAN, MIT, USA
MICHAEL CARBIN, MIT, USA

1 MOTIVATION

Probabilistic Programming with Streams. Prior work introduced a probabilistic programming language, ProbZelus, to extend probabilistic programming functionality to unbounded streams of data [Baudart et al. 2020]. A key innovation was to demonstrate that *delayed sampling* [Murray et al. 2018] could be extended to work with streams to provide high-quality inference procedures. Delayed sampling is an inference algorithm that combines Bayesian networks – graphs that encode exact distributions of probabilistic models – with particle filtering [Del Moral et al. 2006] – an approximate inference algorithm.

Bounded-Memory Delayed Sampling. The challenge in adapting delayed sampling to computations on streams is that such computations run for indefinite periods of time and are often subject to stringent limits on resources, such as memory. Baudart et al. [2020] showed that, in many cases, only a finite number of nodes in delayed sampling's graph data structures were reachable at any given time, and the rest could not influence the computation in the future and could be removed from memory. However, this behavior depends on the probabilistic model under consideration; delayed sampling is not guaranteed to maintain a bounded amount of memory for all programs. The result is then that though probabilistic programming languages are designed to hide the complexities of developing probabilistic inference algorithms, certain compositions of a model and the inference algorithm will result in undesirable behaviors that the developer did not anticipate. Moreover, the developer has no means to reason about these behaviors except by inspecting the implementation of the inference algorithm.

Our goal is to identify the semantic conditions under which applying delayed sampling to probabilistic programs that operate on streams will execute in bounded memory and to define a static analysis to enforce them.

In the following we present on an example the programming model of ProbZelus (Section 2) and how we can statically check that the delayed sampling inference algorithm can execute in bounded memory (Section 3).

2 PROBABILISTIC PROGRAMMING WITH STREAMS

The stream function `hmm` presented in Figure 1 specifies a *hidden Markov model* [Baum and Petrie 1966], a common probabilistic model for tracking applications in which the goal is to estimate the trajectory of an object given noisy measurements of the object's position. We present the example in μF, a purely functional calculus for probabilistic programming with streams. The program is a set of *stream function* definitions that each consist of (1) an initializer, and (2) a step function that given the previous state and an input value produces an output value and a new state [Mealy 1955].

Authors' addresses: Eric Atkinson, MIT, USA; Guillaume Baudart, INRIA, École normale supérieure – PSL University, France; Louis Mandel, MIT-IBM Watson AI Lab, IBM Research, USA; Charles Yuan, MIT, USA; Michael Carbin, MIT, USA.

```
1 val hmm = stream {
2   init = 0.0;
3   step (pre_x, obs) =
4     let x = sample (gaussian (pre_x, 1.0)) in
5     let () = observe (gaussian (x, 1.0), obs) in
6     (x, x)
7 }
```

Fig. 1. A streaming probabilistic model (HMM) in μF.

The stream function's state consists of a *latent* random variable, `pre_x`, that denotes the position at the previous iteration. The state is latent in that it is not possible to directly observe the current position; instead we must leverage a noisy measurement or *observation* of the position to infer a probability distribution over potential states.

Inside the definition of `hmm`, the program models the latent nature of `x` by sampling the current position from a Gaussian distribution centered around its previous position `pre_x` (L.4). The program models the observation by taking the observed sensor value as input, `obs`, and supplying it as an input to the `observe` operator. In this example, the `observe` specifies that `obs` is an observation from a Gaussian distribution centered around the position `x`. The `observe` operator conditions the program's execution on the observed value (L.5) in that it adjusts the distribution that will be inferred for `x`.

The sequence of diagrams in Figure 2 illustrates the evolution of a representation of the hidden Markov model over the first four iterations of the program. Each light grey node denotes a latent random variable for `pre_x` or `x` at a given iteration. Each dark grey node denotes an observation at the given iteration. Each solid black arrow signifies a dependence between random variables as in a traditional Bayesian network representation of probabilistic graphical model [Koller and Friedman 2009]. Of note, each observation at each iteration depends on the current position and the state at a given iteration depends only on the position at the previous iteration.

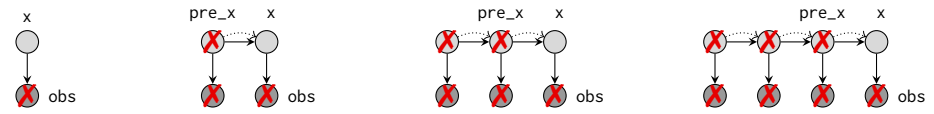
3 INFERENCE WITH DELAYED SAMPLING

The `hmm` probabilistic model alone is not enough to be able to reason about the estimated position. Instead, we must perform inference on the model to compute a posterior distribution of `x` conditioned on observations. In this paper, we study delayed sampling [Baudart et al. 2020; Murray et al. 2018] as the algorithmic implementation of the inference.

Delayed sampling leverages symbolic execution to reason about the relationship between random values to perform exact inference in the form of returning, if possible, a closed-formed distribution. Delayed sampling operates by dynamically maintaining a graph — i.e., a Bayesian network – that records the dependence relationships between the random variables in the program (Figure 2). The key idea is that rather than sample a concrete value for each random variable in the program (e.g., `x`), delayed sampling instead returns a reference to a node in the graph. This node contains a closed-form representation of the distribution that the `sample` operator sampled from, along with the distribution's dependence on other random variables in the program.

3.1 Bounded-Memory Delayed Sampling

A key concern when applying delayed sampling to streams, which may execute for an indefinite number of iterations, is if the size of the delayed sampling graph is bounded from above by a fixed constant for all iterations of the program. If not, then the delayed sampling graph may not consume bounded memory and, hence, the program may exhaust its resources if permitted to execute indefinitely as streams are often permitted to do.



(a) iteration 1 (b) iteration 2 (c) iteration 3 (d) iteration 4

Fig. 2. The evolution of the delayed sampling graph for the hidden Markov model in Figure 1 (`hmm`) as implemented by Baudart et al. [2020]. Each node denotes either a value (dark gray) or a distribution (light gray). A plain arrow denotes a dependency in the underlying Bayesian network. A dotted arrow denotes a pointer in the implementation of the delayed sampling graph. Each label indicates the program variable that correponds to a node. An ✗ on a node denotes that the node is not reachable from the program state.

```
val hmm_first = stream {
  init = (true, 0.0, 0.0);
  step ((first, i, pre_x), obs) =
    let (i, pre_x) =
      if first then (let i = sample (gaussian (0.0, 1.0)) in (i, i))
      else (i, pre_x) in
    let x = sample (gaussian (pre_x, 1.0)) in
    let () = observe (gaussian (x, 1.0), obs) in
    (x, (false, i, x))
}
```

Fig. 3. Model with unbounded memory consumption.

In general, bounding memory use is challenging because the underlying Bayesian network can in fact be unbounded. Nevertheless, a delayed sampling implementation can maintain bounded memory for some programs, depending on the operation of said programs.

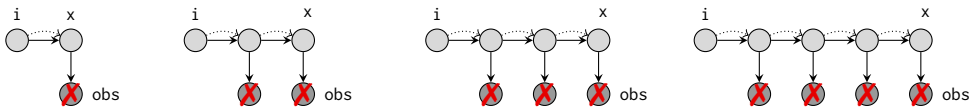
Bounded-Memory Example. Figure 2 shows how delayed sampling maintains bounded memory for the program in Figure 1. The delayed sampling implementation must keep in memory all the nodes that are *reachable* from any node referenced in the program state. The dashed lines in Figure 2 visualize the reachability relation: the node each line points to is reachable from the node the line points from. As the program evolves its state and changes the variables the state contains, nodes in the delayed sampling graph may become unreachable. Nodes marked with an ✗ are unreachable.

Figure 2a shows the delayed sampling graph after the first iteration. The graph consists of two nodes: one introduced by sampling the variable `x`, and one introduced by the observation of `obs`. At the end of the step, only the variable `x` is in the program state and reachable. The node introduced by the observation represents a concrete value that is no longer reachable.

Figure 2b shows the delayed sampling graph after the second iteration. The program has added two nodes to the graph for sampling `x` and observing `obs`. Again, only the variable `x` is in the program state and reachable. The nodes left over from the first iteration and the node introduced for the observation are still in the graph, but are no longer reachable.

Figures 2c and 2d show the delayed sampling graph at iterations 3 and 4, respectively. In each case the most recently introduced node for `x` is reachable, and the nodes from the previous iterations and the observation `obs` are unreachable. This pattern continues at future iterations. In general, the program ensures that at any iteration, the most recently introduced node is reachable, and the rest are unreachable. Because the set of reachable nodes is at most one for all iterations, inference executes in bounded memory.

Unbounded-Memory Example. Figure 3 presents an example of a program that does not execute in bounded memory. This is a modified version of `hmm` from Figure 1 that samples an initial latent



(a) iteration 1 (b) iteration 2 (c) iteration 3 (d) iteration 4

Fig. 4. The evolution of the delayed sampling graph for the variant of a HMM probabilistic model in Figure 3. Nodes and edges have the same meaning as in Figure 2.

$$\tau_2 = \mathtt{x}_1 \leftsquigarrow \mathtt{nil} :: \mathtt{y}_1 \leftsquigarrow \mathtt{x}_0 :: \mathtt{obs}\ \mathtt{y}_0 :: \mathtt{x}_2 \leftsquigarrow \mathtt{x}_1 :: \mathtt{y}_2 \leftsquigarrow \mathtt{x}_2 :: \mathtt{obs}\ \mathtt{y}_2$$

Fig. 5. A depiction of a trace of the program in Figure 1. The figure depicts the trace τ_2 at the end of iteration 2. The trace is a ::-separated list of primitive operations, where each primitive operation is a sampling operation $\leftsquigarrow$ or an observation operation obs. In this diagram, we use $\mathtt{x}_n$ and $\mathtt{y}_n$ to refer to the random variables introduced at iteration n by, respectively, sampling x and observing obs in Figure 1.

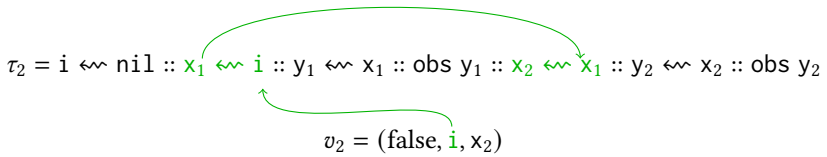


Fig. 6. A depiction of a trace of the program in Figure 3. The figure depicts the trace τ_2 and the value of the program state v_2 at the end of iteration 2. In this diagram, we use i, $\mathtt{x}_n$, $\mathtt{y}_n$, respectively, to refer to the random variable introduced by sampling i, the variable introduced at iteration n by sampling x, and the variable introduced at iteration n by observing obs in Figure 3. We have highlighted the elements of the unseparated paths between i and $\mathtt{x}_2$ in green.

position i from a Gaussian distribution and keeps a reference to this random variable in the state. Figure 4 shows how the program in Figure 3 fails to maintain bounded memory.

Figure 4a shows the delayed sampling graph after the first iteration. The graph consists of three nodes introduced by sampling the variable i, sampling the variable x, and by the observation of obs. The program state comprises variables i and x, and their associated nodes are reachable.

Figure 4b shows the delayed sampling graph after the second iteration. The program has added two nodes to the graph for sampling x and observing obs. Since the variable i is in the program state, the node between i and x is reachable.

Figures 4c and 4d show that in the next iterations two new nodes are introduced at each step and one remains reachable. The primary observation to note is that the number of introduced nodes increases at every iteration. Therefore, there is no bound on the size of the delayed sampling graph and, hence, the program does not execute in bounded memory.

3.2 Analyzing Delayed Sampling

We propose an analysis that can show that the program in Figure 1 maintains bounded memory while the program in Figure 3 does not. For that, we define two dataflow properties that encode whether a program execution achieves bounded memory: the *unseparated path* property and the *m-consumed* property. We then show how these dataflow properties can be verified using a static analysis.

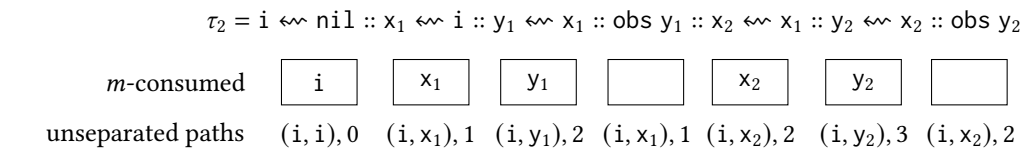


Fig. 7. A depiction of the abstract graphs of the program in Figure 3, with the same trace as Figure 6. At each operation, we depict the m-consumed abstract graph, which is a set of nodes that have been introduced but not consumed. Because this set is empty at the end of any iteration, the program satisfies the m-consumed semantic property. The unseparated paths abstract graph is a mapping, for each unseparated path in the graph, from its pair of endpoints to its length. We depict the longest path residing in the mapping. After each iteration, this longest path continues to lengthen, so the program does not satisfy the unseparated paths semantic property.

Traces. We formalize the dataflow properties as properties of *traces*. A trace is a recording of the important features of a program execution. In our case, a trace records all sampling and observation operations that the program has executed, as well as the variables that were involved in these operations. Figure 5 illustrates a trace of the execution of the program in Figure 1.

Unseparated Paths. An *unseparated path* in a trace is a sequence of variables $\{x_i\}$, where the trace specifies that each variable x_i was sampled from its predecessor x_{i-1} and no x_i is observed. The *unseparated path property* states that there is a uniform bound c so that for all iterations no variable in the program state starts an unseparated path with more than c variables in it.

Figure 6 illustrates the trace for the program in Figure 3. This program carries the variable $\mathtt{i}$ in the program state, and because the trace specifies that $\mathtt{x}_1$ was sampled from $\mathtt{i}$, and $\mathtt{x}_2$ was sampled from $\mathtt{x}_1$, the sequence $\mathtt{i}, \mathtt{x}_1, \mathtt{x}_2$ is an unseparated path with 3 variables. In general, at iteration n, the program in Figure 3 maintains that $\mathtt{i}$ is in the program state and starts an unseparated path with length $n + 2$. Because no bound can exist on the length of this path for an arbitrary number of iterations, this program fails the unseparated path property.

m-consumed. A variable is *m-consumed* if it is no more than m sampling operations away from a variable that is *consumed* by an observe statement. The *m-consumed property* states that there is a uniform bound $\overline{m}$ such every variable introduced by a sampling operation is m-consumed for some $m \leq \overline{m}$. We note that the traces in Figures 5 and 6 satisfy the m-consumed property, because every variable is at most 2-consumed. For all t, $\mathtt{y}_t$ is 0-consumed because it is directly observed, and $\mathtt{x}_t$ is 1-consumed because $\mathtt{y}_t$ is sampled from $\mathtt{x}_t$ and $\mathtt{y}_t$ is 0-consumed. The variable $\mathtt{i}$ is 2-consumed because $\mathtt{x}_1$ is sampled from $\mathtt{i}$, and $\mathtt{x}_1$ is 1-consumed.

Analysis. Our goal is ultimately to analyze whether a given program will execute in bounded memory. This property is satisfied iff it satisfies both the unseparated path and m-consumed properties. This reduces the problem of analyzing the bounded-memory behavior of a program to analyzing these dataflow properties. Our analysis utilizes an abstract delayed sampling graph with the key aspects of these properties. For the m-consumed property, the abstract graph maintains a set of variables that have been introduced but not yet consumed, and for unseparated paths, it maintains an upper bound on their length. For example, the abstract graphs for the trace in Figure 6 are given in Figure 7.

4 CONCLUSION

Probabilistic programming has been augmented by constructs that perform inference over unbounded iterations on streams of data. Underlying this programming model is delayed sampling, which combines the benefits of exact inference and the flexibility of sampling.

We introduce the m-consumed and unseparated path semantic properties, which show that delayed sampling can execute in bounded memory for reactive probabilistic programs. To the best of our knowledge, our work is the first to develop a resource analysis for a probabilistic program in relation to its probabilistic programming system's underlying inference algorithm. We hope this work will enable automatic inference mechanisms whose performance is better understood by model developers in probabilistic programming languages.

REFERENCES

Guillaume Baudart, Louis Mandel, Eric Atkinson, Benjamin Sherman, Marc Pouzet, and Michael Carbin. 2020. Reactive Probabilistic Programming. In *Conference on Programming Language Design and Implementation*.

Leonard E. Baum and Ted Petrie. 1966. Statistical Inference for Probabilistic Functions of Finite State Markov Chains. *The Annals of Mathematical Statistics* 37, 6 (1966).

Pierre Del Moral, Arnaud Doucet, and Ajay Jasra. 2006. Sequential Monte Carlo samplers. *J. Royal Statistical Society: Series B (Statistical Methodology)* 68, 3 (2006), 411–436.

Daphne Koller and Nir Friedman. 2009. *Probabilistic Graphical Models - Principles and Techniques*. MIT Press.

George H. Mealy. 1955. A method for synthesizing sequential circuits. *The Bell System Technical Journal* 34, 5 (1955), 1045–1079.

Lawrence M. Murray, Daniel Lundén, Jan Kudlicka, David Broman, and Thomas B. Schön. 2018. Delayed Sampling and Automatic Rao-Blackwellization of Probabilistic Programs. In *International Conference on Artificial Intelligence and Statistics*.