Statically Bounded-Memory Delayed Sampling for 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

Probabilistic programming languages aid developers performing Bayesian inference. These languages provide programming constructs and tools for probabilistic modeling and automated inference. Prior work introduced a probabilistic programming language, ProbZelus, to extend probabilistic programming functionality to unbounded streams of data. This work demonstrated that the *delayed sampling* inference algorithm could be extended to work in a streaming context. ProbZelus showed that while delayed sampling could be effectively deployed on some programs, depending on the probabilistic model under consideration, delayed sampling is not guaranteed to use a bounded amount of memory over the course of the execution of the program.

In this paper, we the present conditions on a probabilistic program's execution under which delayed sampling will execute in bounded memory. The two conditions are dataflow properties of the core operations of delayed sampling: the *m-consumed property* and the *unseparated paths property*. A program executes in bounded memory under delayed sampling if, and only if, it satisfies the *m*-consumed and unseparated paths properties. We propose a static analysis that abstracts over these properties to soundly ensure that any program that passes the analysis satisfies these properties, and thus executes in bounded memory under delayed sampling.

CCS Concepts: • **Theory of computation** → **Program analysis**; **Streaming models**; • **Software and its engineering** → **Data flow languages**.

Additional Key Words and Phrases: Probabilistic programming, reactive programming, streaming inference, semantics, program analysis

1 INTRODUCTION

Probabilistic programming languages aid developers performing Bayesian inference [Atkinson et al. 2018; Bingham et al. 2019; Cusumano-Towner et al. 2019; Ge et al. 2018; Gelman et al. 2015; Goodman et al. 2008; Goodman and Stuhlmüller 2014; Gordon et al. 2014; Huang et al. 2017; Mansingkha et al. 2018; Milch et al. 2007; Narayanan et al. 2016; Nori et al. 2015; Pfeffer 2009; Tran et al. 2017]. These languages provide programming constructs and tools for probabilistic modeling and automated inference. Researchers have developed probabilistic programming languages for several domains,

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.

including data science [Gelman et al. 2015], machine learning [Bingham et al. 2019; Tran et al. 2017], scientific simulation [Baydin et al. 2019], and real-time control [Baudart et al. 2020].

Probabilistic Programming with Streams. In this paper, we consider programs that accept inputs and compute outputs at discrete time steps, with the outputs of each step flowing into the environment to affect future inputs to the program. Mathematically, one can model these programs as computations that operate on and produce infinite *streams*. Computing with streams is a common computational model for applications in real-time control, such as robotics and avionics [Colaço et al. 2017]. For example, control for an airplane fly-by-wire system can be implemented as a program transforming a stream of altitude measurements into a stream of commands to the engine.

Baudart et al. [2020] introduced a probabilistic programming language, ProbZelus, to enable probabilistic programming in this domain of computations on streams. A key innovation of ProbZelus 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 both exact and approximate inference; it takes advantage of exact inference when efficient known closed-formed solutions exist and falls back on sampling-based, approximate inference when required. Specifically, delayed sampling combines Bayesian networks – graphs that encode exact distributions of probabilistic models – with particle filtering [Del Moral et al. 2006] – an approximate inference algorithm.

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 combinations 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.

Bounded-Memory Delayed Sampling. In this paper, we formalize semantic conditions under which applying delayed sampling to probabilistic programs with streams will execute in bounded memory.

The two conditions are dataflow properties of the core operations of delayed sampling: *assume*, *observe*, and *value*, which respectively add a new random variable to the delayed sampling graph, observe a random variable, and evaluate a random variable to produce a sampled value. The *m-consumed property* states that all variables introduced with *assume* are eventually consumed by an *observe* or a *value*, or are passed to other *assume*s resulting in new variables that are themselves $(m-1)$-consumed. An *unseparated path* is a sequence of random variables, each passed as parameter to the *assume* operation of the next, where no variable is passed to an *observe* or *value* operation. The *unseparated paths property* states that no variable maintained in the program state starts an unseparated longer than some fixed bound n. A program executes in bounded memory under delayed sampling if, and only if, it satisfies the *m*-consumed and unseparated paths properties.

Static Analysis. We propose a static analysis that checks the *m*-consumed and unseparated paths properties to soundly ensure that any program that passes the analysis satisfies these properties, and thus executes in bounded memory under delayed sampling.

Contributions. In this paper, we present the following contributions:

- We introduce and formalize the *m*-consumed and unseparated paths properties, and show these are necessary and sufficient for a program to have bounded-memory execution.
- We present a static analysis to check these properties, and prove that the analysis is sound.
- We implement the analysis and evaluate it against several probabilistic inference benchmarks. Our results show that for eight of nine benchmarks, the analysis determines whether the semantic properties necessary for bounded-memory execution are satisfied, and we identify the precision limitation of conservative static analysis on the remaining benchmark.

This work brings probabilistic programming to control settings with the new benefit of static guarantees on the system's resource consumption. 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.

The remainder of the paper is structured as follows. In Section 2, we give an example program to illustrate the concepts in the paper. In Section 3, we present the syntax and semantics of a language for probabilistic programming with streams, adapted from the *μF* language from Baudart et al. [2020]. In Section 4, we review background on delayed sampling, based on the contributions from Murray et al. [2018] and Baudart et al. [2020]. In Section 5, we present the *m*-consumed and unseparated paths semantic properties. In Sections 6 and 7, we present and evaluate the static analysis. Sections 8 and 9 summarize related work and present conclusions. An extended version of this paper with all appendices is available [Atkinson et al. 2021].

2 EXAMPLE

Figure 1 presents the example of a `robot` designed to navigate to a desired position `target` using measurements `obs` from a noisy position sensor. The robot issues a command `u` that indicates the acceleration to apply to change its position. The robot (1) estimates its current position with a probabilistic model `kalman` and (2) uses this estimate to compute the command `u` with a deterministic `controller` (e.g., a Linear-Quadratic Regulator [Sontag 2013], the implementation of which we have elided for simplicity). We present the example in *μF*, a purely functional core 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]. The operators `init` and `infer` instantiate a stream function by creating an internal state. A stream function can be applied to an input stream to generate an output stream with the operator `unfold`, which applies the step function using the internal state and the input values. Unlike `init`, the step function of an instance created using `infer` performs probabilistic inference and thus returns at each iteration a distribution of outputs and a distribution of states.

```
1  val kalman = 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  }
8  val robot = stream {
9    init = (0.0, init controller, infer kalman);
10   step ((c, k), (obs, target)) =
11     let x_dist, k' = unfold (k, obs) in
12     let u, c' = unfold (c, (target, mean (x_dist))) in
13     (u, (c', k'))
14 }
```

Fig. 1. *μF* program with main stream function `robot`.

The main stream function, robot, has a state composed of two stream function instances: c the deterministic controller, and k the kalman probabilistic model. The robot initializer creates these two instances (L.9). The transition function of instance k performs probabilistic inference to infer a distribution of the robot's state x_dist and an updated instance k' (L.11). Then the transition function of instance c computes a command u to go toward the destination target using statistics of the position distribution and an updated instance c' (L.12). The transition function of robot returns the command u and the updated state (L.13).

2.1 Probabilistic Model

The stream function kalman 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.

The stream function's state consists of a *latent* random variable, pre_x, that denotes the position of the robot at the previous iteration. The robot's state is latent in that the robot is unable to directly observe its position; instead it must leverage a noisy measurement or *observation* of its position to infer a probability distribution over its potential states.

Inside the definition of kalman, 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 a probabilistic graphical model [Koller and Friedman 2009]. Of note, each observation at each iteration depends on the current position and the robot's state at a given iteration depends only on its position at the previous iteration.

2.2 Inference with Delayed Sampling

The kalman probabilistic model is not sufficient for the robot to reason about its position. Instead, the robot must perform inference on the model to compute a posterior distribution of x conditioned on its observations. As mentioned, the infer operator in the robot stream function applies inference to the probabilistic model it receives as input. In this paper, we study delayed sampling [Baudart et al. 2020; Murray et al. 2018] as the algorithmic implementation of the infer operator.

Delayed sampling is an extension of a particle filtering algorithm that leverages symbolic execution to reason about the relationship between random values and perform exact inference if possible. A particle filter estimates the posterior distribution from a set of *particles*, i.e., independent executions of the model. For each particle, 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. If a symbolic computation fails, delayed sampling can fall back to a particle filter by drawing concrete values for the random variables.
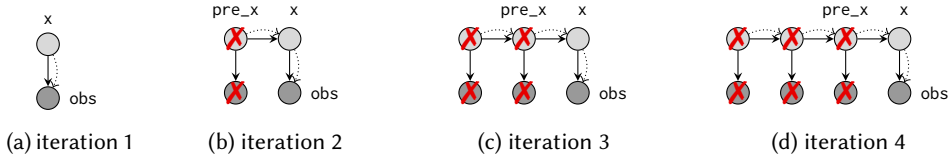
Fig. 2. The evolution of the delayed sampling graph for the hidden Markov model in Figure 1 (kalman) as implemented by Baudart et al. [2020]. Each node denotes either a value (dark gray) or a distribution (light gray). A plain arrow denotes a dependency in the underlying Bayesian network. A dotted arrow denotes a pointer in the implementation of the delayed sampling graph. Each label indicates the program variable that corresponds to a node. An ✗ on a node denotes that the node is not reachable from the program state.

2.3 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 the program may exhaust its resources if permitted to execute indefinitely.

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. In this subsection, we review the delayed sampling implementation presented by Baudart et al. [2020] which can execute in bounded memory for some programs.

Bounded-Memory Example. Figure 2 shows how delayed sampling maintains bounded memory for the program in Figure 1. For each particle, 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, where 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, marked ✗.

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, both are in the program state and 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. The nodes left over from the first iteration 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 nodes for x and obs are reachable, and the nodes from the previous iterations are unreachable. In general, the program ensures that at any iteration, the most recently introduced nodes are reachable, and the rest are unreachable. Because there are at most two reachable nodes 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 kalman from Figure 1 that samples an initial latent 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 reachable nodes introduced by sampling the variables i and x and by the observation of obs.

```
val kalman_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.

(a) iteration 1   (b) iteration 2   (c) iteration 3   (d) iteration 4
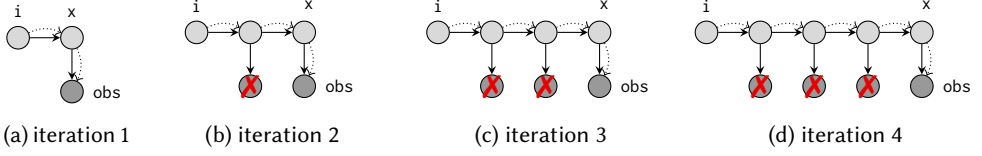
Fig. 4. The evolution of the delayed sampling graph for the variant of a Kalman probabilistic model in Figure 3. Nodes and edges have the same meaning as in Figure 2.
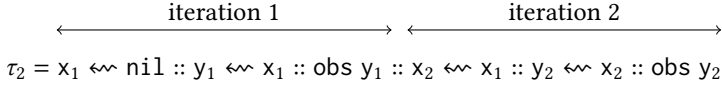
$$\overbrace{}^{\text{iteration 1}} \quad \overbrace{}^{\text{iteration 2}}$$

$$\tau_2 = x_1 \looparrowleft \text{nil} :: y_1 \looparrowleft x_1 :: \text{obs } y_1 :: x_2 \looparrowleft x_1 :: y_2 \looparrowleft x_2 :: \text{obs } y_2$$

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 $\looparrowleft$ or an observation operation obs. In this diagram, we use x_n and y_n to refer to the random variables introduced at iteration n by, respectively, sampling x and observing obs in Figure 1.

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.

2.4 Analyzing Delayed Sampling

In this paper, we present 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 executes in bounded memory: the *unseparated paths* property and the *m-consumed* property. We then show how these properties can be verified using a static analysis.

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.
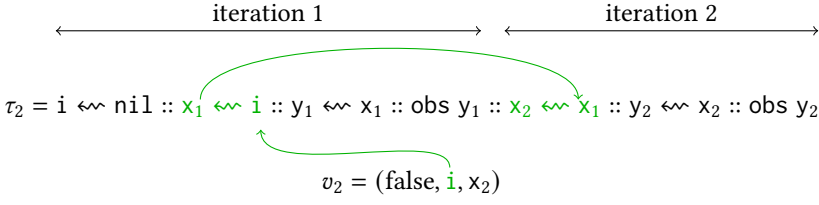
Fig. 6. A depiction of a trace of the program in Figure 3. The figure depicts the trace τ_2 and the value of the program state v_2 at the end of iteration 2. In this diagram, we use i, x_n, y_n, respectively, to refer to the random variable introduced by sampling i, the variable introduced at iteration 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 path between i and x_2 in green.

The *unseparated paths 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 i in the program state, and because the trace specifies that x_1 was sampled from i, and x_2 was sampled from x_1, the sequence i, x_1, x_2 is an unseparated path with 3 variables. In general, at iteration n, the program in Figure 3 maintains that i is in the program state and starts an unseparated path with length $n + 1$. 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, y_t is 0-consumed because it is directly observed, and x_t is 1-consumed because y_t is sampled from x_t and y_t is 0-consumed. The variable i is 2-consumed because x_1 is sampled from i, and x_1 is 1-consumed.

The *Outlier* benchmark presented in Section 7 is an example of a program that fails the *m-consumed* property, and thus does not execute in bounded memory. This program sometimes observes values close to the true latent state but otherwise observes values from an outlier distribution. When the program observes a value from the outlier distribution, it fails to observe any dependencies of the latent state, and thus cannot guarantee that the latent state is *m-consumed*. Over time, if the program performs latent state updates that remain unobserved (due to the program always observing from the outlier distribution), the lack of this guarantee results in there being no uniform bound $\overline{m}$ under which the latent state could be $\overline{m}$-consumed.

Analysis. Our goal is ultimately to analyze whether a given program executes in bounded memory. As we show in Section 5, a program execution maintains bounded memory if and only if 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, formally defined in Section 6, with the key aspects of these properties. For *m-consumed*, 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.
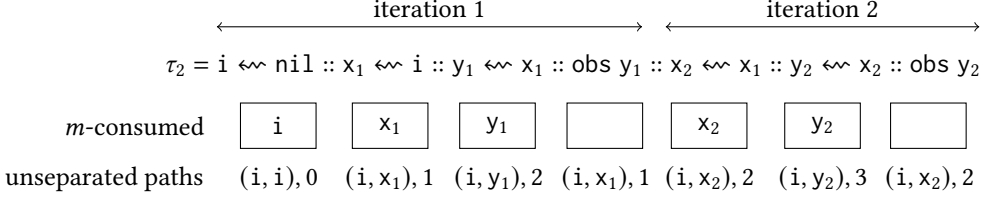
$$\overset{\xleftarrow{\hspace{4cm} \text{iteration 1} \hspace{4cm}}}{} \quad \overset{\xleftarrow{\hspace{3cm} \text{iteration 2} \hspace{3cm}}}{}$$

$\tau_2 = $ i $\xleftsquigarrow$ nil :: x_1 $\xleftsquigarrow$ i :: y_1 $\xleftsquigarrow$ x_1 :: obs y_1 :: x_2 $\xleftsquigarrow$ x_1 :: y_2 $\xleftsquigarrow$ x_2 :: obs y_2

m-consumed	i	x_1	y_1		x_2	y_2	

unseparated paths	$(i,i),0$	$(i,x_1),1$	$(i,y_1),2$	$(i,x_1),1$	$(i,x_2),2$	$(i,y_2),3$	$(i,x_2),2$

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, 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 endpoints to its length. We depict the longest path in the mapping. After each iteration, this longest path continues to lengthen, so the program does not satisfy the unseparated paths semantic property.

3 LANGUAGE MODEL

In this section, we present a semantics for probabilistic programs with streams using the language μF. We have adapted μF from Baudart et al. [2020]'s core calculus for probabilistic programs and extended it with syntax for explicit streams.

3.1 Syntax

The syntax of the μF language is defined according to the following grammar:

$$
\begin{aligned}
\textit{program} \quad &::= \quad d^*\, m \\
d \quad &::= \quad \texttt{val}\ p = e \mid \texttt{val}\ f = \texttt{fun}\ p\ \texttt{->}\ e \mid \texttt{val}\ m = \texttt{stream}\ \{\ \texttt{init} = e\ ;\ \texttt{step}(p,p) = e\ \} \\
e \quad &::= \quad v \mid op(v) \mid f(v) \mid \texttt{if}\ v\ \texttt{then}\ e\ \texttt{else}\ e \mid \texttt{let}\ p\ \texttt{=}\ e\ \texttt{in}\ e \\
&\qquad \mid \texttt{init}(m) \mid \texttt{unfold}(x,v) \mid \texttt{sample}(v) \mid \texttt{observe}(v,v) \mid \texttt{infer}(m) \\
v \quad &::= \quad c \mid x \mid (v,v) \\
p \quad &::= \quad x \mid (p,p)
\end{aligned}
$$

A program is a set of value, function, and stream function definitions followed by the name of the main stream function. A *stream function* m is composed of an initial state (`init`) and a transition function (`step`). Given a state and an input, the transition function returns an output and a new state. An expression is either a value (constant, variable, or pair), the application of a primitive operator (arithmetic operator, distribution, etc.), a function call, a conditional, or a local definition.

The expression $\texttt{init}(m)$ creates an *instance* of a stream function, and $\texttt{unfold}(x,v)$ applies the instance x of a stream function on an input and returns the next element and the updated instance. Finally, the set of expressions comprises the probabilistic operators `sample`, `observe`, and `infer`. Nested inference and higher-order functions on streams are not allowed in the language. We require that arguments for all syntactic operators are values to simplify the presentation of the semantics. Since new variables can always be introduced to capture the value of any expression, this choice does not reduce the expressiveness of the language.

3.2 Semantics

The execution of a program $p = d^*m$ comprises three steps. First, declarations d^* are evaluated to produce an environment γ which contains the definition of the main stream function m. Second, an instance of the stream function m is created.

$$\llbracket \text{val } x = e \rrbracket_\gamma \qquad\qquad = \gamma[x \leftarrow \llbracket e \rrbracket_\gamma]$$

$$\llbracket \text{val } f = \text{fun } p \; \text{->} \; e \rrbracket_\gamma = \gamma[f \leftarrow (\lambda v.\ \llbracket e \rrbracket_{\gamma + [v/p]})]$$

$$\llbracket \text{val } m = \text{stream} \{ \text{ init} = e_{init} \; ; \; \text{step}(p_{state}, p_{input}) = e \; \} \rrbracket_\gamma$$
$$= \gamma[m \leftarrow \text{stream} \{ \text{ init} = e_{init} \; ; \; \text{step}(p_{state}, p_{input}) = e \; \}_\gamma]$$

$$\llbracket \text{init}(m) \rrbracket_\gamma \qquad\quad = \textit{let } \text{stream} \{ \text{ init} = e_{init} \; ; \; \text{step}(p_{state}, p_{input}) = e \; \}_{\gamma'} = \llbracket m \rrbracket_\gamma \textit{ in}$$
$$\textit{let } s_{init} = \llbracket e_{init} \rrbracket_{\gamma'} \textit{ in } (s_{init}, \lambda(s, v).\ \llbracket e \rrbracket_{\gamma' + [s/p_{state}, v/p_{input}]})$$
$$\textit{if } e \textit{ is deterministic}$$

$$\llbracket \text{init}(m) \rrbracket_\gamma \qquad\quad = \textit{let } \text{stream} \{ \text{ init} = e_{init} \; ; \; \text{step}(p_{state}, p_{input}) = e \; \}_{\gamma'} = \llbracket m \rrbracket_\gamma \textit{ in}$$
$$\textit{let } s_{init} = \llbracket e_{init} \rrbracket_{\gamma'} \textit{ in } (s_{init}, \lambda(s, v).\ \{\!|e|\!\}_{\gamma' + [s/p_{state}, v/p_{input}]})$$
$$\textit{if } e \textit{ is probabilistic}$$

$$\llbracket \text{unfold}(x, v) \rrbracket_\gamma \quad = \textit{let } v_{state}, f = \llbracket x \rrbracket_\gamma \textit{ in}$$
$$\textit{let } v_{output}, v'_{state} = f(v_{state}, \llbracket v \rrbracket_\gamma) \textit{ in}$$
$$(v_{output}, (v'_{state}, f))$$

$$\llbracket \text{infer}(m) \rrbracket_\gamma \qquad = \textit{let } \text{stream} \{ \text{ init} = e_{init} \; ; \; \text{step}(p_{state}, p_{input}) = e \; \}_{\gamma'} = \llbracket m \rrbracket_\gamma \textit{ in}$$
$$\textit{let } s_{init} = \llbracket e_{init} \rrbracket_{\gamma'} \textit{ in } (\delta_{s_{init}}, \textit{infer}(\lambda(s, v).\ \{\!|e|\!\}_{\gamma' + [s/p_{state}, v/p_{input}]}))$$
$$\textit{where } \textit{infer}(f) = \lambda(\sigma, v).\ \textit{let } \mu = \lambda U.\ \int_S \sigma(ds) f(s, v)(U) \textit{ in}$$
$$\textit{let } \nu = \lambda U.\ \mu(U)/\mu(\top) \textit{ in}$$
$$(\pi_{1*}(\nu), \pi_{2*}(\nu))$$

Fig. 8. Deterministic semantics of μF (complete definition in Figure 16).

Third, the instance is iteratively applied on an input stream $(i_n)_{n\in\mathbb{N}}$ to produce an output stream $(o_n)_{n\in\mathbb{N}}$, defined in the following way:

$$\llbracket p \rrbracket(i)_n = o_n \quad \text{where} \quad \begin{aligned} p &= d^* m \\ s_0 &= \llbracket \text{init}(m) \rrbracket_\gamma \end{aligned} \qquad \begin{aligned} \gamma &= \llbracket d^* \rrbracket_\emptyset \\ o_n, s_{n+1} &= \llbracket \text{unfold}(s_n, i_n) \rrbracket_\gamma \quad \forall n \geq 0 \end{aligned}$$

Figure 8 defines the semantics of declarations and deterministic expressions $\llbracket \cdot \rrbracket$. The declarations build the evaluation environment γ which maps names to values, functions, and stream functions.

The semantics of deterministic expressions corresponds to a first order functional language with new constructs to handle streams and the $\text{infer}(\cdot)$ operator (the complete definition is given in Figure 16 of Appendix A). The expression $\text{init}(m)$ creates an instance of the stream function m: a pair corresponding to the current state, and the transition function. The current state is initialized with the value of the init field. The expression $\text{unfold}(x, v)$ executes the transition function of the instance x on its current state and the input v. This expression produces a pair composed of the transformed value and the updated instance.

The ideal semantics of μF probabilistic expressions $\{\!|\cdot|\!\}$ is a measure-based semantics similar to the one presented by Staton [2017] (the complete definition is given in Appendix A). Given an environment γ, an expression is interpreted as a measure $\{\!|e|\!\}_\gamma : \Sigma_D \rightarrow [0, \infty)$, that is, a function which associates a positive number to each measurable set $U \in \Sigma_D$, where Σ_D denotes the Σ-algebra of the domain of the expression D (i.e., the set of measurable sets of possible values). $\text{sample}(v)$ returns the distribution $\llbracket v \rrbracket_\gamma$. $\text{observe}(v_1, v_2)$ weights execution paths using the likelihood of the observation $\llbracket v_2 \rrbracket_\gamma$ w.r.t. the distribution $\llbracket v_1 \rrbracket_\gamma$ (for a distribution μ we denote its probability density function as μ_{pdf}). Local definitions are interpreted as integration, and we use the Dirac delta measure to interpret deterministic expressions.

$$\{\![v]\!\}_\gamma \qquad\qquad\qquad = \lambda g, w.\ (\![v]\!]_\gamma, g, w)$$

$$\{\![op(v)]\!\}_\gamma \qquad\qquad\quad = \lambda g, w.\ (\text{app}(op, [\![v]\!]_\gamma), g, w)$$

$$\{\![f(v)]\!\}_\gamma \qquad\qquad\qquad = \lambda g, w.\ \gamma(f)([\![v]\!]_\gamma)(g, w)$$

$$\{\![\texttt{let } p = e_1 \texttt{ in } e_2]\!\}_\gamma = \lambda g, w.\ let\ v_1, g_1, w_1 = \{\![e_1]\!\}_\gamma(g, w)\ in\ \{\![e_2]\!\}_{\gamma + [v_1/p]}(g_1, w_1)$$

$$\{\![\texttt{if } v \texttt{ then } e_1 \texttt{ else } e_2]\!\}_\gamma = \lambda g, w.\ let\ b, g_b = value([\![v]\!]_\gamma, g)\ in$$
$$\qquad\qquad\qquad\qquad\qquad\qquad if\ b\ then\ \{\![e_1]\!\}_\gamma(g_b, w)\ else\ \{\![e_2]\!\}_\gamma(g_b, w)$$

$$\{\![\texttt{unfold}(x,v)]\!\}_\gamma \qquad = \lambda g, w.\ let\ v_{state}, f = [\![x]\!]_\gamma\ in$$
$$\qquad\qquad\qquad\qquad\qquad\qquad let\ (v_{output}, v'_{state}), g'w' = f(v_{state}, [\![v]\!]_\gamma)(g, w)\ in$$
$$\qquad\qquad\qquad\qquad\qquad\qquad ((v_{output}, (v'_{state}, f)), g'w')$$

$$\{\![\texttt{sample}(v)]\!\}_\gamma \qquad\quad = \lambda g, w.\ let\ X, g' = assume([\![v]\!]_\gamma, g)\ in\ (X, g', w)$$

$$\{\![\texttt{observe}(v_1,v_2)]\!\}_\gamma \quad = \lambda g, w.\ let\ X, g_x = assume([\![v_1]\!], g)\ in$$
$$\qquad\qquad\qquad\qquad\qquad\qquad let\ v, g_v = value([\![v_2]\!], g_x)\ in$$
$$\qquad\qquad\qquad\qquad\qquad\qquad let\ g' = observe(X, v, g_v)\ in\ ((), g', w * \mu_{\text{pdf}}(v))$$

Fig. 9. Delayed sampling semantics. Probabilistic expressions are functions from a graph and a weight to a triplet (value, graph, weight).

The $infer(m)$ operator creates an instance of a probabilistic stream: the initial state is a Dirac delta distribution on the initial state of m, and the transition function is $infer(f)$ where f is the transition function of m. The body of f (the expression e) is interpreted with the probabilistic semantics which defines a measure over pairs of output values and states. The function $infer(f)$ takes as arguments a distribution of states σ and an input v and returns a distribution of outputs and a distribution of new states. These two distributions are obtained by integrating the transition function f along the distribution σ of possible states (domain S) to build a measure μ which is then normalized to build a distribution ν of pairs (outputs, states). The distribution ν is then split into a pair of marginal distributions using the pushforward of ν across the projections π_1 and π_2.

4 DELAYED SAMPLING

In this section, we present the details of delayed sampling that underpin this work. This is a new formalization of results that were presented by Murray et al. [2018] and Baudart et al. [2020].

Delayed sampling is a semi-symbolic algorithm combining exact inference and – when exact computation fails – approximate inference with particle filtering [Del Moral et al. 2006]. A particle filter launches multiple executions of the model. Each execution — or particle — is associated to a weight. In the operational semantics, $sample(d)$ statements draw samples from the corresponding distributions, and $observe(x, d)$ statements update the weight to reflect the quality of the samples. At the end of the executions the results of all the particles are normalized according to their weights to form a categorical distribution that approximates the posterior distribution of the model.

In delayed sampling, each particle contains a graph of random variables and their dependencies that can be used to compute closed-form distributions. Observations can be incorporated by analytically *conditioning* the network. If symbolic conditioning fails, inference falls back to a particle filter, drawing concrete samples for required random variables.

4.1 Operational Semantics

The definition of `infer` in Figure 8 makes use of an intractable integral. The delayed sampling semantics replaces this integral by a discrete sum over the set of particles of the particle filter. Compared to traditional particle filtering, delayed sampling performs exact computations when possible. Thus, we extend values v with symbolic terms. Symbolic terms include random variables (X) — the nodes of the delayed sampling graph — and applications of operators.

$$v \quad ::= \quad ... \mid X \mid \text{app}(op, v)$$

The semantics in Figure 9 rely on the following high-level operations to update the graph g.

$v', g' = value(v, g)$ samples all the random variables in v to produce a concrete value.
$g' = observe(X, v, g)$ conditions the graph on the fact that the random variable X takes the value v.
$X, g' = assume(d, g)$ adds and returns a new random variable X with distribution d.

Probabilistic semantics. The semantics of a probabilistic expressions are defined in Figure 9. The semantics of an expression $\{\!\{e\}\!\}_{Y,g,w}$ takes two additional arguments: g, the delayed sampling graph, and w, the weight for the particle filter, and returns a symbolic value, an updated graph, and an updated weight. Operator application $op(v)$ introduces a symbolic expression $\text{app}(op, v)$. `if` uses the *value* operation to sample a concrete value for the condition. $sample(v)$ introduces a new random variable in the graph with distribution v. $observe(v_1, v_2)$ introduces a fresh random variable X with distribution v_1, and conditions the graph on the fact that X takes the value v_2.

Inference. Given a transition function f, a distribution over states σ from the previous iteration, and inputs v_i, the `infer` operator computes a distribution of outputs and new distribution over states for the next iteration. First, the inference draws N states from σ. Each of theses states s_n is associated with a delayed sampling graph g_n. Second, the transition function f returns a symbolic output value v_n, a new state s'_n, the updated graph g'_n, and the importance weight w_n. Third, the $distribution(o_n, g'_n)$ function returns a distribution of values without altering the graph, and the new distribution over states is a Dirac delta distribution on the pair (s'_n, g'_n). Finally, results are accumulated in a mixture distribution using the weights w_n and this distribution is split into a distribution of values and a distribution of next states.[1]

$$
\begin{aligned}
\llbracket \text{infer}(m) \rrbracket_Y \;=\; & let \; \text{stream} \{ \; \text{init} = e_{init} \; ; \; \text{step}(p_{state}, p_{input}) = e \; \}_{Y'} = \llbracket m \rrbracket_Y \; in \\
& let \; s_{init} = \llbracket e_{init} \rrbracket_{Y'} \; in \; (\delta_{(s_{init}, \emptyset)}, infer(\lambda(p_{state}, p_{input}). \; \{\!\{e\}\!\}_{Y'}))
\end{aligned}
$$

$$
\text{where } infer(f) = \lambda(\sigma, v_i). \; let \; \mu = \lambda U. \; \sum_{n=1}^{N} \; \begin{aligned} & let \; s_n, g_n = draw(\sigma) \; in \\ & let \; (o_n, s'_n), g'_n, w_n = f(s_n, v_i)(g_n, 1) \; in \\ & let \; d_n = distribution(o_n, g'_n) \; in \\ & \overline{w_n} * d_n(\pi_1(U)) * \delta_{s'_n, g'_n}(\pi_2(U)) \end{aligned}
$$
$$
in \; (\pi_{1*}(\mu), \pi_{2*}(\mu))
$$

4.2 Graph Manipulation

We now describe the graph manipulation functions that are required to define the high-level operations *value*, *assume*, and *observe* used in the semantics of Figure 9. Lundén [2017] and Murray et al. [2018] provide detailed explanations of these operations.

Notation. In this section and those that follow, $frv(v)$ denotes the *free random variables* of a program value v, i.e., the set of variables used in the symbolic expression v.

---

[1]we write $\overline{w_i} = w_i / \sum_{i=1}^{N} w_i$ for the normalized weights.

Graph Data Structure. A delayed sampling graph g is defined by a tuple (V, E, q) where V is a set of vertices – the random variables, E is a set of directed edges – the dependencies between random variables, and q is a relation mapping each node to a state: *Initialized*, *Marginalized*, or *Realized*.

A node *Initialized*$(p_{X|Y})$ represents a random variable X with a conditional distribution $p_{X|Y}$ where Y is the unique parent of X. A node *Marginalized*(p_X) represents a random variable X with a marginal distribution p_X. A *Marginalized* node has at most one parent. If there is a parent node, the distribution p_X incorporates its distribution. A node *Realized*(v) represents a random variable X associated to a concrete value v. By construction, a delayed sampling graph is a forest – a set of trees (each node has at most one parent).

value. The operation *value*(v, g) converts the symbolic expression v into a concrete value by sampling all the random variables in v. All these random variables become *Realized* nodes in the graph, and the distributions depending on these variables are updated.

$$
\begin{array}{lll}
value(v, g) & = & (v, g) \quad \text{if } v \text{ is a concrete value} \\
value(\mathrm{app}(op, v), g) & = & let\ v', g' = value(v, g)\ in\ (op(v'), g') \\
value(X, g) & = & let\ V, E, q = g\ in \\
& & if\ q(X) = Realized(v)\ then\ (v, g) \\
& & else\ let\ V', E', q'[X \leftarrow Marginalized(\mu)] = graft(X, g)\ in \\
& & \quad let\ v = draw(\mu)\ in \\
& & \quad (v, (V', E', q'[X \leftarrow Realized(v)]))
\end{array}
$$

If v is already a concrete value, there is nothing to do. If v is the application of an operator, *value* recursively samples a concrete value for the argument and applies the operator to this value. If v is a random variable X that is already realized, *value* returns the corresponding value. Otherwise, *value* (1) calls the *graft* function defined in Appendix C to marginalize X and all its ancestors, (2) draws a sample from the marginalized distribution, and (3) returns this value and turns X into a *Realized* node. Note that *graft* might have to realize some nodes since it marginalizes all its ancestors and a marginal node has a single marginalized child. During marginalization, *graft* also removes edges between *Marginalized* nodes and their *Realized* child if any.

assume. The operation *assume*(v, g) adds a new random variable X with distribution v in graph g.

$$
\begin{array}{ll}
assume(v, g) \quad = & let\ (V, E, q) = g\ in \\
& let\ X = fresh(V)\ in \\
& if\ frv(v) = \emptyset\ then\ (X, (V \cup \{X\}, E, q[X \leftarrow Marginalized(v)])) \\
& else\ if\ frv(v) = \{Y\}\ \wedge\ conj(v, Y, g)\ then \\
& \quad (X, (V \cup \{X\}, E \cup \{(X, Y)\}, q[X \leftarrow Initialized(v)])) \\
& else \\
& \quad let\ v', (V,' E', q') = value(v, (V \cup \{X\}, E, q))\ in \\
& \quad (X, (V', E', q'[X \leftarrow Marginalized(v')]))
\end{array}
$$

The distribution v is a symbolic expression which can be a marginal distribution that does not depends on other random variables — e.g., app($bernoulli$, 0.5)— or a conditional distribution — e.g., app($bernoulli$, Y) where Y is a random variable. If v is a marginal distribution, *assume* just adds a new marginalized node in the graph. If v is a conditional distribution, *assume* tries to keep track of the dependency between X and a random variable used in v (the delayed sampling graph is a forest where each node has at most one parent).

The value v thus represents a distribution $p_{X|Y}$ where X depends on a unique random variable Y. If the distribution $p_{X|Y}$ and p_Y are conjugate ($conj(v, Y, g)$) — e.g., app($bernoulli$, Y) with $Y \sim beta(\alpha, \beta)$ — marginalization and conditioning are tractable operations, and *assume* adds an edge

between Y and a new initialized node X to the graph. Otherwise, symbolic computation is not possible; *assume* calls *value* to sample a concrete value, thus breaking the dependency, and adds a new independent *Marginalized* node to the graph.

observe. The operation *observe*(X, v, g) assigns the concrete value v to X and updates the distributions depending on X accordingly.

$$observe(X, v, g) \quad = \quad let\ (V, E, q) = graft(X, g)\ in\ (V, E, q[X \leftarrow Realized(v)])$$

Similarly to *value*, the *observe* operation uses the function *graft* to marginalize the variable X and then turns X to a *Realized* node associated with the value v.

4.3 Memory Usage

Baudart et al. [2020] proposed an implementation of delayed sampling where an *Initialized* node only has a pointer to its parent, a *Marginalized* node only has a pointer to its unique *Marginalized* or *Realized* child, if any, and a *Realized* node has no pointers to its parent or any of its children.

Garbage Collection. A node in the delayed sampling graph can be safely removed if none of the program variables depend on its value. We assume the existence of a garbage collection routine that deallocates the nodes of the graph that are not *reachable* as soon as possible.

Definition 4.1 (Reachability). Given a set of root variables r and a delayed sampling graph $g = (V, E, q)$, the set of *reachable variables* – written *reachable*(g, r) – is defined as follows:

$$
\begin{aligned}
R \quad &= \quad \{(X, Y) \mid \big((X, Y) \in E \wedge q(X) = Initialized\big) \vee \big((Y, X) \in E \\
&\qquad \wedge\ q(X) = Marginalized \wedge (q(Y) = Marginalized \vee q(Y) = Realized)\big) \\
reachable(g, r) \quad &= \quad \{Y \mid (R^*(X, Y)) \wedge X \in r \wedge Y \in V\}
\end{aligned}
$$

where R^* denotes the reflexive transitive closure of the relation R.

If we consider the graph in Figure 2b, *reachable*$(g, \{\texttt{x}\}) = \{\texttt{x}\}$. In the example of Figure 4b, we have *reachable*$(g, \{\texttt{i}, \texttt{x}\}) = \{\texttt{i}, \texttt{pre\_x}, \texttt{x}\}$, where $\texttt{pre\_x}$ is the gray node in between the nodes for $\texttt{i}$ and $\texttt{x}$. Reachability is the core property used in Definition 5.1 to define what it means for a program to run in bounded memory.

Graph Expansion. The only operation that increases the size of the graph is *assume* which introduces new nodes. The operations *value* and *observe* can only marginalize and realize nodes. If g' is the graph resulting from the application of *value* or *observe* on a graph g, g and g' have the same structure but *Initialized* nodes can be *Marginalized* or *Realized*, and *Marginalized* nodes can be *Realized*. The reachability relation of the graph implies that *value* and *observe* reduce the number of dependencies in the delayed sampling graph, that is, *reachable*$(g', r) \subseteq$ *reachable*(g, r).

Initialized and Marginalized Chains. Two patterns can yield unbounded memory consumption. First, it is possible to keep adding nodes without realizing them (via observation or sampling), thus forming *initialized chains*. An initialized chain is a sequence of initialized nodes, each of which holds a pointer to its parent and thereby expands the number of random variables that are reachable. Second, it is possible that nodes are only indirectly used to realize one of their children. These marginalized nodes can form *marginalized chains*. A marginalized chain is a sequence of marginalized nodes, each of which holds a pointer to its child and thus expands the number of random variables that are reachable. The last node of a marginalized chain may be realized.

5 SEMANTIC PROPERTIES

In this section, we define conditions under which delayed sampling executes in bounded memory. We define these conditions as properties of *executions*. An execution is a sequence of pairs of a state and a delayed sampling graph $(s_n, g_n)_{n \in \mathbb{N}}$, where each state is a semi-symbolic value as defined in Section 4.1. An execution defines the sequence of states and graphs a model — i.e., an argument of an `infer` — goes through.

The inference step function $infer(f)$ in $\llbracket \texttt{infer}(m) \rrbracket$ may operate over multiple executions of f (see Section 4.1). However, $infer(f)$ executes in bounded memory if every execution of f is bounded-memory. This is because $infer(f)$ always updates its state by mapping f over states and graphs from the distribution at the previous iteration. Thus, any state and graph in the distribution at the next iteration must have come from *some* execution of f, and if all executions of f are bounded-memory, all states and graphs in the distribution must have bounded memory. We have formalized this in more details in Appendix E.1.

Based on this notion of execution, we introduce two notions of bounded-memory executions of delayed sampling, and semantic properties which are necessary and sufficient for bounded-memory execution. In Section 5.1 we present a low-level definition of bounded memory that directly corresponds to how the delayed sampling runtime executes. In Section 5.2 we present an alternative high-level definition in terms of dataflow properties of the high-level delayed sampling operators: the *m-consumed* and *unseparated paths* properties. In Section 5.3 we show that the high-level and low-level formulations are equivalent. In particular, Section 5.3 shows a correspondence between the *m*-consumed property and a bound on the length of initialized chains, as well as a correspondence between the unseparated paths property and a bound on the length of marginalized chains.

5.1 Low-Level Bounded Memory

A program executes in bounded memory if the delayed sampling graph maintains a bounded number of reachable variables over time. We formalize this as follows:

Definition 5.1 (Low-level Bounded-Memory). An execution $(s_n, g_n)_{n \in \mathbb{N}}$ of a model is *low-level bounded-memory* if

$$\exists k. \ \forall n \geq 0 \ |reachable(g_n, s_n)| \leq k * |frv(s_n)|$$

This definition states that at each iteration, the size of the set of reachable nodes in the delayed sampling graph may be at most a constant multiple of the number of free random variables in the state. We do not consider the runtime to violate bounded memory in the trivial case that the program state is intrinsically unbounded, i.e., when $|frv(s_n)|_{n \in \mathbb{N}}$ is unbounded. Such a program would not execute in bounded memory under any inference algorithm; even a particle filter would require unbounded memory to store the program state.

5.2 High-Level Definitions

In this section, we present an alternative high-level definition of bounded memory that is easier to reason about. The high-level definition is in terms of dataflow properties of delayed sampling operations. We have formalized these dataflow properties by augmenting the delayed sampling operations with tracing. A trace is defined as follows:

$$\tau ::= \tau :: \tau_1 \mid \texttt{nil}$$
$$\tau_1 ::= X \leftsquigarrow X \mid X \leftsquigarrow \texttt{nil} \mid \texttt{eval}(X) \mid \texttt{obs}(X)$$

$$\overline{assume}(v, (g, \tau)) \quad = \quad let\ X', g' = assume(v, g)\ in$$
$$\begin{cases} X', (g', \tau :: X' \twoheadleftarrow \text{nil}) & frv(v) = \emptyset \\ X', (g', \tau :: X' \twoheadleftarrow X) & \{X\} = frv(v) \wedge conj(v, X, g) \\ X', (g', \tau :: \text{eval}(frv(v)) :: X' \twoheadleftarrow \text{nil}) & otherwise \end{cases}$$

$$\overline{value}(v, (g, \tau)) \quad = \quad let\ (v', g') = value(v, g)\ in\ v', (g', \tau :: \text{eval}(frv(v)))$$

$$\overline{observe}(X, v, (g, \tau)) \quad = \quad observe(X, v, g), (\tau :: \text{obs}(X))$$

Fig. 10. Tracing semantics of delayed sampling operators.

A trace is a list of primitive operations, where each primitive is one of:

- Assumption, written $X \twoheadleftarrow X'$ when X is assumed from another random variable X' or $X \twoheadleftarrow \text{nil}$ when it is assumed without a parent.
- Evaluation using the eval keyword, which refers to evaluating a set of random variables $\mathcal{X}$.
- Observation using the obs keyword, which refers to observing a random variable X.

We define an augmented semantics that operates on a pair of a delayed sampling graph and a trace. Figure 10 defines augmented versions of the *assume*, *value*, and *observe* operations, and the full semantics (written $\overline{[\![\cdot]\!]}$ and $\overline{\{\!\{\cdot\}\!\}}$) is defined by replacing these operators in Figure 9 with their traced counterparts from Figure 10.

The m-consumed Property. The *m-consumed property* is used to enforce that every variable introduced with *assume* is eventually consumed either by directly being passed to a *value* or *observe* or transitively by being passed to a *assume* that introduces a variable that is also *m*-consumed.

Definition 5.2 (m-consumed). A variable X is *m*-consumed in a trace τ under the following circumstances:

- X is 0-consumed if it is observed or evaluated (i.e., τ has eval($\mathcal{X}$) where $X \in \mathcal{X}$ or obs(X)).
- X is 0-consumed if it is never used (i.e., there is no $X' \twoheadleftarrow X$, eval(X), or obs(X) in τ).
- X is *m*-consumed if it is passed to the *assume* statement that introduces another variable X' (i.e., $X' \twoheadleftarrow X$ is in τ), and X' is $(m-1)$-consumed.

The Unseparated Paths Property. The *unseparated paths property* states the existence of a sequence of variables, each assumed from the previous, with no variable in the sequence observed or evaluated.

Definition 5.3 (Unseparated Paths). An unseparated path in τ is a sequence of variables $X_0, X_1, \dots, X_n$ such that each X_{i+1} was assumed from X_i (i.e., $X_{i+1} \twoheadleftarrow X_i$ is in τ) and no X_i is directly observed or evaluated (i.e., τ does not contain any eval or obs operations that reference X_i).

High-level Bounded Memory. We now present the high-level bounded memory property. This property states that all variables must eventually be *m*-consumed, and there must be a uniform bound across iterations on the length of an unseparated path starting from a program state variable.

Definition 5.4 (High-level Bounded-Memory). A program execution $(s_n, (g_n, \tau_n))_{n \in \mathbb{N}}$ is *high-level bounded-memory* if and only if

- There exists an m such that for every iteration n and every variable introduced before n (i.e., X such that $X \twoheadleftarrow X'$ or $X \twoheadleftarrow \text{nil}$ is in τ_n), there exists a $n' \geq n$ such that for all $n'' \geq n'$, X is *m*-consumed in $\tau_{n''}$.
- There exists a c such that for all n, no random variable referenced in s_n starts an unseparated path in τ_n of length more than c.

5.3 Equivalence of Low-Level and High-Level Definitions

In this section, we show the equivalence of the low-level and high-level definitions. We do so by showing that both properties are equivalent to the delayed sampling graph having a uniform bound (i.e., a bound that holds across all iterations) on the length of initialized and marginalized chains as defined in Section 4.3.

5.3.1 Low-Level Bounded Memory vs. Infinite Chains.

Lemma 5.5. *If the delayed sampling graph is constructed using assume, observe, and value operations, then each random variable starts either an initialized chain, a marginalized chain, or an initialized chain followed by a marginalized chain.*

Proof. The *assume*, *observe*, and *value* operations can only make the following modifications to a delayed sampling graph g. (1) Add a independent *Marginalized* node which creates a marginalized chain of length zero. (2) Attach a new *Initialized* node X to a node Y with a conjugate distribution. It means that Y is either *Initialized* or *Marginalized* and thus it creates either a longer initialized chain or an initialized chain followed by a marginalized chain. (3) Perform a *graft* which ensures that every ancestor of a node is marginalized and has a single marginalized child. Every non-ancestor variable is either as it was before or becomes realized, so this operation preserves the structure of the previous graph and cannot increase the length of the chains. (4) convert a *Marginalized* node into a *Realized* node which can only break a chain.   □

Theorem 5.6. *A program is low-level bounded-memory iff there is a uniform bound m on the length of an initialized chain and a uniform bound c on the length of a marginalized chain.*

Proof. Assuming a uniform bound, when the number of variables is bounded by N, according to Lemma 5.5, the number of reachable nodes in the graph is bounded by $N \times (c + m)$.

Conversely, if no uniform bound exists (i.e., for every potential bounds c and m, there exists a iteration n such that chains may exceed the bound at n), the execution cannot be low-level bounded-memory, because even if the number of root variables is bounded by N, the reachable variables may exceed $N \times (c + m)$.   □

5.3.2 High-Level Bounded Memory vs. Infinite Chains.

Theorem 5.7 (High-level Soundness). *In a program execution that is high-level bounded-memory, no infinite chains can exist in any of the delayed sampling graphs.*

Proof. All initialized chains must be shorter than m, where m is from the m-consumed property of high-level bounded-memory. This is because when a variable's descendant is subject to *observe* or *value*, the variable becomes marginalized. Such a descendant can be at most m variables away because of the definition of m-consumed.

All marginalized chains must be shorter than $c+m$, where c is from the unseparated path property of high-level bounded-memory and m is from the m-consumed property. By Lemma 5.5, every marginalized chain must start at either a root or an initialized chain. If it starts at a root, the unseparated path property ensures that the path between the root and the end of the chain can contain at most c variables. This is because any observed or valued variables become realized and become the end of the chain. If it starts at an initialized chain, by the above reasoning that chain has length at most m, and there was a previous iteration at which the marginalized chain started at a root and had length at most c, giving an overall length of at most $c + m$.   □

Lemma 5.8. *If there exists a variable that is not m-consumed, then the program produces a graph at some iteration with an initialized chain of length m.*

Proof. If a variable is not m-consumed, then by the definition of m-consumed must start an assume chain of length m. All of the nodes in this chain must be initialized, and therefore form an initialized chain of length m. □

Lemma 5.9. *If every variable is m-consumed, and there exists a variable that starts an unseparated path of length c where $c > m$, then there exists an iteration with a marginalized chain that has length at least $c - m$.*

Proof. Note that the first $c - m$ variables in the unseparated path must be either marginalized or realized. Otherwise, there would be more than m initialized variables in the tail of the unseparated path that are initialized, which would violate soundness of m-consumed. Let X be the variable that starts the unseparated path and X' be the last marginalized or realized variable in the unseparated path, and consider the iteration n' when X' was first marginalized. It must be true that (1) X is in the program state at iteration n' because it is in the state at the current iteration $n > n'$, and (2) a marginalized chain runs from X to X'. Thus, at n', the marginalized chain had length $c - m$. □

Theorem 5.10 (High-level Completeness). *If a program execution is not high-level bounded-memory, the delayed sampling graph has either unbounded initialized chains or marginalized chains.*

Proof. If the execution is not high-level bounded-memory, it either fails the m-consumed property or the unseparated path property. If it fails the m-consumed property, apply Lemma 5.8. Otherwise, apply Lemma 5.9. □

Theorem 5.11. *A program execution is high-level bounded-memory if and only if it is low-level bounded-memory.*

Proof. Apply Theorems 5.6, 5.7, and 5.10. □

6 ANALYSIS

In this section, we develop an analysis to check that a μF program executes in bounded memory. We approach this problem by developing two independent analyses within a shared analysis framework. One analysis checks the m-consumed property of a program and the other checks the unseparated paths property, which together ensure that the program executes in bounded memory (Section 5).

Our shared analysis framework abstracts the execution of a program as the execution of abstract operations on an *abstract graph*. An abstract graph abstracts the dynamic state of a program's delayed sampling graph. We implement the analysis framework by means of a type system, such that well-typed programs satisfy the m-consumed and unseparated paths properties, given each analysis's respective instantiation of the abstract graph. The typing judgment

$$\Gamma, \mathcal{G} \vdash_\alpha e : t, \mathcal{G}'$$

asserts that in a *context* Γ, and for an abstract graph $\mathcal{G}$, that an expression e accesses the random variables denoted by the *type* t and yields a new abstract graph $\mathcal{G}'$. The parameter α is either mc to denote the m-consumed analysis or up to denote the unseparated paths. We write $\Gamma \vdash_\alpha e : t$ as shorthand for $\Gamma, \mathcal{G} \vdash_\alpha e : t, \mathcal{G}$ when e has no effect on the graph.

6.1 Types and Contexts

A *type t* captures the random variables the expression could refer to as well as its shape, as primitive data, a product, a function, or a stream instance.

$$t ::= r \mid () \mid t_1 \times t_2 \mid t_1 \rightarrow t_2 \mid \mathsf{stream}(t, s) \mid \mathsf{bounded}$$
$$s ::= \mathsf{stepfn}(p_{state}, p_{in}, \Gamma_e, e)$$

The type of a primitive expression is a *reference set*, denoted r, which specifies the random variables to which the expression refers. We distinguish two types of stream instances, before and after bounded-memory checking. The first is $\text{stream}(t, s)$, where t is the type of the current state and s is a step function representation to be described later. The second is bounded, representing instances that have passed bounded memory analysis and hide their inner structure.

Reference Sets. A *reference set* of a μF expression, denoted r, specifies the random variables that are affected when the expression is observed or evaluated. In the presence of branches, we define r to be a pair of sets (lb, ub), where the lower bound lb contains all random variables which *must* be affected and the upper bound ub all random variables which *may* be affected. For example, a constant value in μF such as 1.5 has the reference set $(\emptyset, \emptyset)$ because it references no random variables. If the program variables x and y correspond to random variables X and Y respectively, then the expression gaussian(x,y), specifying a distribution with two parameters, has reference set $(\{X, Y\}, \{X, Y\})$, meaning that observing it will observe the random variables X and Y.

Contexts. The context $\Gamma, x : t$ maps variable x to type t. As μF syntactic patterns p may be variables or pairs, we use the shorthand $\Gamma, p : t$ to define types for variables in p by structural correspondence with t, as defined by the first rule below. We also define a judgment $\vdash_p p : t$ that synthesizes a deterministic type t from a pattern p.

$$\frac{\Gamma, p_1 : t_1, p_2 : t_2 \vdash_\alpha e : t}{\Gamma, (p_1, p_2) : t_1 \times t_2 \vdash_\alpha e : t} \qquad \frac{}{\vdash_p x : (\emptyset, \emptyset)} \qquad \frac{\vdash_p p_1 : t_1 \qquad \vdash_p p_2 : t_2}{\vdash_p (p_1, p_2) : t_1 \times t_2}$$

6.2 Abstract Graphs

An *abstract graph* $\mathcal{G}$ is an abstraction of the delayed sampling graph that tracks which random variables have been consumed and active paths between random variables, properties relevant to the semantic properties. For each analysis α there exists an abstract graph type, $\mathcal{G}$, and a set of operations that form its interface (Figure 11).

Specifically, in the m-consumed analysis we define $\mathcal{G}$ to be a pair of sets *in* and *con* which respectively represent an over-approximation of variables introduced into the graph and an under-approximation of the variables consumed by observation or sampling (Figure 12). In the unseparated paths analysis, we define $\mathcal{G}$ to be a set *sep* of separators containing consumed random variables and a partial path function p mapping a pair of random variables to an upper bound on the length of an unseparated path between them (Figure 13).

Operations on the abstract graph manipulate random variables, graphs, and reference sets. The function *assume* returns a new graph with a random variable X from a distribution with reference set r added to $\mathcal{G}$, *observe* returns a graph where X is observed with a value with reference set r, and *value* returns a graph where an expression with reference set r is evaluated. The join operator $\sqcup_\alpha$ represents a conservative choice between two graphs.

m-consumed Graph Operations. In Figure 12, $assume_{mc}(X, r, \mathcal{G})$ marks the random variable X as introduced. In all cases, the lower bound of random variables in the input is marked consumed. To join two states, we union the introduced variables and intersect the consumed variables.

Unseparated Paths Graph Operations. In Figure 13, $observe_{up}$ and $value_{up}$ mark input variables as separators. In $assume_{up}$, we set the length of the path from the new variable X to itself to zero. For a parent X_p that is not a separator, we set the length of the path from any variable X_i to X to one more than the length from X_i to X_p. To join two states, we intersect the separators and take the maximum length between the results of the two path functions (where defined, or 0 otherwise).

$$\mathcal{G} ::= \{in \subseteq \text{RV}; con \subseteq \text{RV}\}$$

$$assume_\alpha : \text{RV} \to r \to \mathcal{G} \to \mathcal{G} \qquad assume_{mc}(X, r, \mathcal{G}) = \{\mathcal{G}.in \cup \{X\}; \mathcal{G}.con \cup r.lb\}$$

$$observe_\alpha : \text{RV} \to r \to \mathcal{G} \to \mathcal{G} \qquad observe_{mc}(X, r, \mathcal{G}) = \{\mathcal{G}.in; \mathcal{G}.con \cup r.lb \cup \{X\}\}$$

$$value_\alpha : r \to \mathcal{G} \to \mathcal{G} \qquad value_{mc}(r, \mathcal{G}) = \{\mathcal{G}.in; \mathcal{G}.con \cup r.lb\}$$

$$\sqcup_\alpha : \mathcal{G} \to \mathcal{G} \to \mathcal{G} \qquad \mathcal{G}_1 \sqcup_{mc} \mathcal{G}_2 = \{\mathcal{G}_1.in \cup \mathcal{G}_2.in; \mathcal{G}_1.con \cap \mathcal{G}_2.con\}$$

Fig. 11. Abstract graph interface.   Fig. 12. m-consumed abstract graph operations.

$$\mathcal{G} ::= \{p : \text{RV} \times \text{RV} \hookrightarrow \mathbb{N}; sep \subseteq \text{RV}\}$$

$$assume_{up}(X, r, \mathcal{G}) = \{p'; \mathcal{G}.sep\} \text{ where } p'(X, X) \mapsto 0,$$
$$p'(X_i, X) \mapsto \mathcal{G}.p(X_i, X_p) + 1 \text{ for all } X_p \in r.ub \setminus \mathcal{G}.sep, X_i \in \text{RV},$$
$$p'(X, Y) \mapsto \mathcal{G}.p(X, Y) \text{ otherwise}$$

$$observe_{up}(X, r, \mathcal{G}) = \{\mathcal{G}.p; \mathcal{G}.sep \cup r.lb \cup \{X\}\}$$
$$value_{up}(r, \mathcal{G}) = \{\mathcal{G}.p; \mathcal{G}.sep \cup r.lb\}$$
$$\mathcal{G}_1 \sqcup_{up} \mathcal{G}_2 = \{p'; \mathcal{G}_1.sep \cap \mathcal{G}_2.sep\} \text{ where } p'(v_1, v_2) \mapsto \max(\mathcal{G}_1.p(v_1, v_2), \mathcal{G}_2.p(v_1, v_2))$$

Fig. 13. Unseparated paths abstract graph operations.

6.3 Typing Rules

In Figure 14 we present the typing rules that are relevant to analyzing probabilistic streams, with the full definition in Appendix D. Constants reference no random variables. `sample` introduces a fresh random variable sampled from its argument and adds it to the graph. `observe` introduces an intermediate random variable for its first argument by the same mechanism as `sample`, and observes it to be the evaluation of its second argument.

Operators and Scalar Folding. We use μF operators op to describe probability distributions and other operations over scalars and assume them to have scalar return values. The auxiliary judgment $\searrow$ folds products and stream instances into scalars by taking unions of variable sets.

$$\frac{}{() \searrow (\emptyset, \emptyset)} \qquad \frac{}{r \searrow r} \qquad \frac{t_1 \searrow (lb, ub) \quad t_2 \searrow (lb', ub')}{t_1 \times t_2 \searrow (lb \cup lb', ub \cup ub')} \qquad \frac{t \searrow (lb, ub)}{\text{stream}(t, s) \searrow (lb, ub)} \qquad \frac{}{\text{bounded} \searrow (\emptyset, \emptyset)}$$

Sequencing. Sequencing using the `let`-expression follows the standard typing rule for `let`, and also threads the output graph of evaluating e into the evaluation of e'.

$$() \sqcup () = ()$$
$$(t_1 \times t_2) \sqcup (t_1' \times t_2') = (t_1 \sqcup t_1') \times (t_2 \sqcup t_2')$$
$$(lb, ub) \sqcup (lb', ub') = (lb \cap lb', ub \cup ub')$$

Fig. 15. Join operator for types.

Conditionals and Join. `if`-expressions evaluate the condition, check both branches in parallel, and join the resulting reference set and graphs. The join operator $\sqcup$ (Figure 15), representing the conservative union of two types, unions the upper bounds and intersects the lower bounds. We disallow `if`-branching over functions and stream instances.

$$\frac{}{\Gamma \vdash_\alpha c : (\emptyset, \emptyset)} \qquad \frac{\Gamma \vdash_\alpha v : r \qquad X = \mathsf{fresh}(\mathcal{G})}{\Gamma, \mathcal{G} \vdash_\alpha \mathsf{sample}(v) : (\{X\}, \{X\}), assume_\alpha(X, r, \mathcal{G})}$$

$$\frac{\Gamma, \mathcal{G} \vdash_\alpha \mathsf{sample}(v_1) : (\{X\}, \{X\}), \mathcal{G}' \qquad \Gamma \vdash_\alpha v_2 : r_2}{\Gamma, \mathcal{G} \vdash_\alpha \mathsf{observe}(v_1, v_2) : (), observe_\alpha(X, r_2, value_\alpha(r_2, \mathcal{G}'))} \qquad \frac{\Gamma \vdash_\alpha v : t \qquad t \searrow r}{\Gamma \vdash_\alpha op(v) : r}$$

$$\frac{\Gamma, \mathcal{G} \vdash_\alpha e : t, \mathcal{G}' \qquad \Gamma, p : t, \mathcal{G}' \vdash_\alpha e' : t', \mathcal{G}''}{\Gamma, \mathcal{G} \vdash_\alpha \mathsf{let}\ p = e\ \mathsf{in}\ e' : t', \mathcal{G}''}$$

$$\frac{\Gamma \vdash_\alpha v : r \qquad \mathcal{G}' = value_\alpha(r, \mathcal{G}) \qquad \Gamma, \mathcal{G}' \vdash_\alpha e_1 : t_1, \mathcal{G}_1 \qquad \Gamma, \mathcal{G}' \vdash_\alpha e_2 : t_2, \mathcal{G}_2}{\Gamma, \mathcal{G} \vdash_\alpha \mathsf{if}\ v\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 : t_1 \sqcup t_2, \mathcal{G}_1 \sqcup_\alpha \mathcal{G}_2}$$

$$\frac{\Gamma \vdash_\alpha m : (t, s)}{\Gamma \vdash_\alpha \mathsf{init}(m) : \mathsf{stream}(t, s)}$$

$$\frac{\Gamma \vdash_\alpha x : \mathsf{stream}(t, \mathsf{stepfn}(p_{state}, p_{in}, \Gamma_e, e)) \qquad \Gamma \vdash_\alpha v : t_{in} \qquad \Gamma_e, p_{state} : t, p_{in} : t_{in}, \mathcal{G} \vdash_\alpha e : t' \times t_{out}, \mathcal{G}'}{\Gamma, \mathcal{G} \vdash_\alpha \mathsf{unfold}(x, v) : t_{out} \times \mathsf{stream}(t', \mathsf{stepfn}(p_{state}, p_{in}, \Gamma_e, e)), \mathcal{G}'}$$

$$\frac{\Gamma \vdash_{mc} m\ \mathsf{bounded} \qquad \Gamma \vdash_{up} m\ \mathsf{bounded}}{\Gamma \vdash_\alpha \mathsf{infer}(m) : \mathsf{bounded}} \qquad \frac{\Gamma \vdash_\alpha x : \mathsf{bounded} \qquad \Gamma \vdash_\alpha v : t \qquad t \searrow (\emptyset, \emptyset)}{\Gamma \vdash_\alpha \mathsf{unfold}(x, v) : (\emptyset, \emptyset) \times \mathsf{bounded}}$$

Fig. 14. Delayed sampling type system.

Streams and Inference. To facilitate typing of stream functions, we define the following auxiliary judgment, which computes, for a stream function, the type of its initial state and the syntactic fragment for its step function.

$$\frac{\Gamma \vdash_\alpha e' : t_{init} \qquad t_{init} \searrow (\emptyset, \emptyset)}{\Gamma \vdash_\alpha \mathsf{stream}\ \{\ \mathsf{init} = e'\ ;\ \mathsf{step}(p_{state}, p_{in}) = e\ \} : (t_{init}, \mathsf{stepfn}(p_{state}, p_{in}, \Gamma, e))}$$

Correspondingly, we define the context $\Gamma, m : (t_{init}, \mathsf{stepfn}(p_{state}, p_{in}, \Gamma_e, e))$ to map the stream function name m to its initial state type and step function.

Instances that are created by `init` expose the type of their internal state and their step function. The `unfold` rule applies the step function to the current state, yielding an output and an instance with the new state. It ensures that the argument v is compatible with the type of the step function.

An `infer` expression marks the entry point of a new sub-analysis for its new delayed sampling graph. The premises of the typing rule for `infer` are the *success conditions* for both analyses that must hold regardless of α. This judgment, $\Gamma \vdash_\alpha m$ bounded, states that the stream function m can be unfolded for an arbitrary number of iterations while satisfying property α starting with an empty delayed sampling graph.

Instances created by `infer` possess a newly instantiated delayed sampling graph. Their internal state contains the delayed sampling graph and bookkeeping information for the inference algorithm. Thus, the state is hidden to the exterior and the instance is assigned the opaque type bounded. `unfold` on a bounded type only requires that the input and output are purely deterministic.

m-consumed Success Condition. We conclude a stream function passes the *m*-consumed analysis when all variables that are introduced are consumed by the program. Because an introduced variable may take several stream iterations to be consumed, we repeatedly execute the analysis until we

consume all variables and succeed or reach a fixed point and fail. Define the iteration judgment $\Gamma \vdash_{\alpha(n)} m : t', \mathcal{G}$, where α is either mc or up, as follows:

$$\frac{\Gamma \vdash_\alpha m : (t, \mathtt{stepfn}(p_{state}, p_{in}, \Gamma_e, e)) \qquad \vdash_p p_{in} : t_{in} \qquad \Gamma_e, p_{in} : t_{in}, p_{state} : t, \bot_\alpha \vdash_\alpha e : t_{out} \times t', \mathcal{G}}{\Gamma \vdash_{\alpha(0)} m : t', \mathcal{G}}$$

$$\frac{\begin{array}{c} \Gamma \vdash_\alpha m : (t, \mathtt{stepfn}(p_{state}, p_{in}, \Gamma_e, e)) \qquad \Gamma \vdash_{\alpha(n-1)} m : t', \mathcal{G} \\ \vdash_p p_{in} : t_{in} \qquad \Gamma_e, p_{in} : t_{in}, p_{state} : t', \mathcal{G} \vdash_\alpha e : t_{out} \times t'', \mathcal{G}' \end{array}}{\Gamma \vdash_{\alpha(n)} m : t'', \mathcal{G}'}$$

On each iteration, this judgment applies the appropriate type rule for the step function and returns the result, using the abstract graph from the previous iteration as the context for the step function rule. The initial iteration uses an empty abstract graph as the context, represented by $\bot_\alpha$. For the m-consumed analysis, we specialize the judgment to $\Gamma \vdash_{mc(n)} m : t', \mathcal{G}$, and define $\bot_{mc}$ to be $(\emptyset, \emptyset)$.

The rule continues iterating until it reaches the success condition. The success condition states that every variable introduced that is kept in the program state must be used with in a bounded number of time steps. We formalize this as the following type rule:

$$\frac{\Gamma \vdash_{mc(0)} m : t, \mathcal{G} \qquad t \searrow (lb, ub) \qquad \Gamma \vdash_{mc(n)} m : t'', \mathcal{G}' \qquad (\mathcal{G}.in \setminus \mathcal{G}'.con) \cap ub = \emptyset}{\Gamma \vdash_{mc} m \text{ bounded}}$$

Alternatively, if evaluating one more iteration does not consume any more variables, we reach a fixed point and return failure. Since every iteration we either consume a variable or reach a fixed point, the analysis is guaranteed to terminate.

Unseparated Paths Success Condition. Like the m-consumed analysis, the unseparated paths analysis is iterative, and we may need to repeat it for some number of iterations. We specialize the iteration judgment defined in the previous section to $\Gamma \vdash_{up(n)} m : t, \mathcal{G}$ and define $\bot_{up}$ to be a pair of an empty map and an empty set. Define $\mathrm{path}(t, \mathcal{G})$ where $t \searrow (lb, ub)$ to be the length of the longest path from any random variable in ub to any other variable in $\mathcal{G}.p$. Then we conclude the program passes the unseparated path analysis when the length of the longest path converges after some finite number of iterations:

$$\frac{\Gamma \vdash_{up(n)} m : t, \mathcal{G} \quad \Gamma \vdash_{up(n+(\mathrm{path}(t,\mathcal{G})*\mathrm{size}(t))+1)} m : t'', \mathcal{G}' \quad \mathrm{path}(t, \mathcal{G}) = \mathrm{path}(t'', \mathcal{G}')}{\Gamma \vdash_{up} m \text{ bounded}}$$

The implementation of this rule repeatedly computes a new abstract graph starting from the previous iteration's output. It exits when the longest path length at the current iteration is equal to the longest path after $(\mathrm{path}(t, \mathcal{G}) * \mathrm{size}(t)) + 1$ additional iterations. The function size determines, for a given type t, how many values of base type are contained in t.

$$\mathrm{size}(r) = 1 \qquad\qquad \mathrm{size}(t_1 \times t_2) = \mathrm{size}(t_1) + \mathrm{size}(t_2)$$

The extra iterations ensure that the path length has stabilized and the analysis can safely conclude that there is a bound on the length of the longest unseparated path.

If the path length check fails, the implementation keeps iterating until a pre-specified bound is reached. Upon reaching this bound, the implementation outputs an analysis failure. Note that the analysis may be imprecise and reject correct programs if the bound is not sufficiently high.

6.4 Example Type Derivation

This section presents an example type derivation used by the analysis to confirm that the program in Figure 1 satisfies the m-consumed property. In particular, we confirm that the stream function kalman passes the m-consumed analysis.

Using e as shorthand for the body of its step function, we derive the following m-consumed success condition for kalman:

$$\frac{\Gamma \vdash_{mc(0)} \text{kalman} : ((\{X\},\{X\}) \times (\{X\},\{X\})) \qquad ((\{X\},\{X\}) \times (\{X\},\{X\})) \searrow (\{X\},\{X\})}{\Gamma \vdash_{mc(0)} \text{kalman} : ((\{X\},\{X\}) \times (\{X\},\{X\})) \qquad (\{X\} \setminus \{X\}) \cap \{X\} = \emptyset}{\Gamma \vdash_{mc} \text{kalman bounded}}$$

The second and fourth premises follow immediately from definitions and set operations. The derivation of the first and third premises is as follows:

$$\frac{\Gamma \vdash_{mc} \text{kalman} : ((\emptyset,\emptyset), \text{stepfn}(\text{pre\_x}, \text{obs}, \Gamma_e, e)) \qquad \vdash_p \text{obs} : (\emptyset,\emptyset)}{\Gamma_e, \text{obs} : (\emptyset,\emptyset), \text{pre\_x} : (\emptyset,\emptyset), (\emptyset,\emptyset) \vdash_{mc} e : ((\{X\},\{X\}) \times (\{X\},\{X\})), (\{X\},\{X\})}{\Gamma \vdash_{mc(0)} \text{kalman} : ((\{X\},\{X\}) \times (\{X\},\{X\}))}$$

The second premise follows from definitions. The derivation of the first premise is as follows:

$$\frac{\Gamma_e \vdash_{mc} 0.0 : (\emptyset,\emptyset) \qquad (\emptyset,\emptyset) \searrow (\emptyset,\emptyset)}{\Gamma \vdash_{mc} \text{stream \{ init = 0.0 ; step(pre\_x,obs) = } e \text{ \} } : ((\emptyset,\emptyset), \text{stepfn}(\text{pre\_x}, \text{obs}, \Gamma_e, e))}$$

where the premises follow immediately. Finally, let Γ' be the context $\Gamma_e, \text{obs} : (\emptyset,\emptyset), \text{pre\_x} : (\emptyset,\emptyset)$ and t be the type $(\emptyset,\emptyset)$. The derivation of the third premise is as follows.

$$\frac{\Gamma', (\emptyset,\emptyset) \vdash_{mc} \text{sample(gaussian(pre\_x,1.0))} : t, (\{X\},\emptyset)}{\Gamma', x : t, (\{X\},\emptyset) \vdash_{mc} \text{let () = observe(gaussian(x,1.0),obs) in (x,x)} : (t \times t), (\{X\},\{X\})}{\Gamma', (\emptyset,\emptyset) \vdash_{mc} e : (t \times t), (\{X\},\{X\})}$$

The first premise follows from the typing rule for sample. The second premise follows from the typing rule for let as follows:

$$\frac{\Gamma', x : t, (\{X\},\emptyset) \vdash_{mc} \text{observe(gaussian(x,1.0),obs)} : (), (\{X\},\{X\})}{\Gamma', x : t, (\{X\},\{X\}) \vdash_{mc} \text{(x,x)} : (t \times t), (\{X\},\{X\})}{\Gamma', x : t, (\{X\},\emptyset) \vdash_{mc} \text{let () = observe(gaussian(x,1.0),obs) in (x,x)} : (t \times t), (\{X\},\{X\})}$$

where the first premise follows from the rule for observe and the second from the rule for pairs.

6.5 Soundness

Here, we outline how we show the type system is sound. We give a high-level overview of the approach; the details are in Appendix E.

Entailment Relations. In Appendix E.2, we establish several *entailment relations* that relate semantic objects to their type-level counterparts. These relations are parameterized by α which is either mc for the m-consumed relation or up for the unseparated path relation. We write $v \vDash_\alpha t$ to mean a value entails a type. We write $\gamma \vDash_\alpha \Gamma$ to mean an environment entails a type context. We write $v, (g, \tau) \vDash_\alpha t, \mathcal{G}$ to mean a value and traced graph (see Section 5.2 for the definition of a traced graph) entail a type and abstract graph. We write $\gamma, (g, \tau) \vDash_\alpha \Gamma, \mathcal{G}$ to mean an environment and traced graph entail a type context and abstract graph.

Soundness. The following theorems establish the soundness of the type system. The first theorem states that the type system soundly ascribes types to values and soundly updates the abstract delayed sampling graph:

THEOREM 6.1 (*m*-CONSUMED AND UNSEPARATED PATH SOUNDNESS). *If* $\gamma, (g, \tau) \vDash_\alpha \Gamma, \mathcal{G}$ *and* $\Gamma, \mathcal{G} \vdash_\alpha e : t, \mathcal{G}'$ *and* $\overline{\{[e]\}}_\gamma (g, \tau), w = v, (g', \tau'), w'$, *then* $v, (g', \tau') \vDash_\alpha t, \mathcal{G}'$.

Next, the type system soundly ensures a stream function maintains bounded memory.

THEOREM 6.2 (ANALYSIS SOUNDNESS). *If* $\gamma \vDash_\alpha \Gamma$ *and* $\Gamma \vdash_\alpha m$: bounded, *then* $\overline{[\![m]\!]}_\gamma \vDash_\alpha$ bounded.

We prove these theorems in Appendix E.2.

6.6 Implementation

We implemented our analysis framework and the *m*-consumed and unseparated paths analyses in OCaml. Our implementation takes as input a μF program and outputs either true or false for each analysis. It also accepts a parameter for the iteration bound for the unseparated paths analysis. The implementation goes beyond the type system laid out in the paper by supporting functions that have probabilistic effects as well as interfaces for list and array operations. μF programs can further be compiled to OCaml and executed using the ProbZelus delayed sampling runtime. The code is available at https://github.com/psg-mit/probzelus-oopsla21.

7 EVALUATION

To evaluate the ability of the analysis to accept only μF programs that can execute in bounded memory, we executed it on several benchmarks reflective of real-world inference tasks.

Research Questions. We used our implementation to answer two research questions. For realistic probabilistic programs, (1) does the type system precisely verify the properties required for bounded-memory execution, and (2) is a small iteration bound sufficient for the unseparated paths analysis?

7.1 Methodology

We executed the analysis on example programs from Baudart et al. [2020] originally written in ProbZelus, a probabilistic programming language featuring probabilistic data streams and delayed sampling. We manually translated them to μF, and they reflect a range of realistic control problems with different memory usage characteristics. For the unseparated paths analysis, we set an iteration count bound of 10, which was sufficient for these programs. We compared the outputs of the analysis to our manual logical reasoning about the ability of each of the following programs to execute in bounded memory. We provide source code for all benchmarks in Appendix F.

Kalman is the simplified core model of Figure 1 and models an agent that estimates position from noisy observations. Applying delayed sampling on this model is equivalent to a Kalman filter [Kalman 1960] where each particle returns the exact solution.

Kalman Hold-First is the example from Figure 3 with a reference to the output of the first iteration.

Gaussian Random Walk is a simplification of Kalman that does not observe of the true position, effectively expressing a Gaussian random walk.

Robot is the full example from Figure 1 that includes the Kalman core model as well as a main stream function that invokes a controller based on the inferred position.

Coin models an agent that estimates the bias of a coin. The model chooses the probability of the coin from a uniform distribution, and thereafter chooses the observations by flipping a coin with that probability. Applying delayed sampling to this model is equivalent to exact inference in a Beta-Bernoulli conjugate model [Fink 1997] where each particle returns the exact solution.

Gaussian-Gaussian estimates the mean and variance of a Gaussian distribution.

Table 1. Bounded memory analysis on benchmark programs.

	m-consumed		unsep. paths		bounded mem.	
	output	actual	output	actual	output	actual
Kalman	✓	✓	✓	✓	✓	✓
Kalman Hold-First	✓	✓	✗	✗	✗	✗
Gaussian Random Walk	✗	✗	✓	✓	✗	✗
Robot	✓	✓	✓	✓	✓	✓
Coin	✓	✓	✓	✓	✓	✓
Gaussian-Gaussian	✓	✓	✓	✓	✓	✓
Outlier	✗	✗	✓	✓	✗	✗
MTT	✗	✗	✓	✓	✗	✗
SLAM	✗	✓	✓	✓	✗	✓

Outlier, adapted from Section 2 of [Minka 2001], models the same situation as the Kalman benchmark, but with a sensor that can occasionally produce invalid readings. The model chooses the probability of an invalid reading from a beta(100,1000) distribution, so that invalid readings occur approximately 10% of the time. At each time step, with the previously chosen probability, the model chooses the observation from either the invalid distribution gaussian(0,100) or the Kalman model. Applying delayed sampling to this model is equivalent to a Rao-Blackwellized particle filter [Doucet et al. 2000b] combining exact inference with approximate particle filtering.

MTT (Multi-Target Tracker) is adapted from [Murray and Schön 2018] and involves a variable number of targets with linear-Gaussian 2D position/velocity motion models that produce measurements of position at each time step. The model randomly introduces targets as a Poisson process and deletes them with fixed probability at each step.

SLAM (Simultaneous Localization and Mapping) is adapted from [Doucet et al. 2000a] and models an agent that estimates its position on a one-dimensional grid and also a map of its environment associating each cell with black or white. The robot uses inference to decide its next move, but its motion commands are noisy with some probability that its wheels may slip, and its observations may also be incorrectly reported.

7.2 Analysis Results

Table 1 displays the analysis outputs for each of the benchmark programs. For each analysis, the "output" column is the result of the implementation, and the "actual" column is the ground truth, i.e., whether the program satisfies the semantic property according to manual analysis. The "bounded memory" columns are the logical conjunction of the two semantic properties.

For the first six benchmarks, the analysis implementation yielded the same answer as manual analysis for whether the program satisfies both semantic properties and thus permits execution in bounded memory. In every case, the output of the implementation is sound with respect to the ground truth. Furthermore, all unseparated-path analyses converged within 10 iterations.

Kalman. For this program, every variable is m-consumed for $m \leq 1$ and starts an unseparated path of length at most 1, and thus it can execute in bounded memory.

Kalman Hold-First. For this program, every variable is m-consumed for $m \leq 1$. However, the analysis detects that unseparated paths starting from the initial value for x grow without bound and fail to converge after 10 iterations, so this program cannot execute in bounded memory.

Gaussian Random Walk. Here, every unseparated path has length at most 1. However, the analysis detects that there is no m such that any variable is m-consumed because no variable is ever observed or evaluated, so this program cannot execute in bounded memory.

Robot. Every variable is 1-consumed and every separated path has length at most 1. The analysis succeeds and indicates this program can execute in bounded memory.

Coin. Every variable is 1-consumed and every unseparated path has length at most 1. The analysis succeeds and indicates this program can execute in bounded memory.

Gaussian-Gaussian. Every variable is 1-consumed and every separated path has length at most 1. The analysis succeeds and indicates this program can execute in bounded memory.

Outlier. Every unseparated path has length at most 1. However, in the event that samples are indefinitely considered outliers, no observation will occur that causes the variable xt to be consumed, so this program cannot execute in bounded memory.

MTT. Every unseparated path has length at most 1. However, not all random variables are guaranteed to be consumed, as the final observe operation is only executed based on a dynamic condition on the lengths of two list data structures. Because this condition is not guaranteed to be met, this program cannot execute in bounded memory.

SLAM. Every unseparated path has length at most 1. The analysis concludes that the environment map array is not consumed because the model makes random choices that are not guaranteed to cover all the entries of the map. However, manual examination shows that an entry of the map that is never covered by a random choice is 0-consumed by virtue of being never used. Thus, the analysis soundly but imprecisely determines that the m-consumed condition fails.

7.3 Discussion

For the *Outlier* and *MTT* benchmarks, even though both fail the m-consumed semantic property and therefore are not guaranteed to execute in bounded memory, they will *almost certainly* execute in bounded memory. For example, in *Outlier*, the only way that the memory consumption of the model will increase indefinitely is if a particular random choice always takes one branch, which is a probability-zero event. In general, our semantic properties and analysis implementation reason about the absence of *any* program execution that yields unbounded memory. However, in practice, almost certain bounded-memory execution may also be a useful property of programs.

In general, the analysis can provide a sound guarantee that a program executes with bounded memory. However, as we saw with *SLAM*, it is not always precise enough such that if it rejects a program, then the program must have unbounded memory consumption. For example, it is possible to deliberately construct pathological programs requiring a large number of iterations for the unseparated paths analysis. Remaining limitations on precision include common static analysis challenges such as path sensitivity due to if statements and aliasing due to complex data structures.

When facing conditional branches, the analysis takes a conservative approach that may not utilize all statically available knowledge. Specifically, it cannot determine that certain branches are taken at most once over the entire input stream or that only certain program paths are valid over multiple sequential branches. The analysis also cannot accurately track variables that are stored into complex data structures, meaning it cannot mark them as consumed. We discuss these challenges in greater detail and provide specific examples in Appendix G.

8 RELATED WORK

Resource Analysis for Probabilistic Programs. Static resource analysis is capable of automatically determining upper bounds for resources such as time or memory required to execute a probabilistic program. Ngo et al. [2018] proposed a weakest-precondition approach to determine the expected memory usage of a probabilistic program, which bounds the number of loop iterations executed

and number of explicit memory allocation ticks encountered. Our analysis, on the other hand, extends static reasoning to the inherent memory usage of the inference algorithm itself.

Reactive Probabilistic Programming. Gupta et al. [1997] first introduced the idea of *reactive* probabilistic programming. They extend a concurrent constraint language with random variables. In contrast, our language is based on a synchronous dataflow model and focus on resource analysis.

Baudart et al. [2020] developed ProbZelus, a reactive probabilistic programming language which operates over streams of data and supports inference at each stream iteration. It uses an implementation of delayed sampling designed to provide bounded-memory inference for a class of reactive probabilistic programs. However, ProbZelus provides no static guarantee of bounded-memory inference. In this work, we define a language that can be used as a target for the compilation of ProbZelus and identify the semantic conditions and a static analysis that makes it possible to provide a static guarantee.

Delayed Sampling and Bounded-Memory Inference. The mechanism of delayed sampling in probabilistic programs was introduced by Murray et al. [2018] and implemented in the Anglican and Birch programming languages, neither of which supports inference over streams. Delayed sampling, a form of Sequential Monte Carlo [Liu and Chen 1998], can execute in bounded memory because it automates the construction of Rao-Blackwellized particle filters [Doucet et al. 2000b], a particularly efficient variant of SMC. By comparison, Markov chain Monte Carlo techniques generally cannot execute in bounded memory because they maintain a sample of the full history of program execution, the size of which can grow without bound for a probabilistic stream. Variational inference has extensions that make it amenable to streaming [Broderick et al. 2013], but we are not aware of any probabilistic programming system that makes use of them.

Other programming languages such as Hakaru [Narayanan et al. 2016] use static program transformations to accomplish the same goal of deferring approximate inference as much as possible. It is unclear if these transformations apply to a streaming context, where dynamic information is necessary to reflect the evolution of the underlying model over many iterations.

9   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.

In our paper, we introduce the *m*-consumed and unseparated path semantic properties, which show that delayed sampling can execute in bounded memory for reactive probabilistic programs. We present a sound static analysis that verifies these two properties with a type system and an abstract delayed sampling graph. 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.

ACKNOWLEDGMENTS

REFERENCES

Eric Atkinson, Guillaume Baudart, Louis Mandel, Charles Yuan, and Michael Carbin. 2021. Statically Bounded-Memory Delayed Sampling for Probabilistic Streams. In *arXiv e-prints*. Available at https://arxiv.org/abs/2109.12473.

Eric Atkinson, Cambridge Yang, and Michael Carbin. 2018. Verifying Handcoded Probabilistic Inference Procedures. In *arXiv e-prints*.

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).

Atilim Güneş Baydin, Lei Shao, Wahid Bhimji, Lukas Heinrich, Lawrence Meadows, Jialin Liu, Andreas Munk, Saeid Naderiparizi, Bradley Gram-Hansen, Gilles Louppe, Mingfei Ma, Xiaohui Zhao, Philip Torr, Victor Lee, Kyle Cranmer, Prabhat, and Frank Wood. 2019. Etalumis: Bringing Probabilistic Programming to Scientific Simulators at Scale. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '19)*.

Eli Bingham, Jonathan P. Chen, Martin Jankowiak, Fritz Obermeyer, Neeraj Pradhan, Theofanis Karaletsos, Rohit Singh, Paul Szerlip, Paul Horsfall, and Noah D. Goodman. 2019. Pyro: Deep Universal Probabilistic Programming. *Journal of Machine Learning Research* 20, 28 (2019).

Tamara Broderick, Nicholas Boyd, Andre Wibisono, Ashia C. Wilson, and Michael I. Jordan. 2013. Streaming Variational Bayes. In *International Conference on Neural Information Processing Systems*.

Jean-Louis Colaço, Bruno Pagano, and Marc Pouzet. 2017. SCADE 6: A formal language for embedded critical software development (invited paper). In *TASE*. IEEE Computer Society, 1–11.

Marco F Cusumano-Towner, Feras A Saad, Alexander K Lew, and Vikash K Mansinghka. 2019. Gen: a General-purpose Probabilistic Programming System with Programmable Inference. In *Conference on Programming Language Design and Implementation*.

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.

Arnaud Doucet, Nando de Freitas, Kevin P. Murphy, and Stuart J. Russell. 2000a. Rao-Blackwellised Particle Filtering for Dynamic Bayesian Networks. In *UAI*.

Arnaud Doucet, Nando de Freitas, Kevin P. Murphy, and Stuart J. Russell. 2000b. Rao-Blackwellised Particle Filtering for Dynamic Bayesian Networks. In *Conference on Uncertainty in Artificial Intelligence*.

Daniel Fink. 1997. A Compendium of Conjugate Priors. (1997).

Hong Ge, Kai Xu, and Zoubin Ghahramani. 2018. Turing: Composable inference for probabilistic programming. In *International Conference on Artificial Intelligence and Statistics*.

Andrew Gelman, Daniel Lee, and Jiqiang Guo. 2015. Stan: A probabilistic programming language for Bayesian inference and optimization. *Journal of Educational and Behavioral Statistics* 40, 5 (2015), 530–543.

Noah D. Goodman, Vikash K. Mansinghka, Daniel M. Roy, Keith Bonawitz, and Joshua B. Tenenbaum. 2008. Church: A language for generative models. In *Conference on Uncertainty in Artificial Intelligence*.

Noah D Goodman and Andreas Stuhlmüller. 2014. The Design and Implementation of Probabilistic Programming Languages. http://dippl.org. Accessed: 2020-10-30.

Andrew D. Gordon, Thore Graepel, Nicolas Rolland, Claudio Russo, Johannes Borgstrom, and John Guiver. 2014. Tabular: a schema-driven probabilistic programming language. In *Symposium on Principles of Programming Languages*.

Vineet Gupta, Radha Jagadeesan, and Vijay A. Saraswat. 1997. Probabilistic Concurrent Constraint Programming. In *CONCUR (Lecture Notes in Computer Science, Vol. 1243)*. Springer, 243–257.

Daniel Huang, Jean-Baptiste Tristan, and Greg Morisett. 2017. Compiling Markov Chain Monte Carlo Algorithms for Probabilistic Modeling. In *Conference on Programming Language Design and Implementation*.

R. E. Kalman. 1960. A New Approach to Linear Filtering and Prediction Problems. *Journal of Basic Engineering* 82, 1 (1960).

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

Jun S. Liu and Rong Chen. 1998. Sequential Monte Carlo Methods for Dynamic Systems. *J. Amer. Statist. Assoc.* 93, 443 (1998), 1032–1044.

Daniel Lundén. 2017. *Delayed sampling in the probabilistic programming language Anglican*. Master's thesis. KTH Royal Institute of Technology.

Vikash Mansingkha, Ulrich Schaechtle, Shivam Handa, Alexey Radul, Yutian Chen, and Martin Rinard. 2018. Probabilistic Programming with Programmable Inference. In *Conference on Programming Language Design and Implementation*.

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

Brian Milch, Bhaskara Marthi, Stuart Russell, David Sontag, Daniel L. Ong, and Andrey Kolobov. 2007. BLOG: Probabilistic models with unknown objects. *Statistical relational learning* (2007).

Thomas P. Minka. 2001. Expectation Propagation for Approximate Bayesian Inference. In *Conference in Uncertainty in Artificial Intelligence*.

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*.

Lawrence M. Murray and Thomas B. Schön. 2018. Automated learning with a probabilistic programming language: Birch. *Annual Reviews in Control* 46 (2018).

Praveen Narayanan, Jacques Carette, Wren Romano, Chung-chieh Shan, and Robert Zinkov. 2016. Probabilistic inference by program transformation in Hakaru (system description). In *International Symposium on Functional and Logic Programming*.

Van Chan Ngo, Quentin Carbonneaux, and Jan Hoffmann. 2018. Bounded Expectations: Resource Analysis for Probabilistic Programs. In *Conference on Programming Language Design and Implementation*.

Aditya V. Nori, Sherjil Ozair, Sriram K. Rajamani, and Deepak Vijaykeerthy. 2015. Efficient Synthesis of Probabilistic Programs. In *Conference on Programming Language Design and Implementation*.

Avi Pfeffer. 2009. Figaro: An object-oriented probabilistic programming language. Vol. 137. 96.

Eduardo D Sontag. 2013. *Mathematical control theory: deterministic finite dimensional systems*. Vol. 6. Springer Science & Business Media.

Sam Staton. 2017. Commutative Semantics for Probabilistic Programming. In *European Symposium on Programming*.

Dustin Tran, Matthew D Hoffman, Rif A Saurous, Eugene Brevdo, Kevin Murphy, and David M Blei. 2017. Deep probabilistic programming. In *International Conference on Learning Representations*.