Reactive Probabilistic Programming

Guillaume Baudart
MIT-IBM Watson AI Lab, IBM
Research
USA

Louis Mandel
MIT-IBM Watson AI Lab, IBM
Research
USA

Eric Atkinson
MIT
USA

Benjamin Sherman
MIT
USA

Marc Pouzet
École Normale Supérieure
France

Michael Carbin
MIT
USA

Abstract

Synchronous modeling is at the heart of programming languages like Lustre, Esterel, or SCADE used routinely for implementing safety critical control software, e.g., fly-by-wire and engine control in planes. However, to date these languages have had limited modern support for modeling uncertainty — probabilistic aspects of the software's environment or behavior — even though modeling uncertainty is a primary activity when designing a control system.

In this paper we present ProbZelus the first *synchronous probabilistic programming language*. ProbZelus conservatively provides the facilities of a synchronous language to write control software, with probabilistic constructs to model uncertainties and perform *inference-in-the-loop*.

We present the design and implementation of the language. We propose a measure-theoretic semantics of probabilistic stream functions and a simple type discipline to separate deterministic and probabilistic expressions. We demonstrate a semantics-preserving compilation into a first-order functional language that lends itself to a simple presentation of inference algorithms for streaming models. We also redesign the delayed sampling inference algorithm to provide efficient *streaming* inference. Together with an evaluation on several reactive applications, our results demonstrate that ProbZelus enables the design of reactive probabilistic applications and efficient, bounded memory inference.

1 Introduction

Synchronous languages [1] were introduced thirty years ago for designing and implementing real-time control software. They are founded on the synchronous abstraction [3] where a system is modeled ideally, as if communications and computations were instantaneous and paced on a global clock. This abstraction is simple but powerful: input, output and local signals are streams that advance synchronously and a system is a stream function. It is at the heart of the data-flow languages Lustre [20] and SCADE [12]; it is also the underlying model behind the discrete-time subset of Simulink.

The data-flow programming style is very well adapted to the direct expression of the classic control blocks of control engineering (e.g., relays, filters, PID controllers, control logic), and a discrete time model of the environment, with the feedback between the two. For example, consider a backward Euler integration method defined by the following stream equations and its corresponding implementation in Zelus [7], a language reminiscent of Lustre:

$$x_0 = xo_0 \qquad\qquad x_n = x_{n-1} + x'_n \times h \quad \forall n \in \mathbb{N}, n > 0$$

```
let node integr(xo, x') = x where rec x = xo -> pre x + (x' * h)
```

integr is a function from input streams xo and x' to output stream x. −> is the initialization operator and pre, a unit-delay. integr can be used to define an other stream function, which can be called in a control structures like a hierarchical automata, e.g., to express a system that switches between automatic and manual control. Compared to a general purpose functional language (or an embedded DSL), the expressiveness of a synchronous language is purposely constrained to modularly ensure safety properties that are critical for the targeted applications: determinacy, deadlock freedom (reactivity), the generation of statically scheduled code that runs in bounded time and space.

However, to date, these languages have had limited support for modeling uncertainty (e.g., a noisy sensor or channel, a variable delay), to simulate the interaction of a software controller and a partially unknown environment, or to infer parameters from noisy observations. Moreover, uncertainty is a first-order design concern for a controller that operate under the assumption of a probabilistic model of their environment (e.g., object tracking). Using a probabilistic environment model and data gathered from observing the environment, a controller can infer a distribution of likely environments given the observations. Existing approaches consist in hand-coding stochastic controllers that have a known solution (e.g., Kalman filters) which can be tedious and error-prone, or to simply perform off-line statistical testing on the generated code of a controller. Alternatively, in recent years, *probabilistic programming* has developed as an approach to endow general purpose languages with the ability to automate inference.

Probabilistic Programming. Probabilistic programming languages are used to describe probabilistic models and automatically infer distributions of latent (i.e., unobserved) parameters from observations.

A popular approach [5, 18, 30, 38–40] consists in extending a general-purpose programming language with three constructs: (1) x = sample(d) introduces a random variable x of distribution d, (2) observe(d, y) measures the likelihood of an observation y with respect to a distribution d, and (3) infer m obs computes the distribution of the output values of a program or *model* m w.r.t. the observation of the input data obs.

Probabilistic programming languages offer a variety of automatic inference techniques ranging from exact inference [17] to approximate inference [31] and include hybrid approaches that combine exact and approximate techniques when part of the program is analytically tractable [29]. However, a standing challenge for these programming languages is that none of them meet the design goals of synchronous reactive languages by being immediately amenable to techniques to ensure that for example a program with an indefinite execution time runs in bounded memory.

Inference in the Loop. In this paper we extend Zelus[1] to provide a synchronous probabilistic programming language, ProbZelus. ProbZelus enables one to combine deterministic reactive programs, such as integr (above), with probabilistic programming constructs to produce reactive probabilistic programs.

Compared to other probabilistic languages (e.g. WebPPL, Church, Stan) where inference is executed on terminating pure functions, our probabilistic models are stateful stream processors. Inference on probabilistic models runs in parallel with deterministic processes that interact with the environment. The distributions computed by infer at each step can thus be used by deterministic processes to compute new inputs for the next inference step. We term this capability *inference-in-the-loop*.

Streaming Inference. ProbZelus provides multiple inference algorithms, most notably the *delayed sampling* inference algorithm [29]. This hybrid strategy combines the approximate inference technique of particle filtering [19] with exact inference when it is possible to symbolically determine the exact distribution for some or all of the latent variables of the program [14].

However, the memory consumption of delayed sampling strictly increases with the number of random variables which is not practical for reactive applications that operate on infinite streams. We propose a novel *streaming* implementation of delayed sampling that can operate over infinite streams in constant memory for a large class of models. ProbZelus therefore provides an expressive language for reactive probabilistic programming with appropriate memory consumption properties.

Contributions. We present the following contributions:

```
let proba kalman (u, acc, gps) = x where
 rec mu = x0 -> (a *@ pre x) +@ (b *@ u)
 and x = sample (mv_gaussian (mu, noise))
 and () = observe (gaussian (vec_get x 2, 1.0), acc)
 and present gps (pos) ->
       do () = observe (gaussian (vec_get x 0, 0.01), pos) done

let node robot (acc, gps) = u where
 rec x_dist = infer kalman (u, acc, gps)
 and u = u0 -> lqr a b (mean (pre x_dist))
```

Figure 1. Robot controller with inference-in-the-loop. +@ and *@ are matrix operations, vec_get x i is the ith projection.

Design, Semantics, Compilation. We present ProbZelus, the first synchronous probabilistic programming language combining language constructs for streams (reactivity) with those for probabilistic programming thus enabling *inference-in-the-loop*. We give a measure-based co-iterative semantics for ProbZelus that forms the basis of a compiler and demonstrate a semantics-preserving compilation strategy to a first-order functional language μF.

Streaming Inference. We define the semantics of multiple inference algorithms on μF including particle filtering and delayed sampling. We then present a novel *streaming delayed sampling* implementation which enables partial exact inference over infinite streams in bounded memory for a large class of models.

Evaluation. We evaluate the performance of ProbZelus on a set of benchmarks that illustrate multiple aspects of the language. We demonstrate that streaming delayed sampling drastically reduce the number of particles required to achieve better accuracy compared to a particle filter.

The result is ProbZelus, a synchronous probabilistic language that enables us to write, in the very same source, a deterministic model for the control software and a probabilistic model with complex interactions between the two. On one hand, a deterministic model of a controller can rely on predictions computed by a probabilistic model. On the other hand, a probabilistic model can be programmed in an expressive reactive language.

2 Example

In this section, we demonstrate how ProbZelus provides probabilistic modeling, inference-in-the-loop, and bounded-memory inference for a robot navigation system. The results of the inference are continuously used by a controller to correct the robot trajectory.

2.1 Inference in the Loop.

We consider a robot equipped with an accelerometer and a GPS. We assume that the motion of the robot can be described as: $x_{t+1} = Ax_t + Bu_t$ where x_t denotes the state of the robot (position, velocity, and acceleration) at a given

[1] Language distribution and manual available at zelus.di.ens.fr

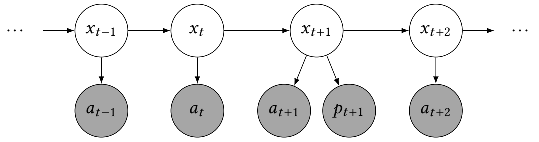


Figure 2. Kalman filter for the robot example. Variables are either latent (white, e.g., state x) or observed (gray, e.g., acceleration a). The position p is only sporadically observed.

time step t, and u_t denotes the command sent to the robot. A and B are constant matrices. In addition, the robot receives at each step noisy *observations* from the accelerometer a_t, and sporadically an estimation of the position from a GPS p_t.

Figure 1 presents a controller, robot, that given inputs from the accelerometer acc and the GPS gps computes a stream u of commands that drives the robot to a given target. The keyword node indicates a deterministic stream function that maps input streams to output streams. The body of robot is the parallel composition of (1) the inference of a probabilistic process kalman that estimates x_dist the stream of distributions over the robot's state, and (2) a deterministic process that computes the stream u of commands. It is written as two mutually recursive equations that define x_dist from u and u from the previous value of x_dist.

The command u is set to one of two values as determined by the *initialization operator*, ->. On the first time step, the operator returns its left-hand side value, the initial command u0. On every time step thereafter, the operator returns its right-hand side expression, the output of a *Linear-Quadratic Regulator* (LQR) [36] — a stable and optimal controller for such dynamic systems — given the estimation of the state at the previous step. The *unit delay operator*, pre, introduces a unit delay to the evaluation of an expression and therefore returns the value of its expression at the previous time step.

Inference. The stream x_dist of distributions of state is inferred from the model defined by the probabilistic node kalman given the command u and the observations acc, and gps. The keyword proba indicates a probabilistic model.

In this example, the model is a *Kalman Filter* illustrated in Figure 2. A Kalman filter is a time-dependent probabilistic model used to describe inference problems such as tracking, in which a tracker estimates the true position of an object given noisy, sensed observations. The robot's state x_t is a *latent* random variable in that the tracker is not able to directly observe it. Each arrow connecting two random variables denotes a dependence of the variable at the head of the arrow on the variable at the tail. In this case, the observations at each time step depends on the current state, and the robot's state at a given time step depends only on its states at the previous time step.

Sampling. Inside kalman, the sample operator samples a value from a probability distribution. In this case, the expression samples the current state x from a multivariate Gaussian with mean obtained by applying the motion model to the previous state and the command. This code models the trajectory of a robot where at each time step, the state is Gaussian-distributed around an estimation computed from the motion model.

Observations. The expression observe conditions the execution on observed data. Its first parameter denotes a distribution that models the observation and its second parameter denotes the observed value itself. In this case, at each step, the first observe statement models a Gaussian-distributed observation of the current acceleration vec_get x 2 given by acc. The input gps is a *signal* that is only emitted when the GPS computes a new position. When a value pos is emitted on gps, the present construct executes its body, further conditioning the model by adding a Gaussian-distributed observation of the current position vec_get x 0 given by pos.

2.2 Streaming Inference

A classic operational interpretation of a probabilistic model is an *importance sampler* that generates random samples from the model together with an importance weight measuring the quality of the sample. In this model, each execution of a sample operator samples a value from the operator's corresponding distribution. Each execution of an observe evaluates the likelihood of the provided observation and multiplies the current importance weight by this value. Then, each execution step of infer yields a distribution represented as a set of pairs (output, weight) or *particles*. The particles can be re-sampled at each step to build a *particle filter* [13].

The integer parameter to infer determines how many particles to use: the more particles the user specifies, the more accurate the estimate of the distribution becomes. The PF points in Figure 3 present this improvement in accuracy as a function of increasing the number of particles for the robot example. However, as the latency results presents, the more particles the user specifies, the more computation is required for each step because each particle requires a full, independent execution of each time step of the model.

Streaming Delayed Sampling. Delayed sampling [29] can reduce the number of particles required to achieve a given desired quality of inference. Specifically, delayed sampling exploits the opportunity to symbolically reason about the relationships between random variables to compute closed-form distributions whenever possible. To capture relationships between random variables, delayed sampling maintains a graph: a *Bayesian network* that can be used to compute closed-form distributions involving subsets of random variables. For instance, this inference scheme is able to compute the exact posterior distribution for our robot example. The SDS dots in Figure 3 show that the accuracy is independent

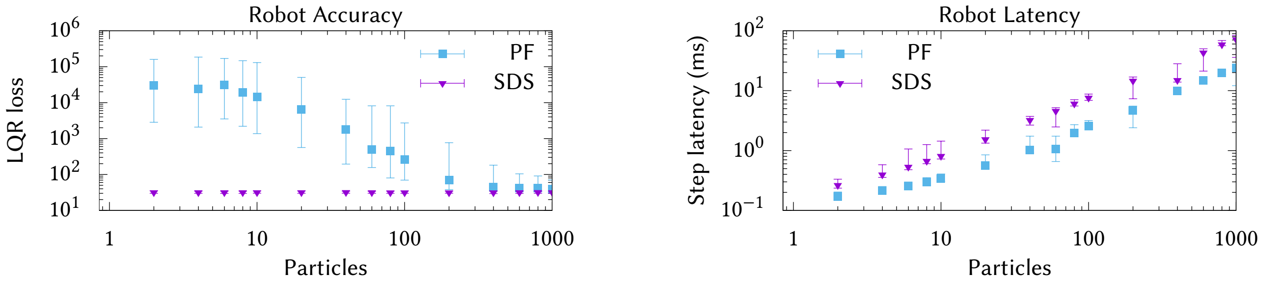


Figure 3. Particle filter (PF) and streaming delayed sampling (SDS) performances for the robot example of Figure 1. Accuracy is measured using the loss function of the LQR.

of the number of particles since each particle computes the exact solution.

Figure 4 illustrates the evolution of the delayed sampling graph as it proceeds through the first four time steps of the robot example (for simplicity we assume that there is no GPS activation in these four steps). A notable challenge with the traditional delayed sampling algorithm is that the graph grows linearly in the number of samples. This property is not tractable in our reactive context because we would like to deploy our programs under the model that they run indefinitely, thus requiring that they execute with bounded resources. To address this problem, we propose a novel *streaming delayed sampling* (SDS) implementation of the delayed sampling algorithm. Specifically, in Figure 4 the node denoting the marginal posterior for x at step 1 can be eliminated from the graph at step 3 because the distributions for pre x and x have fully incorporated its effect on their values and, moreover, the program no longer maintains a reference to the node.

While the standard delayed sampling algorithm will keep this node alive through the edge pointers it maintains, SDS builds a *pointer-minimal* graph representation with a minimal number of edges that 1) ensure that the graph has sufficient connectivity to support operations in the traditional delayed sampling algorithm and 2) only maintain the reachability of nodes that can effect the distribution of future nodes in the graph. The result is that the memory consumption of SDS is constant across the number of steps while the memory consumption of the original delayed sampling implementation DS increases linearly in the number of steps (Figure 5).

3 Language: Syntax, Typing, Semantics

ProbZelus is a reactive probabilistic language with inference-in-the-loop which enables interaction between probabilistic models and deterministic processes. This capability introduces two design requirements. First, a probabilistic model must be able to receive inputs from an evolving environment. Second, instead of awaiting the final result of the inference, deterministic processes running in parallel need access to intermediate results. The resulting inference-in-the-loop enables feedback loops between inferred distributions from probabilistic models and deterministic processes, which our design controls by enforcing a separation between the semantics of probabilistic and deterministic execution.

In this section, we formalize the syntax of ProbZelus, introduce a type system that imposes a clear separation between deterministic and probabilistic expressions, and define the semantics of the language in a co-iteration framework where the semantics of probabilistic processes is adapted from Staton's measure-theoretic semantics for probabilistic programs [37]. The co-iterative semantics forms the basis of a compiler that is described in Section 4.

3.1 Syntax

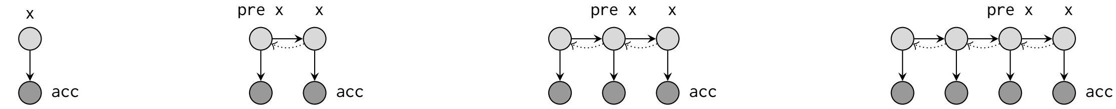
We focus on the following kernel of ProbZelus. The missing constructs (e.g., pre and ->) can be compiled into this kernel via a source-to-source transformation.

$$
\begin{aligned}
d &::= \texttt{let node } f\ x = e \ \mid\ \texttt{let proba } f\ x = e \ \mid\ d\ d \\
e &::= c \ \mid\ x \ \mid\ (e{,}e) \ \mid\ op(e) \ \mid\ f(e) \ \mid\ \texttt{last } x \ \mid\ e \texttt{ where rec } E \\
&\quad \mid\ \texttt{present } e \texttt{ -> } e \texttt{ else } e \ \mid\ \texttt{reset } e \texttt{ every } e \\
&\quad \mid\ \texttt{sample}(e) \ \mid\ \texttt{observe}(e, e) \ \mid\ \texttt{infer}(e) \\
E &::= x = e \ \mid\ \texttt{init } x = c \ \mid\ E \texttt{ and } E
\end{aligned}
$$

A program is a sequence of *declarations* d of stream functions (node) and probabilistic stream functions (proba). An *expression* e is either a constant (c), a variable (x), a pair, an external operator application (op), a function application ($f(e)$), a delay (last x) that returns a value (x) from the previous step, or a set of locally recursive equations (e where rec E). A set of *equations* E is either an equation $x = e$ that define x with the stream e, the initialization of a variable with a constant init x = c, or parallel composition of sets of equations.

The control structure present e -> e_1 else e_2 is an *activation condition* that executes the expression e_1 only when the value of e is true and executes e_2 otherwise. It differs from if e then e_1 else e_2, where both e_1 and e_2 are computed at each step and the returned value is chosen based on the value of e.[2] The reset e_1 every e construct re-initializes the values of the init equations and the corresponding last expressions in e_1 each time e is true. The language is extended with the classic probabilistic expressions: sample to draw from a distribution, observe to measure the likelihood of an

[2] The if construct can thus be considered as an external operator.



(a) step 1 **(b)** step 2 **(c)** step 3 **(d)** step 4

Figure 4. Evolution of the delayed sampling graph for the model of Figure 1. Each node denotes either a value (dark gray) or a distribution (light gray). Plain arrows represent dependencies in the underlying Bayesian network. The dotted arrow represents the pointers in the original data-structure implementing the graph. Labels indicate the program variables.

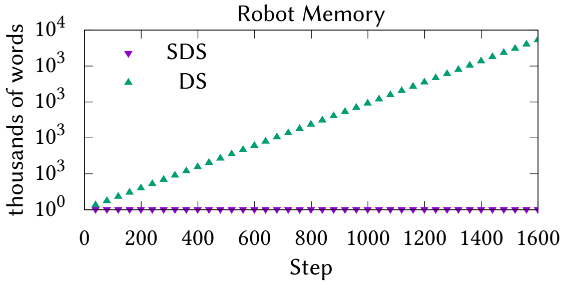


Figure 5. Delayed sampling (DS) and streaming delayed sampling (SDS) memory consumption in thousands of live words in the heap per steps for the robot example.

observation w.r.t. a distribution, and infer to compute the distribution described by a model.

Scheduling. In the expression e where rec E, E is a set of mutually recursive equations. In practice, a scheduler reorders the equations according to their dependencies. Initializations init $x_j = c_j$ are grouped at the beginning, and an equation $x_j = e_j$ must be scheduled after the equation $x_i = e_i$ if the expression e_j uses x_i and not last x_i. A program satisfying this partial order is said to be *scheduled*. The compiler can also introduce additional equations to relax the scheduling constraints and rejects programs that cannot be statically scheduled [4]. After scheduling, the expression e where rec E has the following form.

```
e where rec init x_1 = c_1 and ... and init x_k = c_k
          and y_1 = e_1 and ... and y_n = e_n
```

For simplicity, we also assume that every initialized variable is defined in a subsequent equation, i.e., $\{x_i\}_{1..k} \cap \{y_j\}_{1..n} = \{x_i\}_{1..k}$. If it is not the case, in this kernel we can always add additional equations of the form x_i = last x_i.

3.2 Typing: Deterministic vs. Probabilistic

Deterministic and probabilistic expressions have distinct interpretations. A dedicated type system discriminates between the two kinds of expressions, assigning one of two *kinds* to each expression: D for deterministic, or P for probabilistic. The typing judgment $G \vdash^k e : T$ states that in the environment G which maps variable names to their type, the expression e has kind k and type T. Function types $T \to^k T'$ are extended with the kind k of the body and we introduce a new datatype T dist for the probability distribution over values of type T.

The expressions sample, and observe are probabilistic but their arguments must be deterministic. Any deterministic expression can be lifted to a probabilistic expression using a sub-typing rule. The transition from probabilistic to deterministic is realized via infer (the complete set of typing rules is presented in Figure 11 of Appendix A.1)

$$\frac{G \vdash^{\mathsf{D}} e : T\ \mathsf{dist}}{G \vdash^{\mathsf{P}} \mathsf{sample}(e) : T} \qquad \frac{G \vdash^{\mathsf{D}} e_1 : T\ \mathsf{dist} \quad G \vdash^{\mathsf{D}} e_2 : T}{G \vdash^{\mathsf{P}} \mathsf{observe}(e_1, e_2) : \mathsf{unit}}$$

$$\frac{G \vdash^{\mathsf{D}} e : T}{G \vdash^{\mathsf{P}} e : T} \qquad \frac{G \vdash^{\mathsf{P}} e : T}{G \vdash^{\mathsf{D}} \mathsf{infer}(e) : T\ \mathsf{dist}}$$

3.3 Co-Iterative Semantics

We now give the semantics of ProbZelus in a co-iteration framework [11]. In this framework, a *deterministic stream* of type T is defined by an initial state of type S and a transition function of type $S \to T \times S$.

$$CoStream(T, S) = S \times (S \to T \times S)$$

Repeatedly executing the transition function from the initial state yields a stream of values of type T.

The semantics of a *deterministic expression* ($kindOf(e) = \mathsf{D}$) is defined using two auxiliary functions. If γ is an environment mapping variable names to values, $[\![e]\!]^i_\gamma$ denotes the initial state, and $[\![e]\!]^s_\gamma$ denotes the transition function:

$$[\![e]\!]_\gamma : CoStream(T, S) = [\![e]\!]^i_\gamma, [\![e]\!]^s_\gamma$$

The deterministic semantics of ProbZelus presented in Figure 6 is an extension of [11] with the control structures present and reset (see also Figure 12 of Appendix A.2).

The transition function of a variable always returns the corresponding value stored in the environment γ.

The present e -> e_1 else e_2 construct introduced in Section 2 returns the value of e_1 when e is true and the value of e_2 otherwise. The state (s, s_1, s_2) stores the state of the three sub-expressions. The transition function lazily executes e_1 or e_2 depending on the value of e and returns the updated state.

The state of a set of scheduled locally recursive definitions e where rec E comprises three parts: the value of the local variables at the previous step which can be accessed via

$$
\begin{aligned}
\llbracket x \rrbracket^i_\gamma &= () \\
\llbracket x \rrbracket^s_\gamma &= \lambda s.\ (\gamma(x), s)
\end{aligned}
$$

$$
\begin{aligned}
&\llbracket \text{present } e \text{ -> } e_1 \text{ else } e_2 \rrbracket^i_\gamma = (\llbracket e \rrbracket^i_\gamma, \llbracket e_1 \rrbracket^i_\gamma, \llbracket e_2 \rrbracket^i_\gamma) \\
&\llbracket \text{present } e \text{ -> } e_1 \text{ else } e_2 \rrbracket^s_\gamma = \\
&\quad \lambda(s, s_1, s_2).\ \mathit{let}\ v, s' = \llbracket e \rrbracket^s_\gamma(s)\ \mathit{in} \\
&\qquad \mathit{if}\ v\ \mathit{then}\ \mathit{let}\ v_1, s'_1 = \llbracket e_1 \rrbracket^s_\gamma(s_1)\ \mathit{in}\ (v_1, (s', s'_1, s_2)) \\
&\qquad\qquad \mathit{else}\ \ \mathit{let}\ v_2, s'_2 = \llbracket e_2 \rrbracket^s_\gamma(s_2)\ \mathit{in}\ (v_2, (s', s_1, s'_2))
\end{aligned}
$$

$$
\begin{aligned}
&\left\llbracket \begin{array}{c} e \text{ where rec init } x_1 = c_1 \text{ and ... and init } x_k = c_k \\ \text{and } y_1 = e_1 \text{ and ... and } y_n = e_n \end{array} \right\rrbracket^i_\gamma = \\
&\quad ((c_1, \ldots, c_k),\ (\llbracket e_1 \rrbracket^i_\gamma, \ldots, \llbracket e_n \rrbracket^i_\gamma),\ \llbracket e \rrbracket^i_\gamma)
\end{aligned}
$$

$$
\begin{aligned}
&\left\llbracket \begin{array}{c} e \text{ where rec init } x_1 = c_1 \text{ and ... and init } x_k = c_k \\ \text{and } y_1 = e_1 \text{ and ... and } y_n = e_n \end{array} \right\rrbracket^s_\gamma = \\
&\quad \lambda((m_1, \ldots, m_k), (s_1, \ldots, s_n), s). \\
&\qquad \mathit{let}\ \gamma_1 = \gamma[m_1/x_1\_\text{last}]\ \mathit{in}\ \ldots \mathit{let}\ \gamma_k = \gamma_{k-1}[m_k/x_k\_\text{last}]\ \mathit{in} \\
&\qquad \mathit{let}\ v_1, s'_1 = \llbracket e_1 \rrbracket^s_{\gamma_k}(s_1)\ \mathit{in}\ \mathit{let}\ \gamma'_1 = \gamma_k[v_1/y_1]\ \mathit{in}\ \ldots \\
&\qquad \mathit{let}\ v_n, s'_n = \llbracket e_n \rrbracket^s_{\gamma'_{n-1}}(s_n)\ \mathit{in}\ \mathit{let}\ \gamma'_n = \gamma'_{n-1}[v_n/y_n]\ \mathit{in} \\
&\qquad \mathit{let}\ v, s' = \llbracket e \rrbracket^s_{\gamma'_n}(s)\ \mathit{in} \\
&\qquad v, ((\gamma'_n[x_1], \ldots, \gamma'_n[x_k]), (s'_1, \ldots, s'_n), s')
\end{aligned}
$$

Figure 6. Semantics of deterministic expressions.

the last operator, the state of the defining expressions, and the state of expression e. The initialization stores the initial values introduced by init and the initial states of all sub-expressions. The transition function incrementally builds the local environment defined by E. First the environment is populated with a set of fresh variables x_i_last initialized with the values stored in the state that can then be accessed via the last operator. Then the environment is extended with the definition of all the variables y_i by executing all the defining expressions (where $\{x_i\}_{1..k} \cap \{y_j\}_{1..n} = \{x_i\}_{1..k}$). Finally, the expression e is executed in the final environment. The updated state contains the value of the initialized variables defined in E that will the be used to start the next step, and the updated state of the sub-expressions.

Probabilistic extension. The semantics of a *probabilistic expression* ($\mathit{kindOf}(e) = \mathsf{P}$) follows the same scheme, but the transition function returns a *measure* over the set of possible pairs (result, state):

$$\mathit{CoPStream}(T, S) = S \times (S \to (\Sigma_{T \times S} \to [0, \infty]))$$

A measure μ associates a positive number to each measurable set $U \in \Sigma_{T \times S}$ where $\Sigma_{T \times S}$ denotes the Σ-algebra of $T \times S$, i.e., the set of measurable sets over pairs (result, state). We use the following notation for the semantics of a probabilistic expression e:

$$\{\!\!\{ e \}\!\!\}_\gamma : \mathit{CoPStream}(T, S) = \{\!\!\{ e \}\!\!\}^i_\gamma, \{\!\!\{ e \}\!\!\}^s_\gamma$$

The semantics of probabilistic expressions is presented in Figure 7 (the complete semantics is in Figure 13 of Appendix A.2). This measure-based semantics is adapted from [37] to explicitly handle the state of the transition functions.

$$
\begin{aligned}
\{\!\!\{ e \}\!\!\}^i_\gamma &= \llbracket e \rrbracket^i_\gamma & &\mathit{if}\ \mathit{kindOf}(\mathrm{e}) = \mathsf{D} \\
\{\!\!\{ e \}\!\!\}^s_\gamma &= \lambda s.\ \lambda U.\ \delta_{\llbracket e \rrbracket^s_\gamma(s)}(U) & &\mathit{if}\ \mathit{kindOf}(e) = \mathsf{D}
\end{aligned}
$$

$$
\begin{aligned}
\{\!\!\{ \text{sample}(e) \}\!\!\}^i_\gamma &= \llbracket e \rrbracket^i_\gamma \\
\{\!\!\{ \text{sample}(e) \}\!\!\}^s_\gamma &= \lambda s.\ \lambda U.\ \mathit{let}\ \mu, s' = \llbracket e \rrbracket^s_\gamma(s)\ \mathit{in}\ \textstyle\int_T \mu(dv)\ \delta_{v,s'}(U)
\end{aligned}
$$

$$
\begin{aligned}
&\{\!\!\{ \text{observe}(e_1, e_2) \}\!\!\}^i_\gamma = (\llbracket e_1 \rrbracket^i_\gamma, \llbracket e_2 \rrbracket^i_\gamma) \\
&\{\!\!\{ \text{observe}(e_1, e_2) \}\!\!\}^s_\gamma = \\
&\quad \lambda(s_1, s_2).\ \lambda U. \\
&\qquad \mathit{let}\ \mu, s'_1 = \llbracket e_1 \rrbracket^s_\gamma(s_1)\ \mathit{in} \\
&\qquad \mathit{let}\ v, s'_2 = \llbracket e_2 \rrbracket^s_\gamma(s_2)\ \mathit{in}\ \mu_{\mathrm{pdf}}(v) * \delta_{(),(s'_1,s_2)}(U)
\end{aligned}
$$

$$
\begin{aligned}
&\left\{\!\!\left\{ \begin{array}{c} e \text{ where rec init } x_1 = c_1 \text{ and ... and init } x_k = c_k \\ \text{and } y_1 = e_1 \text{ and ... and } y_n = e_n \end{array} \right\}\!\!\right\}^s_\gamma = \\
&\quad \lambda((m_1, \ldots, m_k), (s_1, \ldots, s_n), s).\ \lambda U. \\
&\qquad \mathit{let}\ \gamma_1 = \gamma[m_1/x_1\_\text{last}]\ \mathit{in}\ \ldots \mathit{let}\ \gamma_k = \gamma_{k-1}[m_k/x_k\_\text{last}]\ \mathit{in} \\
&\qquad \mathit{let}\ \mu_1 = \{\!\!\{ e_1 \}\!\!\}^s_{\gamma_k}(s_1)\ \mathit{in} \\
&\qquad \textstyle\int \mu_1(dv_1, ds'_1) \mathit{let}\ \gamma'_1 = \gamma_k[v_1/y_1]\ \mathit{in}\ \ldots \\
&\qquad\quad \mathit{let}\ \mu_n = \llbracket e_n \rrbracket^s_{\gamma'_{n-1}}(s_n)\ \mathit{in} \\
&\qquad\quad \textstyle\int \mu_n(dv_n, ds'_n) \mathit{let}\ \gamma'_n = \gamma'_{n-1}[v_n/y_n]\ \mathit{in} \\
&\qquad\qquad \mathit{let}\ \mu = \{\!\!\{ e \}\!\!\}^s_{\gamma'_n}(s)\ \mathit{in} \\
&\qquad\qquad \textstyle\int \mu(dv, ds')\ \delta_{v,((\gamma'_n[x_i],\ldots,\gamma'_n[x_k]),(s'_1,\ldots,s'_n),s')}(U)
\end{aligned}
$$

Figure 7. Semantics of probabilistic expressions.

First, any deterministic expression can be lifted as a probabilistic expression. The transition function returns the Dirac delta measure ($\delta_x(U) = 1$ if $x \in U$, 0 otherwise) on the pair returned by the deterministic transition function applied on the current state: $\llbracket e \rrbracket^s_\gamma(s) : T \times S$.

The probabilistic operator sample(e) evaluates e which returns a distribution $\mu : T$ dist and a new state $s' : S$, and returns a measure over the pair (result, state) where the state is fixed to the value s'. observe(e_1, e_2) evaluates e_1 and e_2 into a distribution μ and a value v, and weight execution paths using the likelihood of v w.r.t. μ (μ_{pdf} denote the density function of the distribution μ).

The state of a set of locally recursive definitions is the same as in Figure 6 and contains the previous value of the initialized variables and the states of the sub-expressions. The transition starts by adding the variables x_i_last to the environment. We note $\int \mu(dv, ds) f(v, s)$ the integral of f w.r.t. the measure μ where variables v and s are the integration variables. The integration measure appears on the right of the integral to maintain the expression order of the source code and we allow local definitions (e.g., *let* $x = v$ *in* $\ldots$) inside the integral to simplify the presentation. Local definitions are interpreted by successively integrating the measure on pairs (value, state) returned by the defining expressions. In other words, we integrate over all possible executions. Integrals need to be nested to capture the eventual dependencies in the successive expressions. The returned value is a measure on pairs (value, state) where the state captures the value of the initialized variables and the state of the sub-expressions.

Inference in the loop. The infer operator is the boundary between the deterministic and the probabilistic expressions. Given a probabilistic model defined by an expression e, at each step the inference computes a distribution of results and a distribution of possible next states. Expression e can contain free variables thus capturing inputs from deterministic processes. The distribution of results can be used by deterministic processes to produce new inputs for the next inference step.

$$\begin{aligned}
[\![\mathsf{infer}(e)]\!]^i_\gamma &= \lambda U.\ \delta_{[\![e]\!]^i_\gamma}(U) \\
[\![\mathsf{infer}(e)]\!]^s_\gamma &= \lambda \sigma.\ let\ \mu = \lambda U.\ \frac{\int_S \sigma(ds)\{\![e]\!\}^s_\gamma(s)(U)}{\int_S \sigma(ds)\{\![e]\!\}^s_\gamma(s)(\top)}\ in \\
&\qquad (\pi_{1*}(\mu), \pi_{2*}(\mu))
\end{aligned}$$

The state of infer(e) is a distribution over the possible states for e. The initial state is the Dirac delta measure on the initial state of e. The transition function integrates the measure defined by e over all the possible states and normalize the result to produce a distribution $\mu : T \times S$ dist ($\top$ denotes the entire space). This distribution is then split into a pair of distributions using the pushforward of μ across the projections π_1 and π_2.

Remark. In ProbZelus deterministic processes that interact with the environment cannot rollback on actions based on past estimations (e.g., the command of the robot controller). However, the inferred distribution of a random variable captured in the state may evolve at each time step based on subsequent observations (e.g., the initial position of the robot). These two properties follow from the separation of the distribution of results and the distribution of states in the semantics of infer.

Alternatively, we could define a fix-point semantics of streams based on a Scott order, as simple lazy streams in Haskell, where the value of a stream can depend on future computations (as illustrated Appendix A.3). However, this approach is not practical in a reactive context where processes interact with the environment [10, 22].

4 Compilation

Following the semantics described in Section 3.3 each expression is compiled into a transition function that can be written in a simple functional first-order language extended with probabilistic operators we call μF. Importantly, the compilation process is the same for deterministic and probabilistic expressions. We can then give a classic interpretation to deterministic terms, and a measure-based semantics to probabilistic terms following [37]. We then show that the semantics of the compiled code coincides with the co-iterative semantics described in Section 3.3.

4.1 A First-Order Functional Probabilistic Language

The syntax of μF is the following:

```
d ::= let f = e | d d
e ::= c | x | (e, e) | op(e) | e(e)
    | if e then e else e | let p = e in e | fun p -> e
    | sample(e) | observe(e, e) | infer((fun x -> e), e)
p ::= x | (p, p)
```

A program is a set of definitions. An expression is either a constant, a variable, a pair, an operator, a function call, a conditional, a local definition, an anonymous function, or one of the probabilistic operators sample, or observe. The infer operator is tailored for ProbZelus and always takes two arguments: a transition function of the form fun x -> e, and a distribution of states. A type system similar to the one of ProbZelus is used to distinguish deterministic from probabilistic expressions (see Appendix B.1).

4.2 Compilation to μF

The compilation function C generates a function that closely follows the transition function defined by the co-iterative semantics presented in Section 3.3. Each expression is compiled into a function of type $S \longrightarrow T \times S$ which given a state returns a value and an updated state (see Appendix C for the complete definition). The compilation of present is thus:

```
C(present e -> e1 else e2) = fun (s,s1,s2) ->
  let v, s' = C(e)(s) in
  if v then let v1, s1' = C(e1)(s1) in (v1, (s', s1', s2))
  else let v2, s2' = C(e2)(s2) in (v2, (s', s1, s2'))
```

The probabilistic operators sample, and observe are treated as external operators. The compilation generates code that simply calls the μF version of these operators. The compilation of infer passes the distribution over state to the μF version of infer. The inference is thus aware of the distribution over state at the previous step.

```
C(infer(e)) = fun sigma ->
  let mu, sigma' = infer(C(e), sigma) in (mu, sigma')
```

The compilation of a node declaration generates two definitions: the transition function f_step and the initial state f_init. The transition function is the result of compiling the body of the node with an additional argument to capture the input. The initial step is generated by the allocation function $\mathcal{A}$ which follows the definition of the initial state in the semantics of Section 3.3 (see the Appendix C for the complete definition).

```
C(let node f x = e)  =  let f_step = fun (s,x) -> C(e)(s)
                        let f_init = 𝒜(e)
```

4.3 Semantics equivalence

We showed how to compile ProbZelus to μF a simple functional language with no loops, no recursion, and no higher-order functions, extended with the probabilistic operators. This language is similar to the kernel presented in [37] for which a measure-based probabilistic semantics is defined (see also Figure 15 of Appendix B.2).

We can now prove that the semantics of the generated code corresponds to the semantics of the source language described in Section 3.3.

Theorem. *For all ProbZelus expression e, for all state s and environment γ:*

- *if $kindOf(e) = \mathtt{D}$ then $\llbracket e \rrbracket^s_\gamma(s) = \llbracket C(e) \rrbracket_\gamma(s)$, and,*
- *if $kindOf(e) = \mathtt{P}$ then $\{\!\{ e \}\!\}^s_\gamma(s) = \{\!\{ C(e) \}\!\}_\gamma(s)$.*

Proof. The proof is done by induction on the structure of e. As an example consider the expression sample(e). If this expression is well-typed and since typing is preserved by compilation (see Lemma C.1) $kindOf(C(e)) = \mathtt{D}$. Using the induction hypothesis on $\llbracket C(e) \rrbracket_\gamma(s) = \llbracket \mathrm{e} \rrbracket_\gamma(s)$ we have:

$$\{\!\{ C(\mathsf{sample}(e)) \}\!\}_\gamma(s) = \left\{\!\!\left\{ \begin{array}{l} \mathsf{fun\ s\ \text{->}\ let\ mu,\ s' =}\ C(e)\mathsf{(s)\ in} \\ \qquad\qquad \mathsf{let\ v = sample(mu)\ in\ (v,\ s')} \end{array} \right\}\!\!\right\}_\gamma (s)$$

$$= \lambda U.\ \int \delta_{\llbracket C(e) \rrbracket_\gamma(s)}(d\mu, ds') \quad \{\!\{ \mathsf{let\ v = sample(mu)\ in\ (v,\ s')} \}\!\}_{\gamma[\mu/\mathsf{mu},\ s'/\mathsf{s'}]}(U)$$

$$= \lambda U.\ \mathit{let}\ (\mu,s) = \llbracket C(e) \rrbracket_\gamma(s)\ \mathit{in} \quad \{\!\{ \mathsf{let\ v = sample(mu)\ in\ (v,\ s')} \}\!\}_{\gamma[\mu/\mathsf{mu},\ s'/\mathsf{s'}]}(U)$$

$$= \lambda U.\ \mathit{let}\ (\mu, s') = \llbracket e \rrbracket^s_\gamma(s)\ \mathit{in}\ \int \mu(dv)\delta_{v,s'}(U)$$

$$= \{\!\{ \mathsf{sample}(e) \}\!\}^s_\gamma(s)$$

□

Remark. The probabilistic semantics of μF is commutative [37, Theorem 4]. We can thus show that the semantics of a ProbZelus program does not depend on the schedule used by the compiler to order the equations of local definitions.

5 Inference

The measure-based semantics of infer presented in Section 3.3 and Section 4.3 includes often intractable integrals. An additional challenge is to design inference techniques that can operate in bounded memory to be practical in a reactive context where the inference is a non-terminating process.

In this section, we show how to adapt particle filtering [13] to explicitly handles the state of the transition functions. We then present a novel implementation of *delayed sampling*, a recently proposed semi-symbolic inference, which enables partial exact inference over infinite streams in bounded memory for a large class of models including state-space models like the robot example of Figure 1 in Section 2.

5.1 Particle Filtering

In conventional probabilistic programming, the operational interpretation of a model is an *importance sampler* that randomly generates a sample of the model together with an importance weight measuring the quality of the sample.

Following the conventions of Section 3.3 we write $\llbracket e \rrbracket_\gamma$ for the semantics of a deterministic expression, and $\{\!\{ e \}\!\}_{\gamma,w}$ for the semantics of a probabilistic expression. The additional argument w captures the weight. The probabilistic operator sample draws a sample from a distribution without changing the score. observe increments the score by the likelihood of the observation. A deterministic expression can be lifted in a probabilistic context: the corresponding sample is the return value of the expression and the score is unchanged. The let construct illustrates that the score is accumulated along the execution path (the complete semantics is presented in Figure 18 of Appendix D).

$$\begin{array}{lcl}
\{\!\{ \mathsf{sample}(e) \}\!\}_{\gamma,w} & = & (\mathrm{draw}(\llbracket e \rrbracket_\gamma), w) \\
\{\!\{ \mathsf{observe}(e_1, e_2) \}\!\}_{\gamma,w} & = & \mathit{let}\ \mu = \llbracket e_1 \rrbracket_\gamma\ \mathit{in}\ ((), w * \mu_{\mathrm{pdf}}(\llbracket e_2 \rrbracket_\gamma)) \\
\{\!\{ e \}\!\}_{\gamma,w} & = & (\llbracket e \rrbracket_\gamma, w) \quad \mathit{if}\ \mathit{kindOf}(\mathrm{e}) = \mathtt{D} \\
\{\!\{ \mathsf{let}\ p = e_1\ \mathsf{in}\ e_2 \}\!\}_{\gamma,w} & = & \mathit{let}\ v_1, w_1 = \{\!\{ e_1 \}\!\}_{\gamma,w}\ \mathit{in}\ \{\!\{ e_2 \}\!\}_{\gamma[v_1/p],w_1}
\end{array}$$

Such a sampler is the basis of a *particle filter* or a *bootstrap filter* [13]. infer independently launches N particles. At each step, each particle samples the distribution of states σ obtained at the previous step and executes the sampler to compute a pair (result, state) along with its weight. The resulting pairs are then normalized according to their weights to form a categorical distribution μ over pairs of values and states (we write $\overline{w_i} = w_i / \sum_{i=1}^N w_i$ for the normalized weights). This distribution is then split into the distribution of returned values and the distribution of next states.

$$\begin{array}{l}
\llbracket \mathsf{infer}(\mathsf{fun}\ s\ \text{->}\ e, \sigma) \rrbracket_\gamma = \\
\quad \mathit{let}\ \mu = \lambda U.\ \sum\limits_{i=1}^{N} \begin{array}{l} \mathit{let}\ s_i = \mathrm{draw}(\llbracket \sigma \rrbracket_\gamma)\ \mathit{in} \\ \mathit{let}\ (v_i, s'_i), w_i = \{\!\{ \mathsf{fun}\ s\ \text{->}\ e \}\!\}_{\gamma,1}(s_i)\ \mathit{in} \\ \overline{w_i} * \delta_{v_i,s'_i}(U) \end{array} \\
\quad \mathit{in}\ (\pi_{1*}(\mu), \pi_{2*}(\mu))
\end{array}$$

Remark. The resampling step requires the ability to clone particles in the middle of the execution. A classic technique is to compile the model in *continuation passing style* (CPS) [35] and use the probabilistic constructs sample and observe as checkpoints for resampling. In our context, the compilation presented in Section 4 externalizes the state of the transition functions. Duplicating the state effectively clone a particle during its execution. The code does not need to be compiled in CPS form and we avoid the alignment problem [25].

$$
\begin{aligned}
&\llbracket op(e) \rrbracket_{\gamma,g,w} \;=\\
&\qquad let\ (e', g_e, w_e) = \llbracket e \rrbracket_{\gamma,g,w}\ in\ (\mathrm{app}(op, e'), g_e, w_e)\\
&\llbracket \texttt{if}\ e\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \rrbracket_{\gamma,g,w} \;=\\
&\qquad let\ e', g_e, w_e = \llbracket e \rrbracket_{\gamma,g,w}\ in\\
&\qquad let\ v, g_v = value(e', g_e)\ in\\
&\qquad if\ v\ then\ \llbracket e_1 \rrbracket_{\gamma,g_v,w_e}\ else\ \llbracket e_2 \rrbracket_{\gamma,g_v,w_e}\\
&\llbracket \texttt{sample}(e) \rrbracket_{\gamma,g,w} \;=\\
&\qquad let\ \mu, g_e, w' = \llbracket e \rrbracket_{\gamma,g,w}\ in\\
&\qquad let\ X, g' = assume(\mu, g_e)\ in\ (X, g', w')\\
&\llbracket \texttt{observe}(e_1, e_2) \rrbracket_{\gamma,g,w} \;=\\
&\qquad let\ \mu, g_1, w_1 \;= \llbracket e_1 \rrbracket_{\gamma,g,w}\ in\ let\ X, g_x = assume(\mu, g_1)\ in\\
&\qquad let\ e_2', g_2, w_2 = \llbracket e_2 \rrbracket_{\gamma,g_x,w_1}\ in\ let\ v, g_v = value(e_2', g_2)\ in\\
&\qquad let\ g' = observe(X, v, g_v)\ in\ ((), g', w_2 * \mu_{\mathrm{pdf}}(v))
\end{aligned}
$$

Figure 8. Delayed sampling sampler. Expressions return a pair (symbolic expression, weight).

5.2 Delayed Sampling

The basis of our streaming inference algorithm is delayed sampling, which we first review to explain its conceptual approach. Delayed sampling is an inference technique combining partial exact inference with approximate particle filtering to reduce estimation errors [24, 29].

In addition to the importance weight, each particle exploits *conjugacy* relationships between pairs of random variables to maintains a graph: a *Bayesian network* representing closed-form distributions involving subsets of random variables. Observations are incorporated by analytically *conditioning* the network. Particles are thus only required to draw sample when forced to, i.e., when exact computation is not possible, or when a concrete value is required.

To perform analytic computations, delayed sampling manipulates symbolic terms where random variables are referenced in the graph. The semantics of an expression $\llbracket e \rrbracket_{\gamma,g,w}$ takes an additional argument g for the graph and returns a symbolic term, an updated weight, and an updated graph. Given a graph, a symbolic term can be evaluated into a concrete value by sampling the random variables that appear in the term. The graph can be accessed and modified using the three following functions defined in [29].

$v, g' = value(e, g)$ evaluate a symbolic term and return a concrete value.

$X, g' = assume(\mu, g)$ add a random variable $X \sim \mu$ to the graph and return the variable.

$g' = observe(X, v, g)$ condition the graph by observing the value v for the variable X.

Compared to the particle filter any expression, probabilistic or deterministic, can contribute to a symbolic term. The evaluation function $\llbracket e \rrbracket_{\gamma,g,w}$ partially presented in Figure 8 must thus be defined on the entire language and not only on probabilistic constructs. For instance, the application of an operator $op(e)$ returns a symbolic term app(op, e') that represents the application of op on the evaluation of e. Some constructs are partially evaluated, e.g., the condition e of if e then e_1 else e_2 must be a concrete value.

The probabilistic sample(e) adds a new random variable to the graph without drawing a sample. observe(e_1, e_2) adds a new random variable $X \sim \mu$ where μ is defined by e_1, then computes a concrete value v for e_2 and conditions the graph by observing the value v for X. As for the particle filter, the score is incremented by the likelihood of the observation.

Symbolic Computations. The functions *value*, *assume*, and *observe* used in Figure 8 rely on the following mutually recursive lower level operations (Y is the parent of X) [29]:

$X, g' = initialize(\mu, Y, g)$ add a new node X with a distribution $p_{X|Y} = \mu$ as a child of Y in g.

$g' = marginalize(X, g)$ compute and store p_X in g' from p_Y and $p_{X|Y}$ where p_Y and $p_{X|Y}$ are in g.

$g' = realize(X, v, g)$ assign in g' a concrete value to a random variable X.

$g' = condition(Y, g)$ compute $p_{Y|X}$ given $p_X, p_{X|Y}$, and a concrete value $X = v$ where v is in g.

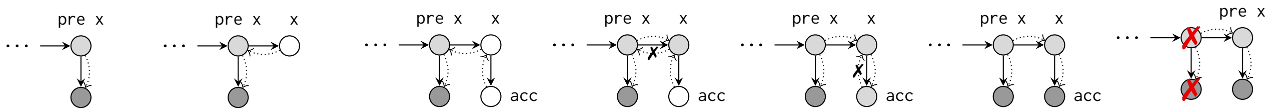
In the class of Bayesian networks maintained by the delayed sampler, marginalization w.r.t. a parent node, and conditioning a parent on the value of a child are tractable operations.

To reflect these operations, nodes are characterized by a state (see Figures 4 and 9). *Initialized* nodes ○ are random variables with a conditional distribution $p_{X|Y}$ where the parent Y has no concrete value yet. *Marginalized* nodes ◍ are random variables with a marginal distribution p_X that incorporate the distributions of the ancestors. *Realized* nodes ● are random variables that have been assigned a concrete value via sampling or observation.

The evaluation function $value(e, g)$ forces the realization by sampling of all the random variables referenced in e to produce a concrete value. Similarly, the function $observe(X, v, g)$ realizes a variable X with a given observation v. The realization of a random variable comprises three steps: (1) compute the distribution $p(x)$ by recursively marginalizing the parents from a root node, (2) sample a value, or use the observation, and (3) use the concrete value to update the children and condition the parent which remove the dependencies.

The function $assume(\mu, g)$ adds a new node to the graph and is defined case by case on the shape of the symbolic term μ. If there is a conjugacy relationship between μ and a random variable Y present in the graph, e.g., $\mu = Bernoulli(Y)$ with $Y \sim Beta(\alpha, \beta)$, a new initialized node $X \sim \mu$ is added as a child of Y. Otherwise, since symbolic computation is not possible, dependencies are broken by realizing the random variables that appear in μ, e.g., $\mu' = Bernoulli(value(Y, g))$, and $X \sim \mu'$ is added as a new root node.

Inference. The inference scheme is similar to the particle filter. At each step, the inference draws N states from σ to execute the transition function. For each particle, execution starts with the graph computed at the previous step and



(a) Initial state **(b)** *init*(x, pre x) **(c)** *init*(y, x) **(d)** *marg*(x) **(e)** *marg*(y) **(f)** *realize*(y) **(g)** Update state

Figure 9. One step of the robot example of Figure 1 with SDS. Plain arrows represent dependencies and dotted arrows represent pointers at runtime. The sample statement adds the initialized nodes x (b). The observe statement adds the initialized node a (c), triggers the marginalizations of x (d) and a (e), and assigns to a its observed value (f). When the state is updated, the value x becomes pre x. The previous values are not referenced anymore and can be removed (g).

returns a pair of symbolic terms (result, state), the particle weight, and the updated graph. The function *distribution*(e, g) returns the distribution corresponding to the expression e without altering the graph. Results are then aggregated in a mixture distribution (concrete values are lifted to Dirac distribution) where the distribution d_i operates on the value component of U and we use the pair (symbolic term, graph) computed by the transition function for the distribution of state. This distribution is then split into the distribution of returned values and the distribution of next states.

$$
\begin{array}{l}
[\![\text{infer}(\text{fun } s \text{ -> } e, \sigma)]\!]_{\gamma} = \\
\quad \mathit{let}\ \mu = \lambda U.\ \sum_{i=1}^{N} \begin{array}[t]{l} \mathit{let}\ s_i, g_i = \text{draw}([\![\sigma]\!]_{\gamma})\ \mathit{in} \\ \mathit{let}\ (e_i, s_i'), w_i, g_i' = \{\!|\text{fun } s \text{ -> } e|\!\}_{\gamma,1,g_i}(s_i)\ \mathit{in} \\ \mathit{let}\ d_i = \mathit{distribution}(e_i, g_i')\ \mathit{in} \\ \overline{w_i} * d_i(\pi_1(U)) * \delta_{s_i', g_i'}(\pi_2(U)) \end{array} \\
\quad \mathit{in}\ (\pi_{1*}(\mu), \pi_{2*}(\mu))
\end{array}
$$

5.3 Streaming Delayed Sampling

As illustrated in Section 2.2, a notable challenge with the traditional delayed sampling algorithm is that that graph grows linearly in the number of samples. In the original formulation of delayed sampling [29], graph edges are only removed when a node is realized. All nodes that have been neither sampled nor observed are thus kept in the graph even if they are no longer referenced by the program. In a reactive programming context, such an implementation can consume unbounded memory.

Bounded Delayed Sampling. A simple mitigation is to limit the scope of symbolic computations to one time step and discard the graph at the end of each time step. We call this inference technique *bounded delayed sampling* (BDS).

BDS performs symbolic computations during the execution of a time step, and whenever possible, delays the sampling until the end of the instant. As for the particle filters, BDS guaranties a bounded-memory execution. For each particle, the size of the graph is bounded by the number of variables introduced during a time step, which by construction, is bounded for any valid ProbZelus program.

Streaming Delayed Sampling. Compared to the original delayed sampling algorithm, BDS looses the ability to perform symbolic computations using variables defined at different time steps. This can results in a significant loss of precision for models with inter-steps dependencies such as the robot example of Figure 1. To adapt delayed sampling to a streaming settings while keeping its maximum accuracy, we designed a delayed sampler that is *pointer-minimal* where nodes that are no longer referenced by the program can be eventually removed. We call this inference *streaming delayed sampling* (SDS). SDS enables partial exact inference in bounded memory for a large class of models.

In the original implementation of delayed sampling, graph nodes need to access their parents and children. Marginalization requires access to the parent to incorporate the ancestor distribution. Realization requires access to both the parent and the children of a node to update their respective distributions with the concrete value assigned to the node.

In the pointer minimal implementation, initialized nodes only keep a pointer to their parent to follow the ancestor chain during marginalization and marginalized nodes only keep a pointer to a marginalized child (Delayed Sampling imposes that a node always has at most one marginalized child). Compared to the original implementation, marginal nodes only keep track of one child, and marginalization turns backward pointers to the parent node into forward pointers to the marginalized child. Note that this implementation prevents updating the children when the parent is realized, and prevent conditioning a parent when a child is realized. Instead, when marginalizing a node, the sampler first checks if the parent is realized to apply the update. Symmetrically, to realize a node, the sampler first checks if the children are realized and, if necessary, condition the distribution before assigning the concrete value.

Figure 9 shows the evolution of the graph during one step for the robot example of Figure 1. At the end of the step, the value of pre x is updated. The previous value is not referenced anymore by the program and the node can be removed from the graph. In the original implementation, backward pointers between marginalized nodes prevent the collection (see Figure 4).

Table 1. Benchmarks with: inference of *fixed* parameters from observations, estimation of a *moving* state (state-space model), and *inference-in-the-loop* (IITL).

	Fixed	Moving	IITL	Metric
Beta-Bernoulli	✓			MSE
Gaussian-Gaussian	✓			MSE
Kalman-1D		✓		MSE
Outlier	✓	✓		MSE
Robot		✓	✓	LQR
SLAM	✓	✓	✓	MSE
MTT		✓		MOTA*

Limitations. With SDS, models like the robot example that only maintain bounded chains of dependencies between variables are guaranteed to be executed in bounded memory. The class of models that can be executed in bounded memory with our pointer-minimal implementation thus already comprises state-space models like Kalman filters, and models for learning unknown constant parameters from a series of observations (e.g., computing the bias of a coin from a succession of flips) where variables introduced at each step are immediately realized.

However, unbounded chains can still be formed if the program keeps a reference to a constant variable that is never realized. In addition, in ProbZelus, at each step the inference returns a snapshot of the current distribution without forcing the realization of any node in the graph. Compared to the original delayed sampling implementation, initialized nodes can be inspected without being realized. It is thus possible to form unbounded chains of initialized nodes which cannot be pruned even when nodes are no longer referenced in the program due to the backward pointers to the parent in initialized nodes.

To mitigate these issues, we can force the realization of trailing nodes at each step as in *bounded delayed sampling* or use a sliding window. Alternatively, the *value* function is available to the programmer and can be used to implement any strategy to force the evaluation of the nodes.

6 Evaluation

We next evaluate the performance of ProbZelus on a set of benchmarks that illustrate multiple aspects of the language: inferring fixed parameters from observations, online trajectory estimation, inference-in-the-loop. For these examples, we compare the accuracy and the latency cost of the three inference techniques: PF, BDS, and SDS. Appendix E details our implementation as an extension of the Zelus compiler.

Benchmarks. The models used in the experiments are summarized in Table 1 (a detailed description along with the code of the benchmarks is given in Appendix F.1). Two models infer fixed parameters from a series of observations. Beta-Bernoulli estimates the parameter of a Bernoulli distribution from a series of binary observations (e.g., the bias of a coin). Gaussian-Gaussian estimates the mean and variance of a Gaussian distribution from a series of observations. The accuracy metric is the *Mean Squared Error* (MSE) of the inferred parameters compared to their exact values.

Two models infer the state of a moving agent from noisy observations. Kalman-1D is a one-dimensional Kalman filter that models an agent that estimates its trajectory from noisy observations. Outlier adapted from [27] models the same situation as Kalman-1D, but the sensor occasionally produces invalid readings. This models infer both the trajectory of the agent, and the bias of the sensor. The accuracy metric is the *Mean Squared Error* (MSE) of the inferred trajectory compared to the exact positions.

Two models use inference-in-the-loop (IITL). Robot is the robot example of Section 2 and the accuracy metric is the LQR loss. SLAM (*Simultaneous Localization and Mapping*) adapted from [15] models an agent that estimates its position and a map of its environment. In this simplified version, the agent moves in a one-dimensional grid where each cell is either black or white. The robot's wheels may slip causing the robot to unknowingly stay in place (noisy motion), and the sensor is not perfect an may accidentally report the wrong color (noisy observations). At each step the robot uses the inferred position to decide its next move. The accuracy metric is the MSE of both the position and the map.

MTT (*Multi-Target Tracker*) adapted from [30] is a model where there are a variable number of targets with linear-Gaussian motion models with a state space of 2D position and velocity, producing linear-Gaussian measurements of the position at each time step. Targets randomly appear according to a Poisson process and each disappear with fixed probability at each step. Measurements do not identify which target they came from, and "clutter" measurements that come not from targets but from some underlying distribution add to observations, complicating inference of which measurements are associated to which targets. The accuracy metric is expected $\mathsf{MOTA}* = (1/\mathsf{MOTA}) - 1$ where $\mathsf{MOTA} \in [0, 1]$ is the *Multiple Object Tracking Accuracy* [2].

Experimental Setup. All the experiments were run on a Google Cloud n1-highmem-4 machine (4 vCPUs, 26 GB memory) on a sole-tenant node. We ran all the benchmarks for $1,600$ steps, except MTT that we ran for 250 steps because of its higher step latency per particle. In all cases, the inference runs in bounded memory (see Appendix F.3).

For each inference algorithm, we evaluated how much time it required to achieve 90% of runs close to a loss target (out of 1000 runs total):

$$|\log(P_{90\%}(loss)) - \log(loss_{target})| < 0.5.$$

For each benchmark, we chose the median loss of SDS at 500 particles as $loss_{target}$ for that benchmark. We measured the number of particles required to achieve this loss, and then measured per-step latency at this particle count (in

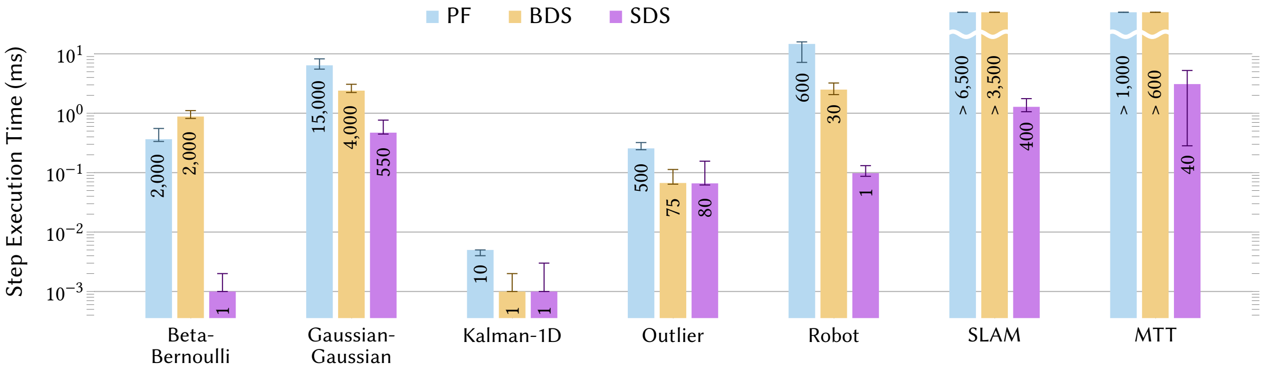


Figure 10. Step latency comparison when 90% of 1000 runs reach an accuracy similar to the baseline (mean accuracy of SDS with 500 particles). The number of particles required to reach this accuracy is shown on top of the bars. The error bars show the 10th and 90th percentiles.

Appendix F.2 we also evaluate loss and step latency across a fixed range of particle counts).

Results. Figure 10 shows the results. The height of each bar is the median step latency, and the error bars are 90% and 10% quantiles, aggregated over 1000 runs. Each bar is labeled with the minimum number of particles required to achieve the accuracy threshold, accurate to 1.5 significant digits (100, 150, 200, 250, . . .). We observe that SDS is able to compute an exact solution for Beta-Bernoulli, Kalman-1D, and Robot. In all these examples 1 particle is already enough to reach the target accuracy. Overall, the latency results show that the number of particles required to reach the desired accuracy with PF implies a significant slowdown compared to SDS. Moreover, the SLAM and MTT benchmark shows that, in some cases, PF is not an option: the target accuracy was not reached with 5, 000 and 1, 000 particles, respectively, at which point PF was already 10 times slower than SDS and we stopped the experiments.

As expected, BDS performance numbers are between those of PF and SDS. At worst, there is no possible intra-step symbolic computations (e.g., Beta-Bernoulli) and BDS behaves like a particle filter which requires as many particles as PF. At best, BDS performs as well as SDS (e.g., Outlier).[3]

Additionally, Figure 10 also shows that for a given number of particles, the overhead induced by managing the delayed sampling graph is significant. Compared to BDS and SDS, depending on the benchmark, it is possible to use 2 to 4 times as many particles for PF with the same latency. However, this is not enough to match the gain in accuracy.

7 Related Work

Probabilistic Programming. Over the last few years there has been a growing interest on probabilistic programming languages. Some languages like BUGS [26], Stan [9], or Augur [21] offer optimized inference technique for a constrained subset of models. Other languages like WebPPL [18], Edward [39], Pyro [5], or Birch [30] focus on expressivity allowing the specification of arbitrary complex models. Compared to these languages, ProbZelus can be used to program *reactive models* that typically do not terminate, and inference can be run in parallel with deterministic components that interact with an environment.

Reactive Languages with Uncertainty. Lutin is a language for describing non-deterministic reactive systems for testing and simulation [34], but while Lutin supports weighted sampling to describe constrained random scenarios, it does not support inference. ProPL [32] is a language to describe probabilistic models for process that evolve over a period of time. This language also extend a probabilistic language with a notion of processes that can be composed in parallel, but compared to ProbZelus, ProPL focuses on a constrained class of models that can be interpreted as *Dynamic Bayesian Networks* (DBN), and relies on standard DBN inference techniques. In the same vein, CTPPL [33] is a language to describe continuous time processes where the amount of time taken by a sub-process can be specified by a probabilistic model. These models cannot be expressed in ProbZelus which relies on the synchronous model of computation. It would be interesting to investigate how to extend ProbZelus to continuous models based on Zelus' support for continuous-time ordinary differential equations (ODE) [7].

Inference. Researchers have proposed streaming inference algorithms, including variational [8], or sampling-based [14, 19] approaches. Popular languages like Stan, Edward, or Pyro, offer support to stream data through the model during inference to handle large datasets. However, compared to ProbZelus, the model must be defined a priori and does not evolve during the inference.

The Anglican and Birch probabilistic programming languages support delayed sampling [29]. These languages do

[3] For this example the difference between SDS and BDS can be explained by the high variance in the results

not support streaming inference or reactive programming. Again, their interfaces only support inference on a complete probabilistic model.

8 Conclusion

Modeling uncertainty is a primary element of control systems for tasks that operate under the assumption of a probabilistic model of their environment (e.g., object tracking). While synchronous languages have developed as a prominent way to develop control applications, to date there has been limited work in these languages on programming language support for modeling uncertainty.

In this paper we present ProbZelus, the first synchronous probabilistic programming language that lifts emerging abstractions for probabilistic programming into the reactive setting thus enabling *inference-in-the-loop*. Moreover, our streaming delayed sampling algorithm provides efficient, hybrid inference while still satisfying a key requirement of control applications in that they must execute with bounded resources.

Our results demonstrate that ProbZelus enables us to write, in the very same source, a deterministic model for the control software and a probabilistic model for its behavior and environment with complex interactions between the two.

References

[1] A. Benveniste, P. Caspi, S.A. Edwards, N. Halbwachs, P. Le Guernic, and R. de Simone. 2003. The synchronous languages 12 years later. *Proc. IEEE* 91, 1 (Jan. 2003).

[2] Keni Bernardin and Rainer Stiefelhagen. 2008. Evaluating Multiple Object Tracking Performance: The CLEAR MOT Metrics. *EURASIP J. Image and Video Processing* 2008 (2008).

[3] G. Berry. 1989. Real time programming: Special purpose or general purpose languages. *Information Processing* 89 (1989), 11–17.

[4] Darek Biernacki, Jean-Louis Colaco, Grégoire Hamon, and Marc Pouzet. 2008. Clock-directed Modular Code Generation of Synchronous Dataflow Languages. In *ACM International Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES).* Tucson, Arizona.

[5] Eli Bingham, Jonathan P. Chen, Martin Jankowiak, Fritz Obermeyer, Neeraj Pradhan, Theofanis Karaletsos, Rohit Singh, Paul A. Szerlip, Paul Horsfall, and Noah D. Goodman. 2019. Pyro: Deep Universal Probabilistic Programming. *Journal of Machine Learning Research* 20 (2019), 28:1–28:6.

[6] Timothy Bourke and Marc Pouzet. 2013. *Zélus, a Hybrid Synchronous Language.* École normale supérieure. http://zelus.di.ens.fr Distribution at: `zelus.di.ens.fr`.

[7] Timothy Bourke and Marc Pouzet. 2013. Zélus, a Synchronous Language with ODEs. In *International Conference on Hybrid Systems: Computation and Control (HSCC 2013).* ACM, Philadelphia, USA.

[8] Tamara Broderick, Nicholas Boyd, Andre Wibisono, Ashia C. Wilson, and Michael I. Jordan. 2013. Streaming Variational Bayes. In *Advances in Neural Information Processing Systems 26: 27th Annual Conference on Neural Information Processing Systems 2013. Proceedings of a meeting held December 5-8, 2013, Lake Tahoe, Nevada, United States.* 1727–1735.

[9] Bob Carpenter, Andrew Gelman, Matthew D Hoffman, Daniel Lee, Ben Goodrich, Michael Betancourt, Marcus Brubaker, Jiqiang Guo, Peter Li, and Allen Riddell. 2017. Stan: A probabilistic programming language. *Journal of Statistical Software* 76, 1 (2017), 1–37.

[10] Paul Caspi. 1992. Clocks in dataflow languages. *Theoretical Computer Science* 94, 1 (1992), 125 – 140. https://doi.org/10.1016/0304-3975(92)90326-B

[11] Paul Caspi and Marc Pouzet. 1998. A Co-iterative Characterization of Synchronous Stream Functions. In *Coalgebraic Methods in Computer Science (CMCS'98) (Electronic Notes in Theoretical Computer Science).*

[12] Jean-Louis Colaco, Bruno Pagano, and Marc Pouzet. 2017. Scade 6: A Formal Language for Embedded Critical Software Development. In *Eleventh International Symposium on Theoretical Aspect of Software Engineering (TASE).* Sophia Antipolis, France.

[13] Pierre Del Moral, Arnaud Doucet, and Ajay Jasra. 2006. Sequential monte carlo samplers. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)* 68, 3 (2006), 411–436.

[14] Arnaud Doucet, Nando de Freitas, Kevin Murphy, and Stuart Russell. 2000. Rao-blackwellised Particle Filtering for Dynamic Bayesian Networks. In *UAI.*

[15] Arnaud Doucet, Nando de Freitas, Kevin P. Murphy, and Stuart J. Russell. 2000. Rao-Blackwellised Particle Filtering for Dynamic Bayesian Networks. In *UAI '00: Proceedings of the 16th Conference in Uncertainty in Artificial Intelligence, Stanford University, Stanford, California, USA, June 30 - July 3, 2000.* 176–183.

[16] Daniel Fink. 1997. A Compendium of Conjugate Priors.

[17] Timon Gehr, Sasa Misailovic, and Martin Vechev. 2016. PSI: Exact Symbolic Inference for Probabilistic Programs. In *Computer Aided Verification (CAV).*

[18] Noah D. Goodman and Andreas Stuhlmüller. 2014. The Design and Implementation of Probabilistic Programming Languages. http://dippl.org Accessed February 2019.

[19] N.J. Gordon, D.J. Salmond, and A.F.M. Smith. 1993. Novel approach to nonlinear/non-Gaussian Bayesian state estimation. In *IEE-F.*

[20] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. 1991. The Synchronous Dataflow Programming Language LUSTRE. *Proc. IEEE* 79, 9 (September 1991), 1305–1320.

[21] Daniel Huang, Jean-Baptiste Tristan, and Greg Morrisett. 2017. Compiling Markov chain Monte Carlo algorithms for probabilistic modeling. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*. 111–125.

[22] Gilles Kahn. 1974. The semantics of a simple language for parallel programming. *Information processing* 74 (1974), 471–475.

[23] R.Ẽ. Kalman. 1960. A New Approach to Linear Filtering and Prediction Problems. In *Journal of Basic Engineering*. 35–45.

[24] Daniel Lundén. 2017. *Delayed sampling in the probabilistic programming language Anglican*. Ph.D. Dissertation. KTH Royal Institute of Technology.

[25] Daniel Lundén, David Broman, Fredrik Ronquist, and Lawrence M. Murray. 2018. Automatic Alignment of Sequential Monte Carlo Inference in Higher-Order Probabilistic Programs. *CoRR* abs/1812.07439 (2018). http://arxiv.org/abs/1812.07439

[26] David Lunn, David Spiegelhalter, Andrew Thomas, and Nicky Best. 2009. The BUGS project: Evolution, critique and future directions. *Statistics in medicine* 28, 25 (2009), 3049–3067.

[27] P. Minka, Thomas. 2001. Expectation Propagation for Approximate Bayesian Inference. In *UAI '01*. 362–369.

[28] Michael Montemerlo, Sebastian Thrun, Daphne Koller, and Ben Wegbreit. 2002. FastSLAM: A Factored Solution to the Simultaneous Localization and Mapping Problem. In *Eighteenth National Conference on Artificial Intelligence*. American Association for Artificial Intelligence, 593–598.

[29] 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 2018*.

[30] Lawrence M. Murray and Thomas B. Schön. 2018. Automated learning with a probabilistic programming language: Birch. *Annual Reviews in Control* 46 (2018), 29–43. https://doi.org/10.1016/j.arcontrol.2018.10.013

[31] Praveen Narayanan, Jacques Carette, Wren Romano, Chung-chieh Shan, and Robert Zinkov. 2016. Probabilistic Inference by Program Transformation in Hakaru (System Description). In *FLOPS*.

[32] Avi Pfeffer. 2005. Functional Specification of Probabilistic Process Models. In *Proceedings, The Twentieth National Conference on Artificial Intelligence and the Seventeenth Innovative Applications of Artificial Intelligence Conference, July 9-13, 2005, Pittsburgh, Pennsylvania, USA*. 663–669.

[33] Avi Pfeffer. 2009. CTPPL: A Continuous Time Probabilistic Programming Language. In *IJCAI 2009, Proceedings of the 21st International Joint Conference on Artificial Intelligence, Pasadena, California, USA, July 11-17, 2009*. 1943–1950.

[34] Pascal Raymond, Yvan Roux, and Erwan Jahier. 2008. Lutin: A Language for Specifying and Executing Reactive Scenarios. *EURASIP Journal of Embedded Sytems* (2008). https://doi.org/10.1155/2008/753821

[35] Daniel Ritchie, Andreas Stuhlmüller, and Noah D. Goodman. 2016. C3: Lightweight Incrementalized MCMC for Probabilistic Programs using Continuations and Callsite Caching. In *Proceedings of the 19th International Conference on Artificial Intelligence and Statistics, AISTATS 2016, Cadiz, Spain, May 9-11, 2016*. 28–37.

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

[37] Sam Staton. 2017. Commutative Semantics for Probabilistic Programming. In *Programming Languages and Systems - 26th European Symposium on Programming, ESOP 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings*. 855–879.

[38] David Tolpin, Jan-Willem van de Meent, Hongseok Yang, and Frank Wood. 2016. Design and Implementation of Probabilistic Programming Language Anglican. In *Symposium on the Implementation and Application of Functional Programming Languages (IFL)*. 6:1–6:12.

[39] Dustin Tran, Matthew D. Hoffman, Rif A. Saurous, Eugene Brevdo, Kevin Murphy, and David M. Blei. 2017. Deep Probabilistic Programming. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*.

[40] Yi Wu, Lei Li, Stuart J. Russell, and Rastislav Bodík. 2016. Swift: Compiled Inference for Probabilistic Programming Languages. In *IJCAI*.

A ProbZelus

In this section, we provide the complete definition of the ProbZelus typing and semantics. It is defined on the language introduced Section 3 extended with the probabilistic operator factor(e) which is equivalent to observe(exp(1), e). Intuitively, it directly updates the weight of the execution path using the value of an expression e.

A.1 Typing

The type system that discriminates deterministic from probabilistic expressions is defined Figure 11. To simplify the presentation, we ignored the polymorphism of the datatypes.

The sub-typing rule indicates that any deterministic expression can be lifted into a probabilistic one. Expressions like constants, variables, and last are deterministic. The kind of classic Zelus expressions (pairs, *op*, local definitions, present, and reset) is the kind of their body. Similarly, the kind of equations is the kind of their defining expression, and parallel composition imposes the same kind for all the equations. Note that it is always possible to compose deterministic and probabilistic computations. The fact that in a rule each sub-expression must share the same kind k enforces the use of the sub-typing rule to lift deterministic expressions.

The expressions sample, factor, and observe are probabilistic. The transition from probabilistic to deterministic is realized via infer. It is a deterministic expression whose body is always probabilistic. Probabilistic expressions thus only exist under an infer.

Other Static Analyses. The Zelus compiler statically checks initialization, and causality of the program [6]. These analyses respectively guarantee that there exists a schedule of parallel equations that makes the streams productive. Extending these analyses to the probabilistic operators is straightforward. For the initialization and causality analyses, probabilistic operators can be treated as external operators.

A.2 Co-iterative Semantics

The co-iterative semantics of ProbZelus's deterministic processes is inspired by [11] and defined Figure 12.

A *node* is a stream function of type $T \longrightarrow^{\mathsf{D}} T'$. In addition to the state, the transition function thus takes an additional input of type T and returns a pair (result, next state)

$$CoNode(T, T', S) = S \times (S \longrightarrow T \longrightarrow T' \times S).$$

The transition function of a variable always returns the corresponding value stored in the environment γ. The semantics of last x is a simple access to a special variable x_last. present e -> e_1 else e_2 introduced in Section 2 returns the value of e_1 when e is true and the value of e_2 otherwise. The state (s, s_1, s_2) stores the state of the three sub-expressions. The transition function lazily executes e_1 or e_2 depending on the value of e and returns the updated state.

The state of a set of scheduled locally recursive definitions e where rec E comprises three parts: the value of the local variables at the previous step which can be accessed via the last operator, the state of the defining expressions, and the state of expression e. The initialization stores the initial values introduced by init and the initial states of all sub-expressions. The transition function incrementally builds the local environment defined by E. First the environment is populated with a set of fresh variables x_i_last initialized with the values stored in the state that can then be accessed via the last operator. Then the environment is extended with the definition of all the variables y_i by executing all the defining expressions (where $\{x_i\}_{1..k} \cap \{y_j\}_{1..n} = \{x_i\}_{1..k}$). Finally, the expression e is executed in the final environment. The updated state contains the value of the initialized variables defined in E that will the be used to start the next step, and the updated state of the sub-expressions.

Probabilistic Extensions. The semantics of the probabilistic part of ProbZelus is defined Figure 13. It follows the same structure as the deterministic semantics but defines measures over all possible executions as in [37]. In particular a succession of computation is interpreted as sequentially integrating over the results of the preceding computations.

As for deterministic nodes, the transition function of a probabilistic node of type $T \longrightarrow^{\mathsf{P}} T'$ takes an additional argument and returns a measure over pairs (result, next state).

$$CoPNode(T, T', S) = S \times (S \to T \to (\Sigma_{T' \times S} \to [0, \infty]))$$

A.3 Alternative semantics

We could give different semantics to ProbZelus. For example consider the following probabilistic node.

```
let proba kahn_vs_scott () = p where
  rec init p = sample(beta(1, 1))
  and () = observe(bernoulli(p), true)
```

With the semantics defined Section 3, this program produces the stream of distribution: $Beta(2, 1), Beta(3, 1), \ldots$ Note that, even though, p is defined as a constant, its distribution evolves at each steps.

Since the observe statement uses the constant true, we know that p is necessarily 1. An alternative semantics could thus returns the constant stream of distributions: δ_1.

B The μF language

Similarly to ProbZelus, we extends μF with the probabilistic operator factor. We now present the complete type system and semantics for μF.

B.1 Typing

The type system defined Figure 14 is similar to the one Figure 11. It is also used to distinguish deterministic from probabilistic expressions, but with additional restrictions since

$$\frac{G \vdash^{\mathsf{D}} e : t}{G \vdash^{\mathsf{P}} e : t} \qquad \frac{\mathit{typeOf}(c) = t}{G \vdash^{\mathsf{D}} c : t} \qquad \frac{G(x) = t}{G \vdash^{\mathsf{D}} x : t} \qquad \frac{G \vdash^{k} e_1 : t_1 \quad G \vdash^{k} e_2 : t_2}{G \vdash^{k} (e_1, e_2) : t_1 \times t_2} \qquad \frac{\mathit{typeOf}(op) = t_1 \rightarrow^{\mathsf{D}} t_2 \quad G \vdash^{k} e : t_1}{G \vdash^{k} op(e) : t_2}$$

$$\frac{G(f) = t_1 \rightarrow^{k} t_2 \quad G \vdash^{\mathsf{D}} e : t_1}{G \vdash^{k} f(e) : t_2} \qquad \frac{G(x) = t}{G \vdash^{\mathsf{D}} \texttt{last}\ x : t} \qquad \frac{G \vdash^{k} E : G' \quad G + G' \vdash^{k} e : t}{G \vdash^{k} e\ \texttt{where rec}\ E : t}$$

$$\frac{G \vdash^{k} e : \text{bool} \quad G \vdash^{k} e_1 : t \quad G \vdash^{k} e_2 : t}{G \vdash^{k} \texttt{present}\ e \texttt{ -> } e_1\ \texttt{else}\ e_2 : t} \qquad \frac{G \vdash^{k} e_1 : t \quad G \vdash^{k} e_2 : \text{bool}}{G \vdash^{k} \texttt{reset}\ e_1\ \texttt{every}\ e_2 : t}$$

$$\frac{G \vdash^{\mathsf{D}} e : t\ \text{dist}}{G \vdash^{\mathsf{P}} \texttt{sample}(e) : t} \qquad \frac{G \vdash^{\mathsf{D}} e_1 : t\ \text{dist} \quad G \vdash^{\mathsf{D}} e_2 : t}{G \vdash^{\mathsf{P}} \texttt{observe}(e_1, e_2) : \text{unit}} \qquad \frac{G \vdash^{\mathsf{D}} e : \text{float}}{G \vdash^{\mathsf{P}} \texttt{factor}(e) : \text{unit}} \qquad \frac{G \vdash^{\mathsf{P}} e : t}{G \vdash^{\mathsf{D}} \texttt{infer}(e) : t\ \text{dist}}$$

$$\frac{G \vdash^{k} e : t}{G \vdash^{k} x = e : [t/x]} \qquad \frac{G \vdash^{k} e : t}{G \vdash^{k} \texttt{init}\ x = e : [t/x]} \qquad \frac{G + G_1 + G_2 \vdash^{k} E_1 : G_1 \quad G + G_1 + G_2 \vdash^{k} E_2 : G_2}{G \vdash^{k} E_1\ \texttt{and}\ E_2 : G_1 + G_2}$$

$$\frac{G + [t_1/x] \vdash^{\mathsf{D}} e : t_2}{G \vdash^{\mathsf{D}} \texttt{let node}\ f\ x = e : G + [t_1 \rightarrow^{\mathsf{D}} t_2/f]} \qquad \frac{G + [t_1/x] \vdash^{\mathsf{P}} e : t_2}{G \vdash^{\mathsf{D}} \texttt{let proba}\ f\ x = e : G + [t_1 \rightarrow^{\mathsf{P}} t_2/f]} \qquad \frac{G \vdash^{\mathsf{D}} d_1 : G_1 \quad G_1 \vdash^{\mathsf{D}} d_2 : G_2}{G \vdash^{\mathsf{D}} d_1\ d_2 : G_2}$$

Figure 11. Typing with deterministic and probabilistic kinds.

the compiled code is in a more constrained form. Whenever possible we require sub-expressions to be deterministic, that is, in pairs, operator applications (including sample, factor, and observe), function calls, and the condition of a if/then/else. These restrictions simplify the presentation of the semantics but do not reduce the expressiveness of the language since it is always possible to introduce additional local definitions to name intermediate probabilistic expressions. For example if sample(bernoulli(0.5)) then ... can be rewritten let b = sample(bernoulli(0.5)) in if b then ...

B.2 Semantics of μF

The semantics of μF follows [37]. In a deterministic context $\mathit{kindOf}(e) = \mathsf{D}$, the semantics $[\![e]\!]_\gamma$ of an expression is the classic interpretation of a strict functional language. In a probabilistic context ($\mathit{kindOf}(e) = \mathsf{P}$), we define a the measure-based semantics $\{\![e]\!\}_\gamma$.

The probabilistic semantics of μF is presented in Figure 15. A deterministic expression is lifted to a probabilistic expression using the the Dirac delta measure applied to the value of the expression computed by the deterministic semantics. As in Section 3.3, a local definition let $x = e_1$ in e_2 is interpreted as integrating e_2 over the measure defined by e_1. The semantics of the probabilistic operators is the following: sample(e) returns the distribution $[\![e]\!]_\gamma$. factor(e) returns a measure defined on the singleton space () whose value is $\exp([\![e]\!]_\gamma)$. observe(e_1, e_2) is similar but the score is given by the density function of the distribution $[\![e_1]\!]_\gamma$ applied to $[\![e_2]\!]_\gamma$.

Inference. infer handles the transition function generated by the compilation of Section 4. The first argument of infer is a transition function, and the second argument a distribution over state σ. The inference first integrates over the distribution σ and then normalizes the result to produce a distribution μ of pairs (result, next state). The special value $\top$ denotes the entire space (value, state). This distribution is then decomposed into a pair of distributions using the pushforward of μ.

C Compilation

Figure 17 presents the entire compilation function from ProbZelus to μF introduced Section 4. Figure 16 is the allocation function.

Lemma C.1. *The compilation preserves the kind (deterministic* D*, or probabilistic* P*) of the expressions. For any expression e, if $G \vdash^{k} e : t$, there exists G' and t' such that $G' \vdash^{k} C(e) : t'$.*

Proof. By induction on the structure of e. □

Remark. The compilation presented in Figure 17 generates a function for each sub-expression. However, in most cases it is possible to simplify the code using static reduction. For instance, a constant can directly be compiled into a constant.

D Inference

D.1 Importance Sampling

Importance sampling. The most simple inference independently launches N *particles.* Each particle executes the importance sampler to compute a pair (result, weight). Results are then normalized in a *categorical distribution*, i.e., a discrete distribution over the results.

$$
\begin{array}{lcl}
\llbracket c \rrbracket^i_\gamma & = & () \\
\llbracket c \rrbracket^s_\gamma & = & \lambda s.\ (c, s) \\[4pt]
\llbracket x \rrbracket^i_\gamma & = & () \\
\llbracket x \rrbracket^s_\gamma & = & \lambda s.\ (\gamma(x), s) \\[4pt]
\llbracket \mathsf{last}\ x \rrbracket^i_\gamma & = & () \\
\llbracket \mathsf{last}\ x \rrbracket^s_\gamma & = & \lambda s.\ (\gamma(x\_\mathsf{last}), s) \\[4pt]
\llbracket (e_1,e_2) \rrbracket^i_\gamma & = & (\llbracket e_1 \rrbracket^i_\gamma, \llbracket e_2 \rrbracket^i_\gamma) \\
\llbracket (e_1,e_2) \rrbracket^s_\gamma & = & \lambda(s_1, s_2).\ \mathit{let}\ v_1, s_1' = \llbracket e_1 \rrbracket^s_\gamma(s_1)\ \mathit{in} \\
& & \qquad\qquad \mathit{let}\ v_2, s_2' = \llbracket e_2 \rrbracket^s_\gamma(s_2)\ \mathit{in}\ ((v_1,v_2), (s_1', s_2')) \\[4pt]
\llbracket op(e) \rrbracket^i_\gamma & = & \llbracket e \rrbracket^i_\gamma \\
\llbracket op(e) \rrbracket^s_\gamma & = & \lambda s.\ \mathit{let}\ v, s' = \llbracket e \rrbracket^s_\gamma(s)\ \mathit{in}\ (op(v), s') \\[4pt]
\llbracket f(e) \rrbracket^i_\gamma & = & (\llbracket e \rrbracket^i_\gamma, \gamma(f\_\mathsf{init})) \\
\llbracket f(e) \rrbracket^s_\gamma & = & \lambda(s_1, s_2).\ \mathit{let}\ v_1, s_1' = \llbracket e \rrbracket^s_\gamma(s_1)\ \mathit{in} \\
& & \qquad\qquad \mathit{let}\ v_2, s_2' = \gamma(f\_\mathsf{step})(v_1)(s_2)\ \mathit{in}\ (v_2, (s_1', s_2')) \\[10pt]
\llbracket \mathsf{present}\ e\ \texttt{->}\ e_1\ \mathsf{else}\ e_2 \rrbracket^i_\gamma & = & (\llbracket e \rrbracket^i_\gamma, \llbracket e_1 \rrbracket^i_\gamma, \llbracket e_2 \rrbracket^i_\gamma) \\
\llbracket \mathsf{present}\ e\ \texttt{->}\ e_1\ \mathsf{else}\ e_2 \rrbracket^s_\gamma & = & \lambda(s, s_1, s_2).\ \mathit{let}\ v, s' = \llbracket e \rrbracket^s_\gamma(s)\ \mathit{in} \\
& & \quad \mathit{if}\ v\ \mathit{then}\ \mathit{let}\ v_1, s_1' = \llbracket e_1 \rrbracket^s_\gamma(s_1)\ \mathit{in}\ (v_1, (s', s_1', s_2)) \\
& & \qquad\quad \mathit{else}\ \ \mathit{let}\ v_2, s_2' = \llbracket e_2 \rrbracket^s_\gamma(s_2)\ \mathit{in}\ (v_2, (s', s_1, s_2')) \\[4pt]
\llbracket \mathsf{reset}\ e_1\ \mathsf{every}\ e_2 \rrbracket^i_\gamma & = & (\llbracket e_1 \rrbracket^i_\gamma, \llbracket e_1 \rrbracket^i_\gamma, \llbracket e_2 \rrbracket^i_\gamma) \\
\llbracket \mathsf{reset}\ e_1\ \mathsf{every}\ e_2 \rrbracket^s_\gamma & = & \lambda(s_0, s_1, s_2).\ \mathit{let}\ v_2, s_2' = \llbracket e_2 \rrbracket^s_\gamma(s_2)\ \mathit{in} \\
& & \qquad\qquad\quad \mathit{let}\ v_1, s_1' = \llbracket e_1 \rrbracket^s_\gamma(\mathit{if}\ v_2\ \mathit{then}\ s_0\ \mathit{else}\ s_1)\ \mathit{in} \\
& & \qquad\qquad\quad (v_1, (s_0, s_1', s_2'))
\end{array}
$$

$$
\left\llbracket
\begin{array}{l}
e\ \mathsf{where} \\
\quad \mathsf{rec\ init}\ x_1 = c_1\ \mathsf{and}\ ... \\
\quad \mathsf{and\ init}\ x_k = c_k \\
\quad \mathsf{and}\ y_1 = e_1\ \mathsf{and}\ ... \\
\quad \mathsf{and}\ y_n = e_n
\end{array}
\right\rrbracket^i_\gamma
=
\begin{pmatrix}
(c_1, \ldots, c_k), \\
(\llbracket e_1 \rrbracket^i_\gamma, \ldots, \llbracket e_n \rrbracket^i_\gamma), \\
\llbracket e \rrbracket^i_\gamma
\end{pmatrix}
$$

$$
\left\llbracket
\begin{array}{l}
e\ \mathsf{where} \\
\quad \mathsf{rec\ init}\ x_1 = c_1\ \mathsf{and}\ ... \\
\quad \mathsf{and\ init}\ x_k = c_k \\
\quad \mathsf{and}\ y_1 = e_1\ \mathsf{and}\ ... \\
\quad \mathsf{and}\ y_n = e_n
\end{array}
\right\rrbracket^s_\gamma
=
\begin{array}{l}
\lambda((m_1, \ldots, m_k), (s_1, \ldots, s_n), s). \\
\quad \mathit{let}\ \gamma_1 = \gamma[m_1/x_1\_\mathsf{last}]\ \mathit{in} \\
\quad \ldots \\
\quad \mathit{let}\ \gamma_k = \gamma_{k-1}[m_k/x_k\_\mathsf{last}]\ \mathit{in} \\
\quad \mathit{let}\ v_1, s_1' = \llbracket e_1 \rrbracket^s_{\gamma_k}(s_1)\ \mathit{in}\ \mathit{let}\ \gamma_1' = \gamma_k[v_1/y_1]\ \mathit{in} \\
\quad \ldots \\
\quad \mathit{let}\ v_n, s_n' = \llbracket e_n \rrbracket^s_{\gamma_{n-1}'}(s_n)\ \mathit{in}\ \mathit{let}\ \gamma_n' = \gamma_{n-1}'[v_n/y_n]\ \mathit{in} \\
\quad \mathit{let}\ v, s' = \llbracket e \rrbracket^s_{\gamma_n'}(s)\ \mathit{in} \\
\quad v, ((\gamma_n'[x_1], \ldots, \gamma_n'[x_k]), (s_1', \ldots, s_n'), s')
\end{array}
$$

$$
\begin{array}{lcl}
\llbracket \mathsf{let\ node}\ f\ \mathsf{x} = e \rrbracket_\gamma & = & \gamma[\llbracket e \rrbracket^i_\gamma / f\_\mathsf{init}, \quad \lambda v.\ \lambda s.\ \llbracket e \rrbracket^s_{\gamma[v/\mathsf{x}]} / f\_\mathsf{step}] \\
\llbracket \mathsf{let\ proba}\ f\ \mathsf{x} = e \rrbracket_\gamma & = & \gamma[\{\!\!\{ e \}\!\!\}^i_\gamma / f\_\mathsf{init}, \quad \lambda v.\ \lambda s.\ \{\!\!\{ e \}\!\!\}^s_{\gamma[v/\mathsf{x}]} / f\_\mathsf{step}] \\
\llbracket d_1\ d_2 \rrbracket_\gamma & = & \mathit{let}\ \gamma_1 = \llbracket d_1 \rrbracket_\gamma\ \mathit{in}\ \llbracket d_2 \rrbracket_{\gamma_1}
\end{array}
$$

Figure 12. Co-iterative semantics of deterministic ProbZelus programs. For local definitions each initialized variable is defined in a subsequent equation, i.e., $\{x_i\}_{1..k} \cap \{y_j\}_{1..n} = \{x_i\}_{1..k}$.

$$
\begin{aligned}
\llbrace e \rrbrace^{i}_{\gamma} &= \llbracket e \rrbracket^{i}_{\gamma} \qquad \mathit{if}\ \mathit{kindOf}(\mathrm{e}) = \mathtt{D}\\
\llbrace e \rrbrace^{s}_{\gamma} &= \lambda s.\ \lambda U.\ \delta_{\llbracket e \rrbracket^{s}_{\gamma}(s)}(U) \qquad \mathit{if}\ \mathit{kindOf}(\mathrm{e}) = \mathtt{D}\\
&= \lambda s.\ \lambda U.\ \begin{cases} 1\ \mathit{if}\ \llbracket e \rrbracket^{s}_{\gamma}(s) \in U \\ 0\ \mathit{otherwise} \end{cases}
\end{aligned}
$$

$$
\begin{aligned}
\llbrace (e_1,e_2) \rrbrace^{i}_{\gamma} &= (\llbrace e_1 \rrbrace^{i}_{\gamma}, \llbrace e_2 \rrbrace^{i}_{\gamma})\\
\llbrace (e_1,e_2) \rrbrace^{s}_{\gamma} &= \lambda(s_1, s_2).\ \lambda U.\ \mathit{let}\ \mu_1 = \llbrace e_1 \rrbrace^{s}_{\gamma}(s_1)\ \mathit{in}\\
&\qquad\qquad \int \mu_1(dv_1, ds'_1)\ \mathit{let}\ \mu_2 = \llbrace e_2 \rrbrace^{s}_{\gamma}(s_2)\ \mathit{in}\\
&\qquad\qquad\qquad\qquad \int \mu_2(dv_2, ds'_2)\ \delta_{(v_1,v_2),(s'_1,s'_2)}(U)\\
\llbrace op(e) \rrbrace^{i}_{\gamma} &= \llbrace e \rrbrace^{i}_{\gamma}\\
\llbrace op(e) \rrbrace^{s}_{\gamma} &= \lambda s.\ \lambda U.\ \mathit{let}\ \mu = \llbrace e \rrbrace^{s}_{\gamma}(s)\ \mathit{in}\ \int \mu(dv, ds')\ \delta_{op(v),s'}(U)\\
\llbrace f(e) \rrbrace^{i}_{\gamma} &= (\llbracket e \rrbracket^{i}_{\gamma}, \gamma(f\_\mathrm{init}))\\
\llbrace f(e) \rrbrace^{s}_{\gamma} &= \lambda(s_1, s_2).\ \lambda U.\ \mathit{let}\ v_1, s'_1 = \llbracket e \rrbracket^{s}_{\gamma}(s_1)\ \mathit{in}\\
&\qquad\qquad \mathit{let}\ \mu_2 = \gamma(f\_\mathrm{step})(v_1)(s_1)\ \mathit{in}\ \int \mu_2(dv_2, ds'_2)\ \delta_{v_2,(s'_1,s'_2)}(U)
\end{aligned}
$$

$$
\begin{aligned}
\llbrace \mathrm{present}\ e \mathrel{-\!\!>} e_1\ \mathrm{else}\ e_2 \rrbrace^{i}_{\gamma} &= (\llbrace e \rrbrace^{i}_{\gamma}, \llbrace e_1 \rrbrace^{i}_{\gamma}, \llbrace e_2 \rrbrace^{i}_{\gamma})\\
\llbrace \mathrm{present}\ e \mathrel{-\!\!>} e_1\ \mathrm{else}\ e_2 \rrbrace^{s}_{\gamma} &= \lambda(s, s_1, s_2).\ \lambda U.\\
&\qquad \mathit{let}\ \mu = \llbrace e \rrbrace^{s}_{\gamma}(s)\ \mathit{in}\\
&\qquad \int \mu(dv, ds')\ \mathit{if}\ v\\
&\qquad\qquad\qquad \mathit{then}\ \mathit{let}\ \mu_1 = \llbrace e_1 \rrbrace^{s}_{\gamma}(s_1)\ \mathit{in}\ \int \mu_1(dv_1, ds'_1)\ \delta_{v_1,(s',s'_1,s_2)}(U)\\
&\qquad\qquad\qquad \mathit{else}\ \ \mathit{let}\ \mu_2 = \llbracket e_2 \rrbracket^{s}_{\gamma}(s_2)\ \mathit{in}\ \int \mu_2(dv_2, ds'_2)\ \delta_{v_2,(s',s_1,s'_2)}(U)\\
\llbrace \mathrm{reset}\ e_1\ \mathrm{every}\ e_2 \rrbrace^{i}_{\gamma} &= (\llbrace e_1 \rrbrace^{i}_{\gamma}, \llbrace e_1 \rrbrace^{i}_{\gamma}, \llbrace e_2 \rrbrace^{i}_{\gamma})\\
\llbrace \mathrm{reset}\ e_1\ \mathrm{every}\ e_2 \rrbrace^{s}_{\gamma} &= \lambda(s_0, s_1, s_2).\ \lambda U.\ \mathit{let}\ \mu_2 = \llbracket e_2 \rrbracket^{s}_{\gamma}(s_2)\ \mathit{in}\\
&\qquad\qquad \int \mu(dv_2, ds'_2)\ \mathit{let}\ \mu_1 = \llbracket e_1 \rrbracket^{s}_{\gamma}(\mathit{if}\ v_2\ \mathit{then}\ s_0\ \mathit{else}\ s_1)\ \mathit{in}\\
&\qquad\qquad\qquad\qquad \int \mu(dv_1, ds'_1)\ \delta_{v_1,(s_0,s'_1,s'_2)}(U)
\end{aligned}
$$

$$
\left\llbrace \begin{array}{l}
e\ \mathrm{where}\\
\quad \mathrm{rec\ init}\ x_1 = c_1\ \mathrm{and}\ \ldots\\
\quad \mathrm{and\ init}\ x_k = c_k\\
\quad \mathrm{and}\ y_1 = e_1\ \mathrm{and}\ \ldots\\
\quad \mathrm{and}\ y_n = e_n
\end{array} \right\rrbrace^{i}_{\gamma}
= \begin{pmatrix} (c_1, \ldots, c_k),\\ (\llbrace e_1 \rrbrace^{i}_{\gamma}, \ldots, \llbrace e_n \rrbrace^{i}_{\gamma}),\\ \llbrace e \rrbrace^{i}_{\gamma} \end{pmatrix}
$$

$$
\left\llbrace \begin{array}{l}
e\ \mathrm{where}\\
\quad \mathrm{rec\ init}\ x_1 = c_1\ \mathrm{and}\ \ldots\\
\quad \mathrm{and\ init}\ x_k = c_k\\
\quad \mathrm{and}\ y_1 = e_1\ \mathrm{and}\ \ldots\\
\quad \mathrm{and}\ y_n = e_n
\end{array} \right\rrbrace^{s}_{\gamma}
= \begin{array}{l}
\lambda((m_1, \ldots, m_k), (s_1, \ldots, s_n), s).\ \lambda U.\\
\quad \mathit{let}\ \gamma_1 = \gamma[m_1/x_1\_\mathrm{last}]\ \mathit{in}\ \ldots\ \mathit{let}\ \gamma_k = \gamma_{k-1}[m_k/x_k\_\mathrm{last}]\ \mathit{in}\\
\quad \mathit{let}\ \mu_1 = \llbrace e_1 \rrbrace^{s}_{\gamma_k}(s_1)\ \mathit{in}\\
\quad \int \mu_1(dv_1, ds'_1)\ \mathit{let}\ \gamma'_1 = \gamma_k[v_1/y_1]\ \mathit{in}\\
\qquad\qquad\qquad \int \ldots\\
\qquad\qquad\qquad\quad \mathit{let}\ \mu_n = \llbracket e_n \rrbracket^{s}_{\gamma'_{n-1}}(s_n)\ \mathit{in}\\
\qquad\qquad\qquad\quad \int \mu_n(dv_n, ds'_n)\ \mathit{let}\ \gamma'_n = \gamma'_{n-1}[v_n/y_n]\ \mathit{in}\\
\qquad\qquad\qquad\qquad\qquad\qquad \mathit{let}\ \mu = \llbrace e \rrbrace^{s}_{\gamma'_n}(s)\ \mathit{in}\\
\qquad\qquad\qquad\qquad\qquad\qquad \int \mu(dv, ds')\ \delta_{v,((\gamma'_n[x_i],\ldots,\gamma'_n[x_k]),(s'_1,\ldots,s'_n),s')}(U)
\end{array}
$$

$$
\begin{aligned}
\llbrace \mathrm{sample}(e) \rrbrace^{i}_{\gamma} &= \llbracket e \rrbracket^{i}_{\gamma}\\
\llbrace \mathrm{sample}(e) \rrbrace^{s}_{\gamma} &= \lambda s.\ \lambda U.\ \mathit{let}\ \mu, s' = \llbracket e \rrbracket^{s}_{\gamma}(s)\ \mathit{in}\ \int_T \mu(dv)\ \delta_{v,s'}(U)\\
\llbrace \mathrm{factor}(e) \rrbrace^{i}_{\gamma} &= \llbracket e \rrbracket^{i}_{\gamma}\\
\llbrace \mathrm{factor}(e) \rrbrace^{s}_{\gamma} &= \lambda s.\ \lambda U.\ \mathit{let}\ v, s' = \llbracket e \rrbracket^{s}_{\gamma}(s)\ \mathit{in}\ \exp(v)\ \delta_{(),s'}(U)\\
\llbrace \mathrm{observe}(e_1, e_2) \rrbrace^{i}_{\gamma} &= (\llbracket e_1 \rrbracket^{i}_{\gamma}, \llbracket e_2 \rrbracket^{i}_{\gamma})\\
\llbrace \mathrm{observe}(e_1, e_2) \rrbrace^{s}_{\gamma} &= \lambda(s_1, s_2).\ \lambda U.\ \mathit{let}\ \mu, s'_1 = \llbracket e_1 \rrbracket^{s}_{\gamma}(s_1)\ \mathit{in}\\
&\qquad\qquad \mathit{let}\ v, s'_2 = \llbracket e_2 \rrbracket^{s}_{\gamma}(s_2)\ \mathit{in}\\
&\qquad\qquad \mu_{\mathrm{pdf}}(v) * \delta_{(),(s'_1,s_2)}(U)
\end{aligned}
$$

Figure 13. Co-iterative semantics of probabilistic ProbZelus expressions (*i.e.*, $\mathit{kindOf}(\mathrm{e}) = \mathtt{P}$). For local definitions each initialized variable is defined in a subsequent equation, i.e., $\{x_i\}_{1..k} \cap \{y_j\}_{1..n} = \{x_i\}_{1..k}$.

$$\frac{G \vdash^{\mathsf{D}} e : t}{G \vdash^{\mathsf{P}} e : t} \qquad \frac{\mathit{typeOf}(c) = t}{G \vdash^{\mathsf{D}} c : t} \qquad \frac{G(x) = t}{G \vdash^{\mathsf{D}} x : t} \qquad \frac{G \vdash^{\mathsf{D}} e_1 : t_1 \quad G \vdash^{\mathsf{D}} e_2 : t_2}{G \vdash^{\mathsf{D}} (e_1, e_2) : t_1 \times t_2} \qquad \frac{\mathit{typeOf}(op) = t_1 \rightarrow^{\mathsf{D}} t_2 \quad G \vdash^{\mathsf{D}} e : t_1}{G \vdash^{\mathsf{D}} op(e) : t_2}$$

$$\frac{G(f) = t_1 \rightarrow^{k} t_2 \quad G \vdash^{\mathsf{D}} e : t_1}{G \vdash^{k} f(e) : t_2} \qquad \frac{G + [t_1/x] \vdash^{k} e_1 : t_2 \quad G \vdash^{\mathsf{D}} e_2 : t_1}{G \vdash^{k} (\mathsf{fun}\ x \mathrel{->} e_1)(e_2) : t_2} \qquad \frac{G \vdash^{\mathsf{D}} e : \mathsf{bool} \quad G \vdash^{k} e_1 : t \quad G \vdash^{k} e_2 : t}{G \vdash^{k} \mathsf{if}\ e\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 : t}$$

$$\frac{G \vdash^{k} e_1 : t_1 \quad G + [t_1/x] \vdash^{k} e_2 : t_2}{G \vdash^{k} \mathsf{let}\ x = e_1\ \mathsf{in}\ e_1 : t_2} \qquad \frac{G + [t_1/x] \vdash^{k} e : t_2}{G \vdash^{\mathsf{D}} \mathsf{fun}\ x \mathrel{->} e : t_1 \rightarrow^{k} t_2} \qquad \frac{G \vdash^{\mathsf{D}} e : t\ \mathsf{dist}}{G \vdash^{\mathsf{P}} \mathsf{sample}(e) : t}$$

$$\frac{G \vdash^{\mathsf{D}} e_1 : t\ \mathsf{dist} \quad G \vdash^{\mathsf{D}} e_2 : t}{G \vdash^{\mathsf{P}} \mathsf{observe}(e_1, e_2) : \mathsf{unit}} \qquad \frac{G \vdash^{\mathsf{D}} e : \mathsf{float}}{G \vdash^{\mathsf{P}} \mathsf{factor}(e) : \mathsf{unit}} \qquad \frac{G \vdash^{\mathsf{P}} e_1 : t \times t_{state} \quad G \vdash^{\mathsf{D}} e_2 : t_{state}\ \mathsf{dist}}{G \vdash^{\mathsf{D}} \mathsf{infer}((\mathsf{fun}\ x \mathrel{->} e_1),e_2) : t\ \mathsf{dist}}$$

$$\frac{G \vdash^{\mathsf{D}} e : t}{G \vdash^{\mathsf{D}} \mathsf{let}\ f = e : G + [t/f]} \qquad \frac{G \vdash^{\mathsf{D}} d_1 : G_1 \quad G_1 \vdash^{\mathsf{D}} d_2 : G_2}{G \vdash^{\mathsf{D}} d_1\ d_2 : G_2}$$

Figure 14. Typing of μF with deterministic and probabilistic kinds.

$$\begin{aligned}
\{\!\![\mathsf{let}\ f = e]\!\!\}_\gamma &= \gamma[\{\!\![e]\!\!\}_\gamma / f] \\
\{\!\![d_1\ d_2]\!\!\}_\gamma &= \mathit{let}\ \gamma_1 = \{\!\![d_1]\!\!\}_\gamma\ \mathit{in}\ \{\!\![d_2]\!\!\}_{\gamma_1} \\
\{\!\![e]\!\!\}_\gamma &= \lambda U.\ \delta_{[\![e]\!]_\gamma}(U)\ \mathit{if}\ \mathit{kindOf}(\mathrm{e}) = \mathsf{D} \\
\{\!\![e_1(e_2)]\!\!\}_\gamma &= \lambda U.\ ([\![e_1]\!]_\gamma([\![e_2]\!]_\gamma))(U) \\
\{\!\![\mathsf{let}\ p = e_1\ \mathsf{in}\ e_2]\!\!\}_\gamma &= \lambda U.\ \int_T \{\!\![e_1]\!\!\}_\gamma(du)\{\!\![e_2]\!\!\}_{\gamma+[u/p]}(U) \\
\{\!\![\mathsf{if}\ e\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2]\!\!\}_\gamma &= \\
&\quad \lambda U.\ \mathit{if}\ [\![e]\!]_\gamma\ \mathit{then}\ \{\!\![e_1]\!\!\}_\gamma(U)\ \mathit{else}\ \{\!\![e_2]\!\!\}_\gamma(U) \\
\{\!\![\mathsf{fun}\ p \mathrel{->} e]\!\!\}_\gamma &= \lambda v.\ \{\!\![e]\!\!\}_{[v/p]} \\
\{\!\![\mathsf{sample}(e)]\!\!\}_\gamma &= \lambda U.\ [\![e]\!]_\gamma(U) \\
\{\!\![\mathsf{observe}(e_1, e_2)]\!\!\}_\gamma &= \\
&\quad \lambda U.\ \mathit{let}\ \mu = [\![e_1]\!]_\gamma\ \mathit{in}\ \mu_{\mathrm{pdf}}([\![e_2]\!]_\gamma) * \delta_{()}(U) \\
\{\!\![\mathsf{factor}(e)]\!\!\}_\gamma &= \lambda U.\ \exp([\![e]\!]_\gamma) * \delta_{()}(U) \\
[\![\mathsf{infer}(\mathsf{fun}\ x \mathrel{->} e_1, e_2)]\!]_\gamma &= \\
&\quad \mathit{let}\ \sigma = [\![e_2]\!]_\gamma\ \mathit{in} \\
&\quad \mathit{let}\ \mu = \lambda U.\ \frac{\int_S \sigma(ds)\{\!\![e]\!\!\}^s_{\gamma+[s/x]}(U)}{\int_S \sigma(ds)\{\!\![e]\!\!\}^s_{\gamma+[s/x]}(\top)}\ \mathit{in}\ (\pi_{1*}(\mu), \pi_{2*}(\mu))
\end{aligned}$$

Figure 15. Probabilistic semantics of μF. The semantics is defined only for probabilistic expressions ($\mathit{kindOf}(\mathrm{e}) = \mathsf{P}$).

$\mathcal{A}(c) = ()$

$\mathcal{A}(x) = ()$

$\mathcal{A}(\mathsf{last}\ x) = ()$

$\mathcal{A}((e_1, e_2)) = (\mathcal{A}(e_1), \mathcal{A}(e_2))$

$\mathcal{A}(e\ \mathsf{where}$
$\quad \mathsf{rec\ init}\ x_1 = c_1\ \ldots\ \mathsf{and\ init}\ x_k = c_k$
$\quad \mathsf{and}\ y_1 = e_1\ \ldots\ \mathsf{and}\ y_n = e_n) =$
$\quad ((c_1,\ldots, c_k),(\mathcal{A}(e_1),\ldots, \mathcal{A}(e_n)),\mathcal{A}(e))$

$\mathcal{A}(\mathsf{present}\ e \mathrel{->} e_1\ \mathsf{else}\ e_2) =$
$\quad (\mathcal{A}(e),\mathcal{A}(e_1),\mathcal{A}(e_2))$

$\mathcal{A}(\mathsf{reset}\ e_1\ \mathsf{every}\ e2) =$
$\quad (\mathcal{A}(e_1),\mathcal{A}(e_1),\mathcal{A}(e_2))$

$\mathcal{A}(op(e)) = \mathcal{A}(e)$

$\mathcal{A}(f(e)) = (f\_\mathrm{init}, \mathcal{A}(e))$

$\mathcal{A}(\mathsf{sample}(e)) = \mathcal{A}(e)$

$\mathcal{A}(\mathsf{factor}(e)) = \mathcal{A}(e)$

$\mathcal{A}(\mathsf{observe}(e_1, e_2)) =$
$\quad (\mathcal{A}(e_1), \mathcal{A}(e_2))$

$\mathcal{A}(\mathsf{infer}(e)) = (\mathcal{A}(e))$

Figure 16. Memory allocation, i.e., initialization for the μF step functions.

The infer operator takes a transition function fun s –> e and an array of pairs (state, weight) S of size N which represents the distribution of possible states across the particles.

$$\begin{aligned}
&[\![\mathsf{infer}(\mathsf{fun}\ s \mathrel{->} e, S)]\!]_\gamma = \\
&\quad \mathit{let}\ \mu = \lambda U.\ \sum_{i=1}^{N} \begin{array}[t]{l} \mathit{let}\ s_i, w_i = [\![S]\!]_\gamma[i]\ \mathit{in} \\ \mathit{let}\ (v_i, s_i'), w_i' = \{\!\![\mathsf{fun}\ s \mathrel{->} e]\!\!\}_{\gamma, w_i}(s_i)\ \mathit{in} \\ \overline{w_i'} * \delta_{v_i}(U) \end{array} \\
&\quad\ \mathit{in}\ (\mu, [(s_i', w_i')]_{1 \le i \le N})
\end{aligned}$$

At each step, the inference executes one step of all the particles and normalizes the scores to return the distribution μ of possible results and an updated array of pairs (state, weight) for the next step.

The weights of the particles are multiplied at each step and never reset. In other words, the inference reports at each step how likely is the execution path since the beginning of the program for each particle w.r.t. the model. Obviously the probability of each individual path quickly collapses to 0 after a few steps which makes this inference technique not practical in a reactive context where the inference process never terminates. The particle filter mitigates this issue by periodically *re-sampling* the set of particles.

```
C(let node f x = e) =
  let f_init = 𝒜(e)
  let f_step = fun (s,x) -> C(e)(s)

C(d1 d2) = C(d1) C(d2)
C(c) = fun s -> (c, s)
C(x) = fun s -> (x, s)
C(last x) = fun s -> (x_last, s)

C((e1, e2)) = fun (s1,s2) ->
  let v1,s1' = C(e1)(s1) in
  let v2,s2' = C(e2)(s2) in
  ((v1,v2), (s1',s2'))

C(op(e)) = fun s ->
  let v,s' = C(e)(s) in
  (op(v), s')

C(f(e)) = fun (s1,s2) ->
  let v1,s1' = C(e)(s1) in
  let v2,s2' = f_step(s2,v) in
  (v2, (s1',s2'))
```

```
C(e where
  rec init x1 = c1 ... and init xk = ck
  and y1 = e1 ... and yn = en) =
fun ((m1,...,mk),(s1, ...,sn),s) ->
  let x1_last = m1 in ...
  let xk_last = mk in
  let y1, s1' = C(e1)(s1) in
  let yn, sn' = C(en)(sn) in
  let v,s' = C(e)(s) in
  (v, (s1', ..., sn'), s')

C(present e -> e1 else e2) =
fun (s,s1,s2) ->
  let v, s' = C(e)(s) in
  if v then let v1,s1' = C(e1)(s1) in
    (v1, (s',s1',s2))
  else let v2,s2' = C(e2)(s2) in
    (v2, (s',s1,s2'))

C(reset e1 every e2) = fun (s0,s1,s2) ->
  let v2,s2' = C(e2)(s2) in
  let v1,s1' = C(e1)(if v2 then s0 else s1) in
  (v1, (s0,s1',s2'))
```

```
C(sample(e)) = fun s ->
  let mu,s' = C(e)(s) in
  let v = sample(mu) in (v, s')

C(observe(e1, e2)) = fun (s1,s2) ->
  let v1,s1' = C(e1)(s1) in
  let v2,s2' = C(e2)(s2) in
  let _ = observe(v1,v2) in ((), (s1',s2'))

C(factor(e)) = fun s ->
  let v,s' = C(e)(s) in
  let _ = factor(v) in ((), s')

C(infer(e)) = fun sigma ->
  let mu,sigma' = infer(C(e), sigma) in
  (mu, sigma')

C(let proba f x = e) =
  let f_init = 𝒜(e)
  let f_step = fun (s,x) -> C(e)(s)
```

Figure 17. Compilation of ProbZelus to μF.

$$\begin{aligned}
\llbracket \text{let } f = e \rrbracket_\gamma &= \gamma[\{\!\!\{e\}\!\!\}_{\gamma,1}/f] \quad \textit{if kindOf}(\mathrm{e})\mathrm{P} \\
\{\!\!\{e\}\!\!\}_{\gamma,w} &= (\llbracket e \rrbracket_\gamma, w) \quad \textit{if kindOf}(\mathrm{e}) = \mathrm{D} \\
\{\!\!\{e_1(e_2)\}\!\!\}_{\gamma,w} &= \textit{let } v_2 = \llbracket e_2 \rrbracket_\gamma \textit{ in } \llbracket e_1 \rrbracket_\gamma(v_2, w) \\
\{\!\!\{\text{if } e \text{ then } e_1 \text{ else } e_2\}\!\!\}_{\gamma,w} &= \\
&\quad \textit{if } \llbracket e \rrbracket_\gamma \textit{ then } \{\!\!\{e_1\}\!\!\}_{\gamma,w} \textit{ else } \{\!\!\{e_2\}\!\!\}_{\gamma,w} \\
\{\!\!\{\text{let } p = e_1 \text{ in } e_2\}\!\!\}_{\gamma,w} &= \textit{let } v_1, w_1 = \{\!\!\{e_1\}\!\!\}_{\gamma,w} \textit{ in } \{\!\!\{e_2\}\!\!\}_{\gamma[v_1/p],w_1} \\
\{\!\!\{\text{fun } p \text{ -> } e\}\!\!\}_{\gamma,w} &= \textit{let } f = \lambda(v, w').\ \{\!\!\{e\}\!\!\}_{[v/p],w'} \textit{ in } (f, w) \\
\{\!\!\{\text{sample}(e)\}\!\!\}_{\gamma,w} &= (\mathrm{draw}(\llbracket e \rrbracket_\gamma), w) \\
\{\!\!\{\text{factor}(e)\}\!\!\}_{\gamma,w} &= ((), w * \exp(\llbracket e \rrbracket_\gamma)) \\
\{\!\!\{\text{observe}(e_1, e_2)\}\!\!\}_{\gamma,w} &= \textit{let } \mu = \llbracket e_1 \rrbracket_\gamma \textit{ in } ((), w * \mu_{\mathrm{pdf}}(\llbracket e_2 \rrbracket_\gamma))
\end{aligned}$$

Figure 18. Importance sampler. Probabilistic expressions return a pair (value, weight). sample draws a sample from a distribution, factor and observe update the weight.

E Implementation

ProbZelus is implemented on top of Zelus (http://zelus.di.ens.fr/). The new constructs sample, observe, and factor are Zelus nodes implemented directly in OCaml. The infer construct is a node that take as argument the Zelus node that represents the probabilistic model. The infer node thus takes as argument the allocation and step functions of the model as argument which corresponds to the compilation described in Section 4.

Artifact. ProbZelus is provided as supplementary material. To install ProbZelus, first install Zelus. In the zelus directory:

```
opam install ocamlfind menhir dune owl
./configure && make && make install
```

To re-run the experiments:

```
make -C probabilistic
make -C probabilistic/examples/benchmarks bench
```

More examples are available in the directory probabilistic/examples (launch with `make; make exec`). The implementation of the inferences algorithms are in probabilistic/inference.

F Performance Evaluation

This section presents the experimental results. We ran each inference algorithm on a series of benchmarks and measured properties of the execution. All the experiments were run on a Google Cloud n1-highmem-4 machine (4 vCPUs, 26 GB memory).

F.1 Benchmarks

Beta-Bernoulli. The Beta-Bernoulli benchmark models an agent that estimates the bias of a coin.

```
let proba coin yobs = xt where
  rec init xt = sample (beta (1., 1.))
  and () = observe (bernoulli xt, yobs)
```

The model samples the variable zt from a *Beta*(1, 1) distribution, and thereafter evaluates the observations with a *Bernoulli* distribution of parameter xt. Running SDS on this model is equivalent to exact inference in a Beta-Bernoulli conjugate model [16] where each particle returns the exact solution. The benchmark's error metric is the mean squared error over time between the true coin probability and the expected probability conditioned on the stream of observations.

Gaussian-Gaussian. The Gaussian-Gaussian benchmark models an agent that estimates the mean and the standard deviation of a Gaussian.

```
let proba gaussian_model o = (mu, sigma) where
  rec init mu = sample (gaussian (0., 10.))
  and init sqrt_sigma = sample (gaussian (0., 1.))
  and sigma = sqrt_sigma *. sqrt_sigma
  and () = observe (gaussian (mu, sigma), o)
```

The initial values for the distribution of the mean mu follows a distribution $\mathcal{N}(0, 10)$ and the distribution of $\sqrt{\text{sigma}}$ is $\mathcal{N}(0, 1)$. The distributions of mu and sigma are conditioned by the observations values that follows a distribution $\mathcal{N}(\text{mu}, \text{sigma})$. In the current implementations of delayed sampling we are doing exact inference only on the mean and not on the standard deviation (even if it would be possible). The benchmark's error metric is the mean squared error over time between the true mean and standard deviation and the expected probability conditioned on the stream of observations.

Kalman. The Kalman benchmark models an agent that estimates its position based on noisy observations.

```
let proba delay_kalman yobs = xt where
  rec xt = sample (gaussian ((0., 50.) -> (pre xt, 1.)))
  and () = observe (gaussian (xt, 1.), yobs)
```

The model chooses an initial position from $\mathcal{N}(0, 50)$, and chooses subsequent positions from $\mathcal{N}(\text{pre x}, 1)$ where pre x denote the previous position. The model draws the observation at each time step from $\mathcal{N}(\text{x}, 1)$ where x is the true position. Running SDS on this model is equivalent to a Kalman filter [23] where each particle returns the exact solution. The benchmark's error metric is the mean squared error over time between the true position and the expected position conditioned on all previous observations.

Outlier. The Outlier benchmark, adapted from Section 2 of [27], models the same situation as the Kalman benchmark, but with a sensor that occasionally produces invalid readings.

```
let proba outlier yobs = (is_outlier, xt) where
  rec xt = sample (gaussian ((0., 50.) -> (pre xt, 1.)))
  and init outlier_prob = sample (beta (100., 1000.))
  and is_outlier = sample (bernoulli outlier_prob)
  and () = present is_outlier -> observe (gaussian (0., 100.), yobs)
              else observe (gaussian (xt, 1.), yobs)
```

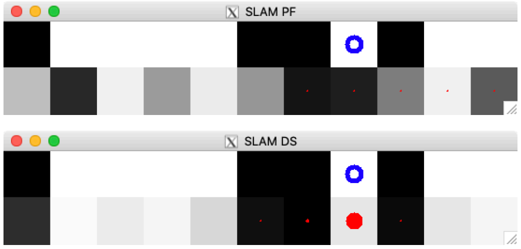


Figure 19. Screenshots of the execution of the SLAM with the PF and SDS inferences. For each screenshot, the top line shows the map, and the blue circle the exact position of the robot. The lower line represents the inferred map where the gray level indicates the probability for the cell to be black and the red dots the probability of presence of the robot on the cell.

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 either chooses the observation from the invalid distribution $\mathcal{N}(0, 100)$, or it chooses the observation from the Kalman model. Running SDS on this model is equivalent to a Rao-Blackwellized particle filter [14] that combines exact inference with approximate particle filtering. The benchmark's error metric is the mean squared error over time between the true position and the expected position conditioned on all previous observations.

Robot. The Robot benchmark is detailed Section 2.

SLAM. The last benchmark is the Simultaneous Location And Mapping (SLAM) [28]. Consider the simple case where a robot evolves in a discrete one-dimensional world and each position corresponds to a black or white cell. The robot can move from left to right and can observe the color of the cell on which it stands with a sensor. There are two sources of uncertainty: (1) the robot's wheels are slippery, so the robot can sometimes stay on the spot thinking about moving, (2) the sensor is making read errors, and can reverse the colors. The controller tries to infer the map (color of the cells) and the current position of the robot (Figure 19).

The robot maintains a map where each box is a random variable that represents the probability of being black or white (gray level in the Figure 19). The *a priori* distribution of these random variables is a *Beta*(1, 1) distribution:

```
let proba beta_priors _ = sample (beta (1., 1.))
```

The robot starts from the position x0 and receives at each step a command Right or Left. It then moves to the left or right following the command with a probability of 10% of remaining in place (modeled by a Bernoulli distribution of parameter 0.1).

```
let proba move (x0, cmd) = x where
 rec slip = sample (bernoulli 0.1)
 and xp = x0 -> pre x
 and x = match cmd with
   | Right -> min max_pos (if slip then xp else xp + 1)
   | Left  -> max min_pos (if slip then xp else xp - 1)
   end
```

The sensor is modeled in the same way with a probability of reading error of 10%:

```
let proba read obs = if sample (bernoulli 0.1) then not obs else obs
```

At each instant, the robot computes its current position x. The observation of the sensor follows a Bernoulli distribution parameterized by the value of the map associated with this position.

```
let proba slam (obs, cmd) = (map, x) where
 rec init map = Array.init (max_pos + 1) beta_priors ()
 and x = move (0, cmd)
 and o = read (obs)
 and () = observe (bernoulli (Array.get map x), o)
```

The benchmark's error metrics is the mean squared error over time between the exact map and position and the expected map and position.

Multi-Target Tracker. MTT (*Multi-Target Tracker*) adapted from [30] is a model where there are a variable number of targets with linear-Gaussian motion models producing linear-Gaussian measurements of the position at each time step. Targets randomly appear according to a Poisson process and each disappear with some fixed probability at each time step. Measurements do not identify which target they came from, and "clutter" measurements that come not from targets but from some underlying distribution add to observations, complicating inference of which measurements are associated to which targets.

We model this with a ProbZelus program that has a state consisting of a list of position-velocity pairs that encode the *track* of each target. In this example we consider two-dimensional targets, giving us a 4-dimensional vector representing position and velocity together.

The first step is to define helper functions that will be mapped over the list of tracks. The first function tells us how frequently tracks die. They do so with probability p_dead which we set to $e^{-.02}$.

```
let proba death_fn _ = sample (bernoulli (const p_dead))
```

We now define how tracks are initialized when they are first created. They are sampled from a multivariate Gaussian distribution with mean mu_new set to zero and covariance sigma_new set to a diagonal with variance 1 on the positions and variance .001 on the velocities.

```
let proba new_track_init_fn _ =
 (new_track_num (),
 sample (mv_gaussian (const mu_new, sigma_new)))
```

Next, we define the motion model and update model. Each track tr is multiplied with the motion matrix a_u which encodes discrete time integration of position and velocity with time constant 1. We then sample a Gaussian distribution around the new position and velocity with covariance sigma_update which is a diagonal matrix with 0.01 variance for the position and 0.1 for the variance of the velocity. For the observation model, we project out the position with the projection matrix proj_pos and observe it with covariance matrix sigma_obs which we set to a diagonal of 0.1.

```
let proba state_update_fn (tr_num, tr) =
 (tr_num,
 sample (mv_gaussian(((const a_u) *@ tr), sigma_update)))

let observe_fn (_, tr) =
 (mv_gaussian ((const proj_pos) *@ tr, sigma_obs))
```

We next define the model fo clutter data. We assume that each clutter point is drawn from a multivariate Gaussian with mean mu_clutter which is zero an covariance sigma_clutter which we set to 10.

```
let proba clutter_init_fn _ =
 (mv_gaussian (const mu_clutter, sigma_clutter))
```

The model proceeds as follows. For every track, we use the filter list operator to remove all the tracks that died in this time step. We then sample the number of new tracksn_new from a Poisson distribution with parameter lambda_new which we set to 0.1. After forcing a sample of this value, we use the list constructer ini to build a list of new tracks. We append the survived and new tracks together, and then use the map list operator to apply the motion and observation models to each track. Next, we determine the amount of clutter by subtracting the number of observations from the number of surviving tracks. We then observe that this comes from a Poisson distribution with parameter lambda_clutter which we set to 1. Note that this sets the particle weight to $-\infty$ if the particle yields a negative amount of clutter. Next, we shuffle the track observations and the clutter together by forcing a sample of the shuffle random primitive, and finally observe that the resulting list yields the observed values.

```
let proba obsfn (var, value) = observe (var, value)

let proba model inp = t where
 rec init t = []
 and t_survived = filter death_fn (last t)
 and n_new = sample (poisson lambda_new)
 and t_new = ini new_track_init_fn n_new
 and t_tot = append t_survived t_new
 and t = map state_update_fn t_tot
 and obs = map observe_fn t
 and n_clutter = (length inp) - (length obs)
 and () = observe(poisson lambda_clutter, n_clutter)
 and clutter = ini clutter_init_fn n_clutter
and obs_shuffled = sample (shuffle (append obs clutter))
and present (not (n_clutter < 0)) −>
  do () = (iter2 obsfn (obs_shuffled, inp)) done
```

The accuracy metric is based on the *Multiple Object Tracking Accuracy* [2]. This evaluates whether a track estimate contains the right targets across all time steps within a sufficient tolerance (we set the tolerance to 5 in our example). Conventional MOTA is in $[0, 1]$ with 1 being the best; we have modified it to be in $[0, \infty]$ with 0 being the best by transforming it to $\mathrm{MOTA}* = 1/\mathrm{MOTA} - 1$.

Because we estimate a distribution of track estimates, we compute an expectation of the MOTA* using a particle estimate with the same number of particle as used for MTT model inference.

Data. For each benchmark except Robot and SLAM, we obtained observation data by sampling from the benchmark’s model. In these benchmarks, every run of each benchmark across all experiments uses the same data as input. For SLAM, we pre-sampled the map from the model, but sampled position data on the fly as this data depends on the controller. For the Robot benchmark, we sampled all observations on the fly because they all depend on the command from the controller. This means that for SLAM and Robot, each run uses different position observations.

F.2 Delayed Samplers vs. Particle Filter

We compare both the accuracy and runtime performance of BDS, SDS, and PF to investigate whether the delayed samplers can achieve better accuracy than the particle filter with the same amount of computational resources.

Accuracy Methodology. For a range of selected particle counts, we execute each benchmark multiple times and record the resulting accuracy. To measure accuracy we use the end-to-end error metrics for each benchmark as described in Section F.1. We record the median and the 90% and 10% quantiles. For the benchmarks in Figure 20 we executed each benchmark 1000 times, and for the benchmarks in Figure 22 we executed each benchmark 100 times.

Accuracy Results. Figures 20 and 22 show the results of the accuracy experiment for the different benchmarks. The error bars show 90% and 10% quantiles, and the center is the median. The vertical lines corresponds to the number reported in Figure 10 where there is enough particles to achieve similar accuracy to delayed sampling with 500 particles. In all cases, SDS is able to achieve equal or better accuracy than BDS which is itself equal or better than PF, but the results vary widely by benchmark. Note that SDS returns the exact posterior distribution for the Coin and Kalman benchmarks therefore its accuracy is independent of the number of particles. On the other-hand, BDS is not exact since the symbolic distributions are sampled at the end of each the step.

Performance Methodology. For a range of selected particle counts, we execute each benchmark multiple times (the same number as for the accuracy experiments described above) after a warm-up of 1 run and record the resulting performance: the latency of one step of computation. In the following graphs we report the median latency as well as the 90% and 10% quantiles of the collected data.

Performance Results. Figures 21 and 23 shows how the latency for a single step varies with the number of particles for each benchmark. The error bars show 90% and 10% quantiles, and the center is the median. With the three algorithms, the execution time increases linearly with the number of particles. In all cases, PF has lower latency than BDS which has lower latency than SDS.

Conclusions. These experiments show that the delayed samplers achieve better accuracy than the particle filter with the same computational resources. For some models SDS is able to compute the exact solution with only one particle (Kalman, Coin). BDS achieves better accuracy when relationships between variables defined in the same step can be exploited (Kalman). At worst the delayed samplers performs as a well as the particle filter (BDS on the Coin, SDS and BDS on the Outlier).

F.3 Streaming Delayed Sampling vs. Delayed Sampling

We next evaluate the performance of SDS and BDS relative to our own OCaml implementation of the original delayed sampler (DS). We compare both the performance and memory consumption of the three algorithms at each time step to investigate whether, as the size of the input stream grows large, they can retain constant performance.

Performance Methodology. We execute each benchmark 100 times after a warm-up of 1 run and record the latency. We execute each benchmark with 100 particles (even if only one particle is necessary for DS and SDS on the Coin and Kalman benchmarks to compute the exact distribution) and plot latency as a function of the time step. We report the median latency as well as the 90% and 10% quantiles of the collected data.

Performance Results. Figures 24 and 26 shows the latency at each step of a run, aggregated over 100 runs. PF, BDS, and SDS show nearly constant performance in time but DS gets linearly worse performance for the Kalman and Outlier benchmarks. For the Coin benchmark, the graph of DS remains of constant size because there is only one sample at the first step and then only observe statements.

Memory Methodology. We next evaluate the memory consumption of the algorithms. For all benchmarks except the multi-target tracker, memory consumption is deterministic

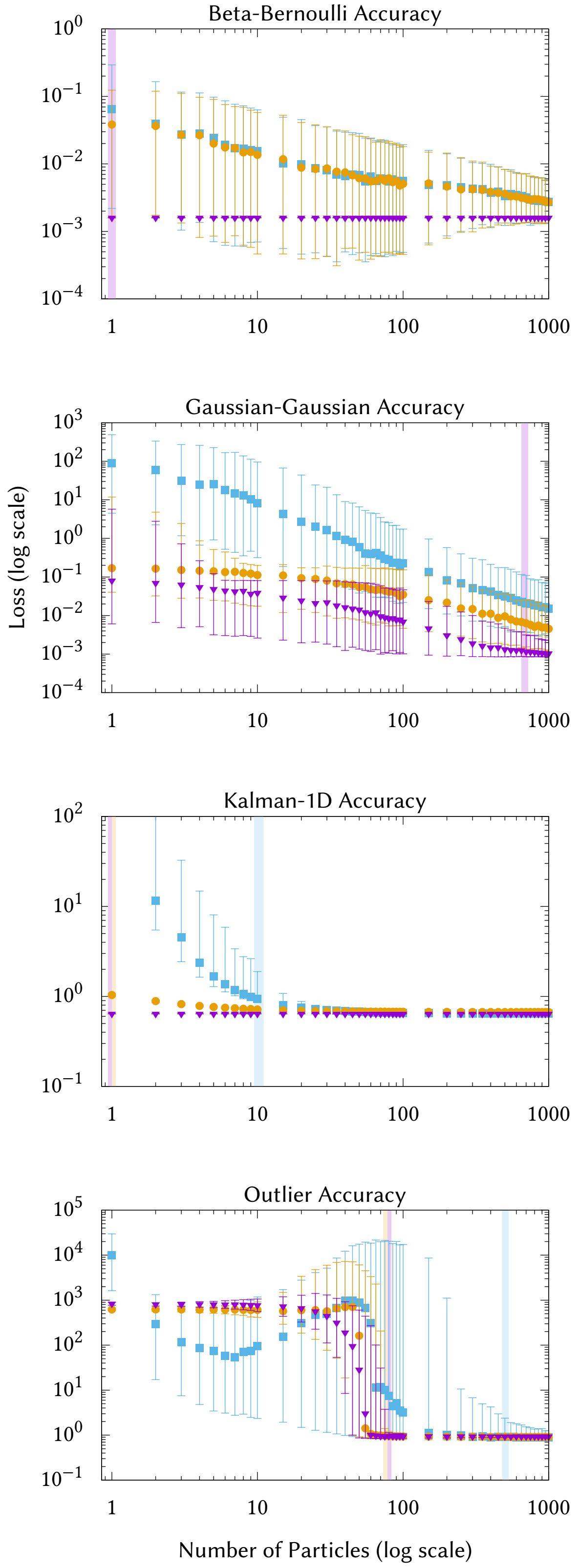


Figure 20. Accuracy as a function of the number of particles.

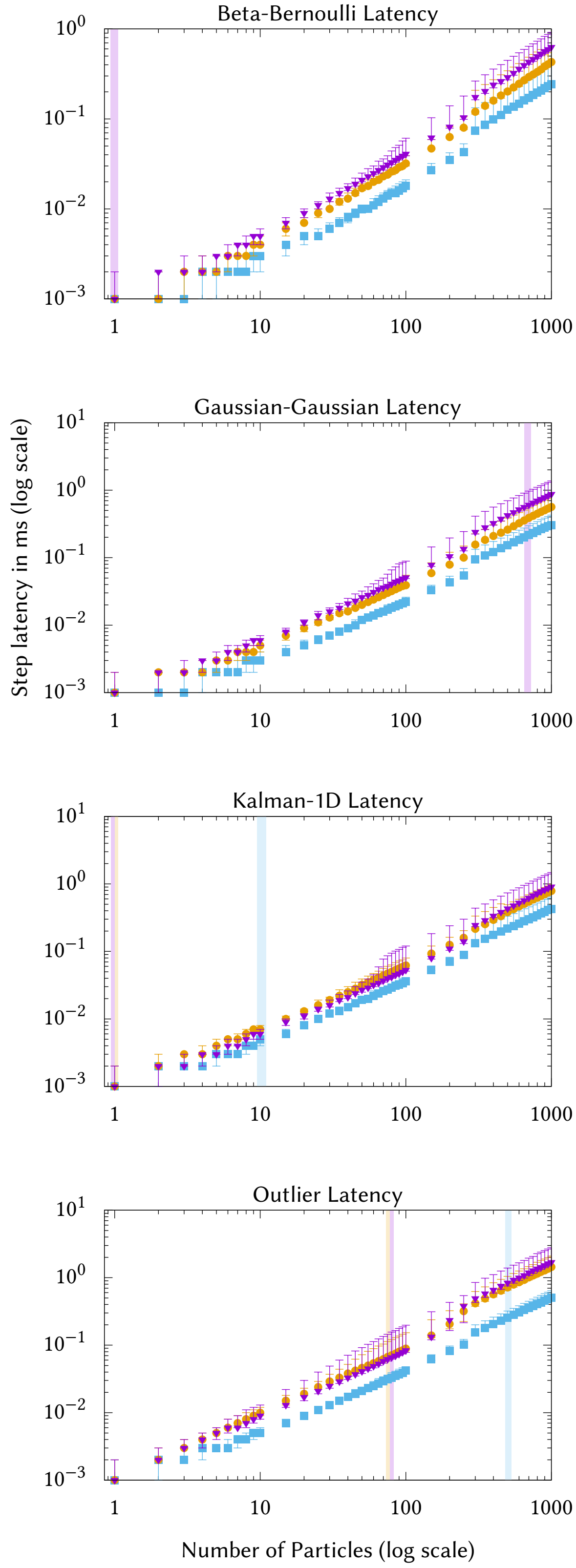


Figure 21. Runtime performance as a function of particles.

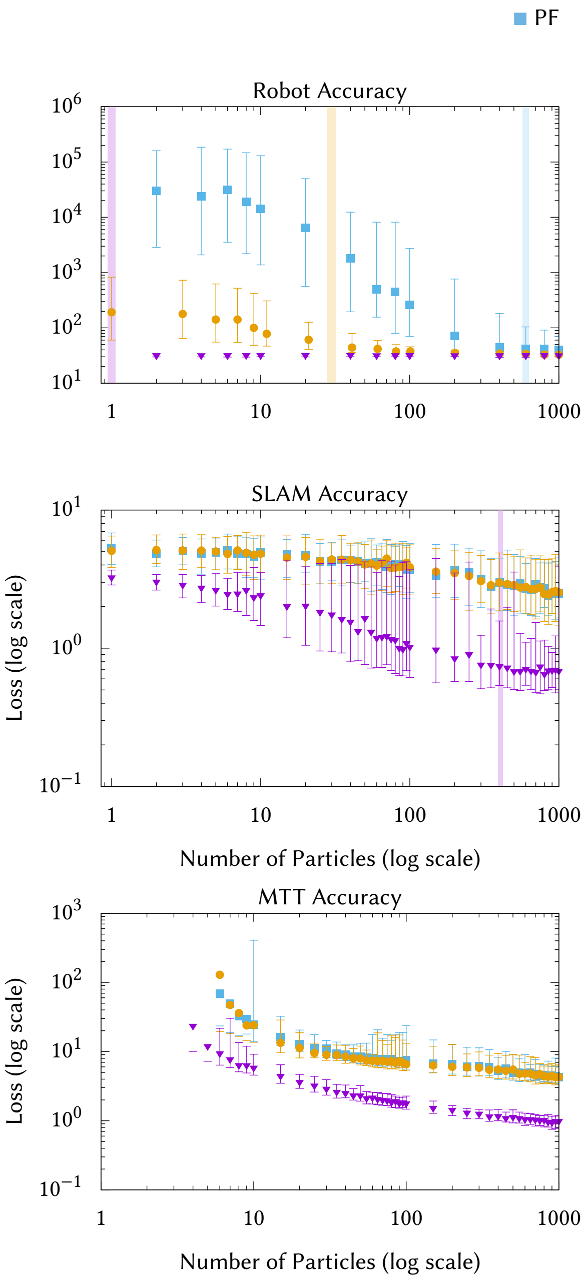


Figure 22. Accuracy as a function of the number of particles.

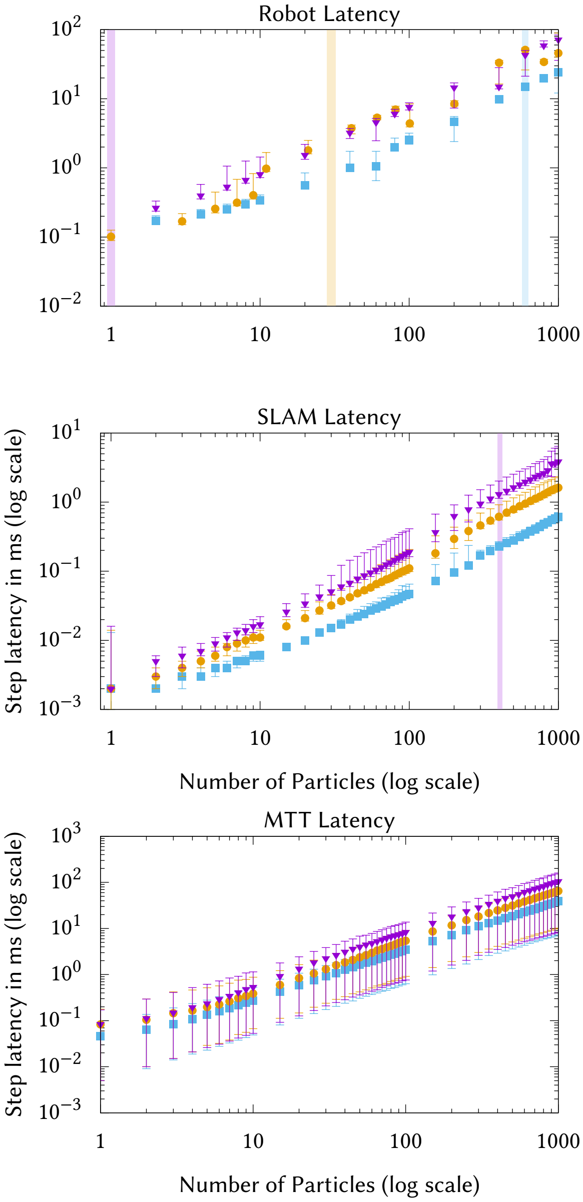


Figure 23. Runtime performance as a function of particles.

even in the presence of random choices. Therefore, we measure the *ideal* memory consumption of the execution of each benchmark after each step. The ideal memory consumption is the total amount of live words in the program's heap. In our implementation, we measure these numbers by forcing a garbage collection after each step. We use OCaml's standard facilities for forcing garbage collection as well as for measuring the amount of live words. We ran each algorithms with 100 particles.

For the multi-target tracker, the memory is not deterministic because it is determined by the number of hypothetical tracks, which is random. We report median and 10% and 90% values for memory consumption for this benchmark.

Memory Results. Figures 25 and 27 shows the results of the memory consumption experiment. For all benchmarks, PF, BDS, SDS use constant memory over time, including for the multi-target tracker where their memory consumption is random at each time step. However, DS has increasing memory consumption over time for the Kalman, Outlier, and Robot benchmarks. The memory consumption of DS is

■ PF ● BDS ▼ SDS ▲ DS

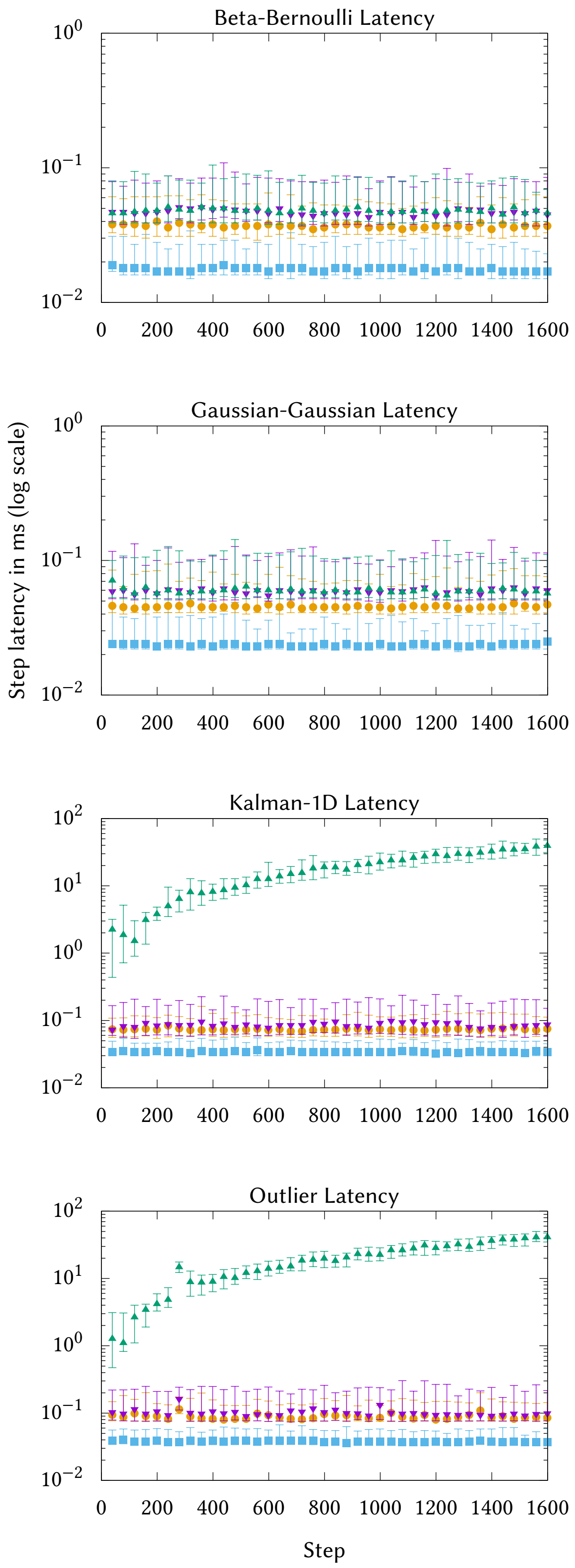


Figure 24. Runtime performance at each step of a run.

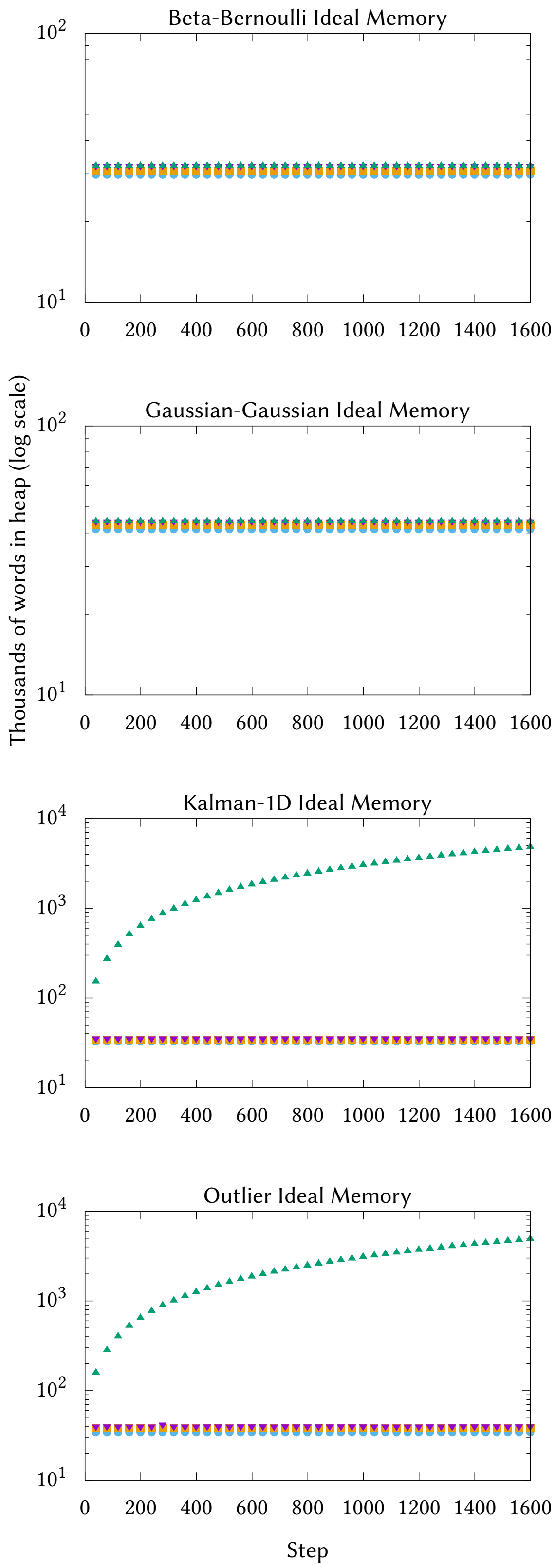


Figure 25. Memory consumption at each step of a run.

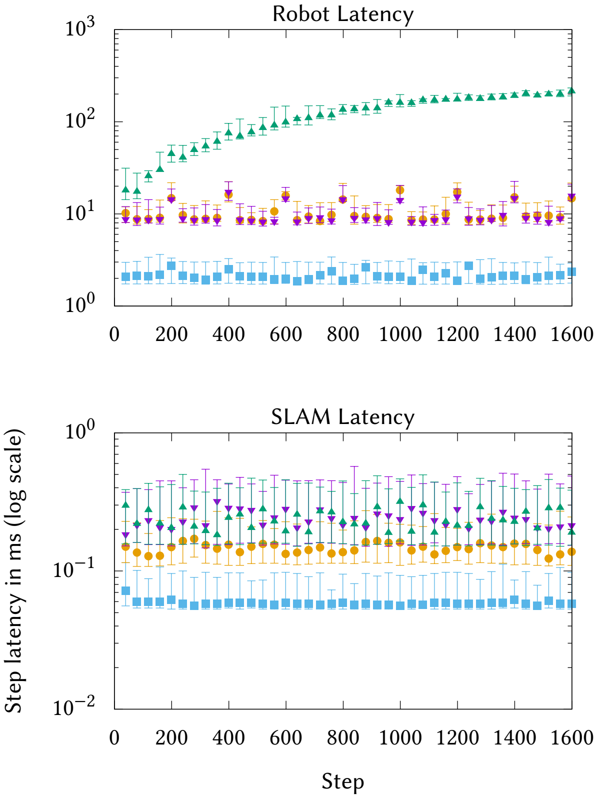


Figure 26. Runtime performance at each step of a run.

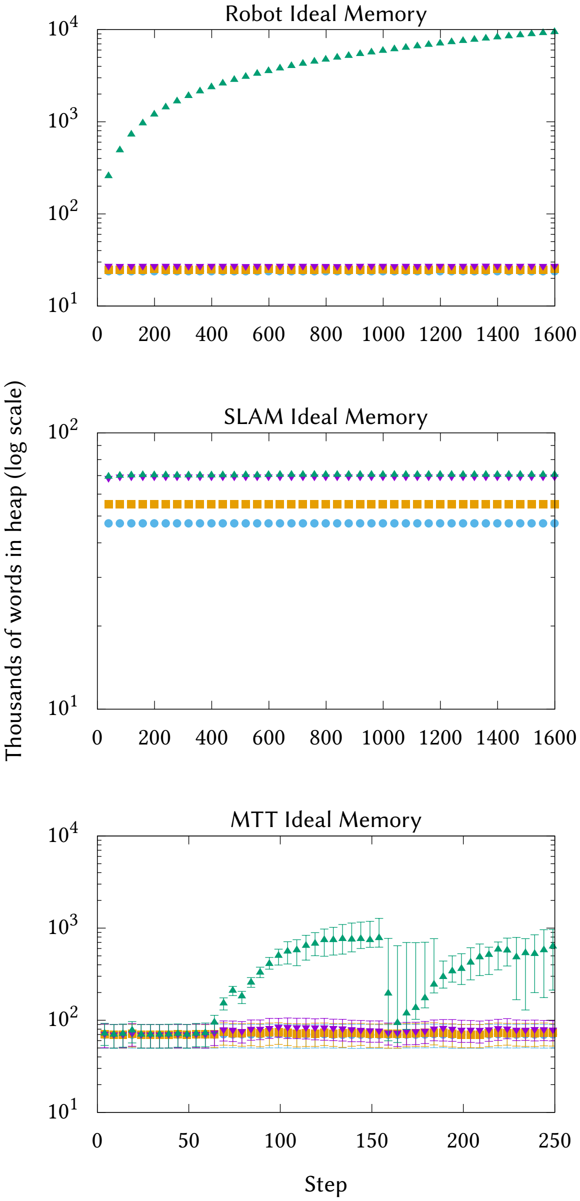


Figure 27. Memory consumption at each step of a run.

constant for the Coin benchmark because the graph remains of constant size.

For the mutli-target tracker, the memory consumption of DS is based both on the number of hypothesized tracks and the length of the hypothesized tracks. We can see that the memory consumption of DS increases as the first generation of tracks becomes longer, but eventually curtails its memory consuption when these tracks die. MTT's memory consumption thereafter increases again as the second generation of tracks starts to increase in length.

Conclusions. The original DS implementation consumes an increasing amount of memory over time for models that introduce new variables at each step (Kalman, Outlier, and Robot) in contrast to BDS and SDS whose memory consumptions are constant over time. For the multi-target tracker, the DS memory consumption is based on the length of the track which is in principle probabilistically bounded. However, DS still consumes much more memory than PF, BDS, and SDS because the tracks are long-lived.

Furthermore, DS step latency increases without bound as the number of steps becomes large on benchmarks where the memory increases. These observations confirm that the original DS implementation is not practical in a reactive settings.