Schedule Agnostic Semantics for Reactive Probabilistic Programming

GUILLAUME BAUDART, Inria, France
LOUIS MANDEL, IBM Research, USA
CHRISTINE TASSON, ISAE Supaero, France

Synchronous languages are now a standard industry tool for critical embedded systems. Designers write high-level specifications by composing streams of values using block diagrams. These languages have been recently extended with Bayesian reasoning to program state-space models which compute a stream of distributions given a stream of observations. Yet, the semantics of probabilistic models is only defined for scheduled equations – a significant limitation compared to dataflow synchronous languages and block diagrams.

In this paper we propose two schedule agnostic semantics for a probabilistic synchronous language. The key idea is to interpret probabilistic expressions as a stream of un-normalized density functions which maps random variable values to a result and positive score. The co-iterative semantics extends the original semantics to interpret mutually recursive equations using a fixpoint operator. The relational semantics directly manipulates streams and is thus a better fit to reason about program equivalence. We use the relational semantics to prove the correctness of a program transformation required to run an optimized inference algorithm for state-space models with constant parameters.

1 INTRODUCTION

Synchronous programming languages [6] were introduced for the design of critical embedded systems. In dataflow languages such as Lustre [36], system designers write high-level specifications by composing infinite streams of values, called *flows*. Flows progress *synchronously*, paced on a global logical clock. Specialized compilers generate efficient and correct-by-construction embedded code with strong guarantees on execution time and memory consumption. This approach was inspired by block diagrams, a popular notation to describe control systems [37]. Built on these ideas, Scade is now a standard tool in automotive and avionic industries to program safety critical embedded software [18]. The synchronous model of computation is also central for the discrete-time subset of Matlab/Simulink [40].

Probabilistic languages extend general purpose programming languages with probabilistic constructs for Bayesian reasoning [7, 23, 33, 34]. Following a Bayesian approach, a program describes a probability distribution, the *posterior* distribution, using initial beliefs on random variables, the *prior* distributions, that are conditioned on observations.

At the intersection of these two lines of research, ProbZelus [4] is a probabilistic extension of the synchronous dataflow language Zelus [13]. ProbZelus combines, in a single source program, deterministic controllers and probabilistic models that can interact with each other to perform *inference-in-the-loop*. A classic example is the *Simultaneous Localization and Mapping problem* (SLAM) [45] where an autonomous agent tries to infer both its position and a map of its environment to adapt its trajectory.

The probabilistic model of the SLAM involves two kinds of parameters. The position is a *state parameter* represented by a stream of random variables. At each instant, a new position must be estimated from the previous position and the observations. The map is a *constant parameter* represented by a random variable whose value is progressively refined from the *prior* distribution with each new observation. This type of problem mixing constant parameters and state parameters are instances of *State-Space Models* (SSM) [16]. Any ProbZelus program can be expressed as a SSM.

Probabilistic semantics and scheduling. The original ProbZelus semantics is defined in a co-iterative framework where expressions are interpreted as state machines [4]. Following [48], a probabilistic expression computes a stream of *measures*. The semantics of an expression with a set of local declarations integrates the semantics of the main expression over all possible values of the local variables. Unfortunately, this semantics yields nested integrals that are only well defined if the declarations are *scheduled*, i.e., ordered according to data dependencies.

This is a significant limitation compared to synchronous dataflow languages where sets of mutually recursive equations are not ordered: a key requirement for commercial synchronous dataflow languages where programs are written using a block diagram graphical interface. Scheduling should not depend on the placement of the blocks which motivate their definition as mutually recursive equations. Besides, the compiler implements a series of source-to-source transformations which often introduces new variables in arbitrary order. Scheduling local declarations is one of the very last compilation passes [13]. The semantics of ProbZelus is thus far from what is exposed to the programmer and prevents reasoning about most program transformations and compilation passes.

In this paper, we show how to extend the schedule agnostic semantics of dataflow synchronous languages [9, 15] for probabilistic programming. We first define a new probabilistic *co-iterative semantics* where sets of equations in arbitrary order can be interpreted with a fixpoint operator. Unfortunately, classic probabilistic fixpoint definitions used to interpret loops and recursions [31, 41, 50] yields incorrect results for the interpretation of mutually recursive equations. The key idea of our approach is to interpret probabilistic expressions as a stream of un-normalized density functions which maps random variable values to a result and positive score. Like its deterministic counterpart [15], this density-based co-iterative semantics is *executable* and has the same structure as the compiled code. A drawback of this semantics is that, at each step, probabilistic state machines compute measures. Proofs of program equivalence must relate measures of states through successive integrations by exhibiting a bisimulation [46]. We introduce an alternative *relational semantics* which abstracts away the state machines and directly manipulates streams which simplifies reasoning about program equivalence. This semantics is based on the deterministic relational semantics used in the Vélus project to prove an end-to-end compiler for the synchronous language Lustre [9–11].

Filtering and constant parameters. As a case study we prove the correctness of a program transformation that is required to run an optimized inference algorithm. To estimate state parameters, Sequential Monte Carlo (SMC) techniques rely on random simulations and filtering to approximate the posterior distribution, i.e., voluntarily dropping information to re-center the inference on the most significant estimations. Unfortunately, this information loss negatively impacts constant parameters estimations.

Inspired by the Assumed Parameter Filter algorithm [32], we can split the inference into two steps: 1) estimate state parameters, and 2) update constant parameters. This technique requires a program transformation to explicitly separate constant from state parameters. A specialized static analysis identifies the constant parameters and their prior distributions. A compilation pass then transforms these parameters into additional inputs of the model. We use the relational semantics to prove the correctness of this transformation, i.e., the transformation preserves the ideal semantics of the program.

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

- We introduce in Section 4 a new density-based co-iterative semantics and show that sets of mutually recursive equations can be interpreted using a fixpoint operator. We prove that this semantics is equivalent to the original ProbZelus semantics.
- We introduce in Section 5 an alternative relational semantics which abstracts away the state machines and directly manipulates streams which simplifies reasoning about program

```
1 proba tracker(y_obs) = x where
2   rec init x = x_init
3   and x = sample(gaussian(f(last x), sx))
4   and y = g(x)
5   and () = observe(gaussian(y, sy), y_obs)

7 node main(y_obs) = u where
8   rec x_dist = infer (tracker (y_obs))
9   and u = controller(x_dist)
```

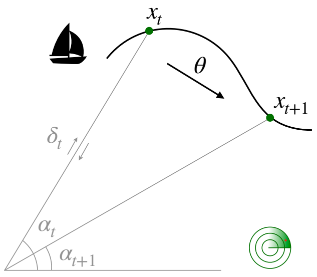

Fig. 1. Tracking a moving boat with a marine radar in ProbZelus. x_t is the position of the boat in Cartesian coordinates. We assume a linear motion model $f(x_{t-1}) = x_{t-1} + \theta$, and $g(x_t) = (\alpha_t, \delta_t)$ returns the radar angle $\alpha_t = \text{atan}(x_t[1]/x_t[0])$, and the echo delay $\delta_t = 2 * \|x_t\|/c$ where c is the speed of light.

equivalence. We prove that this semantics is equivalent to the density-based co-iterative semantics and thus to the original semantics.

- We define in Section 6 a program transformation required to run an optimized inference algorithm for state-space models with constant parameters. We use the relational semantics to prove the correctness of the transformation.

2 EXAMPLE

To motivate our approach, consider the ProbZelus model of Figure 1 adapted from [16][Section 2.4.1]. The goal is to estimate at each instant the position of a moving boat given noisy observations from a marine radar. A rotating antenna sweeps a beam of microwaves and detects the boat when the beam is reflected back to the antenna. The radar then estimates the position from noisy measurements of its angle and of the echo delay.

The keyword `proba` indicates the definition of a probabilistic stream function. Line 1, the model `tracker` takes as input a stream of observations `y_obs` and returns a stream of positions `x`. The model uses a function `f` to estimate the current position (e.g., using a linear motion model), and a function `g` to compute the observable quantities from the state (e.g., angle and echo delay). Line 3 uses `sample` to specify that `x` is Gaussian distributed around `f(last x)` where `last x` refers to the previous position of the boat initialized Line 2 with the `init` keyword. Line 5 uses `observe` to condition the model assuming that the observations `y_obs` are Gaussian distributed around `y = g(x)`. The initial position `x_init` and the noise parameters `sx` and `sy` are global constants.

2.1 Kernel-based co-iterative semantics

The original ProbZelus semantics [4] is a co-iterative semantics where expressions are interpreted as state machines characterized by an initial state and a transition function. Given the current state, the transition function of deterministic expression returns the next state and a value. Following [48], the transition function of a probabilistic expression returns a *measure* over all possible pairs (next state, value).

For instance, if `f` and `g` are deterministic and stateless, the transition function of `tracker` is the following (omitting empty states for stateless expressions) where $\mathcal{N}$ is the normal distribution,

δ the Dirac delta distribution, and pdf_d the probability density function of d.

$$\begin{aligned}\{\![\texttt{tracker(y\_obs)}]\!\}^{\text{step}}_{\gamma}(p_x) &= \int \mathcal{N}(f(p_x), s_x)(dx) \int \delta_{g(x)}(dy)\ pdf_{\mathcal{N}(y,s_x)}(y_{\text{obs}}) * \delta_{x,x} \\ &= \int \mathcal{N}(f(p_x), s_x)(dx)\ pdf_{\mathcal{N}(g(x),s_x)}(y_{\text{obs}}) * \delta_{x,x} \qquad (1)\end{aligned}$$

Given the initial context γ (mapping variable names to values) and the current state p_x (the previous value of `x`), the transition function integrates over all possible values for `x`, then all possible values for `y`, weights each execution by the likelihood of the observation (i.e., the value of the density function on `y_obs`), and returns a measure over the new state x (that will be used as the previous position in the next step), and the results x.

Inference. The explicit `infer` operator computes the stream of distributions described by a model. The state of `infer` is a measure over current states. The transition function integrates the semantics of the model over all possible states, normalizes the resulting measure, and splits the result into a distribution of next states and a distribution of values. The runtime iterates this process from a distribution of initial states to compute a stream of distributions. Line 9 is an example of *inference-in-the-loop* where the distribution `x_dist` is used to compute the command `u` that propels the boat which thus changes the next observation.[1]

Scheduling. The original co-iterative semantics for dataflow synchronous languages [15] interprets mutually recursive equations in arbitrary order with a fixpoint operator in a flat complete partial order (CPO) where variables are either undefined or set to a value. Numerous works define a probabilistic fixpoint operator to interpret loops and recursions [31, 41, 50] in a CPO over measures. Unfortunately, with these definitions the semantics of a set of equations is always the null measure.

To avoid this problem, the ProbZelus semantics defined in [4] thus focuses on a kernel language where local declarations are all *scheduled*. Local declarations such as `x` and `y` yield nested integrals in Equation (1) that are ordered according to data dependencies and therefore well-defined. But imposing a valid schedule is a significant limitation compared to synchronous dataflow languages which manipulate mutually recursive equations in arbitrary order. In this paper, we propose a solution to overcome this limitation.

2.2 Density-based co-iterative semantics

We first propose a new density-based co-iterative semantics for ProbZelus inspired by existing probabilistic semantics [8, 35, 42] where a program defines an un-normalized density function over the random variables. Instead of manipulating measures via integration, probabilistic expressions are now similar to deterministic expressions, but the transition function takes one additional argument — a random element for all random variables — and returns one additional output — a positive score, or weight, which measures the quality of the output w.r.t. the model.

On the example of Figure 1 we have:

$$\begin{aligned}(\!|\texttt{tracker(y\_obs)}|\!)^{\text{step}}_{\gamma}(p_x, r) \ =\ & \mathit{let}\ \mu_x = \mathcal{N}(f(p_x), s_x)\ \mathit{in} \\ & \mathit{let}\ x = \mathit{icdf}_{\mu_x}(r)\ \mathit{in} \\ & \mathit{let}\ \mu_y = \mathcal{N}(g(x), s_y)\ \mathit{in} \\ & x, x, pdf_{\mu_y}(y_{\text{obs}})\end{aligned} \qquad (2)$$

The additional argument r corresponds to the random element for the `sample` operator, i.e., an element of the interval $[0, 1]$ mapped to a sample of a distribution d using inverse transform

[1]A more complex motion model could also use the value of the command inside the tracker.

sampling [29]. The model computes the sample x associated to the random element ($icdf_d$ is the generalized inverse cumulative distribution function of d), and returns the new state x, the result x, and a weight capturing the likelihood of the observation (the density of the distribution $\mathcal{N}(g(x), s_y)$ at y_{obs}).

Inference. At each step, the `infer` operator first computes the un-normalized measure which associates each pair (state, result) to its weight, i.e., for a model e, if $(\!(e)\!)_\gamma\ (m, r) = m', v, w$, $\texttt{infer}(e)$ computes the measure $\int_{[0,1]^p} w * \delta_{m',v}\ dr$ where p is the number of random variables in e. On the example of Figure 1, we can check that this measure corresponds to the original semantics of Equation (1).

$$\begin{aligned}
&\int_{[0,1]} \mathit{let}\ x, x, w = (\!(\texttt{tracker(y\_obs)})\!)^{\text{step}}_\gamma\ (p_x, r)\ \mathit{in}\ w * \delta_{x,x}\ dr \\
&\qquad = \int_{[0,1]} \mathit{let}\ x = \mathit{icdf}_{\mathcal{N}(f(p_x), s_x)}(r)\ \mathit{in}\ \ \mathit{pdf}_{\mathcal{N}(g(x), s_y)}(y_{\text{obs}}) * \delta_{x,x}\ dr \\
&\qquad = \int \mathcal{N}(f(p_x), s_x)(dx)\ \mathit{pdf}_{\mathcal{N}(g(x), s_y)}(y_{\text{obs}}) * \delta_{x,x}
\end{aligned}$$

The semantics of `infer` is then similar to its interpretation in the original kernel-based semantics, i.e., 1) integrate over all possible states, 2) normalize the measure, 3) split the result into a distribution of next states and a distribution of values. We prove in Section 4.2 that this semantics is equivalent to the kernel-based semantics, i.e., the `infer` operator yields the same stream of distributions.

Mutually recursive equations. In the density-based semantics, the transition functions of probabilistic equations are similar to their deterministic counterparts with additional inputs/outputs. Compared to the kernel-based semantics, there are no longer nested integrals and a deterministic fixpoint operator can be used to interpret sets of equations.

Consider a variant of the example of Figure 1 where we swap Lines 3 and 4. For a state p_x and a random element r, the semantics of the local declarations in `tracker` is the fixpoint of the following function F starting from the least element $[x \leftarrow \bot, y \leftarrow \bot]$.

$$\begin{aligned}
F(\rho) \;=\;& \mathit{let}\ y = g(\rho(\mathsf{x}))\ \mathit{in} \\
& \mathit{let}\ \mu_x = \mathcal{N}(f(p_x), s_x)\ \mathit{in} \\
& \mathit{let}\ x = \mathit{icdf}_{\mu_x}(r)\ \mathit{in} \\
& [\mathsf{x} \leftarrow x, \mathsf{y} \leftarrow y]
\end{aligned}
\qquad
\begin{aligned}
\rho_0 &= [x \leftarrow \bot, y \leftarrow \bot] \\
\rho_1 &= [x \leftarrow \mathit{icdf}_{\mu_x}(r), y \leftarrow \bot] \\
\rho_2 &= [x \leftarrow \mathit{icdf}_{\mu_x}(r), y \leftarrow g(\mathit{icdf}_{\mu_x}(r))] \\
\rho_3 &= [x \leftarrow \mathit{icdf}_{\mu_x}(r), y \leftarrow g(\mathit{icdf}_{\mu_x}(r))]
\end{aligned}$$

The fixpoint converges after 3 iterations. Using the resulting environment, the semantics of `tracker` then computes the next state, the resulting value, and the weight which, after simplification, yields the same results as Equation (2).

$$\begin{aligned}
(\!(\texttt{tracker(y\_obs)})\!)^{\text{step}}_\gamma\ (p_x, r) \;=\;& \mathit{let}\ \rho = [x \leftarrow \mathit{icdf}_{\mu_x}(r), y \leftarrow g(\mathit{icdf}_{\mu_x}(r))]\ \mathit{in} \\
& \mathit{let}\ x = \rho(\mathsf{x})\ \mathit{in} \\
& \mathit{let}\ \mu_y = \mathcal{N}(\rho(\mathsf{y}), s_y)\ \mathit{in} \\
& \rho(\mathsf{x}), \rho(\mathsf{x}), \mathit{pdf}_{\mu_y}(y_{\text{obs}})
\end{aligned} \tag{3}$$

Program equivalence. Since deterministic expressions are interpreted as state machines, to prove program equivalence one must exhibit a bisimulation [46], i.e., a relation between the states of the two state machines. Two deterministic expressions are equivalent if there exists a relation such that 1) the initial states are in relation, and 2) given two states in relation the transition function produces new states in relation and the same output. The proof is done by unfolding the definition of the transition function.

Two probabilistic expressions are equivalent if they describe the same stream of measures obtained by integrating at each step the transition function over all possible states computed at the previous step. The bisimulation must thus relate measures of states through successive integrations.

2.3 Density-based relational semantics

An alternative to the co-iterative semantics is to directly manipulate streams of values. This is the approach used in the Vélus project[2] to prove an end-to-end compiler for the dataflow synchronous language Lustre [9–11]. The semantics of a stream function is defined as a relation between input streams and output streams. In Vélus, most of the compilation passes are proven correct using this *relational semantics*. The translation to state machines is one of the very last passes and focuses on a normalized, scheduled subset of the language.

We extend this relational semantics to probabilistic streams. The key idea is to lift the density-based semantics to streams. Given a context H mapping variable names to streams of values, and an array R of random streams, the semantics of an expression returns a stream of pairs (value, weight): $H, R \vdash e \Downarrow (v, w)$.

In the example of Figure 1, R is a single stream of independent random elements $R_0 \cdot R_1 \cdot R_2 \cdot \ldots$ in $[0,1]$ (the operator $\cdot$ represents the concatenation of stream elements). We can interpret `tracker` in a context H that contains the observations `y_obs`:

$$[\texttt{y\_obs} \leftarrow y_{\text{obs}}], R \vdash \texttt{tracker(y\_obs)} \Downarrow \quad (x_0, w_0) \quad \cdot \quad (x_1, w_1) \quad \cdot \quad (x_2, w_2) \quad \cdot \ldots$$

$$\begin{array}{lrcl}
\textit{where} & \mu_x & = & \mathcal{N}(f(x_{\text{init}}), s_x) \cdot \mathcal{N}(f(x_0), s_x) \cdot \mathcal{N}(f(x_1), s_x) \cdot \ldots \\
& x & = & \textit{icdf}_{\mu_{x_0}}(R_0) \cdot \textit{icdf}_{\mu_{x_1}}(R_1) \cdot \textit{icdf}_{\mu_{x_2}}(R_2) \cdot \ldots \\
& y & = & g(x_0) \cdot g(x_1) \cdot g(x_2) \cdot \ldots \\
& \mu_y & = & \mathcal{N}(y_0, s_x) \cdot \mathcal{N}(y_1, s_x) \cdot \mathcal{N}(y_2, s_x) \cdot \ldots \\
& w & = & \textit{pdf}_{\mu_{y_0}}(y_{\text{obs}0}) \cdot \textit{pdf}_{\mu_{y_1}}(y_{\text{obs}1}) \cdot \textit{pdf}_{\mu_{y_2}}(y_{\text{obs}2}) \cdot \ldots
\end{array}$$

The semantics now directly manipulates streams. At each step, the result is similar to the expression in Equation (2), but states are abstracted away. The result is a stream of pairs (value, weight).

Inference. The semantics of `infer` now operates on a stream of pairs (value, weight): $(v_0, w_0) \cdot (v_1, w_1) \cdot (v_2, w_2) \cdot \ldots$. The `infer` operator 1) associates to each value v_k the total weight of its prefix using a cumulative product $\overline{w_k} = \Pi_{i=0}^{k} w_i$, 2) computes the un-normalized measure which associates each pair (state, result) to its total weight, and 3) normalizes it to obtain a distribution of values. The key difference with the density-based co-iterative semantics is that the integral is now over the infinite domain of streams. We prove in Section 5.3 that this semantics is equivalent to the co-iterative density-based semantics, i.e., the `infer` operator yields the same stream of distributions.

Mutually recursive equations. Given the random streams R, the semantics of a set of probabilistic mutually recursive equations $H, R \vdash E : W$ checks that a context H mapping variable names to stream of values is compatible with all the equations in E, and that the combined weight of all sub-expressions is the stream W. Since variables in a context are not ordered, there is nothing special to do to interpret mutually recursive equations. By construction the order of equations does not matter which greatly simplifies reasoning about compilation passes that introduce new equations in arbitrary order. Of course, compared to the state machines of the co-iterative semantics, the relational semantics is not executable since equations are only checked a posteriori for a given context.

[2] https://velus.inria.fr

$$
\begin{array}{rcl}
d & ::= & \texttt{let}\ x = e \mid \texttt{node}\ f\ x = e \mid \texttt{proba}\ f\ x = e \mid d\ d \\
e & ::= & c \mid x \mid (e,e) \mid op(e) \mid \texttt{last}\ x \mid f(e) \mid e\ \texttt{where}\ \texttt{rec}\ E \\
 & \mid & \texttt{present}\ e \ \rightarrow\ e\ \texttt{else}\ e \mid \texttt{reset}\ e\ \texttt{every}\ e \\
 & \mid & \texttt{sample}(e) \mid \texttt{factor}(e) \mid \texttt{infer}(e) \\
E & ::= & x = e \mid \texttt{init}\ x = e \mid E\ \texttt{and}\ E
\end{array}
$$

Fig. 2. ProbZelus Syntax.

Program equivalence. In the relational semantics, states are abstracted away and a probabilistic expression computes a stream of pairs (value, weight) where each element only depends on the random streams. Two probabilistic expressions are equivalent if they describe the same stream of measures obtained by integrating at each step the result of the relational semantics over all possible random streams. If we can map the random streams of one expression to the random streams of the other, program equivalence can be reduced to the comparison of the streams of pairs (value, weight) computed by each expression.

3 BACKGROUND

In this section we briefly summarize the key elements of the co-iterative semantics of ProbZelus. Importantly, this semantics is only defined if all equations are ordered according to data dependencies. We then recall the original co-iterative semantics of synchronous dataflow languages where sets of mutually recursive equations in arbitrary order are interpreted using a fixpoint operator.

3.1 Syntax

The syntax of ProbZelus is presented in Figure 2. A program is a series of declarations d. A declaration can be a global variable `let`, a deterministic stream function `node`, or a probabilistic model `proba`. Each declaration has a unique name. An expression can be a constant c, a variable x, a pair, an operator application $op(e)$, the previous value of a variable `last` x, a function call $f(e)$, a local declaration e `where rec` E where E is a set of mutually recursive equations, a lazy conditional `present` $c \ \rightarrow\ e_1$ `else` e_2, or a reset construct `reset` e_1 `every` e_2. An equation is either a simple definition $x = e$, an initialization `init` $x = e$ (the delay operator `last` x can only be used on initialized variables), or a set of equations E_1 `and` E_2. In a set of equations, every initialized variable must be defined by another equation.

We add the classic probabilistic constructs to the set of expressions: `sample`(d) creates a random variable with distribution d, `factor`(s) increments the log-density of the model, and `infer`(m) computes the posterior distribution of a model m. If d is a distribution with a density function, we use the syntactic shortcut `observe`(d,x) for `factor`$(pdf_d(x))$ which conditions the model on the assumption that x was sampled from d. Recursion, loops, and nested inference are not allowed in the language [1].

3.2 Co-iterative semantics

The semantics of ProbZelus presented in [4] extends the co-iterative semantics of dataflow synchronous languages [15, 20]. The main advantage of the co-iterative semantics is that the state machine interpretation is executable. Recent works demonstrated an interpreter from this semantics that can be used to test and validate a compiler against a reference semantics [17].

A type system statically identifies deterministic and probabilistic expressions [4, Section 3.2] which have different interpretations. In an environment γ mapping variable names to values, a

$$
\begin{aligned}
\{\!|e|\!\}^{\text{init}}_{\gamma} &= \llbracket e \rrbracket^{\text{init}}_{\gamma} \\
\{\!|e|\!\}^{\text{step}}_{\gamma}(m) &= \mathit{let}\ m', v = \llbracket e \rrbracket^{\text{step}}_{\gamma}(m)\ \mathit{in}\ \delta_{m',v} \quad \text{if } e \text{ is deterministic} \\[1ex]
\{\!|\texttt{sample}(e)|\!\}^{\text{init}}_{\gamma} &= \llbracket e \rrbracket^{\text{init}}_{\gamma} \\
\{\!|\texttt{sample}(e)|\!\}^{\text{step}}_{\gamma}(m) &= \mathit{let}\ m', \mu = \llbracket e \rrbracket^{\text{step}}_{\gamma}(m)\ \mathit{in} \int \mu(dv)\ \delta_{m',v} \\[1ex]
\{\!|\texttt{factor}(e)|\!\}^{\text{init}}_{\gamma} &= \llbracket e \rrbracket^{\text{init}}_{\gamma} \\
\{\!|\texttt{factor}(e)|\!\}^{\text{step}}_{\gamma}(m) &= \mathit{let}\ m', v = \llbracket e \rrbracket^{\text{step}}_{\gamma}(m)\ \mathit{in}\ v * \delta_{m',()} \\[1ex]
\left\{\!\!\left|\begin{array}{l} e\ \texttt{where rec init}\ x = c \\ \qquad\quad \texttt{and}\ x = e_x \\ \qquad\quad \texttt{and}\ y = e_y \end{array}\right|\!\!\right\}^{\text{init}}_{\gamma} &= c, \left(\{\!|e|\!\}^{\text{init}}_{\gamma}, \{\!|e_x|\!\}^{\text{init}}_{\gamma}, \{\!|e_y|\!\}^{\text{init}}_{\gamma}\right) \\
\left\{\!\!\left|\begin{array}{l} e\ \texttt{where rec init}\ x = c \\ \qquad\quad \texttt{and}\ x = e_x \\ \qquad\quad \texttt{and}\ y = e_y \end{array}\right|\!\!\right\}^{\text{step}}_{\gamma}(p_x, (m, m_x, m_y)) &= \int \{\!|e_x|\!\}^{\text{step}}_{\gamma+[x.\texttt{last}\leftarrow p_x]}(m_x)(dm'_x, dv_x) \\
&\qquad \int \{\!|e_y|\!\}^{\text{step}}_{\gamma+[x.\texttt{last}\leftarrow p_x, x\leftarrow v_x]}(m_y)(dm'_y, dv_y) \\
&\qquad\quad \int \{\!|e|\!\}^{\text{step}}_{\gamma+[x.\texttt{last}\leftarrow p_x, x\leftarrow v_x, y\leftarrow v_y]}(m)(dm', d_v) \\
&\qquad\quad\ \delta_{(v_x, (m', m'_x, m'_y)), v} \\[1ex]
\llbracket \texttt{infer}(e) \rrbracket^{\text{init}}_{\gamma} &= \llbracket e \rrbracket^{\text{init}}_{\gamma} \\
\llbracket \texttt{infer}(e) \rrbracket^{\text{step}}_{\gamma}(\sigma) &= \mathit{let}\ \nu = \int \sigma(dm)\ \{\!|e|\!\}^{\text{step}}_{\gamma}(m)\ \mathit{in} \\
&\quad\ \mathit{let}\ \overline{\nu} = \nu / \nu(\top)\ \mathit{in} \\
&\quad\ (\pi_{1*}(\overline{\nu}), \pi_{2*}(\overline{\nu}))
\end{aligned}
$$

Fig. 3. A simplified excerpt of the original ProbZelus co-iterative probabilistic semantics [4].

deterministic expression e is interpreted as a state machine characterized by an initial state $\llbracket e \rrbracket^{\text{init}}_{\gamma}$ of type S and a transition function $\llbracket e \rrbracket^{\text{step}}_{\gamma}$ of type $S \to S \times V$ which given the current state returns the next step and a value. A stream of values is then obtained by iteratively applying the transition function from the initial state.

$$\begin{tikzcd}
(\llbracket e \rrbracket^{\text{init}}_{\gamma_0} = m_0) \arrow[r, "\llbracket e \rrbracket^{\text{step}}_{\gamma_1}"] & m_1 \arrow[r, "\llbracket e \rrbracket^{\text{step}}_{\gamma_2}"] & m_2 \arrow[r, "\llbracket e \rrbracket^{\text{step}}_{\gamma_3}"] & m_3 \arrow[r] & \dots \\
 & v_1 & v_2 & v_3 &
\end{tikzcd}$$

Following [48], the semantics of a probabilistic expression is a state machine which computes a stream of *kernels*. Given the current state, the transition function $\{\!|e|\!\}^{\text{step}}_{\gamma}$ of type $S \to \Sigma_{S\times V} \to [0, \infty)$ returns a measure over pairs (next state, value),[3] i.e., a function mapping measurable sets of pairs (next state, value) to a score.

[3] Σ_A denotes the Borel σ-algebra over values of type A.

Figure 3 shows a simplified excerpt of the semantics of probabilistic expressions from [4]. In a probabilistic context, a deterministic expression is interpreted as the Dirac delta[4] measure on the pair (state, value) returned by the deterministic semantics. `sample` evaluates its argument into a new state m' and a distribution of values μ, and returns a measure over pairs (new state, value). `factor` evaluates its argument into a new state m' and a real value v, and returns a Dirac delta measure on the pair $(m', ())$ weighted by v. To simplify the semantics, the type system ensures that the arguments of the probabilistic operators are always deterministic expressions. To illustrate local declarations, Figure 3 shows the semantics of a simple expression with two local variables `x` and `y`. The state captures the previous value of the initialized variable x, and the state of all sub-expressions. The transition function starts in a context where the previous value of x is bound to a special variable x`.last`, and integrates over all possible executions of the sub-expressions to compute the main expression.

Inference. So far, probabilistic expressions describe a stream of un-normalized measures over pairs (state, value). At each step, the `infer` operator normalizes the measure to obtain a distribution ($\top$ denotes the entire space), that is then split into a distribution of next states, and a distribution of values. The corresponding stream of distributions is obtained by iteratively integrating the transition function along the distribution of states.

$$\begin{tikzcd}
(\{\![e]\!\}^{\text{init}}_{\gamma_0} = \sigma_0) \arrow[r, "\int \sigma_0(dm)\{\![e]\!\}^{\text{step}}_{\gamma_1}(m)"] & \sigma_1 \arrow[r, "\int \sigma_1(dm)\{\![e]\!\}^{\text{step}}_{\gamma_2}(m)"] & \sigma_2 \arrow[r, "\int \sigma_2(dm)\{\![e]\!\}^{\text{step}}_{\gamma_3}(m)"] & \sigma_3 \arrow[r] & \dots \\
 & \mu_1 & \mu_2 & \mu_3 &
\end{tikzcd}$$

If the model is ill-defined, the normalization constant can be 0 or ∞, which triggers an exception and stops the execution. It is the programmer's responsibility to avoid such error cases when defining the model.

3.3 Equations and fixpoints

In the interpretation of local declarations in Figure 3, the nested integrals are only well defined if equations are ordered according to data dependencies. The original semantics in [4] thus focuses on a kernel language where local declarations are all *scheduled*: initializations are grouped at the beginning and an equation $y = e_y$ must appear after $x = e_x$ if x appears in e_y outside a `last`. In the compiler, a specialized type system, the *causality analysis* statically checks that a program is *causal*, i.e., that all local declarations can be scheduled [22]. The kernel-based semantics is commutative, i.e., yields the same results for any valid schedule [48], but imposing a scheduled order on equations is a significant limitation compared to block diagrams or synchronous dataflow languages which manipulate set of equations in arbitrary order.

Deterministic equations. The original co-iterative semantics [15] and recent works [17] interpret mutually recursive equations using a fixpoint operator. Values $v \in V$ are interpreted in a flat domain $V_\perp = V + \{\perp\}$ with $\perp$ as the minimal element and the flat order $\leq$: $\forall v \in V.\ \perp \leq v$. $(V_\perp, \perp, \leq)$ is a complete partial order (CPO). This flat CPO is lifted to environments defining the same set of variables: $\forall \rho_1, \rho_2$ such that $dom(\rho_1) = dom(\rho_2) = X$, $\rho_1 \leq \rho_2$ iff $\forall x \in X,\ \rho_1(x) \leq \rho_2(x)$ and the least element is $\perp = [x \leftarrow \perp]_{x \in X}$.

Figure 4 shows the semantics rules for deterministic equations adapted from [20]. The initial state of an equation is the initial state of its defining expression. Given a state m and and environment γ, the transition function returns a new state and an environment containing the variables defined by the equation. The initial state of the composition of two sets of equations E_1 `and` E_2 is the union of the states of E_1 and E_2. The transition function evaluates E_1 and E_2 on their respective parts of the

[4] $\delta_x(U) = 1$ if $x \in U$ and 0 otherwise.

$$
\begin{array}{lcl}
[\![x = e]\!]^{\text{init}}_{\gamma} & = & [\![e]\!]^{\text{init}}_{\gamma} \\
[\![x = e]\!]^{\text{step}}_{\gamma}(m) & = & \mathit{let}\ m', v = [\![e]\!]^{\text{step}}_{\gamma}(m)\ \mathit{in}\ m', [x \leftarrow v] \\
\\
[\![E_1\ \texttt{and}\ E_2]\!]^{\text{init}}_{\gamma} & = & \mathit{let}\ M_1 = [\![E_1]\!]^{\text{init}}_{\gamma}\ \mathit{in} \\
 & & \mathit{let}\ M_2 = [\![E_2]\!]^{\text{init}}_{\gamma}\ \mathit{in} \\
 & & (M_1, M_2) \\
[\![E_1\ \texttt{and}\ E_2]\!]^{\text{step}}_{\gamma}(M_1, M_2) & = & \mathit{let}\ M_1', \gamma_1 = [\![E_1]\!]^{\text{step}}_{\gamma}(M_1)\ \mathit{in} \\
 & & \mathit{let}\ M_2', \gamma_2 = [\![E_2]\!]^{\text{step}}_{\gamma}(M_2)\ \mathit{in} \\
 & & (M_1', M_2'), \gamma_1 + \gamma_2 \\
\\
[\![e\ \texttt{where rec}\ E]\!]^{\text{init}}_{\gamma} & = & \mathit{let}\ m = [\![e]\!]^{\text{init}}_{\gamma}\ \mathit{in} \\
 & & \mathit{let}\ M = [\![E]\!]^{\text{init}}_{\gamma}\ \mathit{in} \\
 & & (m, M) \\
[\![e\ \texttt{where rec}\ E]\!]^{\text{step}}_{\gamma}(m, M) & = & \mathit{let}\ F(\rho) = \left(\mathit{let}\ M', \rho' = [\![E]\!]^{\text{step}}_{\gamma+\rho}(M)\ \mathit{in}\ \rho'\right)\ \mathit{in} \\
 & & \mathit{let}\ \rho = \mathit{fix}(F)\ \mathit{in} \\
 & & \mathit{let}\ M', \rho = [\![E]\!]^{\text{step}}_{\gamma+\rho}(M)\ \mathit{in} \\
 & & \mathit{let}\ m', v = [\![e]\!]^{\text{step}}_{\gamma+\rho}(m)\ \mathit{in} \\
 & & (m', M'), v
\end{array}
$$

Fig. 4. Co-iterative semantics of deterministic equations (adapted from [20]).

state but on the same environment γ. This function returns the updated state and the environment containing the variables defined in both sets of equations.

To interpret an expression with a set of local declarations e `where rec` E, the transition function first computes the environment defined by E with a fixpoint operator. Given a state M, the function $F(\rho) = \mathit{let}\ M', \rho' = [\![E]\!]^{\text{step}}_{\gamma+\rho}(M)\ \mathit{in}\ \rho'$ is continuous and has a minimal fixpoint $\rho = \mathit{fix}(F) = \lim_{n\to\infty}(F^n(\bot))$. After convergence, the transition function evaluates $[\![E]\!]^{\text{step}}_{\gamma+\rho}(M)$ once more to compute the next state M' (leaving ρ unchanged by definition of the fixpoint) and finally evaluates the main expression e in the environment $\gamma + \rho$. If the program is *causal* a valid schedule exists for E, and by monotonicity, each fixpoint iteration computes the value of at least one variable, the fixpoint is thus reached after a finite number of iterations [20].

Probabilistic equations. In a probabilistic context, the semantics operates on measures. Existing works define a fixpoint operator to interpret loops and recursions in probabilistic lambda calculi [31, 41, 50]. The least element is the null measure, i.e., for all measurable set U, $\bot(U) = 0$, and the partial order is $\mu_1 \leq \mu_2$ iff $\forall U.\mu_1(U) \leq \mu_2(U)$. Unfortunately, using this CPO the semantics of a set of equations is always the null measure. Therefore, we cannot directly define a schedule agnostic kernel semantics. We show in the next section how to recover mutually recursive equations in a probabilistic context with the density-based co-iterative semantics

4 DENSITY-BASED CO-ITERATIVE SEMANTICS

In this section we detail a new density-based co-iterative semantics for probabilistic expressions. We show that, in this semantics we can now interpret sets of mutually recursive equations with a

$$
\begin{array}{lcll}
(\!| e |\!)^{\text{init}}_{\gamma} & = & [\![e]\!]^{\text{init}}_{\gamma}, 0 & \\
(\!| e |\!)^{\text{step}}_{\gamma}(m, []) & = & \mathit{let}\ m', v = [\![e]\!]^{\text{step}}_{\gamma}(m)\ \mathit{in}\ m', v, 1 & \text{if } e \text{ is deterministic} \\
\\
(\!| \texttt{sample}(e) |\!)^{\text{init}}_{\gamma} & = & \mathit{let}\ m = [\![e]\!]^{\text{init}}_{\gamma}\ \mathit{in}\ m, 1 & \\
(\!| \texttt{sample}(e) |\!)^{\text{step}}_{\gamma}(m, [r]) & = & \mathit{let}\ m', \mu = [\![e]\!]^{\text{step}}_{\gamma}(m)\ \mathit{in}\ m', \mathit{icdf}_{\mu}(r), 1 & \\
\\
(\!| \texttt{factor}(e) |\!)^{\text{init}}_{\gamma} & = & \mathit{let}\ m = [\![e]\!]^{\text{init}}_{\gamma}\ \mathit{in}\ m, 0 & \\
(\!| \texttt{factor}(e) |\!)^{\text{step}}_{\gamma}(m, []) & = & \mathit{let}\ m', v = [\![e]\!]^{\text{step}}_{\gamma}(m)\ \mathit{in}\ m', (), v & \\
\\
(\!| f(e) |\!)^{\text{init}}_{\gamma} & = & \mathit{let}\ m_f, p_f = \gamma(f.\texttt{init})\ \mathit{in} & \\
 & & \mathit{let}\ m_e, p_e = (\!| e |\!)^{\text{init}}_{\gamma}\ \mathit{in} & \\
 & & (m_f, m_e), p_f + p_e & \\
(\!| f(e) |\!)^{\text{step}}_{\gamma}((m_f, m_e), [r_f : r_e]) & = & \mathit{let}\ m'_e, v_e, w_e = (\!| e |\!)^{\text{step}}_{\gamma}(m_e, r_e)\ \mathit{in} & \\
 & & \mathit{let}\ m'_f, v, w_f = \gamma(f.\texttt{step})(v_e, m_f, r_f)\ \mathit{in} & \\
 & & (m'_f, m'_e), v, w_e * w_f &
\end{array}
$$

Fig. 5. Density-based co-iterative semantics of expressions (full version in Figure 13 of the appendix).

fixpoint operator as in the original co-iterative semantics. We then prove that the density-based semantics is equivalent to the kernel-based semantics, i.e., describes the same stream of distributions.

4.1 Probabilistic semantics with fixpoint

The key idea of the density-based semantics is to externalize all sources of randomness. Compared to the deterministic case, the transition function of a probabilistic expression takes one additional argument: an array of random elements with one random element for each random variable introduced by `sample`. To capture the effect of the `factor` operator, the transition function also returns a weight which measures the quality of the result w.r.t. the model.

Expressions. More formally, the initialization function of a probabilistic expression e, $(\!| e |\!)^{\text{init}}_{\gamma} : S \times \mathbb{N}$ returns the initial state and the number of random variables in e. Since loops and recursive calls are not allowed in the language of Figure 2, the number of calls to `sample` can be statically computed. Given the current state and a value for all random elements (an array of p values in $[0, 1]$ where p is the number of random variables computed by the initialization function) the transition function $(\!| e |\!)^{\text{step}}_{\gamma} : S \times [0, 1]^p \to S \times V \times [0, \infty)$ returns a triple (next state, value, weight).

An excerpt of the density-based co-iterative semantics is presented in Figure 5. If e is deterministic, there is no random variable and no conditioning. The transition function takes an empty array of random elements, evaluates the expression, and returns the next state, the value, and a weight of 1. `sample` defines one random variable. The transition function takes an array containing one random element, evaluates the argument into a distribution, converts the random element into a sample of the distribution, and returns the next state, the sample, and a weight of 1. `factor` updates the weight. The transition function evaluates it arguments into a real value v, and returns the next state, an empty value (), and the score v. The initialization of a function call $f(e)$ evaluates the initialization functions of f and of e, combines the initial states and sums the numbers of random variables. The transition function takes an array containing the random elements for e and f,[5] evaluates the argument e into a value v_e and a weight w_e, uses the value to evaluate the transition function of f which returns a result v and a weight w_f, and returns the combined next states, the result, and the total weight $w_e * w_f$.

Mutually recursive equations. The semantics of probabilistic equations is presented in Figure 6. As for probabilistic expressions, the initialization function returns the initial state, and the number of random variables. Given a state and an array of random elements, the transition function returns a tuple (next state, environment, weight). The equation $x = e$ defines a single variable. The transition function evaluates the defining expression e into a tuple (next state, value, weight), and returns the next state, an environment where x is bound to the value, and the weight. The `init` $x = e$ equation manages the special variable `last` x which refers to the value of x at the previous time step. Compared to the original ProbZelus semantics, we do not require initial values to be constants. The state contains the previous value of x initialized with an undefined value of the correct type *nil*, and the initial state m_0 of the expression e. There are two cases for the transition function. At the first time step, or after a `reset`, the state contains *nil* and the transition function evaluates e using m_0 to compute the initial value i and the corresponding weight, and returns a new state containing the current value of x, an environment where x`.last` is bound to i, and the weight. In any other case, the previous value v of x stored in the state is defined. The transition function returns a new state containing the current value of x, an environment where x`.last` is bound to v, and a weight of 1.

Compared to the original kernel-based semantics described in Section 3.2 which combines measures via integration, the density-based semantics only manipulates deterministic values for which the flat CPO on environments described in Section 3.3 is well defined. The initialization function of an expression with a set of local declarations e `where rec` E combines the initial states of e and E and returns the total number of random variables. The transition function takes an array containing the random elements for e and E, computes the environment ρ defined by E with a fixpoint operator, evaluates $(\!|E|\!)_{\gamma+\rho}\ (M, r)$ once more to compute the next state M' and the weight W, evaluates the main expression e in the environment $\gamma + \rho$ which returns a value v and a weight w, and returns the combined next states, the value v, and the total weight $w * W$. The only difference with the deterministic case is that the transition functions of e and E now take the random elements as arguments and return the weights.

Scheduling. To compare the density-based semantics with the kernel-based semantics, it is useful to define an alternative semantics for a scheduled language without a fixpoint operator. This alternative semantics $(\!|e|\!)^{\text{s step}}$ exactly matches the density-based semantics except for the two following rules:

$$
\begin{aligned}
(\!|E_1\ \texttt{and}\ E_2|\!)^{\text{s step}}_{\gamma}\ ((M_1, M_2), [r_1 : r_2]) \quad &= \quad \textit{let}\ M'_1, \rho_1, w_1 = (\!|E_1|\!)^{\text{s step}}_{\gamma}\ (M_1, r_1)\ \textit{in} \\
&\qquad \textit{let}\ M'_2, \rho_2, w_2 = (\!|E_2|\!)^{\text{s step}}_{\gamma+\rho_1}\ (M_2, r_2)\ \textit{in} \\
&\qquad (M'_1, M'_2), \rho_1 + \rho_2, w_1 * w_2 \\[1em]
(\!|e\ \texttt{where rec}\ E|\!)^{\text{s step}}_{\gamma}\ ((m, M), [r_e : r_E]) \quad &= \quad \textit{let}\ M', \rho, W = (\!|E|\!)^{\text{s step}}_{\gamma}\ (M, r_E)\ \textit{in} \\
&\qquad \textit{let}\ m', v, w = (\!|e|\!)^{\text{s step}}_{\gamma+\rho}\ (m, r)\ \textit{in} \\
&\qquad (m', M'_1, M'_2), v, w * W
\end{aligned}
$$

[5]We note $[r_1 : r_2]$ the concatenation of two arrays.

$$
\begin{array}{lcl}
(\!|x = e|\!)^{\text{init}}_{\gamma} & = & (\!|e|\!)^{\text{init}}_{\gamma} \\
(\!|x = e|\!)^{\text{step}}_{\gamma}\ (m, r) & = & let\ m', v, w = (\!|e|\!)^{\text{step}}_{\gamma}\ (m, r)\ in\ m', [x \leftarrow v], w \\
\\
(\!|\texttt{init}\ x = e|\!)^{\text{init}}_{\gamma} & = & let\ m_0, p = (\!|e|\!)^{\text{init}}_{\gamma}\ \ in\ (nil, m_0), p \\
(\!|\texttt{init}\ x = e|\!)^{\text{step}}_{\gamma}\ ((nil, m_0), r) & = & let\ m', i, w = (\!|e|\!)^{\text{step}}_{\gamma}\ (m_0, r)\ in\ (\gamma(x), m_0), [x.\texttt{last} \leftarrow i], w \\
(\!|\texttt{init}\ x = e|\!)^{\text{step}}_{\gamma}\ ((v, m_0), r) & = & (\gamma(x), m_0), [x.\texttt{last} \leftarrow v], 1 \\
\\
(\!|E_1\ \texttt{and}\ E_2|\!)^{\text{init}}_{\gamma} & = & let\ M_1, p_1 = (\!|E_1|\!)^{\text{init}}_{\gamma}\ \ in \\
 & & let\ M_2, p_2 = (\!|E_2|\!)^{\text{init}}_{\gamma}\ \ in \\
 & & (M_1, M_2), p_1 + p_2 \\
(\!|E_1\ \texttt{and}\ E_2|\!)^{\text{step}}_{\gamma}\ ((M_1, M_2), [r_1 : r_2]) & = & let\ M'_1, \rho_1, w_1 = (\!|E_1|\!)^{\text{step}}_{\gamma}\ (M_1, r_1)\ in \\
 & & let\ M'_2, \rho_2, w_2 = (\!|E_2|\!)^{\text{step}}_{\gamma}\ (M_2, r_2)\ in \\
 & & (M'_1, M'_2), \rho_1 + \rho_2, w_1 * w_2 \\
\\
(\!|e\ \texttt{where}\ \texttt{rec}\ E|\!)^{\text{init}}_{\gamma} & = & let\ m, p_e = (\!|e|\!)^{\text{init}}_{\gamma}\ \ in \\
 & & let\ M, p_E = (\!|E|\!)^{\text{init}}_{\gamma}\ \ in \\
 & & (m, M), p_e + p_E \\
(\!|e\ \texttt{where}\ \texttt{rec}\ E|\!)^{\text{step}}_{\gamma}\ ((m, M), [r_e : r_E]) & = & let\ F(\rho) = \Big(let\ M', \rho, w = (\!|E|\!)_{\gamma+\rho}\ (M, r_E)\ in\ \rho\Big)\ \ in \\
 & & let\ \rho = fix(F)\ in \\
 & & let\ M', \rho, W = (\!|E|\!)^{\text{step}}_{\gamma+\rho}\ (M, r_E)\ in \\
 & & let\ m', v, w = (\!|e|\!)^{\text{step}}_{\gamma+\rho}\ (m, r_e)\ in \\
 & & (m', M'), v, w * W
\end{array}
$$

Fig. 6. Density-based co-iterative semantics of equations.

Since all equations are scheduled, the environment produced by a set of equations can be computed incrementally and there is no need for a fixpoint operator to interpret local declarations.

Proposition 4.1. *For an expression where all equations are scheduled, the scheduled density-based semantics is equal to the density-based semantics with a fixpoint.*

Proof. This result is a consequence of the following lemma:

Lemma 4.2. *For all scheduled equations set E, the scheduled semantics yields the same environment as the fixpoint operator, i.e., for an environment γ, a state M and an array of random elements r:*

$$fix(\lambda\rho.\ let\ M', \rho', w = (\!|E|\!)^{step}_{\gamma+\rho}\ (M, r)\ in\ \rho') \quad = \quad let\ M', \rho, w = (\!|E|\!)^{s\,step}_{\gamma}\ (M, r)\ in\ \rho$$

The proof is by induction on E. It is sufficient to focus on the case E_1 and E_2. Since equations are scheduled, E_1 does not depend on variables defined in E_2 and we have for an environment γ, a state (M_1, M_2) and an array of random elements $[r_1 : r_2]$:

$$
\begin{array}{rl}
& \mathit{fix}\ (\lambda(\rho_1 + \rho_2).\ \mathit{let}\ M_1', \rho_1', w_1 = (\!|E_1|\!)^{\text{step}}_{\gamma+\rho_1+\rho_2}\ (M_1, r_1)\ \mathit{in} \\
& \qquad\qquad\qquad \mathit{let}\ M_2', \rho_2', w_2 = (\!|E_2|\!)^{\text{step}}_{\gamma+\rho_1+\rho_2}\ (M_2, r_2)\ \mathit{in}\ \rho_1' + \rho_2') \\
= & \mathit{fix}\ (\lambda(\rho_1 + \rho_2).\ \mathit{let}\ M_1', \rho_1', w_1 = (\!|E_1|\!)^{\text{step}}_{\gamma+\rho_1}\ (M_1, r_1)\ \mathit{in} \\
& \qquad\qquad\qquad \mathit{let}\ M_2', \rho_2', w_2 = (\!|E_2|\!)^{\text{step}}_{\gamma+\rho_1+\rho_2}\ (M_2, r_2)\ \mathit{in}\ \rho_1' + \rho_2') \\
= & \mathit{let}\ \rho_1'' = \mathit{fix}(\lambda\rho_1.\ \mathit{let}\ M_1', \rho_1', w_1 = (\!|E_1|\!)^{\text{step}}_{\gamma+\rho_1}\ (M_1, r_1)\ \mathit{in}\ \rho_1')\ \mathit{in} \\
& \mathit{let}\ \rho_2'' = \mathit{fix}(\lambda\rho_2.\ \mathit{let}\ M_2', \rho_2', w_2 = (\!|E_2|\!)^{\text{step}}_{\gamma+\rho_1''+\rho_2}\ (M_2, r_2)\ \mathit{in}\ \rho_2')\ \mathit{in}\ \rho_1'' + \rho_2'' \\
= & \mathit{let}\ M_1', \rho_1', w_1 = (\!|E_1|\!)^{\text{s step}}_{\gamma}\ (M_1, r_1)\ \mathit{in} \\
& \mathit{let}\ M_2', \rho_2', w_2 = (\!|E_2|\!)^{\text{s step}}_{\gamma+\rho_1'}\ (M_2, r_2)\ \mathit{in}\ \rho_1' + \rho_2'
\end{array}
$$

□

4.2 Inference

The `infer` operator first turns the result of the density-based semantics into an un-normalized measure, and then performs the same operation as in the kernel-based semantics: 1) integrate over all possible states, 2) normalize the measure, 3) split the result into a distribution of next states and a distribution of values.

$$
\begin{array}{rcl}
[\![\texttt{infer}(e)]\!]^{\text{D init}}_{\gamma} & = & \mathit{let}\ m, p = (\!|e|\!)^{\text{init}}_{\gamma}\ \mathit{in}\ \delta_m, p \\
[\![\texttt{infer}(e)]\!]^{\text{D step}}_{\gamma}\ (\sigma, p) & = & \mathit{let}\ \psi(m) = \displaystyle\int_{[0,1]^p} \mathit{let}\ m', v, w = (\!|e|\!)^{\text{step}}_{\gamma}\ (m, r)\ \mathit{in}\ w * \delta_{(m',v)}\ dr\ \mathit{in} \\
& & \mathit{let}\ \nu = \displaystyle\int \sigma(dm)\ \psi(m)\ \mathit{in} \\
& & \mathit{let}\ \overline{\nu} = \nu/\nu(\top)\ \mathit{in} \\
& & (\pi_{1*}(\overline{\nu}), p), \pi_{2*}(\overline{\nu})
\end{array}
\tag{4}
$$

The state of the `infer` operator contains the number of random variables in the model p (which remains constant) and a distribution of possible states. The initial distribution of states is a Dirac delta measure over the initial state of the model. The transition function first computes a function ψ mapping a state to the un-normalized measure which associates each pair (next state, value) to its weight. The `infer` operator then integrates this function along all possible values of the state, normalizes it, and splits it into a pair of distributions.

Correctness. The previous definition is very similar to its kernel-based semantics counterpart where the function $\psi(m)$ in Equation (4) plays the role of the semantics of the model. We now show that these two notions coincide.

Proposition 4.3. *For all probabilistic expression e with p random variables where all equations are scheduled, the density-based semantics is the density of the measure computed by the kernel semantics, i.e., for all environment γ and state m:*

$$
\left(\int_{[0,1]^p} \mathit{let}\ m', v, w = (\!|e|\!)^{\mathit{step}}_{\gamma}\ (m)\ \mathit{in}\ w * \delta_{m',v}\ dr\right) \quad = \quad \{\![e]\!\}^{\mathit{step}}_{\gamma}\ (m)
$$

Proof. The kernel-based semantics is only defined for a scheduled language. We first prove by induction on the structure of e that the scheduled density-based semantics coincide with the kernel-based semantics. We can then conclude with Proposition 4.1.

The case `sample`(μ) is a simple variable substitution $x = \mathit{icdf}_{\mu}(r)$ where icdf_{μ} is the inverse of the cumulative function of μ. Indeed, any real continuous distribution μ is the pushforward by

$icdf_{\mu}$ of the uniform distribution over $[0,1]$ denoted λ:

$$\int_{[0,1]} \delta_{icdf_{\mu}(r)}\, dr = \int \delta_{icdf_{\mu}(r)}\, \lambda(dr) = \int \delta_x \, icdf_{\mu *}(\lambda)(dx) = \int \delta_x \, \mu(dx) = \mu$$

This property generalizes to discrete distributions, multivariate distributions, and any distributions over Polish spaces. By analogy, we use the notation $icdf_{\mu}$ in all cases.

The case E_1 `and` E_2 is a consequence of Fubini's theorem.

$$\begin{aligned}
&\int \{\!|E_1|\!\}^{\text{step}}_{\gamma}(M_1)(dM_1', d\rho_1) \int \{\!|E_2|\!\}^{\text{step}}_{\gamma+\rho_1}(M_2)(dM_2', d\rho_2)\ \delta_{(M_1',M_2'),\rho_1+\rho_2}\\
&\quad = \int \left(\int_{[0,1]^{p_1}} \mathit{let}\ M_1', \rho_1, w_1 = (\!|E_1|\!)^{\text{step}}_{\gamma}(M_1)\ \mathit{in}\ w_1 * \delta_{M_1,\rho_1}\ dr_1\right)(dM_1', d\rho_1)\\
&\qquad \int \left(\int_{[0,1]^{p_2}} \mathit{let}\ M_1', \rho_2, w_1 = (\!|E_2|\!)^{\text{step}}_{\gamma+\rho_1}(M_2)\ \mathit{in}\ w_2 * \delta_{M_2,\rho_2}\ dr_2\right)(dM_2', d\rho_2)\\
&\qquad \delta_{(M_1',M_2'),\rho_1+\rho_2}\\
&\quad = \int_{[0,1]^{p_1}} \int_{[0,1]^{p_2}} \int \left(\mathit{let}\ M_1', \rho_1, w_1 = (\!|E_1|\!)^{\text{step}}_{\gamma}(M_1)\ \mathit{in}\ w_1 * \delta_{M_1,\rho_1}\right)(dM_1', d\rho_1)\\
&\qquad\qquad \int \left(\mathit{let}\ M_1', \rho_2, w_1 = (\!|E_2|\!)^{\text{step}}_{\gamma+\rho_1}(M_2)\ \mathit{in}\ w_2 * \delta_{M_2,\rho_2}\right)(dM_2', d\rho_2)\ dr_1 dr_2\\
&\qquad\qquad \delta_{(M_1',M_2'),\rho_1+\rho_2}\\
&\quad = \int_{[0,1]^{p_1+p_2}} \ \mathit{let}\ M_1', \rho_1, w_1 = (\!|E_1|\!)^{\text{step}}_{\gamma}(M_1)\ \mathit{in}\\
&\qquad\qquad \mathit{let}\ M_2', \rho_2, w_2 = (\!|E_2|\!)^{\text{step}}_{\gamma+\rho_1}(M_2)\ \mathit{in}\\
&\qquad\qquad w_1 * w_2 * \delta_{(M_1',M_2'),\rho_1+\rho_2}\ dr_1 dr_2
\end{aligned}$$

Other cases are similar. □

We can now state the main correctness theorem, i.e., the `infer` operator yields the same stream of distributions in the density-based semantics and in the kernel based semantics.

Theorem 4.4 (Co-iterative semantics correctness). *For every probabilistic model e where all equations sets are scheduled, for all distribution of states σ:*

$$\begin{array}{lcl}
\forall\gamma,\ [\![\mathtt{infer}(e)]\!]^{D\,init}_{\gamma} & = & [\![\mathtt{infer}(e)]\!]^{init}_{\gamma}, p\\
\forall\gamma,\ [\![\mathtt{infer}(e)]\!]^{D\,step}_{\gamma}(\sigma, p) & = & [\![\mathtt{infer}(e)]\!]^{step}_{\gamma}(\sigma)
\end{array}$$

Proof. By construction, in the density-based semantics, the first element of the initial state of `infer` is a Dirac delta measure on the initial state of the model which corresponds to the initial state of `infer` in the kernel-based semantics.

By Proposition 4.3 the un-normalized measure defined by the density-based semantics matches the measure computed by the kernel-based semantics. Given this measure, the rest of the transition function of `infer` is the same in both cases. □

4.3 Program equivalence

Two expressions are equivalent if they compute the same stream of output values. The semantics is defined with an initial state and a transition function. To prove equivalence of two state machines one must exhibit a bisimulation [46] that relates the states and ensure the equality of output values.

Definition 4.5. Deterministic expressions are equivalent if there is a bisimulation, that is a relation on states $\mathcal{P} \subseteq S \times S$ such that:

- $\forall\gamma,\ (m_1^0, m_2^0) \in \mathcal{P}$ where $m_1^0 = [\![e_1]\!]^{\text{init}}_{\gamma}$ and $m_2^0 = [\![e_2]\!]^{\text{init}}_{\gamma}$
- $\forall\gamma$, if $(m_1, m_2) \in \mathcal{P}$, then $(m_1', m_2') \in \mathcal{P}$ and $v_1 = v_2$ where

$$\begin{aligned}
m_1', v_1 &= [\![e_1]\!]^{\text{step}}_{\gamma}(m_1)\\
m_2', v_2 &= [\![e_2]\!]^{\text{step}}_{\gamma}(m_2)
\end{aligned}$$

Two probabilistic expressions are equivalent if they describe the same output measures obtained by integrating at each step the pairs (value, weight) computed by the density-based co-iterative semantics.

Definition 4.6. Two probabilistic expressions e_1 and e_2 are equivalent if there is a bisimulation $\mathcal{P}$ on measures of states such that:

- $\forall\gamma$, $(\delta_{m_1^0}, \delta_{m_2^0}) \in \mathcal{P}$ where $m_1^0, p_1 = (\!|e_1|\!)_\gamma^{\text{init}}$ and $m_2^0, p_2 = (\!|e_2|\!)_\gamma^{\text{init}}$
- $\forall\gamma$, if $(\sigma_1, \sigma_2) \in \mathcal{P}$, then $(\sigma_1', \sigma_2') \in \mathcal{P}$ and $\mu_1 = \mu_2$ where

$$\begin{aligned}
\sigma_1', \mu_1 \quad &= \quad \mathit{let}\ \psi_1(m_1) = \int_{[0,1]^{p_1}} \mathit{let}\ m_1', v_1, w_1 = (\!|e_1|\!)_\gamma^{\text{step}}\ (m_1, r_1)\ \mathit{in}\ w_1 * \delta_{(m_1', v_1)}\ dr_1\ \mathit{in} \\
&\qquad \mathit{let}\ \nu_1 = \int \sigma_1(dm_1)\ \psi_1(m_1)\ \mathit{in} \\
&\qquad \pi_{1*}(\nu_1), \pi_{2*}(\nu_1) \\
\sigma_2', \mu_2 \quad &= \quad \mathit{let}\ \psi_2(m_2) = \int_{[0,1]^{p_2}} \mathit{let}\ m_2', v_2, w_2 = (\!|e_2|\!)_\gamma^{\text{step}}\ (m_2, r_2)\ \mathit{in}\ w_2 * \delta_{(m_2', v_2)}\ dr_2\ \mathit{in} \\
&\qquad \mathit{let}\ \nu_2 = \int \sigma_2(dm_2)\ \psi_2(m_2)\ \mathit{in} \\
&\qquad \pi_{1*}(\nu_2), \pi_{2*}(\nu_2)
\end{aligned}$$

In the co-iterative density semantics, for a given context, each triplet (state, value, weight) is a function of the previous state and of the random elements. If we can map states of e_1 to states of e_2 and random elements of e_1 to the random elements of e_2 (while preserving uniform distributions), then program equivalence can be reduced to the comparison of the stream of pairs (value, weight).

Proposition 4.7 (Probabilistic equivalence). *Probabilistic expressions e_1 and e_2, with p_1 the number of random variables in e_1 and p_2 the number of random variables in e_2, are equivalent if there is a pair of measurable functions $\phi : S \longrightarrow S$ and $f : [0,1]^{p_1} \longrightarrow [0,1]^{p_2}$ such that:*

- *The Lebesgue measure over $[0,1]^{p_2}$, λ^{p_2} is the pushforward of λ^{p_1} along f, i.e., $\lambda^{p_2} = f_*\lambda^{p_1}$*
- *$\forall\gamma$, $\phi(m_1^0) = m_2^0$ where $m_1^0, p_1 = (\!|e_1|\!)_\gamma^{init}$ and $m_2^0, p_2 = (\!|e_2|\!)_\gamma^{init}$*
- *$\forall\gamma$, if $\phi(m_1) = m_2$ and $f(r_1) = r_2$, then $\phi(m_1') = m_2'$, $v_1 = v_2$ and $w_1 = w_2$, where*

$$\begin{aligned}
m_1', v_1, w_1 &= (\!|e_1|\!)_\gamma^{step}\ (m_1, r_1) \\
m_2', v_2, w_2 &= (\!|e_2|\!)_\gamma^{step}\ (m_2, r_2)
\end{aligned}$$

Proof. We define the bisimulation using the pushforward measure along ϕ: $(\sigma_1, \sigma_2) \in \mathcal{P}$ iff $\sigma_2 = \phi_*\sigma_1$.

Since $\delta_{m_2^0} = \delta_{\phi(m_1^0)} = \phi_*\delta_{m_1^0}$ we have $(\delta_{m_1^0}, \delta_{m_2^0}) \in \mathcal{P}$.

At each step, we have:

$$\begin{aligned}
\nu_1 &= \int \sigma_1(dm_1)\ w_1 * \delta_{m_1', v_1}\ dr_1 \\
\nu_2 &= \int \sigma_2(dm_2)\ w_2 * \delta_{m_2', v_2}\ dr_2
\end{aligned}$$

If $(\sigma_1, \sigma_2) \in \mathcal{P}$, that is $\sigma_2 = \phi_*\sigma_1$ we can apply the changes of variables $r_2 = f(r_1)$ and $m_2 = \phi(m_1)$ in ν_2.

$$\nu_2 = \int \phi_*\sigma_1(dm_2)\ w_2 * \delta_{m_2', v_2}\ f_*\lambda^{p_1}(dr_2) = \int \sigma_1(dm_1)\ w_1 * \delta_{\phi(m_1'), v_1} dr_1$$

We can check that $(\sigma'_1, \sigma'_2) \in \mathcal{P}$ and $\mu_1 = \mu_2$:

$$\sigma'_2 = \pi_{1*}\nu_2 = \int \sigma_1(dm_1)\ w_1 * \delta_{\phi(m'_1)} dr_1 = \phi_*(\pi_{1*}\nu_1) = \phi_* \sigma'_1$$
$$\mu_2 = \pi_{2*}\nu_2 = \int \sigma_1(dm_1)\ w_1 * \delta_{v_1} dr_1 = \pi_{2*}\nu_1 = \mu_1$$

□

Finding such a mapping is in general difficult. A useful simple case is when the two programs involve the same random variables in different orders, e.g., a program and its compiled version after a source-to-source transformation. Then, the measurable function is a permutation of the random elements, and two expressions are equivalent if they compute the same stream of pairs (value, weight).

Example. If x and y are not free variables in e_1 and e_2:

$$\texttt{sample}(e_1) + \texttt{sample}(e_2)\ \sim\ x + y\ \texttt{where rec}\ x = \texttt{sample}(e_2)\ \texttt{and}\ y = \texttt{sample}(e_1)$$

We define the following:

$$m_1^0, 1 = (\!|e_1|\!)_\gamma^{\text{init}} \qquad m'_1, v_1, 1 = (\!|e_1|\!)_\gamma^{\text{step}}\ (m_1, r_1) \qquad v_y = icdf_{v_1}(r_1)$$
$$m_2^0, 1 = (\!|e_2|\!)_\gamma^{\text{init}} \qquad m'_2, v_2, 1 = (\!|e_2|\!)_\gamma^{\text{step}}\ (m_2, r_2) \qquad v_x = icdf_{v_2}(r_2)$$

The left hand side term $e_\ell = \texttt{sample}(e_1) + \texttt{sample}(e_2)$ is interpreted by the state machine:

$$(\!|e_\ell|\!)_\gamma^{\text{init}} = (m_1^0, m_2^0), 2$$
$$(\!|e_\ell|\!)_\gamma^{\text{step}}\ ((m_1, m_2), [r_1 : r_2]) = (m'_1, m'_2), v_y + v_x, 1$$

The right hand side term $e_r = x + y\ \texttt{where rec}\ x = \texttt{sample}(e_2)\ \texttt{and}\ y = \texttt{sample}(e_1)$ is interpreted by the state machine:

$$(\!|e_r|\!)_\gamma^{\text{init}} = (((), ((), ())), (m_2^0, m_1^0)), 2$$
$$(\!|e_r|\!)_\gamma^{\text{step}}\ ((((), ((), ())), (m_2, m_1)), [r_2 : r_1]) = (((), ((), ())), (m'_2, m'_1)), v_x + v_y, 1$$

With $\phi : (m_1, m_2) \mapsto (((), ((), ())), (m_2, m_1))$ and $f : [r_1 : r_2] \mapsto [r_2 : r_1]$ we have:

- f preserves the uniform distribution $f_* \lambda^{p_1} = \lambda^{p_2}$
- ϕ relates initial states $\phi(m_1^0, m_2^0) = (((), ((), ())), (m_2^0, m_1^0))$
- if ϕ relates the current states $\phi(m_1, m_2) = (((), ((), ())), (m_2, m_1))$ and the random elements are permuted, then ϕ relates the next states $\phi(m'_1, m'_2) = (((), ((), ())), (m'_2, m'_1))$, and the two state machines yield the same pairs of values $v_y + v_x = v_x + v_y$ and weights $1 = 1$.

We can thus apply Proposition 4.7 to conclude that the two probabilistic expressions are equivalent.

5 DENSITY-BASED RELATIONAL SEMANTICS

An alternative to the operational view of the co-iterative semantics where expressions are interpreted as state machines is to define a relational semantics where expressions directly return streams of values [19]. This formalism has been used in the Vélus project to prove an end-to-end dataflow synchronous compiler within the Coq proof assistant [9–11].

In this section, we first present a relational semantics for the deterministic expressions of our language. We then define a relational density-based semantics for probabilistic expressions and prove that this semantics is equivalent to the co-iterative density-based semantics, i.e., the `infer` operator yields the same stream of distributions.

$$G, H \vdash c \downarrow c \qquad G, H \vdash x \downarrow H(x) \qquad \frac{x \notin H}{G, H \vdash x \downarrow G(x)} \qquad \frac{G, H \vdash e_1 \downarrow s_1 \qquad G, H \vdash e_2 \downarrow s_2}{G, H \vdash (e_1, e_2) \downarrow (s_1, s_2)}$$

$$\frac{G, H \vdash e \downarrow s}{G, H \vdash op(e) \downarrow op(s)} \qquad \frac{H(x.\mathtt{last}) = s}{G, H \vdash \mathtt{last}\ x \downarrow s}$$

$$\frac{G, H \vdash e \downarrow s_e \qquad G(f) = \mathtt{node}\ f\ x = e_f \qquad G, [x \leftarrow s_e] \vdash e_f \downarrow s}{G, H \vdash f(e) \downarrow s}$$

$$\frac{G, H + H_E \vdash E \qquad G, H + H_E \vdash e \downarrow s}{G, H \vdash e\ \mathtt{where\ rec}\ E \downarrow s}$$

$$\frac{G, H \vdash e \downarrow H(x)}{G, H \vdash x = e} \qquad \frac{G, H \vdash e \downarrow i \cdot s \qquad H(x.\mathtt{last}) = i \cdot H(x)}{G, H \vdash \mathtt{init}\ x = e} \qquad \frac{G, H \vdash E_1 \qquad G, H \vdash E_2}{G, H \vdash E_1\ \mathtt{and}\ E_2}$$

Fig. 7. Deterministic relational semantics (full version in Figure 16 of the appendix).

Notations. In the following, V^ω is the type of infinite streams of values of type V. The infix operator $(\cdot) : V \to V^\omega \to V^\omega$ is the stream constructor (e.g., $1 \cdot 2 \cdot 3 \cdot ...$). Constants are lifted to constant streams (e.g., $1 = 1 \cdot 1 \cdot 1 \cdot ...$) and when the context is clear we write $f(s) = f(s_0) \cdot f(s_1) \cdot ...$ for $map\ f\ s$, and $(s, t) = (s_1, t_1) \cdot (s_2, t_2) \cdot ...$ to cast a pair of streams into a stream of pairs.

5.1 Deterministic relational semantics

In the relational semantics, deterministic expressions compute streams of values. In a context H which maps variables names to stream of values, the semantics of a deterministic expression e returns a stream s: $G, H \vdash e \downarrow s$. The additional context G stores global declarations (global constants and function definitions). The semantics of a set of equations E checks that the context H is compatible with all the equations: $G, H \vdash E$. The semantics of a set of equations thus defines a *relation* between the streams stored in the context. Compared to the co-iterative semantics, the relational semantics is not executable since the context must be guessed a priori and validated against the equations.

Figure 7 presents the relational semantics for deterministic expressions and equations. A constant is interpreted as a constant stream, and a variable returns the corresponding stream in the context. The semantics of a pair evaluates each component independently and packs the results into a stream of pairs. The application of an operator evaluates its argument into a stream of values and maps the operator on the result. `last` x fetches a special variable x`.last` in the context. A function call first evaluates its argument, and then evaluates the body of the function in a context where the parameter is bound to the argument value.

To interpret the expression e `where rec` E, equations E are evaluated in a new context H_E that is also used to evaluate the main expression e. The semantics of a simple equation checks that a variable is associated to the stream computed by its defining expression. The initialization operator `init` $x = e$ prepends an initial value i to the stream associated to x and checks that the special variable x`.last` is bound to this delayed version of x. In the relational semantics, contexts are un-ordered maps and scheduling equations does not change the semantics.

5.2 Probabilistic relational semantics

The key idea of the probabilistic relational semantics is similar to the density-based co-iterative semantics: instead of manipulating streams of measures, probabilistic expressions compute streams of pairs (value, score) using external streams of random elements, and integration is deferred to the `infer` operator.

Figure 8 presents the density-based relational semantics for probabilistic expressions and equations. In a context H which maps variables names to stream of values, the semantics of a probabilistic expression e takes an array of random streams R and returns a stream of pairs (value, weight): $G, H, R \vdash e \Downarrow (s, w)$. The semantics of a set of equations E takes an array containing the random streams of all sub-expressions, checks that the context H is compatible with all the equations, and returns the total weight w_E of all sub-expressions: $G, H, R \vdash E : w_E$.

The semantics of deterministic expressions (e.g., constant or variable) returns the expected stream of values associated to a constant weight of 1. The semantics of `sample` takes an array containing one random stream R, evaluates its argument into a stream of distributions s_μ, and uses the random stream R to compute a stream of samples associated to the constant weight 1: $(icdf_{s_{\mu_0}}(R_0), 1) \cdot (icdf_{s_{\mu_1}}(R_1), 1) \cdot \ldots$ The semantics of `factor` evaluates its arguments into a stream of values w which is used as the weight associated to a stream of empty values: $((), w_0) \cdot ((), w_1) \cdot \ldots$ The semantics of a function call is similar to the deterministic case, but the random streams are split between the argument and the function body, and the total weight captures the weight of the argument and the weight of the function body. Similarly, for an expression with a set of local definitions the random streams are split between sub-expressions and the weight is the total weight of all sub-expressions.

By construction, for any probabilistic expression e, the size of the array of random streams is the number of random variables defined in e, i.e., the number of `sample`. This information can be computed statically (as in the initialization functions of the co-iterative semantics in Section 4.1), and in the following $\mathrm{RV}(e)$ returns the number of random variables in an expression e.

5.3 Inference

As in the density-based co-iterative semantics, the `infer` operator is defined by integrating at each step an un-normalized density function over all possible values of the streams of random elements. The semantics of a probabilistic model returns a pair of stream functions (value, weight) which both depend on the random streams. Given the random streams, at each time step, the semantics of `infer` first computes the total weight of the prefix to capture all the conditioning since the beginning of the execution: $\overline{w} = \Pi\ w = w_0 \cdot (w_0 * w_1) \cdot (w_0 * w_1 * w_2) \cdot \ldots$ Then the function *integ* 1) turns the current value v_n and the total weight $\overline{w}_n$ into an un-normalized measure by integrating over all possible values of the random streams, and 2) normalizes the result to obtain a stream of distributions of values. If $p = \mathrm{RV}(e)$ is the number of random variables in the model and λ_ω^p is the uniform measure over the cube of random streams $([0,1]^\omega)^p$, then:

$$integ_p\ (\overline{w}_n \cdot \overline{ws})\ (v \cdot vs) \quad = \quad \left(let\ \mu = \int \overline{w}_n(H,R)\delta_{v(H,R)}\ \lambda_\omega^p(dR)\ in\ \mu/\mu(\top)\right) \cdot (integ\ \overline{ws}\ vs) \qquad (5)$$

Cube of random streams. The uniform measure over the cube of random streams is defined as follows. Let $[0,1]^\omega$ be the countable product of the measurable spaces on the interval $[0,1]$ endowed with the Lebesgue σ-algebra, i.e., the coarsest σ-algebra such that projections are measurable. We define λ_ω as the uniform distribution on the *continuous cube* defined by a Kolmogorov extension such that for any $k \in \mathbb{N}$, the pushforward measure of λ_ω along the projection $\pi_{\leq k} : [0,1]^\omega \longrightarrow [0,1]^k$ on the first k coordinates is the Lebesgue measure on $[0,1]^k$: $\lambda_{\leq k} = \pi_{\leq k *}(\lambda_\omega)$. For any measurable

$$\frac{G, H \vdash e \downarrow s}{G, H, [] \vdash e \Downarrow (s, 1)} \qquad \frac{G, H \vdash e \downarrow s_\mu}{G, H, [R] \vdash \texttt{sample}(e) \Downarrow (icdf_{s_\mu}(R), 1)}$$

$$\frac{G, H \vdash e \downarrow w}{G, H, [] \vdash \texttt{factor}(e) \Downarrow ((), w)}$$

$$\frac{G, H, R_e \vdash e \downarrow (s_e, w_e) \qquad G(f) = \texttt{proba}\ f\ x = e_f \qquad G, [x \leftarrow s_e], R_f \vdash e_f \Downarrow (s, w)}{G, H, [R_e : R_f] \vdash f(e) \Downarrow (s, w * w_e)}$$

$$\frac{G, H + H_E, R_E \vdash E : w_E \qquad G, H + H_E, R_e \vdash e \Downarrow (s, w)}{G, H, [R_e : R_E] \vdash e\ \texttt{where rec}\ E \Downarrow (s, w * w_E)} \qquad \frac{G, H, R \vdash e \Downarrow (H(x), w)}{G, H, R \vdash x = e : w}$$

$$\frac{G, H, R \vdash e \Downarrow (i \cdot s, w_i \cdot w) \qquad H(x.\text{last}) = i \cdot H(x)}{G, H, R \vdash \texttt{init}\ x = e : w_i \cdot 1} \qquad \frac{G, H, R_1 \vdash E_1 : w_1 \qquad G, H, R_2 \vdash E_2 : w_2}{G, H, [R_1 : R_2] \vdash E_1\ \texttt{and}\ E_2 : w_1 * w_2}$$

$$\frac{p = \text{RV}(e) \qquad [G, H, R \vdash e \Downarrow (s, w) \qquad \overline{w} = \Pi\ w]_{R \in ([0,1]^\omega)^p}}{G, H \vdash \texttt{infer}(e) \downarrow integ_p\ \overline{w}\ s}$$

Fig. 8. Probabilistic relational semantics (full version in Figure 17 of the appendix).

function $g : [0, 1]^k \longrightarrow V$ we have the following change of variable formula:

$$\int g(\pi_{\leq k}(R))\ \lambda_\omega(dR) = \int g(R_{\leq k})\ \lambda_{\leq k}(dR_{\leq k})$$

Integrating a function which only depends on the k first coordinates of R can thus be reduced to integrating over these coordinates. We can then define the uniform measure on the cube of random streams λ_ω^p as the p-ary product measure of λ_ω, and lift the change of variable formula.

Correctness. For a probabilistic expression e, we first relate the co-iterative semantics of Section 4.1 and the relational semantics of Section 5.1. If H is a context mapping variables names to streams of values, H_k is the context where streams are projected on their k-th coordinate and $H_{\leq k}$ is the context where streams are truncated at k. We define similarly $R_{\leq k}$, and R_k for an array of random streams R. The following proposition states that if a program is causal, i.e., if all equations can be scheduled, the co-iterative semantics and the relational semantics coincide.

Proposition 5.1. *For a causal model e, if $G, H, R \vdash e \Downarrow (s, w)$, then there is a co-iterative execution trace $m_0 = (\!|e|\!)_G^{init}$ and $\forall k > 0$, $(m_{k+1}, v_{k+1}, w'_{k+1}) = (\!|e|\!)_{H_{k+1}}^{step}(m_k, R_{k+1})$ such that $\forall k > 0$, m_k, v_k, w'_k only depend on $H_{\leq k}$ and $R_{\leq k}$, $s_k(H, R) = v_k(H_{\leq k}, R_{\leq k})$ and $w_k(H, R) = w'_k(H_{\leq k}, R_{\leq k})$.*

Proof Sketch. Probabilistic constructs aside, the kernel language of Figure 2 is a subset of the language defined in [12]. We compile the model to this language. Probabilistic nodes become deterministic nodes with additional inputs (the random streams) and one additional output (the score). The `sample` operator is also compiled to a function call with one additional input for the random stream, and the `factor` operator simply updates the score. Following the semantics of Figure 8, the distribution of random streams in sub-expressions is performed by the compilation function. We can then apply the correctness theorem of [9, 12] : *if a relational semantics exists, there exists a compiled state machine whose execution matches the relational semantics.*

The execution of the compiled state machine is deterministic and corresponds to the co-iterative semantics of the normalized scheduled program which does not require any fixpoint computation. Since, the normalization and scheduling passes preserve the co-iterative semantics [20], the execution of the state machine also corresponds to the co-iterative semantics of the original unscheduled program.

The property also states that at each instant, the output of a causal model only depends on past inputs and states which is proved by induction on the structure of the program. □

As in Section 4.2, we can now state the main correctness theorem, i.e., the `infer` operator yields the same stream of distributions in the co-iterative semantics and in the relational semantics.

Theorem 5.2 (Relational semantics correctness). *For a causal model e, and for all contexts H, if $G, H \vdash \texttt{infer}(e) \downarrow \mu$ then the co-iterative execution trace yields the same stream of distributions, i.e., $\sigma_0, p = [\![\texttt{infer}(e)]\!]^{D\,init}_{G}$ and $\forall k > 0$, $(\sigma_{k+1}, p), \mu_{k+1} = [\![\texttt{infer}(e)]\!]^{D\,step}_{H_{k+1}}(\sigma_k, p)$.*

Proof. If p is the number of random variables in the model, we show $\forall k > 0$:

$$\sigma_{k+1}(H_{k+1}, \sigma_k) \propto \int_{([0,1]^k)^p} \overline{w_k}(H_{\le k}, R_{\le k}) * \delta_{m_k(H_{\le k}, R_{\le k})} \lambda^p_{\le k}(dR_{\le k})$$

$$\mu_{k+1}(H_{k+1}, \sigma_k) \propto \int_{([0,1]^k)^p} \overline{w_k}(H_{\le k}, R_{\le k}) * \delta_{v_k(H_{\le k}, R_{\le k})} \lambda^p_{\le k}(dR_{\le k})$$

By the induction hypothesis, $\forall k$, H, R, $s(H,R)_k = v_k(H_{\le k}, R_{\le k})$ and $w(H,R)_k = w'_k(H_{\le k}, R_{\le k})$. From the definition of ψ in Equation (4) and Fubini's theorem we have:

$$\begin{aligned}
v_{k+1} &= \int \sigma_k(dm)\ \psi(m) \\
&\propto \int_{([0,1]^k)^p} \overline{w_k}(H_{\le k}, R_{\le k}) * \psi(m_k(H_{\le k}, R_{\le k}))\ \lambda^n_{\le k}(dR_{\le k}) \\
&\propto \int_{([0,1]^k)^p} \int_{[0,1]^p} \overline{w_k}(H_{\le k}, R_{\le k}) * w_{k+1}(H_{k+1}, R_{k+1}) \\
&\qquad\qquad * \delta_{m_{k+1}(H_{\le k+1}, R_{\le k+1}), v_{k+1}(H_{\le k+1}, R_{\le k+1})}\ \lambda^p(dR_{k+1}) \lambda^p_{\le k}(dR_{\le k}) \\
&\propto \int_{([0,1]^{k+1})^p} \overline{w_{k+1}}(H_{\le k+1}, R_{\le k+1}) * \delta_{m_{k+1}(H_{\le k+1}, R_{\le k+1}), v_{k+1}(H_{\le k+1}, R_{\le k+1})}\ \lambda^p_{\le k+1}(dR_{\le k+1})
\end{aligned}$$

The normalization and marginalization by π_{1*} and π_{2*} concludes the inductive case. Then using the change of variable formula on the cube of random streams we get:

$$\mu_k \propto \int \overline{w_k}(R) \delta_{s_k(R)} \lambda^p_\omega(dR)$$

which corresponds to Equation (5) and concludes the proof. □

5.4 Program equivalence

Compared to the co-iterative semantics where proving the equivalence between two state machines requires a bisimulation, in the relational semantics, to prove the equivalence between two programs one need only to check that they define the same streams.

Definition 5.3. Deterministic expressions e_1 and e_2 are equivalent if for any context H and $\forall k > 0$, $s_{1k}(H) = s_{2k}(H)$, where $G, H \vdash e_1 \downarrow s_1$ and $G, H \vdash e_2 \downarrow s_2$.

Two probabilistic expressions are equivalent if they describe the same stream of measures obtained by integrating at each step the streams of pairs (value, weight) computed by the density-based relational semantics.

Definition 5.4. Probabilistic expressions e_1 and e_2 with $\mathrm{RV}(e_1) = p_1$ and $\mathrm{RV}(e_2) = p_2$ are equivalent if for all contexts H, and $\forall k > 0$,

$$\int \overline{w_1}_k(H, R_1) * \delta_{s_{1k}(H,R_1)}\, d\lambda_\omega^{p_1}(R_1) = \int \overline{w_2}_k(H, R_2) * \delta_{s_{2k}(H,R_2)}\, d\lambda_\omega^{p_2}(R_2)$$

where $G, H, R_1 \vdash e_1 \Downarrow (s_1, w_1)$ and $G, H, R_2 \vdash e_2 \Downarrow (s_2, w_2)$.

In the relational semantics, for a given context, each pair (value, weight) is a function of the random streams. Since, random streams are uniformly distributed, if we can map the random streams of e_1 to the random streams of e_2 while preserving uniform distributions, program equivalence can be reduced to the comparison of the streams of pairs (value, weight) computed by each expression.

Proposition 5.5 (Probabilistic equivalence). *Probabilistic expressions e_1 and e_2 with $\mathrm{RV}(e_1) = p_1$ and $\mathrm{RV}(e_2) = p_2$ are equivalent if there is $f : ([0,1]^\omega)^{p_1} \longrightarrow ([0,1]^\omega)^{p_2}$ measurable such that,*

- *$\lambda_\omega^{p_2}$ is the pushforward of $\lambda_\omega^{p_1}$ along f, i.e., $\lambda_\omega^{p_2} = f_*(\lambda_\omega^{p_1})$*
- *for all contexts H and arrays of random streams R, $\forall k > 0$, $s_{1k}(H, R) = s_{2k}(H, f(R))$ and $w_{1k}(H, R) = w_{2k}(H, f(R))$, where $G, H, R \vdash e_1 \Downarrow (s_1, w_1)$ and $G, H, f(R) \vdash e_2 \Downarrow (s_2, w_2)$.*

Proof. For all H, $\forall k > 0$, by the change of variable formula:

$$\begin{aligned}\int \overline{w_1}_k(H, R) * \delta_{s_{1k}(H,R)}\, d\lambda_\omega^{p_1}(R) &= \int \overline{w_2}_k(H, f(R)) * \delta_{s_{2k}(H,f(R))}\, d\lambda_\omega^{p_1}(R) \\ &= \int \overline{w_2}_k(H, R) * \delta_{s_{2k}(H,R)}\, df_*(\lambda_\omega^{p_1})(R)\end{aligned}$$

We conclude as $f_*(\lambda_\omega^{p_1}) = \lambda_\omega^{p_2}$. □

Finding such a mapping is in general difficult. As in Section 4.3, a useful simple case is when the two programs involve the same random variables in different orders. Then, the measurable function is a permutation of the random streams.

The relational semantics of an expression is described by a derivation tree where each relation is a consequence of smaller relations on all the sub-expressions, up to atomic expressions. Two expressions compute the same streams if from the derivation tree of the first, one can build a derivation tree for the second and vice-versa.

Example. If x and y are not free variables in e_1 and e_2:

$$\texttt{sample}(e_1) + \texttt{sample}(e_2) \;\sim\; x + y \;\texttt{where rec}\; x = \texttt{sample}(e_2) \;\texttt{and}\; y = \texttt{sample}(e_1)$$

Let R_i be the array of random streams associated to the expressions $\texttt{sample}(e_i)$. For all contexts H, if $G, H \vdash e_i \downarrow \mu_i$, then we define $s_i = icdf_{\mu_i}(R_i)$. Then, the derivation tree for the lhs expression is:

$$\frac{G, H, R_1 \vdash \texttt{sample}(e_1) \Downarrow (s_1, 1) \qquad G, H, R_2 \vdash \texttt{sample}(e_2) \Downarrow (s_2, 1)}{G, H, [R_1 : R_2] \vdash \texttt{sample}(e_1) + \texttt{sample}(e_2) \Downarrow (s_1 + s_2, 1)}$$

With $H_E = [x \leftarrow s_2, y \leftarrow s_1]$, the derivation tree for the rhs expression is:

$$\dfrac{G, H + H_E, [\,] \;\vdash x + y \Downarrow (s_2 + s_1, 1) \qquad \dfrac{\dfrac{G, H + H_E, R_2 \vdash \texttt{sample}(e_2) \Downarrow (s_2, 1)}{G, H + H_E, R_2 \vdash x = \texttt{sample}(e_2) : 1} \qquad \dfrac{G, H + H_E, R_1 \vdash \texttt{sample}(e_1) \Downarrow (}{G, H + H_E, R_1 \vdash y = \texttt{sample}(e_1}}{G, H + H_E, [R_2 : R_1] \vdash x = \texttt{sample}(e_2) \;\texttt{and}\; y = \texttt{sample}(e_1) : 1}}{G, H, [R_2 : R_1] \vdash x + y \;\texttt{where rec}\; x = \texttt{sample}(e_2) \;\texttt{and}\; y = \texttt{sample}(e_1) \Downarrow (s_2 + s_1, 1)}$$

Since both programs compute the same stream of pairs (value, weight) and the permutation $f([R_1 : R_2]) = [R_2 : R_1]$ preserves the uniform distribution, the two programs are equivalent.

6 APPLICATION: ASSUMED PARAMETER FILTER

In ProbZelus, state-space models can involve two kinds of random variables. *State parameters* are represented by a stream of random variables which evolve over time depending on the previous values and the observations. *Constant parameters* are represented by a random variable whose value is progressively refined from the *prior* distribution with each new observation.

Example. Consider the example of Section 2 where the boat is drifting at a constant speed θ. We want to estimate the moving position (state parameter), and the drift speed (constant parameter). The motion model `f` is now defined as follows (where the noise parameter `st` is a global constant):

```
let proba f(pre_x) = pre_x + theta where
  rec init theta = sample(gaussian(zeros, st))
  and theta = last theta
```

Filtering. To estimate state parameters, Sequential Monte Carlo (SMC) inference algorithms rely on filtering techniques [16, 27]. Filtering is an approximate method which consists of deliberately losing information on the current approximation to refocus future estimations on the most significant information. These methods are particularly well suited to the reactive context where a system in interaction with its environment never stops and must execute with bounded resources. All ProbZelus inference methods are SMC algorithms [2, 4, 5]. Unfortunately, this loss of information is harmful for the estimation of constant parameters which do not change over time.

The most basic SMC algorithm, the *particle filter*, approximates the posterior distribution by launching a set of independent executions, called *particles*. At each step, each particle returns a value associated to a score which measures the quality of the value w.r.t. the model. To recenter the inference on the most significant particles, the inference runtime periodically resamples the set of particles according to their weights. The most significant particles are then duplicated while the least interesting ones are dropped.

Unfortunately, constant parameters are only sampled at the beginning of the execution of each particle. After each resampling step, the duplicated particles share the same value for `theta`. The quantity of information used to estimate constant parameters thus strictly decreases over time until eventually, there is only one possible value left. The upper part of the Figure 9 graphically illustrates this phenomenon.

Assumed Parameter Filter. To mitigate this issue, the *Assumed Parameter Filter* (APF) maintains a symbolic distribution for the constant parameters and splits the inference into two steps: 1) estimate the state parameters distributions, and 2) update the constant parameters distributions [32]. The lower part of Figure 9 illustrates the results of APF on the estimation of the drift speed for our radar example.

The APF algorithm assumes that constant parameters are an input of the model, and their prior distributions an input of a new inference operator `APF.infer` (the APF algorithm is described in Appendix B.1). In this section, we present a program transformation which generates models that are exploitable by the APF algorithm. First, a static analysis identifies the constant parameters and their prior distributions. Then a compilation pass transforms these parameters into additional inputs of the model. We use the relational semantics of Section 5 to prove the correctness of this transformation, i.e., the transformation preserves the ideal semantics of the program.

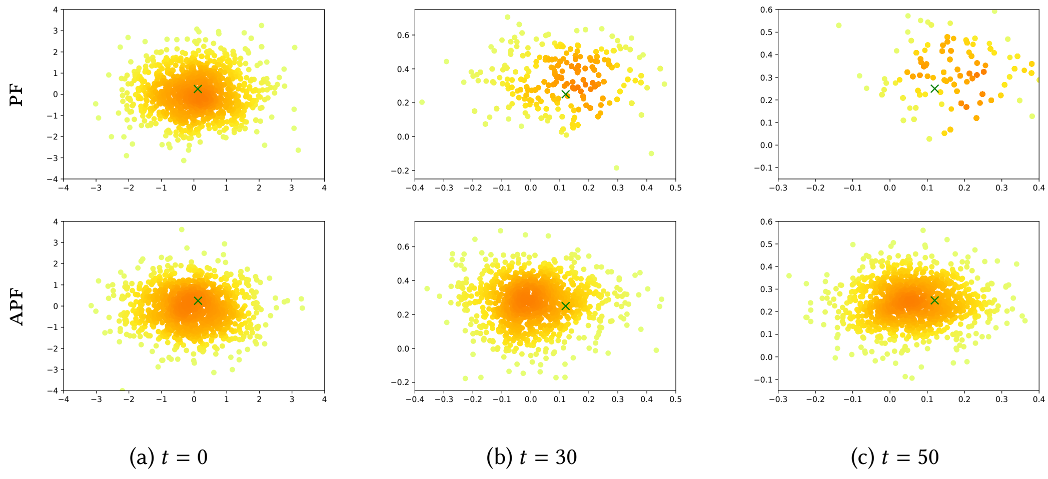

Fig. 9. Estimates of the theta parameter of the radar example over time with a particle filter (PF at the top) and assumed parameter filter (APF at the bottom) . The true drift speed is indicated by a green cross. The color gradient represents the dot density. The scale shrinks over time. Results may differ in between runs.

Example. The radar model (see Figure 1) is compiled into the following model for APF. The constant parameter theta is an argument of the model and the corresponding prior distribution is defined outside the model.

```
let f_prior = gaussian(zeros, st)
proba f_model(theta, pre_x) = pre_x + theta

let tracker_prior = f_prior
proba tracker_model(theta, y_obs) = x where
  rec init x = x_init
  and x = sample(gaussian(f_model(theta, last x), sx))
  and y = g(x)
  and () = observe(gaussian(y, sy), y_obs)

node main(y_obs) = msg where
  rec x_dist = APF.infer(tracker_model, tracker_prior, y_obs)
  and msg = controller(x_dist)
```

6.1 Static Analysis

The goal of the static analysis is to identify the constant parameters of each probabilistic node, i.e., initialized random variables (`init` x = `sample`(e)) that are also constant (x = `last` x). For a program *prog* the judgement $\emptyset, \emptyset \vdash prog : \Phi, C$ builds the environment Φ which associates to each probabilistic node a type ϕ which maps constant parameters to their prior distributions. The environment C contains the global constant variables that can be used to define the prior distributions. An excerpt of the type system is given in Figure 10.

Constants: The auxiliary judgement $C \vdash^c E : C'$ identifies constant streams C' in the set of equations E given the constant variables C. A stream x is constant if it is always equal to its

$$\frac{C \vdash^c e}{\Phi, C \vdash \texttt{let}\ x = e : \Phi, C + \{x\}} \qquad \frac{C \vdash e : \phi}{\Phi, C \vdash \texttt{proba}\ f\ x = e : \Phi + \{f \leftarrow \phi\}, C}$$

$$\frac{C \vdash e_2 : \phi}{C \vdash \texttt{reset}\ e_1\ \texttt{every}\ e_2 : \phi} \qquad \frac{}{C \vdash f_\theta(e) : \{\theta \leftarrow f.\texttt{prior}\}}$$

$$\frac{C \vdash e : \phi_e \qquad C \vdash^c E : D \qquad C, D \vdash E : \phi_E}{C \vdash e\ \texttt{where}\ \texttt{rec}\ E : \phi_e + \phi_E} \qquad \frac{x \in D \qquad C \vdash^c e}{C, D \vdash \texttt{init}\ x = \texttt{sample}(e) : \{x \leftarrow e\}}$$

$$\frac{C \vdash e : \phi}{C, D \vdash x = e : \phi} \qquad \frac{C, D \vdash E_1 : \phi_1 \qquad C, D \vdash E_2 : \phi_2}{C, D \vdash E_1\ \texttt{and}\ E_2 : \phi_1 + \phi_2}$$

Fig. 10. Extract constant parameters and associated prior distributions (full type system in Appendix B.2).

$$\frac{}{C \vdash^c c} \qquad \frac{x \in C}{C \vdash^c x} \qquad \frac{C \vdash^c e_1 \qquad C \vdash^c e_2}{C \vdash^c (e_1, e_2)} \qquad \frac{C \vdash^c e}{C \vdash^c op(e)}$$

$$\frac{C + dom(E) \vdash^c e \qquad C \vdash^c E : dom(E)}{C \vdash^c e\ \texttt{where}\ \texttt{rec}\ E} \qquad \frac{}{C \vdash^c \texttt{init}\ x = e : \emptyset} \qquad \frac{}{C \vdash^c x = \texttt{last}\ x : \{x\}}$$

$$\frac{C \vdash^c e}{C \vdash^c x = e : \{x\}} \qquad \frac{C \nvdash^c e}{C \vdash^c x = e : \emptyset} \qquad \frac{C + C_2 \vdash^c E_1 : C_1 \qquad C + C_1 \vdash^c E_2 : C_2}{C \vdash^c E_1\ \texttt{and}\ E_2 : C_1 + C_2}$$

Fig. 11. Constant expressions and equations

previous value ($x = \texttt{last}\ x$) or if it is defined by a constant expression. The auxiliary judgement $C \vdash^c e$ checks that an expression defines a constant stream. An expression with a set of local declarations is constant if all the equations define constant streams.

Declarations: A global declaration `let` $x = e$ typed in the environment Φ, C adds the name x to the global constant set C if the expression e is constant. A probabilistic node `proba` $f\ x = e$ is associated to the map ϕ computed by the judgement $C \vdash e : \phi$.

Expressions: Typing an expression collects the constant parameters of the sub-expressions. To simplify the analysis, we associate a unique instance name θ to each function call and we assume that all variables and instances names are unique, e.g., $f(1) + f(2)$ becomes $f_{\theta_1}(1) + f_{\theta_2}(2)$. The rule for $f_\theta(e)$ associates to θ the prior distribution of the constant parameters of the body of f: $f.\texttt{prior}$ which is defined as a global variable by the compilation pass. The rule for `reset` e_1 `every` e_2 focuses only on the condition e_2 because e_1 can be re-initialized and thus is not constant.

$$
\begin{array}{lcl}
C_\Phi(\texttt{proba}\ f\ x = e) & = & \texttt{let}\ f.\texttt{prior} = im(\phi) \qquad\qquad \text{with } \phi = \Phi(f) \\
 & & \texttt{proba}\ f.\texttt{model}\ (dom(\phi), x) = C_\phi(e) \\[1ex]
C_\phi(e\ \texttt{where rec}\ E) & = & C_\phi(e)\ \texttt{where rec}\ C_\phi(E) \\[1ex]
C_\phi(\texttt{init}\ x = e) & = & \begin{cases} \emptyset & \text{if } x \in dom(\phi) \\ \texttt{init}\ x = C_\phi(e) & \text{otherwise} \end{cases} \\[1ex]
C_\phi(x = e) & = & \begin{cases} \emptyset & \text{if } x \in dom(\phi) \\ x = C_\phi(e) & \text{otherwise} \end{cases} \\[1ex]
C_\phi(f_\theta(e)) & = & \begin{cases} f(C_\phi(e)) & \text{if } f \text{ is deterministic} \\ f.\texttt{model}(\theta, C_\phi(e)) & \text{if } \theta \in dom(\phi) \\ f.\texttt{model}(\theta, C_\phi(e))\ \texttt{where} & \text{otherwise} \\ \quad \texttt{rec init}\ \theta = \texttt{sample}(f.\texttt{prior}) & \\ \quad \texttt{and}\ \theta = \texttt{last}\ \theta & \end{cases} \\[1ex]
C_\phi(\texttt{infer}(f(e))) & = & \texttt{APF.infer}(f.\texttt{model},\ f.\texttt{prior},\ C_\phi(e))
\end{array}
$$

Fig. 12. APF compilation (full definition in Appendix B.3).

Equations: The typing of e `where rec` E identifies the set D of constant variables in E then types the equations with the judgement $C, D \vdash E : \phi$ where C is the set of constant free variables in E. If a variable x is introduced by the equation `init` x = `sample`(e) and is constant ($x \in D$), then x is a constant parameter and the result type maps x to the distribution e.

Example. On our example, the variable `theta` is identified as a constant parameter of the node `f` and is propagated through the node `tracker` that calls `f`. The final environment is:

$$\Phi = \{\texttt{f} \leftarrow \{\texttt{theta} \leftarrow \texttt{gaussian(...)}\}, \texttt{tracker} \leftarrow \{\theta \leftarrow \texttt{f.prior}\}\}$$

6.2 Compilation

To run the APF algorithm, constant parameters must become additional inputs of the model. The inference runtime can thus execute the model multiple times with different values of the constant parameters to update their distributions. The compilation function is defined in Figure 12 by induction on the syntax and relies on the result of the static analysis. The compilation function C is thus parameterized by the typing environment Φ for declarations and the type ϕ for expressions.

A model `proba` $f\ x = e$ such that $\Phi(f) = \phi$ (*i.e.*, $C \vdash e : \phi$) is compiled into two statements: 1) the prior distribution of the constant parameters in e, `let` f.`prior` = $im(\phi)$, and 2) a new model that takes the constant parameters $dom(\phi)$ as additional arguments, `proba` f.`model` $(dom(\phi), x) = C_\phi(e)$.

The compilation function of an expression C_ϕ removes the definitions of the constant parameters $x \in dom(\phi)$. The `where`/`rec` case effectively removes the constant parameters by keeping only the equations defining variables $x \notin dom(\phi)$. The main difficulty is to handle constant parameters introduced by a function call.

- If the node is deterministic, there is no constant parameter.
- If the constant parameters of the callee are also constant parameters of the caller, we have $\theta \in dom(\phi)$ and we just replace the call to f_θ with a call to f.`model` using the instance name for the constant parameters.

- Otherwise, the constant parameters of the callee are not constant for the caller because the instance f_θ is used inside a `reset`/`every` or a `present`/`else`. In this case, we redefine these parameters locally by sampling their prior distribution f.`prior`.

Finally, the call to `infer`(f(`e`)) is replaced by a call to `APF.infer`(f.`model`, f.`prior`, e).

6.3 Correctness

We use the relational semantics to prove the correctness of the APF compilation pass. First, we prove that any probabilistic expression is equivalent to its compiled version computed in an environment which already contains the definition of the constant parameters. The main theorem which relates `infer`(f(e)) and `APF.infer`(f.`model`, f.`prior`, e) then corresponds to the case $f_\theta(e)$ in Figure 12.

Definition 6.1. For a model f that is compiled into f.`prior` and f.`model`, the ideal semantics of `APF.infer` externalizes the definition of the constant parameters: `APF.infer`(f.`model`, f.`prior`, e) denotes the expression:

$$\texttt{infer}(f.\texttt{model}(\theta, e)\ \texttt{where rec init}\ \theta = \texttt{sample}(f.\texttt{prior}))$$

We first prove the two following correctness lemmas for expressions and equations in parallel.

Lemma 6.2. *For all probabilistic functions f such that $\Phi(f) = \phi$, for all expressions e in the body of f such that $C \vdash e : \phi_e$, there is a permutation $R \longrightarrow [R' : R^p]$ such that*

$$G, H, R \vdash e \Downarrow (s, w) \iff G^+, H + H_f, R' \vdash C_\phi(e) \Downarrow (s, w).$$

where $\vec{p} = dom(\phi_e)$ is the list of constant parameters in e, $\vec{\mu} = im(\phi_e)$ are the corresponding prior distributions and H_f is the context that already contains the definitions of all the constant parameters in f including $\vec{p}$, i.e., $H_f(\vec{p}) = icdf_{\vec{\mu}}(R^p_0)$.

Lemma 6.3. *For all equations E in the body of f such that $C, D \vdash E : \phi_E$, there is a permutation $R \longrightarrow [R' : R^p]$ such that*

$$G, H, R \vdash E : W \iff G^+, H + H_f, R' \vdash C_\phi(E) : W.$$

Proof. The proof is by induction on the expression and the size of the context, i.e., the number of declarations before the expression. For each induction case, we give the mapping between R and $R' + R^p$, and show that a semantics derivation for e in a context G, H, R is equivalent to a semantics derivation for $C_\phi(e)$ in the context $G^+, H + H_f, R'$. We focus on the most interesting cases, i.e., expressions altered by the compilation function.

Case `init` x = `sample`(d) `and` x = `last` x *where* $x \in dom(\phi)$. The permutation is $[R_x] \longrightarrow [] : [R_x]$ and with $v_x = icdf_\mu(R_{x0})$

$$\dfrac{\dfrac{\dfrac{}{G, H \vdash d \downarrow \mu \cdot s_\mu}}{G, H + [x \leftarrow v_x], [R_x] \vdash \texttt{init}\ x = \texttt{sample}(d) : 1} \qquad G, H + [x \leftarrow v_x], [] \vdash x = \texttt{last}\ x : 1}{G, H + [x \leftarrow v_x], [R_x] \vdash \texttt{init}\ x = \texttt{sample}(d)\ \texttt{and}\ x = \texttt{last}\ x : 1}$$

On the other hand, because x is a constant parameter, $C_\phi(d) = \emptyset$ and $H_f(x) = v_x$.

$$\dfrac{H_f(x) = v_x \qquad \dfrac{}{G^+, H + H_f, [] \vdash \emptyset : 1}}{G^+, H + H_f, [] \vdash C_\phi(\texttt{init}\ x = \texttt{sample}(d)\ \texttt{and}\ x = \texttt{last}\ x) : 1}$$

This results can then be generalized to arbitrary sets of equations where the two equations are not necessarily grouped together at the cost of an additional permutation.

Case $g_\theta(e)$. By induction we have the two permutations $R_e \longrightarrow [R'_e : R^p_e]$ and $R_g \longrightarrow [R'_g, R^p_g]$. With `proba` $g\ x = e_g$ and $C \vdash e_g : \phi_g$, we can apply the induction hypothesis on e_g because there is no possible recursive call. The callee context for e_g is thus strictly smaller than the caller context. We also have $H_g = [\vec{p}_g \leftarrow \vec{v}_p]$ with $\vec{p}_g = dom(\phi_g)$, $\vec{\mu}_g = im(\phi_g)$, and $\vec{v}_p = icdf_{\vec{\mu}_g}({R^p_g}_0)$.

$$\frac{\dfrac{G^+, H + H_f, R'_e \vdash C_\phi(e) \Downarrow (s_e, w_e)}{G, H, R_e \vdash e \Downarrow (s_e, w_e)} \qquad \dfrac{G^+, [x \leftarrow s_e] + [\vec{p}_g \leftarrow \vec{v}_p], R'_g \vdash C_{\phi_g}(e_g) \Downarrow (s, w)}{G, [x \leftarrow s_e], R_g \vdash e_g \Downarrow (s, w)}}{G, H, [R_e : R_g] \vdash g_\theta(e) \Downarrow (s, w_e * w)}$$

On the other end, by construction we have $G(g.\mathtt{model})$ = `proba` $g.\mathtt{model}\ (\vec{p}_g, x) = C_{\phi_g}(e_g)$ and there are two cases. If $\theta \in dom(\phi)$, then the constant parameters are already in the context and $H_f(\theta) = \vec{v}_p$. The permutation is $[R_e : R_g] \longrightarrow [R'_e : R'_g] : [R^p_e : R^p_g]$ and we have:

$$\frac{\dfrac{\dfrac{\dfrac{}{G^+, H + H_f, [] \vdash \theta \Downarrow (\vec{v}_p, 1)} \qquad G^+, H + H_f, R'_e \vdash C_\phi(e) \Downarrow (s_e, w_e)}{G^+, H + H_f, R'_e \vdash (\theta, C_\phi(e)) \Downarrow ((\vec{v}_p, s_e), w_e)} \qquad G^+, [x \leftarrow s_e, \vec{p}_g \leftarrow \vec{v}_p], R'_g \vdash C_{\phi_g}(e_g) \Downarrow (s,}{G^+, H + H_f, [R'_e, R'_g] \vdash g.\mathtt{model}(\theta, C_\phi(e)) \Downarrow (s, w_e * w)}}{G^+, H + H_f, [R'_e, R'_g] \vdash C_\phi(g_\theta(e)) \Downarrow (s, w_e * w)}$$

Finally if $\theta \notin dom(\phi)$, the constant parameters are not in the context and the compilation adds a defining equation for θ. The permutation is $[R_e : R_g] \longrightarrow [R'_e : R'_g : R^p_g] : [R^p_e]$, and we have:

$$\frac{\dfrac{\dfrac{\dots}{G^+, H + H_f + [\theta \leftarrow \vec{v}_p], [R'_e, R'_g] \vdash g.\mathtt{model}(\theta, C_\phi(e)) \Downarrow (s, w)} \qquad \dfrac{\dfrac{G^+(g.\mathtt{prior}) = \vec{\mu}_g}{G^+, H + H_f, R^p_g \vdash \mathtt{sample}(g.\mathtt{prior}) \Downarrow (\vec{v}_p, 1)}}{G^+, H + H_f + [\theta \leftarrow \vec{v}_p], R^p_g \vdash \mathtt{init}\ \theta = \mathtt{sample}(g.\mathtt{prior}}}{G^+, H + H_f, [R'_e, R'_g, R^p_g] \vdash g.\mathtt{model}(\theta, C_\phi(e))\ \mathtt{where\ rec\ init}\ \theta = \mathtt{sample}(g.\mathtt{prior}) \Downarrow (s, w)}}{G^+, H + H_f, [R'_e, R'_g, R^p_g] \vdash C_\phi(g_\theta(e)) \Downarrow (s, w)}$$

□

We can now state and prove the correctness of the APF compilation pass.

Theorem 6.4 (APF compilation). *For all probabilistic nodes f,*

$$G, H \vdash \mathtt{infer}(f(e)) \downarrow d \iff G^+, H \vdash \mathtt{APF.infer}(f.\mathtt{model},\ f.\mathtt{prior},\ e) \downarrow d$$

Proof. From Definition 6.1, we need to show that for all random streams R:

$$G, H, R \vdash f_\theta(e) \Downarrow (s, w) \iff G^+, H + [\theta \leftarrow \vec{v}_p], R' \vdash f.\mathtt{model}(\theta, e) \Downarrow (s, w)$$

with $G^+(f.\mathtt{prior}) = \vec{\mu}$ and $\vec{v}_p = icdf_{\vec{\mu}}(R^p_0)$. This corresponds to the case $f_\theta(e)$ with $\theta \in dom(\phi)$. □

7 RELATED WORK

Probabilistic Semantics. In the seminal work [42], two semantics are introduced for a probabilistic imperative language. The first one is already sampling semantics that first picks an infinite stack of random numbers and then executes the program deterministically. In the second semantics, programs are interpreted as distribution transformer: input distributions are transformed to an output sub-probability distribution. This second semantics is defined using measurable functions and kernels.

Probabilistic Coherent Spaces is a generalization of this idea to higher-order types but for discrete probability [25]. This setting has been extended to continuous distributions with models based on positive cones [24, 31], a variation on Banach spaces with positive scalars [47]. To interpret the sampling operation, cones have to be equipped with a measurability structure such that measures and integration can be defined for any types [30].

Quasi-Borel spaces (QBS) define an alternative semantics of higher-order probabilistic programs with conditioning [38, 49] based on measurable spaces and kernels. A probabilistic expression is interpreted as a quasi-Borel measure, i.e., an equivalence class of pairs $[\alpha, \mu]$ where μ is a measure over $\mathbb{R}$, and α is a measurable function from $\mathbb{R}$ to values. Intuitively, the corresponding distribution is obtained as the pushforward of μ along α. Recent work extends this formalism to capture lazy data structures and streams in a functional probabilistic language [26].

Our density-based semantics rely on a similar representation: probabilistic expressions are interpreted by pushingforward a uniform measure over $[0, 1]^n$ along a measurable function. The main difference is that we focus on a domain specific dataflow synchronous language. The set of random variables can be computed statically, and integration is entirely deferred to the `infer` operator. Importantly, we recover the fact that equations sets can be interpreted in any order, a key property for dataflow synchronous languages.

The density-based semantics are also closely related to the sampling semantics of [8] for a higher-order lambda calculus where a sequence of random draws is a parameter of the evaluation rules. The main difference is that we focus on a stream based reactive language. On the one hand, we show how to lift the sampling semantics to the reactive setting. On the other hand, we show how to adapt the reactive semantics (co-iterative and relational) to the probabilistic setting. In addition, there is no recursion and no loop in our language. We thus don't need to define the semantics using step-indexing.

Other works [21, 51, 52] define a similar semantics using $[0, 1]^\omega$ as the *entropy* which describes an idealized random numbers generator. The stream head returns a random element, and a stream can be split (e.g., filtering odd/even indices) to generate two new entropies. This technique is used to interpret programs where random variables can be dynamically created (e.g., in recursive calls). An originality of our language is that we can statically compute the number of streams of random variables in a program, i.e., the calls to `sample`. The random streams exactly correspond to these sample sites and there is no need for a splittable entropy source. This is reminiscent of the Stan language [14] where all random variables must be declared in the `parameters` block, except that in our case, parameters are streams of random variables.

Mutually recursive equations. Positive cones and ωQBS are endowed with a structure of CPO resulting in adequate denotational semantics for probabilistic programming with sampling from continuous distributions, recursive types and term recursion [31, 50]. Although our language does not support term recursion, fixpoint operators in positive cones or ωQBS might be adapted to give a probabilistic semantics of mutually recursive equations related to our density co-iterative semantics. We leave for future investigations these connections.

Program Equivalence. Probabilistic bisimulation has been introduced for testing equivalence of discrete probabilistic systems [44] and generalized to study Labelled Markov Process (continuous systems) [28]. Following [43] which defines a notion of bisimulation for a (higher-order) probabilistic lambda calculus, we could define a notion of probabilistic bisimulation for the co-iterative semantics and compare it to the equivalence of probabilistic expressions we defined.

Probabilistic coupling, is an alternative classic proof technique for probabilistic programs equivalence [3, 39]. A coupling describes correlated executions by associating pairs of samples. Proposition 5.5 that we use to prove the correctness of the APF compilation pass, is a coupling where the relation is made explicit through a measurable function.

A logical relation is proposed in [21, 51, 52] to reason about contextual equivalence for probabilistic programs. In particular [51] uses this framework to prove that reordering declarations preserves the semantics with a permutation of the entropy stream. This reasoning is reminiscent of the proof of Lemma 6.2 (APF Correctness). We thus show that we can apply similar techniques with the relational semantics which manipulates infinite streams and mutually recursive equations. Extending these works to define an equational theory to reason about reactive program equivalence beyond permutations of the random variables streams is a promising future work.

8 CONCLUSION

In this paper we proposed two semantics for a reactive probabilistic programming languages: a density-based co-iterative semantics and a density-based relational semantics. Both semantics are schedule agnostic, i.e., sets of mutually recursive equations can be interpreted in arbitrary order, a key property of synchronous dataflow languages. We defined for both semantics equivalence of programs. The coiterative semantics manipulates state machines and equivalence reasoning requires the description of bisimulations on states. The relational semantics directly manipulates streams, which can lighten program equivalence reasoning for probabilistic expressions. We then defined a program transformation required to run an optimized inference algorithm for state-space models with constant parameters and used the relational semantics to prove the correctness of the transformation.

REFERENCES

[1] Eric Atkinson, Guillaume Baudart, Louis Mandel, Charles Yuan, and Michael Carbin. 2021. Statically bounded-memory delayed sampling for probabilistic streams. *Proc. ACM Program. Lang.* 5, OOPSLA (2021), 1–28.

[2] Eric Atkinson, Charles Yuan, Guillaume Baudart, Louis Mandel, and Michael Carbin. 2022. Semi-symbolic inference for efficient streaming probabilistic programming. *Proc. ACM Program. Lang.* 6, OOPSLA2 (2022), 1668–1696.

[3] Gilles Barthe, Benjamin Grégoire, Justin Hsu, and Pierre-Yves Strub. 2017. Coupling proofs are probabilistic product programs. In *POPL*. ACM, 161–174.

[4] Guillaume Baudart, Louis Mandel, Eric Atkinson, Benjamin Sherman, Marc Pouzet, and Michael Carbin. 2020. Reactive Probabilistic Programming. In *PLDI*.

[5] Guillaume Baudart, Louis Mandel, and Reyyan Tekin. 2022. JAX Based Parallel Inference for Reactive Probabilistic Programming. In *LCTES*.

[6] Albert Benveniste, Paul Caspi, Stephen A. Edwards, Nicolas Halbwachs, Paul Le Guernic, and Robert de Simone. 2003. The synchronous languages 12 years later. *Proc. IEEE* 91, 1 (2003), 64–83.

[7] 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. *J. Mach. Learn. Res.* 20 (2019), 28:1–28:6.

[8] Johannes Borgström, Ugo Dal Lago, Andrew D. Gordon, and Marcin Szymczak. 2016. A lambda-calculus foundation for universal probabilistic programming. In *ICFP*. ACM, 33–46.

[9] Timothy Bourke, Lélio Brun, Pierre-Évariste Dagand, Xavier Leroy, Marc Pouzet, and Lionel Rieg. 2017. A formally verified compiler for Lustre. In *PLDI*.

[10] Timothy Bourke, Lélio Brun, and Marc Pouzet. 2020. Mechanized semantics and verified compilation for a dataflow synchronous language with reset. *Proc. ACM Program. Lang.* 4, POPL (2020), 44:1–44:29.

[11] Timothy Bourke, Paul Jeanmaire, Basile Pesin, and Marc Pouzet. 2021. Verified Lustre Normalization with Node Subsampling. *ACM Trans. Embed. Comput. Syst.* 20, 5s (2021), 98:1–98:25.

[12] Timothy Bourke, Basile Pesin, and Marc Pouzet. 2023. Verified Compilation of Synchronous Dataflow with State Machines. *ACM Trans. Embed. Comput. Syst.* 22, 5s (2023), 137:1–137:26.

[13] Timothy Bourke and Marc Pouzet. 2013. Zélus: a synchronous language with ODEs. In *HSCC*.

[14] 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. *J. Statistical Software* 76, 1 (2017), 1–37.

[15] Paul Caspi and Marc Pouzet. 1998. A Co-iterative Characterization of Synchronous Stream Functions. In *CMCS (Electronic Notes in Theoretical Computer Science, Vol. 11)*. Elsevier.

[16] Nicolas Chopin and Omiros Papaspiliopoulos. 2020. *An introduction to sequential Monte Carlo.* Springer.

[17] Jean-Louis Colaço, Michael Mendler, Baptiste Pauget, and Marc Pouzet. 2023. A Constructive State-based Semantics and Interpreter for a Synchronous Data-flow Language with State machines. In *EMSOFT.*

[18] Jean-Louis Colaço, Bruno Pagano, and Marc Pouzet. 2017. SCADE 6: A formal language for embedded critical software development. In *TASE.*

[19] Jean-Louis Colaço and Marc Pouzet. 2003. Clocks as First Class Abstract Types. In *EMSOFT (Lecture Notes in Computer Science, Vol. 2855)*. Springer, 134–155.

[20] Jean-Louis Colaco, Michael Mendler, Baptiste Pauget, and Marc Pouzet. 2023. A Constructive State-based Semantics and Interpreter for a Synchronous Data-flow Language with State machines. In *EMSOFT.* ACM.

[21] Ryan Culpepper and Andrew Cobb. 2017. Contextual Equivalence for Probabilistic Programs with Continuous Random Variables and Scoring. In *ESOP (Lecture Notes in Computer Science, Vol. 10201)*. Springer, 368–392.

[22] Pascal Cuoq and Marc Pouzet. 2001. Modular Causality in a Synchronous Stream Language. In *ESOP (Lecture Notes in Computer Science, Vol. 2028)*. Springer, 237–251.

[23] Marco F. Cusumano-Towner, Feras A. Saad, Alexander K. Lew, and Vikash K. Mansinghka. 2019. Gen: a general-purpose probabilistic programming system with programmable inference. In *PLDI.*

[24] Fredrik Dahlqvist and Dexter Kozen. 2020. Semantics of higher-order probabilistic programs with conditioning. *Proc. ACM Program. Lang.* 4, POPL (2020), 57:1–57:29.

[25] Vincent Danos and Thomas Ehrhard. 2011. Probabilistic coherence spaces as a model of higher-order probabilistic computation. *Inf. Comput.* 209, 6 (2011), 966–991.

[26] Swaraj Dash, Younesse Kaddar, Hugo Paquet, and Sam Staton. 2023. Affine Monads and Lazy Structures for Bayesian Programming. *Proc. ACM Program. Lang.* 7, POPL (2023), 1338–1368.

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

[28] Josée Desharnais, Abbas Edalat, and Prakash Panangaden. 2002. Bisimulation for Labelled Markov Processes. *Inf. Comput.* 179, 2 (2002), 163–193.

[29] Luc Devroye. 2006. Nonuniform random variate generation. *Handbooks in operations research and management science* 13 (2006), 83–121.

[30] Thomas Ehrhard and Guillaume Geoffroy. 2023. *Integration in cones.* Technical Report. IRIF.

[31] Thomas Ehrhard, Michele Pagani, and Christine Tasson. 2018. Measurable cones and stable, measurable functions: a model for probabilistic higher-order programming. *Proc. ACM Program. Lang.* 2, POPL (2018), 59:1–59:28.

[32] Yusuf Bugra Erol, Yi Wu, Lei Li, and Stuart Russell. 2017. A Nearly-Black-Box Online Algorithm for Joint Parameter and State Estimation in Temporal Models. In *AAAI.*

[33] Hong Ge, Kai Xu, and Zoubin Ghahramani. 2018. Turing: A Language for Flexible Probabilistic Inference. In *Proceedings of Machine Learning Research.*

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

[35] Maria I. Gorinova, Andrew D. Gordon, and Charles Sutton. 2019. Probabilistic programming with densities in SlicStan: efficient, flexible, and deterministic. *Proc. ACM Program. Lang.* 3, POPL (2019), 35:1–35:30.

[36] Nicolas Halbwachs, Paul Caspi, Pascal Raymond, and Daniel Pilaud. 1991. The Synchronous Dataflow Programming Language Lustre. *Proc. of the IEEE* 79, 9 (September 1991), 1305–1320.

[37] David Harel. 1987. Statecharts: A Visual Formalism for Complex Systems. *Sci. Comput. Program.* 8, 3 (1987), 231–274.

[38] Chris Heunen, Ohad Kammar, Sam Staton, and Hongseok Yang. 2017. A convenient category for higher-order probability theory. In *LICS.*

[39] Justin Hsu. 2017. *Probabilistic Couplings for Probabilistic Reasoning.* Ph. D. Dissertation. University of Pennsylvania.

[40] The MathWorks Inc. 2022. *Simulation and Model-Based Design (R2024a).* Natick, Massachusetts, United States. https://www.mathworks.com/products/simulink.html

[41] Claire Jones and Gordon D. Plotkin. 1989. A Probabilistic Powerdomain of Evaluations. In *LICS.* IEEE Computer Society, 186–195.

[42] Dexter Kozen. 1981. Semantics of Probabilistic Programs. *J. Comput. Syst. Sci.* 22, 3 (1981), 328–350.

[43] Ugo Dal Lago and Francesco Gavazzo. 2019. On Bisimilarity in Lambda Calculi with Continuous Probabilistic Choice. In *MFPS (Electronic Notes in Theoretical Computer Science, Vol. 347)*. Elsevier, 121–141.

[44] Kim Guldstrand Larsen and Arne Skou. 1989. Bisimulation Through Probabilistic Testing. In *POPL*. ACM Press, 344–352.

[45] Michael Montemerlo, Sebastian Thrun, Daphne Koller, and Ben Wegbreit. 2002. FastSLAM: A Factored Solution to the Simultaneous Localization and Mapping Problem. In *AAAI*.

[46] David Michael Ritchie Park. 1981. Concurrency and Automata on Infinite Sequences. In *Theoretical Computer Science*.

[47] Peter Selinger. 2004. Towards a semantics for higher-order quantum computation.. In *QPL*.

[48] Sam Staton. 2017. Commutative Semantics for Probabilistic Programming. In *ESOP*.

[49] Sam Staton, Hongseok Yang, Frank D. Wood, Chris Heunen, and Ohad Kammar. 2016. Semantics for probabilistic programming: higher-order functions, continuous distributions, and soft constraints. In *LICS*.

[50] Matthijs Vákár, Ohad Kammar, and Sam Staton. 2019. A domain theory for statistical probabilistic programming. *Proc. ACM Program. Lang.* 3, POPL (2019), 36:1–36:29.

[51] Mitchell Wand, Ryan Culpepper, Theophilos Giannakopoulos, and Andrew Cobb. 2018. Contextual equivalence for a probabilistic language with continuous random variables and recursion. *Proc. ACM Program. Lang.* 2, ICFP (2018), 87:1–87:30.

[52] Yizhou Zhang and Nada Amin. 2022. Reasoning about ”reasoning about reasoning”: semantics and contextual equivalence for probabilistic programs with nested queries and recursion. *Proc. ACM Program. Lang.* 6, POPL (2022), 1–28.

$$
\begin{array}{lcl}
(\!|e|\!)_\gamma^{\text{init}} & = & [\![e]\!]_\gamma^{\text{init}}, 0 \\
(\!|e|\!)_\gamma^{\text{step}}(m, []) & = & let\ m', v = [\![e]\!]_\gamma^{\text{step}}(m)\ in\ m', v, 1 \quad \text{if } e \text{ is determini} \\
\\
(\!|\mathtt{sample}(e)|\!)_\gamma^{\text{init}} & = & let\ m = [\![e]\!]_\gamma^{\text{init}}\ in\ m, 1 \\
(\!|\mathtt{sample}(e)|\!)_\gamma^{\text{step}}(m, [r]) & = & let\ m', \mu = [\![e]\!]_\gamma^{\text{step}}(m)\ in\ m', icdf_\mu(r), 1 \\
\\
(\!|\mathtt{factor}(e)|\!)_\gamma^{\text{init}} & = & let\ m = [\![e]\!]_\gamma^{\text{init}}\ in\ m, 0 \\
(\!|\mathtt{factor}(e)|\!)_\gamma^{\text{step}}(m, []) & = & let\ m', v = [\![e]\!]_\gamma^{\text{step}}(m)\ in\ m', (), v \\
\\
(\!|f(e)|\!)_\gamma^{\text{init}} & = & let\ m_f, p_f = \gamma(f.\mathtt{init})\ in \\
 & & let\ m_e, p_e = (\!|e|\!)_\gamma^{\text{init}}\ in \\
 & & (m_f, m_e), p_f + p_e \\
(\!|f(e)|\!)_\gamma^{\text{step}}((m_f, m_e), [r_f : r_e]) & = & let\ m'_e, v_e, w_e = (\!|e|\!)_\gamma^{\text{step}}(m_e, r_e)\ in \\
 & & let\ m'_f, v, w_f = \gamma(f.\mathtt{step})(v_e, m_f, r_f)\ in \\
 & & (m'_f, m'_e), v, w_e * w_f \\
\\
(\!|\mathtt{present}\ e \ \rightarrow\ e_1\ \mathtt{else}\ e_2|\!)_\gamma^{\text{init}} & = & let\ m_1, p_1 = (\!|e_1|\!)_\gamma^{\text{init}}\ in \\
 & & let\ m_2, p_2 = (\!|e_2|\!)_\gamma^{\text{init}}\ in \\
 & & ([\![e]\!]_\gamma^{\text{init}}, m_1, m_2), p_1 + p_2 \\
(\!|\mathtt{present}\ e \ \rightarrow\ e_1\ \mathtt{else}\ e_2|\!)_\gamma^{\text{step}}((m, m_1, m_2), [r_1 : r_2]) & = & let\ m', v = [\![e]\!]_\gamma^{\text{step}}(m)\ in \\
 & & if\ v\ then\ let\ (m'_1, v_1, w) = (\!|e_1|\!)_\gamma^{\text{step}}(m_1, r_1) \\
 & & \qquad\qquad in\ (m', m'_1, m_2), v_1, w \\
 & & \qquad else\ \ let\ (m'_2, v_2, w) = (\!|e_2|\!)_\gamma^{\text{step}}(m_2, r_2) \\
 & & \qquad\qquad in\ (m', m_1, m'_2), v_2, w \\
\\
(\!|\mathtt{reset}\ e_1\ \mathtt{every}\ e_2|\!)_\gamma^{\text{init}} & = & let\ m_1, p = (\!|e_1|\!)_\gamma^{\text{init}}\ in\ (m_1, m_1, [\![e_2]\!]_\gamma^{\text{init}}), p \\
(\!|\mathtt{reset}\ e_1\ \mathtt{every}\ e_2|\!)_\gamma^{\text{step}}((m_0, m_1, m_2), w, r) & = & let\ m'_2, v_2 = [\![e_2]\!]_\gamma^{\text{step}}(m_2)\ in \\
 & & let\ m'_1, v_1, w = (\!|e_1|\!)_\gamma^{\text{step}}(\ if\ v_2\ then\ m_0\ else\ m_1, r)\ in \\
 & & (m_0, m'_1, m'_2), v_1, w
\end{array}
$$

Fig. 13. Density-based co-iterative semantics for ProbZelus probabilistic expressions.

A SEMANTICS

A.1 Density-based co-iterative semantics

The density-based co-iterative semantics is presented in Section 4. Figures 13 and 14 presents the full semantics for expressions and equations. The additional rules are for `present` and `reset`.

The initialization of `present` $e \ \rightarrow\ e_1$ `else` e_2 allocates memory for e, e_1 and e_2 and count the number of random variables in e_1 and e_2 (e is deterministic and does not have any random variable). The step function first executes e and depending on its value executes e_1 or e_2. The initialization of `reset` e_1 `every` e_2 duplicates the memory needed to execute e_1. That way, in the step function, only the second copy is updated by the transition and if e_1 is reset, the execution restarts from the initial memory state.

$$
\begin{array}{lcl}
\llparenthesis x = e \rrparenthesis_{\gamma}^{\text{init}} & = & \llparenthesis e \rrparenthesis_{\gamma}^{\text{init}} \\
\llparenthesis x = e \rrparenthesis_{\gamma}^{\text{step}}\ (m, r) & = & \mathit{let}\ m', v, w = \llparenthesis e \rrparenthesis_{\gamma}^{\text{step}}\ (m, r)\ \mathit{in}\ m', [x \leftarrow v], w \\
\\
\llparenthesis \texttt{init}\ x = e \rrparenthesis_{\gamma}^{\text{init}} & = & \mathit{let}\ m_0, p = \llparenthesis e \rrparenthesis_{\gamma}^{\text{init}}\ \mathit{in}\ (\mathit{nil}, m_0), p \\
\llparenthesis \texttt{init}\ x = e \rrparenthesis_{\gamma}^{\text{step}}\ ((\mathit{nil}, m_0), r) & = & \mathit{let}\ m', i, w = \llparenthesis e \rrparenthesis_{\gamma}^{\text{step}}\ (m_0, r)\ \mathit{in}\ (\gamma(x), m_0), [x.\texttt{last} \leftarrow i], w \\
\llparenthesis \texttt{init}\ x = e \rrparenthesis_{\gamma}^{\text{step}}\ ((v, m_0), r) & = & (\gamma(x), m_0), [x.\texttt{last} \leftarrow v], 1 \\
\\
\llparenthesis E_1\ \texttt{and}\ E_2 \rrparenthesis_{\gamma}^{\text{init}} & = & \mathit{let}\ M_1, p_1 = \llparenthesis E_1 \rrparenthesis_{\gamma}^{\text{init}}\ \mathit{in} \\
& & \mathit{let}\ M_2, p_2 = \llparenthesis E_2 \rrparenthesis_{\gamma}^{\text{init}}\ \mathit{in} \\
& & (M_1, M_2), p_1 + p_2 \\
\llparenthesis E_1\ \texttt{and}\ E_2 \rrparenthesis_{\gamma}^{\text{step}}\ ((M_1, M_2), [r_1 : r_2]) & = & \mathit{let}\ M_1', \rho_1, w_1 = \llparenthesis E_1 \rrparenthesis_{\gamma}^{\text{step}}\ (M_1, r_1)\ \mathit{in} \\
& & \mathit{let}\ M_2', \rho_2, w_2 = \llparenthesis E_2 \rrparenthesis_{\gamma}^{\text{step}}\ (M_2, r_2)\ \mathit{in} \\
& & (M_1', M_2'), \rho_1 + \rho_2, w_1 * w_2 \\
\\
\llparenthesis e\ \texttt{where rec}\ E \rrparenthesis_{\gamma}^{\text{init}} & = & \mathit{let}\ m, p_e = \llparenthesis e \rrparenthesis_{\gamma}^{\text{init}}\ \mathit{in} \\
& & \mathit{let}\ M, p_E = \llparenthesis E \rrparenthesis_{\gamma}^{\text{init}}\ \mathit{in} \\
& & (m, M), p_e + p_E \\
\llparenthesis e\ \texttt{where rec}\ E \rrparenthesis_{\gamma}^{\text{step}}\ ((m, M), [r_e : r_E]) & = & \mathit{let}\ F(\rho) = \Big(\mathit{let}\ M', \rho, w = \llparenthesis E \rrparenthesis_{\gamma+\rho}\ (M, r_E)\ \mathit{in}\ \rho\Big)\ \mathit{in} \\
& & \mathit{let}\ \rho = \mathit{fix}(F)\ \mathit{in} \\
& & \mathit{let}\ M', \rho, W = \llparenthesis E \rrparenthesis_{\gamma+\rho}^{\text{step}}\ (M, r_E)\ \mathit{in} \\
& & \mathit{let}\ m', v, w = \llparenthesis e \rrparenthesis_{\gamma'}^{\text{step}}\ (m, r_e)\ \mathit{in} \\
& & (m', M'), v, w * W
\end{array}
$$

Fig. 14. Density-based co-iterative semantics for ProbZelus equations.

A.2 Density-based relational semantics

Stream functions. The density-based relational semantics is presented in Section 5. The definition of this semantics relies on a few stream functions presented Figure 15
The function *tl* drops the first element of a stream (tl^n drops the n first elements). The function *map f s* applies f to each element of the stream s. *merge cs as bs* merges the streams *as* and *bs* according to the condition *cs*. *as when cs* keeps the values of *as* only when the condition *cs* is true. The function *slicer ss cs* is used to define the semantics of `reset` e_1 `every` e_2. The first argument *ss* is a stream of streams where each stream represents e_1 restarted at each time step, and *cs* the the reset condition. When the condition is false, the first value of the first stream of *ss* is returned and the second stream of *ss* is discarded. We progress by one step in e_1 and the stream representing e_1 restarted at the current iteration is not useful since the expression was not reset. When the condition is true, the first stream of *ss* which represents the current state of e_1 is discarded and the execution restarts with the first value of the second stream of *ss* which represents e_1 restarted at the current step.

Environment. An *environment* H is a map from variable names to streams of values, for any bound variable $x \in dom(H)$, $H(x) : A^{\omega}$. When the context is clear, we write $f\ H$ for *map* $f\ H$, e.g., for all $x \in dom(H)$, $(tl\ H)(x) = tl\ (H(x))$.

Relational semantics. The full deterministic and probabilistic density-based relational semantics including the rules for `present` and `reset` are given in Figures 16 and 17. The semantics of the

$$\begin{array}{rcl}
tl & : & A^\omega \longrightarrow A^\omega \\
tl\ (a \cdot as) & = & as \\
\\
map & : & (A \longrightarrow B) \longrightarrow (A^\omega \longrightarrow B^\omega) \\
map\ f\ (a \cdot as) & = & f(a) \cdot (map\ f\ as) \\
\\
merge & : & \mathbb{B}^\omega \longrightarrow A^\omega \longrightarrow A^\omega \longrightarrow A^\omega \\
merge\ (T \cdot cs)\ (a \cdot as)\ bs & = & a \cdot (merge\ cs\ as\ bs) \\
merge\ (F \cdot cs)\ as\ (b \cdot bs) & = & b \cdot (merge\ cs\ as\ bs) \\
\\
when & : & \mathbb{A}^\omega \longrightarrow \mathbb{B}^\omega \longrightarrow A^\omega \\
(a \cdot as)\ when\ (T \cdot cs) & = & a \cdot (as\ when\ cs) \\
(a \cdot as)\ when\ (F \cdot cs) & = & as\ when\ cs \\
\\
slicer & : & (A^\omega)^\omega \longrightarrow \mathbb{B}^\omega \longrightarrow A^\omega \\
slicer\ ((a \cdot as) \cdot bs \cdot ss)\ (F \cdot cs) & = & a \cdot (slicer\ (as \cdot ss)\ cs) \\
slicer\ (as \cdot (b \cdot bs) \cdot ss)\ (T \cdot cs) & = & b \cdot (slicer\ (bs \cdot ss)\ cs)
\end{array}$$

Fig. 15. Stream functions for the density-based relational semantics.

$$G, H \vdash c \downarrow c \qquad G, H \vdash x \downarrow H(x) \qquad \frac{x \notin H}{G, H \vdash x \downarrow G(x)} \qquad \frac{G, H \vdash e_1 \downarrow s_1 \qquad G, H \vdash e_2 \downarrow s_2}{G, H \vdash (e_1, e_2) \downarrow (s_1, s_2)}$$

$$\frac{G, H \vdash e \downarrow s}{G, H \vdash op(e) \downarrow op(s)} \qquad \frac{H(x.\texttt{last}) = s}{G, H \vdash \texttt{last}\ x \downarrow s}$$

$$\frac{G, H \vdash e \downarrow s_e \qquad G(f) = \texttt{node}\ f\ x = e_f \qquad G, [x \leftarrow s_e] \vdash e_f \downarrow s}{G, H \vdash f(e) \downarrow s}$$

$$\frac{G, H + H_E \vdash E \qquad G, H + H_E \vdash e \downarrow s}{G, H \vdash e\ \texttt{where rec}\ E \downarrow s}$$

$$\frac{G, H \vdash e \downarrow s_c \qquad G, (H\ when\ s_c) \vdash e_1 \downarrow s_1 \qquad G, (H\ when\ not\ s_c) \vdash e_2 \downarrow s_2}{G, H \vdash \texttt{present}\ e\ \longrightarrow\ e_1\ \texttt{else}\ e_2 \downarrow merge\ s_c\ s_1\ s_2}$$

$$\frac{[G, (tl^{\,n}\ H) \vdash e_1 \downarrow s_n]_{n \in \mathbb{N}} \qquad G, H \vdash e_2 \downarrow s_c}{G, H \vdash \texttt{reset}\ e_1\ \texttt{every}\ e_2 \downarrow slicer\ (s_0 \cdot s_0 \cdot s_1 \cdot s_2 \cdot \ldots)\ s_c} \qquad \frac{G, H \vdash e \downarrow H(x)}{G, H \vdash x = e}$$

$$\frac{G, H \vdash e \downarrow v_i \cdot s_i \qquad H(x.\texttt{last}) = v_i \cdot H(x)}{G, H \vdash \texttt{init}\ x = e} \qquad \frac{G, H \vdash E_1 \qquad G, H \vdash E_2}{G, H \vdash E_1\ \texttt{and}\ E_2}$$

Fig. 16. Deterministic relational semantics.

$$G, H, [\,] \vdash c \Downarrow (c, 1) \qquad G, H, [\,] \vdash x \Downarrow (H(x), 1) \qquad \frac{x \notin H}{G, H, [\,] \vdash x \Downarrow (G(x), 1)}$$

$$\frac{G, H, R_e \vdash e \downarrow (s_e, w_e) \qquad G(f) = \texttt{proba}\ f\ x = e_f \qquad G, [x \leftarrow s_e], R_f \vdash e_f \Downarrow (s, w)}{G, H, [R_e : R_f] \vdash f(e) \Downarrow (s, w * w_e)}$$

$$\frac{G, H + H_E, R_E \vdash E : w_E \qquad G, H + H_E, R_e \vdash e \Downarrow (s, w)}{G, H, [R_e : R_E] \vdash e\ \texttt{where rec}\ E \Downarrow (s, w * w_E)}$$

$$\frac{G, H, R_e \vdash e \downarrow s_c \qquad G, (H, R_1\ \mathit{when}\ s_c) \vdash e_1 \Downarrow sw_1 \qquad G, (H, R_2\ \mathit{when\ not}\ s_c) \vdash e_2 \Downarrow sw_2}{G, H, [R_1 : R_2] \vdash \texttt{present}\ e \ \longrightarrow\ e_1\ \texttt{else}\ e_2 \Downarrow \mathit{merge}\ s_c\ sw_1\ sw_2}$$

$$\frac{\begin{array}{c}[G, (tl^{\,n}\ H, R_1) \vdash e_1 \Downarrow sw_n]_{n \in \mathbb{N}} \\ G, H, R_2 \vdash e_2 \downarrow s_c \qquad (s, w) = \mathit{slicer}\ (sw_0 \cdot sw_0 \cdot sw_1 \cdot sw_2 \cdot \ldots)\ s_c\end{array}}{G, H, [R_1 : R_2] \vdash \texttt{reset}\ e_1\ \texttt{every}\ e_2 \Downarrow (s, w)}$$

$$\frac{G, H \vdash e \downarrow s_\mu}{G, H, [R] \vdash \texttt{sample}(e) \Downarrow (\mathit{icdf}_{s_\mu}(R), 1)} \qquad \frac{G, H \vdash e \downarrow w}{G, H, [\,] \vdash \texttt{factor}(e) \Downarrow ((), w)}$$

$$\frac{G, H, R \vdash e \Downarrow (H(x), w)}{G, H, R \vdash x = e : w} \qquad \frac{G, H, R \vdash e \Downarrow (i \cdot s, w_i \cdot w) \qquad H(x.\texttt{last}) = i \cdot H(x)}{G, H, R \vdash \texttt{init}\ x = e : w_i \cdot 1}$$

$$\frac{G, H, R_1 \vdash E_1 : w_1 \qquad G, H, R_2 \vdash E_2 : w_2}{G, H, [R_1 : R_2] \vdash E_1\ \texttt{and}\ E_2 : w_1 * w_2}$$

$$\frac{p = \mathrm{RV}(e) \qquad [G, H, R \vdash e \Downarrow (s, w) \qquad \overline{w} = \Pi\ w]_{R \in ([0,1]^\omega)^p}}{G, H \vdash \texttt{infer}(e) \downarrow \mathit{integ}_p\ \overline{w}\ s}$$

Fig. 17. Probabilistic relational semantics.

control structure `present` $e \ \longrightarrow\ e_1$ `else` e_2 uses the *when* function on the environment H such that the execution of e_1 and e_2 respectively progress only when the condition is true or false. Then the value of these two streams are merged using the *merge* function. The semantics of `reset` e_1 `every` e_2 is based on the *slicer* function. $[G, (tl^{\,n}\ H) \vdash e_1 \downarrow s_n]_{n \in \mathbb{N}}$ represents the stream of streams where s_n is the stream of values computed by e_1 restarted at time step n. In the slicer, the stream s_0 is duplicated because `reset` e_1 `every` e_2 returns the same value whether or not e_2 is true at the initial step.

B APPLICATION: ASSUMED PARAMETERS FILTERING

B.1 Algorithm

The inference methods proposed by ProbZelus [2, 4, 5] belong to the family of SMC algorithms. These methods rely on a set of independent simulations, called *particles*. Each particle returns an

```
Data: probablisitic model model, observation $y_t$, and previous result $\mu_{t-1}$.
Result: $\mu_t$ an approximation of the distribution of $p_t$.
for each particle $i = 1$ to $N$ do
    $p^i_{t-1}$ = sample()
    $p^i_t, w^i_t$ = model($y_t \mid p^i_{t-1}$)
$\mu_t = \mathcal{M}(\{w^i_t, p^i_t\}_{1 \le i \le N})$
return $\mu_t$
```

Algorithm 1: Particle Filter.

output value associated with a score. The score represents the quality of the simulation. A large number of particles makes it possible to approximate the desired distribution.

More concretely, the `sample(d)` construct randomly draws a value from the `d` distribution, and the `factor(x)` construct multiplies the current score of the particle by `x`. At each instant, the `infer` operator accumulates the values calculated by each particle weighted by their scores to approximate the posterior distribution.

If the model calls on the operator `sample` at each instant, for example to estimate the position of the boat in the radar example (Section 2), the previous method implements a simple random walk for each particle. As time progresses, it becomes increasingly unlikely that one of the random walks will coincide with the stream of observations. The score associated with each particle quickly goes down towards 0.

To solve this issue, sequential Monte Carlo methods (SMC) add a filtering step. Algorithm 1 describes the execution of one instant for a particle filter, the most basic SMC algorithm. At each instant t, a particle $1 \le i \le N$ corresponds to a possible value of the parameters (*i.e.*, random variables) p^i_t of the model. We begin by sampling a new set of particles in the distribution obtained at the previous step. The most probable particles are thus duplicated and the less probable ones are eliminated. This refocuses the inference around the most significant information while maintaining the same number of particles throughout the execution. Knowing the previous state p^i_{t-1}, each particle then executes a step of the model to obtain a sample of the parameters p^i_t associated with a score w^i_t. At the end of the instant, we construct a distribution μ_t where each particle is associated with its score. $\mathcal{M}(\{w^i_t, p^i_t\}_{1 \le i \le N})$ is a multinomial distribution, where the value p^i_t is associated with the probability $w^i_t / \sum_{i=1}^N w^i_t$.

Unfortunately, this approach generates a loss of information for the estimation of constant parameters. On our radar example, at the first instant, each particle draws a random value for the parameter `theta`. At each instant, the duplicated particles share the same value for `theta`. The quantity of information useful for estimating `theta` therefore decreases with each new filtering and, after a certain time, only one possible value remains.

Rather than sampling at the start of execution a set of values for the constant parameters that will impoverish with each filtering, in the APF algorithm, each particle computes a symbolic distribution of constant parameters. At runtime, the inference then alternates between a sampling pass to estimate the state parameters, and an optimization pass which updates the constant parameters. This avoids impoverishment for the estimation of the constant parameters.

More formally, Algorithm 2 describes the execution of one step of APF. At each instant t, a particle $1 \le i \le N$ corresponds to a possible value of the state parameters x^i_t and a distribution of constant parameters Θ^i_t. As for the particle filter, we begin by sampling a set of particles in the distribution obtained at the previous instant. We then sample a value θ^i in Θ^i_{t-1}. Knowing the value of the constant parameters θ^i and the previous state x^i_{t-1}, we can execute a step of

```
Data: probabilistic model model, observation $y_t$, and previous result $\mu_{t-1}$.
Result: $\mu_t$ an approximation of the distributions of state parameter $x_t$ and constant
        parameter $\theta$.
for each particle $i = 1$ to $N$ do
    $x^i_{t-1}, \Theta^i_{t-1} = \texttt{sample}()$
    $\theta^i = \texttt{sample}()$
    $x^i_t, w^i_t = \texttt{model}(y_t \mid \theta^i, x^i_{t-1})$
    $\Theta^i_t = \mathit{Udpate}(\Theta^i_{t-1}, \lambda\theta.\ \texttt{model}(y_t \mid \theta, x^i_{t-1}, x^i_t))$
$\mu_t = \mathcal{M}(\{w^i_t, (x^i_t, \Theta^i_t)\}_{1 \le i \le N})$
return $\mu_t$
```

Algorithm 2: *Assumed Parameter Filter* [32].

$$\frac{C \vdash^c e}{\Phi, C \vdash \texttt{let}\ x = e : \Phi, C + \{x\}} \qquad \frac{}{\Phi, C \vdash \texttt{node}\ f\ x = e : \Phi + \{f \leftarrow \emptyset\}, C}$$

$$\frac{C \vdash e : \phi}{\Phi, C \vdash \texttt{proba}\ f\ x = e : \Phi + \{f \leftarrow \phi\}, C} \qquad \frac{\Phi, C \vdash d_1 : \Phi_1, C_1 \qquad \Phi_1, C_1 \vdash d_2 : \Phi', C'}{\Phi, C \vdash d_1\ d_2 : \Phi', C'}$$

$$\frac{}{C \vdash c : \emptyset} \qquad \frac{}{C \vdash x : \emptyset} \qquad \frac{C \vdash e_1 : \phi_1 \qquad C \vdash e_2 : \phi_2}{C \vdash (e_1, e_2) : \phi_1 + \phi_2} \qquad \frac{C \vdash e : \phi}{C \vdash op(e) : \phi} \qquad \frac{C \vdash e : \phi}{C \vdash \texttt{sample}(e) : \phi}$$

$$\frac{C \vdash e : \phi}{C \vdash \texttt{factor}(e) : \phi} \qquad \frac{}{C \vdash \texttt{last}\ x : \emptyset} \qquad \frac{C \vdash e_1 : \phi}{C \vdash \texttt{present}\ e_1 \ \rightarrow\ e_2\ \texttt{else}\ e_3 : \phi}$$

$$\frac{C \vdash e_2 : \phi}{C \vdash \texttt{reset}\ e_1\ \texttt{every}\ e_2 : \phi} \qquad \frac{}{C \vdash f_\theta(e) : \{\theta \leftarrow f.\texttt{prior}\}}$$

$$\frac{C \vdash e : \phi_e \qquad C \vdash^c E : D \qquad C, D \vdash E : \phi_E}{C \vdash e\ \texttt{where rec}\ E : \phi_e + \phi_E} \qquad \frac{x \in D \qquad C \vdash^c e}{C, D \vdash \texttt{init}\ x = \texttt{sample}(e) : \{x \leftarrow e\}}$$

$$\frac{C \vdash e : \phi}{C, D \vdash \texttt{init}\ x = e : \phi} \qquad \frac{C \vdash e : \phi}{C, D \vdash x = e : \phi} \qquad \frac{C, D \vdash E_1 : \phi_1 \qquad C, D \vdash E_2 : \phi_2}{C, D \vdash E_1\ \texttt{and}\ E_2 : \phi_1 + \phi_2}$$

Fig. 18. Extract constant parameters and associated prior distributions.

the model to obtain a sample of the state parameters x^i_t associated with a score w^i_t. We can then update Θ^i_t by exploring the other possible values for θ knowing that the particle has chosen the transition $x^i_{t-1} \rightarrow x^i_t$. At the end of the instant, we construct a distribution μ_t where each particle is associated with its score. $\mathcal{M}(\{w^i_t, (x^i_t, \Theta^i_t)\}_{1 \le i \le N})$ is a multinomial distribution where the pair of values (x^i_t, Θ^i_t) is associated to the probability $w^i_t / \sum_{i=1}^N w^i_t$.

B.2 Static Analysis

The full type system is given in Figures 18 and 19. The interesting cases are presented in Section 6.1.

$$\frac{}{C \vdash^c c} \qquad \frac{x \in C}{C \vdash^c x} \qquad \frac{C \vdash^c e_1 \qquad C \vdash^c e_2}{C \vdash^c (e_1, e_2)} \qquad \frac{C \vdash^c e}{C \vdash^c op(e)}$$

$$\frac{C + dom(E) \vdash^c e \qquad C \vdash^c E : dom(E)}{C \vdash^c e \texttt{ where rec } E} \qquad \frac{}{C \vdash^c \texttt{init } x = e : \emptyset} \qquad \frac{}{C \vdash^c x = \texttt{last } x : \{x\}}$$

$$\frac{C \vdash^c e}{C \vdash^c x = e : \{x\}} \qquad \frac{C \nvdash^c e}{C \vdash^c x = e : \emptyset} \qquad \frac{C + C_2 \vdash^c E_1 : C_1 \qquad C + C_1 \vdash^c E_2 : C_2}{C \vdash^c E_1 \texttt{ and } E_2 : C_1 + C_2}$$

Fig. 19. Constant expressions and equations.

$$
\begin{array}{lcl}
C_\phi(c) & = & c \\
C_\phi(x) & = & x \\
C_\phi((e_1, e_2)) & = & (C_\phi(e_1), C_\phi(e_2)) \\
C_\phi(op(e)) & = & op(C_\phi(e)) \\
C_\phi(\texttt{last } x) & = & \texttt{last } x \\
C_\phi(\texttt{present } e_1 \ \rightarrow\ e_2 \texttt{ else } e_3) & = & \texttt{present } C_\phi(e_1) \ \rightarrow\ C_\phi(e_2) \texttt{ else } C_\phi(e_3) \\
C_\phi(\texttt{reset } e_1 \texttt{ every } e_2) & = & \texttt{reset } C_\phi(e_1) \texttt{ every } C_\phi(e_2) \\
C_\phi(\texttt{sample}(e)) & = & \texttt{sample}(C_\phi(e)) \\
C_\phi(\texttt{factor}(e)) & = & \texttt{factor}(C_\phi(e)) \\
C_\phi(e \texttt{ where rec } E) & = & C_\phi(e) \texttt{ where rec } C_\phi(E) \\
C_\phi(\texttt{init } x = e) & = & \begin{cases} \emptyset & \text{if } x \in dom(\phi) \\ \texttt{init } x = C_\phi(e) & \text{otherwise} \end{cases} \\
C_\phi(x = e) & = & \begin{cases} \emptyset & \text{if } x \in dom(\phi) \\ x = C_\phi(e) & \text{otherwise} \end{cases} \\
C_\phi(f_\theta(e)) & = & \begin{cases} f(C_\phi(e)) & \text{if } f \text{ is deterministic} \\ f\texttt{.model}(\theta, C_\phi(e)) & \text{if } \theta \in dom(\phi) \\ f\texttt{.model}(\theta, C_\phi(e)) \texttt{ where} & \text{otherwise} \\ \quad \texttt{rec init } \theta = \texttt{sample}(f\texttt{.prior}) & \\ \quad \texttt{and } \theta = \texttt{last } \theta & \end{cases} \\
C_\phi(\texttt{infer}(f(e))) & = & \texttt{APF.infer}(f\texttt{.model}, f\texttt{.prior}, C_\phi(e)) \\
C_\phi(E_1 \texttt{ and } E_2) & = & C_\phi(E_1) \texttt{ and } C_\phi(E_2) \\
C_\Phi(\texttt{let } x = e) & = & \texttt{let } x = e \\
C_\Phi(\texttt{node } f\ x = e) & = & \texttt{node } f\ x = C_\emptyset(e) \\
C_\Phi(\texttt{proba } f\ x = e) & = & \texttt{let } f\texttt{.prior} = im(\phi) \qquad \text{with } \phi = \Phi(f) \\
 & & \texttt{proba } f\texttt{.model } (dom(\phi), x) = C_\phi(e)
\end{array}
$$

Fig. 20. APF Compilation.

B.3 Compilation

The complete compilation function to transformation a ProbZelus model into a model compatible with `APF.infer` is given in Figure 20. Most cases simply call the compilation functions on all sub-expressions. The interesting cases are presented in Section 6.2.