Schedule Agnostic Semantics for Reactive Probabilistic Programming

GUILLAUME BAUDART, Université Paris Cité, Inria, CNRS, IRIF, France
LOUIS MANDEL, IBM Research, USA
CHRISTINE TASSON, ISAE-Supaero, Université de Toulouse, 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 a new operational semantics and a new denotational semantics for a probabilistic synchronous language that are both schedule agnostic. The key idea is to externalize the source of randomness and interpret a probabilistic expression as a stream of functions mapping random elements to a value and positive score. The operational semantics interprets expressions as state machines where mutually recursive equations are evaluated using a fixpoint operator. The denotational semantics directly manipulates streams and is thus a better fit to reason about program equivalence. We use the denotational semantics to prove the correctness of a program transformation required to run an optimized inference algorithm for state-space models with constant parameters.

CCS Concepts: • **Software and its engineering** → **Semantics**; **Data flow languages**; • **Mathematics of computing** → *Sequential Monte Carlo methods*; • **Theory of computation** → *Probabilistic computation.*

Additional Key Words and Phrases: Probabilistic programming languages, Dataflow synchronous languages, Probabilistic stream semantics, Assumed Parameter Filter

ACM Reference Format:
Guillaume Baudart, Louis Mandel, and Christine Tasson. 2025. Schedule Agnostic Semantics for Reactive Probabilistic Programming. 1, 1 (May 2025), 44 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Synchronous programming languages [7] were introduced for the design of critical embedded systems. In dataflow languages such as Lustre [39], 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 [40]. Built on these ideas, Scade is now a standard tool in automotive and avionic industries to program safety critical embedded software [20]. The synchronous model of computation is also central for the discrete-time subset of Matlab/Simulink [43].

Authors' Contact Information: Guillaume Baudart, Université Paris Cité, Inria, CNRS, IRIF, France; Louis Mandel, IBM Research, USA; Christine Tasson, ISAE-Supaero, Université de Toulouse, France.

Manuscript submitted to ACM

Probabilistic languages extend general purpose programming languages with probabilistic constructs for Bayesian reasoning [8, 25, 36, 37]. 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 [5] is a probabilistic extension of the synchronous dataflow language Zelus [16]. 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) [49] 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) [19]. Any ProbZelus program can be expressed as a SSM.

Probabilistic semantics and scheduling. In the original ProbZelus semantics expressions are interpreted as state machines [5]. Following [52], 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 data-flow languages where programs are written using a block diagram graphical interface. Scheduling should not depend on the placement of the blocks, this motivates 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 [16]. 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 [11, 18] for probabilistic programming. We first define a new operational semantics where sets of equations in arbitrary order can be interpreted with a fixpoint operator. The key idea of our approach is to interpret probabilistic expressions as a stream of functions mapping random elements in $[0, 1]$ to a value and positive score. Like its deterministic counterpart [18], this operational 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 [50]. We introduce a denotational semantics which abstracts away the state machines and directly manipulates streams which simplifies reasoning about program equivalence. This semantics extends the semantics used in the Vélus project to prove an end-to-end compiler for the synchronous language Lustre [11–13].

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.

```
1  proba motion(f) = x where
2    rec init x = xo and init v = vo
3    and v = sample(gaussian(last v +. f *. h, sv))
4    and x = sample(gaussian(last x +. last v *. h, sx))

6  proba tracker(y_obs) = x where
7    rec x = motion(f)
8    and f = current_force(x)
9    and () = observe(gaussian(g(x), sy), y_obs)

11 node main(y_obs) = u where
12   rec x_dist = infer (tracker (y_obs))
13   and u = controller(x_dist)
```

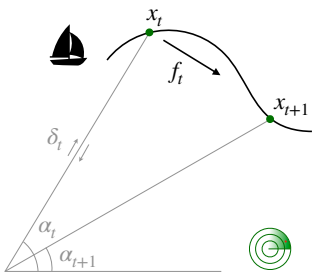


Fig. 1. Tracking a drifting boat with a marine radar in ProbZelus. x_t is the position of the boat in Cartesian coordinates at time t. The speed of the boat v_t is computed from the current force f_t which depends on the position of the boat. The observation $g(x_t) = (\alpha_t, \delta_t)$ comprises 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.

Inspired by the Assumed Parameter Filter algorithm [35], 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 our denotational 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 operational semantics and show that sets of mutually recursive probabilistic equations can be interpreted using a fixpoint operator.
- We introduce in Section 5 a new denotational semantics that abstracts away the state machines and directly manipulates streams which simplifies reasoning about program equivalence.
- We prove in Section 6 that these two semantics coincide, i.e., they both define the same streams, and that the operational semantics is equivalent to the original ProbZelus semantics for the subset of scheduled programs.
- We define in Section 7 a program transformation required to run an optimized inference algorithm for state-space models with constant parameters. We use our denotational semantics to prove the correctness of the transformation.

2 EXAMPLE

To motivate our approach, consider the ProbZelus program of Figure 1 adapted from [19][Section 2.4.1]. ProbZelus is a synchronous dataflow language à la Lustre [39] where programs execute in lock-step on a global logical clock. In this example, a discrete-time probabilistic model is used to track a drifting boat given noisy observations from a marine radar. At each instant, 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 of the boat from noisy measurements of its angle and of the echo delay. We assume that the strength of the current only depends on the position of the boat.

Probabilistic model. The keyword `proba` indicates the definition of a probabilistic stream function. The `motion` model (Line 1) takes as input the stream `f` of the current force and returns the stream of estimated positions using two noisy integrators to compute the speed `v` and position `x` of the boat. Line 3 uses `sample` to specify that `v` is Gaussian distributed around the (discrete) integral of the current force `f` where `h` is the integration step, and `last v` refers to the previous speed of the boat initialized Line 2 with the `init` keyword. Similarly, Line 4 specifies that `x` is Gaussian distributed around the integral of `v`.

The `tracker` model uses the motion model to estimate the position of the boat (Line 7) and computes the current force from the position using a deterministic function `current_force` (Line 8). The `observe` operator (Line 9) conditions the model assuming that the observations `y_obs` are Gaussian distributed around `g(x)`, where `g` estimates the observable quantities (e.g., angle and echo delay) from the estimated position. The integration step `h`, the initial values `xo` and `vo`, and the noise parameters `sx`, `sv` and `sy` are global constants.

Inference-in-the-loop. The tracker model can be used to program a deterministic controller. The keyword `node` indicates the definition of a deterministic stream function that takes as input the stream of radar measurements and returns a stream of commands. Line 12, the explicit `infer` operator computes the stream of distributions `x_dist` defined by the probabilistic model `tracker` conditioned on the observations `y_obs`. Line 13 is an example of *inference-in-the-loop* where the distribution `x_dist` of estimated positions is used to compute the command `u` (e.g., using the mean or the variance of the distribution) that propels the boat which then influences the next observation.[1]

Mutually recursive equations. The original ProbZelus semantics focuses on a kernel language where local declarations are all *scheduled*, i.e., ordered according to data dependencies: an equation $x_i = e_i$ must appear after $x_j = e_j$ if e_j uses x_i without a `last` operator [5, Section 3.1]. But imposing a valid schedule is a significant limitation compared to synchronous dataflow languages which manipulate mutually recursive equations in arbitrary order. In particular, some programs can only be scheduled after several compilation passes which introduce additional equations to relax the scheduling constraints. For instance, in the `tracker` model, `x` depends on `f` (Line 7), but `f` also depends on `x` (Line 8). To break this dependency cycle, the compiler must inline the definition of `motion` to compute `x`, then `f`, then `v`. Such programs cannot be interpreted with the original semantics. In this paper, we propose a new operational semantics, and a new denotational semantics that are both defined on programs before compilation and overcome this limitation.

2.1 Operational semantics

We first propose in Section 4 a new operational semantics for ProbZelus inspired by existing probabilistic semantics [10, 38, 45, 48] where the source of randomness is externalized. Following the original ProbZelus co-iterative semantics [5] expressions are interpreted as state machines characterized by an initial state and a transition function. The interpretation of expressions depends on the context: probabilistic (e.g., inside the models `motion` or `tracker`) or deterministic (e.g., inside the controller `main`). Given the current state, the transition function of deterministic expressions returns the next state and a value. The transition function of probabilistic expressions takes one additional argument — a random element for every random variable — and returns one additional output — a positive score, or weight, which measures the quality of the output w.r.t. the model.

On the `motion` model of Figure 1, the state of the transition function memorizes (p_v, p_x) the previous values of `v` and `x` (required to interpret the `last` operator).[2] The additional argument $[r_v, r_x]$ contains one random element for

[1] A more complex motion model could also use the value of the command.
[2] After simplifications, omitting empty states for stateless expressions, see Section 4.

each `sample` operator, i.e., an element r of the interval $[0, 1]$ that can be mapped to a sample[3] of a distribution d using inverse transform sampling [32]. The model computes the samples v and x associated to the random elements, then returns the new state (v, x) (i.e., the next previous values for v and x), the result x, and a weight 1 (there is no `observe`).

$$\begin{aligned}(\!|\texttt{motion}|\!)^{\text{step}}_{\gamma}\ (f, (p_v, p_x), [r_v, r_x]) \ =\ & \mathit{let}\ \mu_v = \mathcal{N}(p_v + f \times h, s_v)\ \mathit{in}\ \mathit{let}\ v = \mathit{icdf}_{\mu_v}(r_v)\ \mathit{in} \\ & \mathit{let}\ \mu_x = \mathcal{N}(p_x + p_v \times h, s_x)\ \mathit{in}\ \mathit{let}\ x = \mathit{icdf}_{\mu_x}(r_x)\ \mathit{in} \\ & (v, x), x, 1\end{aligned} \tag{1}$$

Mutually recursive equations. Following the original co-iterative semantics for dataflow synchronous languages [18] we interpret mutually recursive equations with a fixpoint operator in a flat complete partial order (CPO) where variables are either undefined or set to a value.

For instance, the semantics of the local declarations in `tracker` is the fixpoint of the following function F starting from the least element $[\texttt{x} \leftarrow \bot, \texttt{f} \leftarrow \bot]$. For simplicity, we assume that the semantics of `current_force` is a known stateless deterministic function c and the semantics of the observation function g is a stateless deterministic function g.

$$\begin{aligned}F(\rho) \ =\ & \mathit{let}\ (v, x), x, 1 = (\!|\texttt{motion}|\!)^{\text{step}}_{\gamma}\ (\rho(\texttt{f}), (p_v, p_x), [r_v, r_x])\ \mathit{in} \\ & \mathit{let}\ f = c(\rho(\texttt{x}))\ \mathit{in} \\ & \mathit{let}\ w_y = \mathit{pdf}_{\mathcal{N}(g(\rho(\texttt{x})), s_y)}(y_{\text{obs}})\ \mathit{in} \\ & [\texttt{x} \leftarrow x, \texttt{f} \leftarrow f]\end{aligned}$$

$$\begin{aligned}\rho_0 &= [\texttt{x} \leftarrow \bot, \texttt{f} \leftarrow \bot] \\ \rho_1 &= [\texttt{x} \leftarrow \mathit{icdf}_{\mu_x}(r_x), \texttt{f} \leftarrow \bot] \\ \rho_2 &= [\texttt{x} \leftarrow \mathit{icdf}_{\mu_x}(r_x), \texttt{f} \leftarrow c(\mathit{icdf}_{\mu_x}(r_x))] \\ \rho_3 &= [\texttt{x} \leftarrow \mathit{icdf}_{\mu_x}(r_x), \texttt{f} \leftarrow c(\mathit{icdf}_{\mu_x}(r_x))]\end{aligned}$$

The fixpoint converges after 3 iterations. Since x only depends on the state (p_v, p_x) in Equation (1), the first iteration returns a value for x even if f is $\bot$. The second iteration can then use this value to compute f. Using the resulting environment, the semantics of `tracker` computes the next state, the result value, and the weight capturing the likelihood of the observation w.r.t. the estimation (the density of the distribution $\mathcal{N}(g(x), s_y)$ at y_{obs}).

$$\begin{aligned}(\!|\texttt{tracker(y\_obs)}|\!)^{\text{step}}_{\gamma}\ ((p_v, p_x), [r_v, r_x]) \ =\ & \mathit{let}\ \rho = [\texttt{x} \leftarrow \mathit{icdf}_{\mu_x}(r_x), \texttt{f} \leftarrow c(\mathit{icdf}_{\mu_x}(r_x))]\ \mathit{in} \\ & \mathit{let}\ (v, x), x, 1 = (\!|\texttt{motion}|\!)^{\text{step}}_{\gamma}\ (\rho(\texttt{f}), (p_v, p_x), [r_v, r_x])\ \mathit{in} \\ & \mathit{let}\ f = c(\rho(\texttt{x}))\ \mathit{in} \\ & \mathit{let}\ w_y = \mathit{pdf}_{\mathcal{N}(g(\rho(\texttt{x})), s_y)}(y_{\text{obs}})\ \mathit{in} \\ & (v, x), x, w_y \\ \ =\ & \mathit{let}\ x = \mathit{icdf}_{\mathcal{N}(p_x + p_v \times h, s_x)}(r_x)\ \mathit{in} \\ & \mathit{let}\ v = \mathit{icdf}_{\mathcal{N}(p_v + c(x) \times h, s_v)}(r_v)\ \mathit{in} \\ & (v, x), x, \mathit{pdf}_{\mathcal{N}(g(x), s_y)}(y_{\text{obs}})\end{aligned} \tag{2}$$

The semantics of a sets of equations is defined via a fixpoint operator even when the solution is obvious like in the `motion` model. If equations are scheduled, the fixpoint yields the same environment as a sequence of local declarations.

Unnormalized measure. We obtain an unnormalized measure over pairs (next state, value) by integrating over all possible random elements, i.e., for a model e, if $(\!|e|\!)^{\text{step}}_{\gamma}\ (m, r) = m', v, w$ we define $\psi(m) = \int_{[0,1]^p} w \times \delta_{(m', v)}\ dr$ where p is the number of random variables in e and $\delta_{(m', v)}$ is the Dirac distribution on the pair of state and value (m', v). On

[3] icdf_d is the generalized inverse cumulative distribution function of d

the example of Figure 1, we get:

$$\begin{aligned}
\psi(p_v, p_x) &= \int_{[0,1]} \int_{[0,1]} let\ (v,x), x, w = (\!|\texttt{tracker(y_obs)}|\!)^{\text{step}}_{\gamma}\ ((p_v,p_x),[r_v,r_x])\ in\ w \times \delta_{(v,x),x}\ dr_x\ dr_v \\
&= \int_{[0,1]} let\ x = icdf_{\mathcal{N}(p_x+p_v\times h, s_x)}(r_x)\ in \\
&\qquad \int_{[0,1]} let\ v = icdf_{\mathcal{N}(p_v+c(x)\times h, s_v)}(r_v)\ in \\
&\qquad\qquad pdf_{\mathcal{N}(g(x),s_y)}(y_{\text{obs}}) \times \delta_{(v,x),x}\ dr_x\ dr_v \\
&= \int \mathcal{N}(p_x + p_v \times h, s_x)(dx) \int \mathcal{N}(p_v + c(x) \times h, s_v)(dv)\ pdf_{\mathcal{N}(g(x),s_y)}(y_{\text{obs}}) \times \delta_{(v,x),x}
\end{aligned}$$

Inference. For a probabilistic model e, `infer(e)` is a deterministic expression whose state is a measure over possible states for e. The transition function 1) integrates the unnormalized measure defined by the model over all possible states, 2) normalizes the measure, and 3) 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. We prove in Section 5.3 that this semantics coincide with the original ProbZelus semantics for the subset of scheduled programs, i.e., they both define the same streams.

Program equivalence. Since deterministic expressions are interpreted as state machines, to prove program equivalence we exhibit a bisimulation [50], 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.

Two probabilistic expressions are equivalent if they describe the same stream of measures of output values obtained by integrating, at each step, the transition functions with respect to the measure over all possible states computed at the previous step. We show in Section 4.4 that we can prove the equivalence of probabilistic expressions with a bisimulation between states, and a *coupling* between the random elements.

2.2 Denotational semantics

The denotational semantics directly manipulates streams of values. This approach has been successfully used in the Vélus project[4] to prove an end-to-end compiler for the dataflow synchronous language Lustre [11–13, 15]. The semantics of a node is defined as a function between input streams and output streams. In Vélus, most of the compilation passes are proven correct using this denotational 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 semantics to probabilistic models in Section 5. Given a context H mapping variable names to streams of values, and an array R of stream of random elements, the semantics of an expression returns a stream of pairs (value, weight): $H, R \vdash e \Downarrow (v_0, w_0) \cdot (v_1, w_1) \cdot (v_2, w_2) \cdot \ldots$ (the operator $\cdot$ represents the concatenation of stream elements).

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

We can interpret `tracker` of Figure 1 in a context $H = [\texttt{y\_obs} \leftarrow y_{\text{obs}}]$ that contains the stream of observations y_{obs} and in the random context given by the array $R = [R_v, R_x]$ of two streams of independent random elements in $[0, 1]$.

$$
\begin{array}{rcccccc}
[\texttt{y\_obs} \leftarrow y_{\text{obs}}], [R_v, R_x] \vdash \texttt{tracker(y\_obs)} \Downarrow & (x_0, w_0) & \cdot & (x_1, w_1) & \cdot & (x_2, w_2) & \cdot \ldots \\
\textit{where } \mu_x = & \mathcal{N}(x_{\text{init}} + v_{\text{init}} \times h, s_x) & \cdot & \mathcal{N}(x_0 + v_0 \times h, s_x) & \cdot & \mathcal{N}(x_1 + v_1 \times h, s_x) & \cdot \ldots \\
x = & icdf_{\mu_{x0}}(R_{x0}) & \cdot & icdf_{\mu_{x1}}(R_{x1}) & \cdot & icdf_{\mu_{x2}}(R_{x2}) & \cdot \ldots \\
\mu_v = & \mathcal{N}(v_{\text{init}} + c(x_0) \times h, s_x) & \cdot & \mathcal{N}(v_0 + c(x_1) \times h, s_x) & \cdot & \mathcal{N}(v_1 + c(x_2) \times h, s_x) & \cdot \ldots \\
v = & icdf_{\mu_{v0}}(R_{v0}) & \cdot & icdf_{\mu_{v1}}(R_{v1}) & \cdot & icdf_{\mu_{v2}}(R_{v2}) & \cdot \ldots \\
\mu_y = & \mathcal{N}(g(x_0), s_x) & \cdot & \mathcal{N}(g(x_1), s_x) & \cdot & \mathcal{N}(g(x_2), s_x) & \cdot \ldots \\
w_y = & pdf_{\mu_{y_0}}(y_{\text{obs}_0}) & \cdot & pdf_{\mu_{y_1}}(y_{\text{obs}_1}) & \cdot & pdf_{\mu_{y_2}}(y_{\text{obs}_2}) & \cdot \ldots
\end{array}
$$

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

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. However, compared to the state machines of the operational semantics, the denotational semantics is not executable since equations are only checked a posteriori for a given context.

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 by integrating over all possible values of the stream of random elements, and 3) normalizes this measure to obtain a distribution of values. The key difference with the operational semantics is that the integral is now over the infinite domain of streams. We prove in Section 5.2 that this semantics is equivalent to the operational semantics, i.e., the `infer` operator yields the same stream of distributions.

Program equivalence. In the denotational semantics, states are abstracted away. Two deterministic expressions are equivalent if they define the same streams of output values. 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 over output values, obtained by integrating at each step the values and the weights over all possible random streams. We show in Section 5.4 that we can prove the equivalence of two probabilistic expressions with a coupling over the random streams.

2.3 Comparison

The following table summarizes the comparison between the original semantics and the two new semantics. The original ProbZelus semantics is similar to the new operational semantics but probabilistic state machines directly computes a stream of unnormalized measures. Compared to the original semantics, the new semantics are not limited to sets of scheduled equations. In the example of Figure 1 switching the order of equations `x` and `f` in `tracker` does not change the fixpoint. The new semantics are also modular and can interpret function call without inlining the body of

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

Fig. 2. ProbZelus Syntax.

the callee as illustrated by the example of Figure 1. These semantics can even interpret programs that are not statically schedulable, e.g., programs where variable dependencies can change dynamically [22].

	original	operational	denotational
Executable	✓	✓	✗
Schedule agnostic	✗	✓	✓
Modularity	✗	✓	✓
Mutual recursion	✗	✓	✓
Randomness	internal	external	external
State	explicit	explicit	implicit

3 LANGUAGE

In this section we recall the syntax and typing rules of ProbZelus [5]. Advanced constructs, e.g., hierarchical automata, can be reduced to this language. The type system statically identifies deterministic and probabilistic expressions which have different interpretations in the operational semantics of Section 4 and the denotational semantics of Section 5. Recursion, loops, and nested inference are not allowed in the language [1].

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` $e\ \rightarrow\ e_1$ `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.

Example. The following node implements a fix-step integrator. The table presents the first few steps of a possible timeline with `h = 0.1`.

```
node integr(x) = i where
  rec init i = 0. and i = (last i) +. x *. h
```

x	0	1	2	3	4	5	...
i	0.0	0.1	0.3	0.6	1.0	1.5	...

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 the distribution d [52].

Example. The following model estimates the bias of a coin from a stream of coin tosses. Initially the model makes no assumption, i.e., the bias p is uniformly distributed over $[0, 1]$ (0.5 corresponds to a perfect coin, 1 is a coin that always returns head). At each time step, we condition the model assuming that the observation o is sampled from a Bernoulli distribution with the same bias p. The deterministic node `main` then computes the mean of the posterior distributions of the bias at each time step.

```
proba coin (o) = p where
  rec init p = sample(uniform(0., 1.))
  and p = last p and () = observe(bernoulli(p), o)

node main(o) = m where
  rec p_dist = infer(coin(o)) and m = mean(p_dist)
```

o	1	0	0	0	0	...
m	0.68	0.51	0.34	0.29	0.25	...

3.2 Typing

Figure 3 – adapted from [5, Figure 12] – presents the ProbZelus type system where each expression is associated to one of two *kinds*: `D` for deterministic, or `P` for probabilistic. This type system guaranties that probabilistic constructs `sample` and `factor` are only used inside `infer`.

The typing judgment $G \vdash^k e : T$ asserts that, in a typing environment G mapping variables names to their types, the expression e has type T and kind k. Function type are annotated with the kind k of their body. The sub-typing rule can be used to lift a deterministic expression to a probabilistic expression. Deterministic expressions, e.g., `x + 1`, can thus be used inside a probabilistic model (except `infer(e)` to prevent nested inference).

The type T `dist` represents distributions over values of type T that can be sampled (`sample` operator). To simplify the presentation of the semantics, we assume that the arguments of the probabilistic operator `sample` and `factor` are deterministic. This is not a limitation since the compiler can always introduce intermediate variables to capture a probabilistic argument, e.g., `() = factor(sample(d))` can be written `x = sample(d) and () = factor(x)`. The `infer` operator returns a deterministic value (a distribution) that can be used in a classic deterministic program (e.g., via standard functions like `mean` or `var`).

To type local declaration, equations are associated to the kind of their defining expression. The typing judgement $G \vdash^k E : G'$ asserts that given an initial environment G, the set of equations E produces a new environment G' with kind k. We require that all equations have the same kind, which may necessitate the use of the sub-typing rule for certain expressions.

4 OPERATIONAL SEMANTICS

In this section, we first recall the original deterministic semantics. We then define a new operational semantics for probabilistic expressions. In this semantics we interpret sets of mutually recursive equations with a fixpoint operator. Finally, we define the equivalence of probabilistic programs using probabilistic bisimulation and coupling.

$$\frac{typeOf(c) = t}{G \vdash^{\mathsf{D}} c : t} \qquad \frac{G(x) = t}{G \vdash^{\mathsf{D}} x : t} \qquad \frac{G(f) = t_1 \to^k t_2 \quad G \vdash^{\mathsf{D}} e : t_1}{G \vdash^k f(e) : t_2}$$

$$\frac{G \vdash^{\mathsf{D}} e : T \qquad e \neq \texttt{infer}(e')}{G \vdash^{\mathsf{P}} e : T} \qquad \frac{G \vdash^{\mathsf{D}} e : T\ \texttt{dist}}{G \vdash^{\mathsf{P}} \texttt{sample}(e) : T} \qquad \frac{G \vdash^{\mathsf{D}} e : \texttt{float}}{G \vdash^{\mathsf{P}} \texttt{factor}(e) : \texttt{unit}} \qquad \frac{G \vdash^{\mathsf{P}} e : T}{G \vdash^{\mathsf{D}} \texttt{infer}(e) : T\ \texttt{dist}}$$

$$\frac{G \vdash^k e : t}{G \vdash^k x = e : [x \leftarrow t]} \qquad \frac{G + G_1 + G_2 \vdash^k E_1 : G_1 \quad G + G_1 + G_2 \vdash^k E_2 : G_2}{G \vdash^k E_1 \texttt{ and } E_2 : G_1 + G_2} \qquad \frac{G \vdash^k E : G' \quad G + G' \vdash^k e : t}{G \vdash^k e \texttt{ where rec } E : t}$$

Fig. 3. Typing rules (adapted from [5, Figure 12])

4.1 Background: deterministic operational semantics

Following the co-iterative semantics of dataflow synchronous languages [18, 22], the deterministic semantics interprets expressions as state machines. The main advantage of the co-iterative semantics is that this 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 [22].

State machine. A deterministic expression e of type V is interpreted as a state machine characterized by an initial state $[\![e]\!]^{\text{init}} : S$ and a transition function $[\![e]\!]^{\text{step}} : \Gamma \to S \to S \times V$ which given the environment mapping variable names to values and the current state returns the next state and a value of type V. A stream of values is obtained by iteratively applying the transition function from the initial state.

$$([\![e]\!]^{\text{init}} = m_0) \xrightarrow{[\![e]\!]^{\text{step}}_{\gamma_0}} \underset{v_1}{m_1} \xrightarrow{[\![e]\!]^{\text{step}}_{\gamma_1}} \underset{v_2}{m_2} \xrightarrow{[\![e]\!]^{\text{step}}_{\gamma_2}} \underset{v_3}{m_3} \to \dots$$

Figure 4 shows a few semantics rules for deterministic expressions adapted from [22]. The state of an expression is defined inductively. The structure of the state matches the syntax of the expression and remains the same at each timestep. The semantics thus captures an aspect of the language that is essential for embedded systems: the memory required at runtime can be statically allocated before the initial step. The state of a variable is empty, and the transition function retrieves the value of the variable in the environment γ at each step. The state of a pair comprises the state of each sub-expression, and at each step the transition function executes the transition function of each sub-expression to compute the pair of next states and the pair of value. The initialization of a function call $f(e)$ combines the initial state of the argument e with the initial state of e_f the body of f. The transition function evaluates the argument e into a value v_e, uses this value to evaluate the transition function of e_f in a fresh environment where the parameter x is bound to v_e which returns a result v, and returns the combined next states and the result.

Figure 5 shows the semantics rules for deterministic equations adapted from [22]. The state of an equation is also defined inductively. The equation $x = e$ defines a single variable. The transition function evaluates the defining expression e into a pair (next state, value), and returns the next state and an environment where x is bound to the value. 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

$$
\begin{array}{lcl}
[\![e : V]\!]^{\text{init}} & : & S \\
[\![e : V]\!]^{\text{step}} & : & \Gamma \to S \times V
\end{array}
$$

$$
\begin{array}{lcl}
[\![c]\!]^{\text{init}} & = & () \\
[\![c]\!]^{\text{step}}_{\gamma}\,(()) & = & (), c
\end{array}
$$

$$
\begin{array}{lcl}
[\![x]\!]^{\text{init}} & = & () \\
[\![x]\!]^{\text{step}}_{\gamma}\,(()) & = & (), \gamma(x)
\end{array}
$$

$$
\begin{array}{lcl}
[\![\texttt{last}\ x]\!]^{\text{init}} & = & () \\
[\![\texttt{last}\ x]\!]^{\text{step}}_{\gamma}\,(()) & = & (), \gamma(x.\texttt{last})
\end{array}
$$

$$
\begin{array}{lcl}
[\![(e_1, e_1)]\!]^{\text{init}} & = & [\![e_1]\!]^{\text{init}}, [\![e_2]\!]^{\text{init}} \\
[\![(e_1, e_1)]\!]^{\text{step}}_{\gamma}\,(m_1, m_2) & = & \mathit{let}\ m_1', v_1 = [\![e_1]\!]^{\text{step}}_{\gamma}\,(m_1)\ \mathit{in} \\
 & & \mathit{let}\ m_2', v_2 = [\![e_2]\!]^{\text{step}}_{\gamma}\,(m_2)\ \mathit{in} \\
 & & (m_1', m_2'), (v_1, v_2)
\end{array}
$$

$$
\begin{array}{lcll}
[\![f(e)]\!]^{\text{init}} & = & [\![e]\!]^{\text{init}}, [\![e_f]\!]^{\text{init}} & \text{where node } f\ x = e_f \\
[\![f(e)]\!]^{\text{step}}_{\gamma}\,(m_e, m_f) & = & \mathit{let}\ m_e', v_e = [\![e]\!]^{\text{step}}_{\gamma}\,(m_e)\ \mathit{in} & \\
 & & \mathit{let}\ m_f', v = [\![e_f]\!]^{\text{step}}_{[x \leftarrow v_e]}\,(m_f)\ \mathit{in} & \\
 & & (m_e', m_f'), v &
\end{array}
$$

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

Fig. 4. Operational semantics of deterministic expressions (adapted from [22]).

transition function evaluates e using m_0 to compute the initial value i and returns a new state containing the current value of x and an environment where $x.\texttt{last}$ is bound to i. 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 and an environment where $x.\texttt{last}$ is bound to v. 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 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.

Mutual recursions. The original co-iterative semantics [18] and recent works [22] 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

$$
\begin{array}{lcl}
[\![E]\!]^{\text{init}} & : & S \\
[\![E]\!]^{\text{step}} & : & \Gamma \to S \times \Gamma
\end{array}
$$

$$
\begin{array}{lcl}
[\![x = e]\!]^{\text{init}} & = & [\![e]\!]^{\text{init}} \\
[\![x = e]\!]^{\text{step}}_{\gamma}\,(m) & = & \mathit{let}\ m', v = [\![e]\!]^{\text{step}}_{\gamma}\,(m)\ \mathit{in}\ m', [x \leftarrow v] \\
 & & \\
[\![\texttt{init}\ x = e]\!]^{\text{init}} & = & \mathit{nil}, [\![e]\!]^{\text{init}} \\
[\![\texttt{init}\ x = e]\!]^{\text{step}}_{\gamma}\,(\mathit{nil}, m_0) & = & \mathit{let}\ m', i = [\![e]\!]^{\text{step}}_{\gamma}\,(m_0)\ \mathit{in}\ (\gamma(x), m_0), [x.\texttt{last} \leftarrow i] \\
[\![\texttt{init}\ x = e]\!]^{\text{step}}_{\gamma}\,(v, m_0) & = & (\gamma(x), m_0), [x.\texttt{last} \leftarrow v] \\
 & & \\
[\![E_1\ \texttt{and}\ E_2]\!]^{\text{init}} & = & [\![E_1]\!]^{\text{init}}, [\![E_2]\!]^{\text{init}} \\
[\![E_1\ \texttt{and}\ E_2]\!]^{\text{step}}_{\gamma}\,(M_1, M_2) & = & \mathit{let}\ M_1', \gamma_1 = [\![E_1]\!]^{\text{step}}_{\gamma}\,(M_1)\ \mathit{in} \\
 & & \mathit{let}\ M_2', \gamma_2 = [\![E_2]\!]^{\text{step}}_{\gamma}\,(M_2)\ \mathit{in} \\
 & & (M_1', M_2'), \gamma_1 + \gamma_2
\end{array}
$$

Fig. 5. Operational semantics of deterministic equations (adapted from [22]).

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 $\bot = [x \leftarrow \bot]_{x \in X}$.

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 $\gamma + \rho$.

Causality. For some programs, the fixpoint operator never converges. For instance, `x where rec x = x` returns $\bot$ at every timestep. In the compiler, a static analysis ensures that dependency cycles between variables are always broken by a delay (`last` operator) [24]. This condition ensures that a valid schedule exists (after several compilation passes, e.g., inlining). If such a valid schedule exists, by monotonicity, each fixpoint iteration computes the value of at least one variable, the fixpoint is thus reached after a finite number of iterations [22].

Definition 4.1. A program is *causal* if all dependency cycles between variables are broken by a delay (`last` operator).

Compared to the original ProbZelus semantics, Definition 4.1 does not require that a valid schedule exists for all sets of mutually recursive equations. For instance, the program of Figure 1 is causal because the cycle `x < f < v < f` is broken by the delay `last v` in the definition of `x` (Section 2).

Example. Consider the `integr` node defined in Section 3.1. For an input expression e with initial state m_e, the initial state of the expression `integr(`e`)` is:

$$
m = \underbrace{m_e}_{\texttt{e}}, (\underbrace{()}_{\texttt{i}}, (\underbrace{(\mathit{nil}, \underbrace{()}_{\texttt{0}})}_{\texttt{last i}}, \underbrace{((), (\underbrace{()}_{\texttt{x}}, \underbrace{()}_{\texttt{h}}))}_{\texttt{integr.init}}))
$$

$$
\begin{array}{lcll}
|\, e : V \,| & : & \mathbb{N} & \\
|\, e \,| & = & 0 & \text{if } e \text{ is deterministic} \\
|\, \texttt{sample}(e) \,| & = & 1 & \\
|\, \texttt{factor}(e) \,| & = & 0 & \\
|\, e \texttt{ where rec } E \,| & = & |\, e \,| + |\, E \,| &
\end{array}
\qquad
\begin{array}{lcl}
|\, E \,| & : & \mathbb{N} \\
|\, x = e \,| & = & |\, e \,| \\
|\, \texttt{init } x = e \,| & = & |\, e \,| \\
|\, E_1 \texttt{ and } E_2 \,| & = & |\, E_1 \,| + |\, E_2 \,|
\end{array}
$$

Fig. 6. Dimension of probabilistic expressions and equations (full version in Figure 16 of the appendix).

During execution, the state at timestep t comprises the state of the input expression m_e and the previous value of `i`. The following table shows the first steps of the timeline.

m_e	m_{e0}	m_{e1}	m_{e2}	…
e	0.0	1.0	2.0	…
m_{integr}	$m_{e0}, ((), (0.0, ())), ((), ((), ())))$	$m_{e1}, ((), ((0.1, ())), ((), ((), ()))))$	$m_{e2}, ((), ((0.3, ())), ((), ((), ()))))$	…
i	0.0	0.1	0.3	…

4.2 Probabilistic operational semantics

The key idea of the probabilistic operational 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.

Model dimension. The dimension of a probabilistic model is defined as the number of random variables in the model, i.e., the number of calls to the `sample` operator. Since loops and recursive calls are not allowed in the language of Figure 2, the dimension of a model e can be statically computed with the function $|\, e \,|$ defined in Figure 6.

Probabilistic state machine. The initialization function of a probabilistic expression e of type V, $(\!|e|\!)^{\text{init}} : S$ returns the initial state. Given an environment, the current state, and a value for all random elements (an array of $|\, e \,|$ values in $[0, 1]$) the transition function $(\!|e|\!)^{\text{step}} : \Gamma \to S \times [0, 1]\ \texttt{array} \to S \times V \times [0, \infty)$ returns a triple (next state, value, weight).

An excerpt of the probabilistic operational semantics is presented in Figure 7. 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. The transition function of `sample(`e`)` takes an array containing one random element, evaluates the argument e into a distribution μ, converts the random element into a sample of the distribution using inverse transform sampling $icdf_\mu(r)$, and returns the next state, the sample, and a weight of 1. The transition function of `factor(`e`)` evaluates its arguments e into a real value v, and returns the next state, an empty value $()$, and the weight v. To simplify the semantics, the type system of Section 3.2 ensures that the arguments of the probabilistic operators are always deterministic expressions. These arguments can thus be interpreted with the deterministic semantics of Section 4.1. The transition function of a function call f`(`e`)` is similar to the deterministic case with an extra inputs for the random elements[5] and an extra output for the weight.

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

$$
\begin{array}{lcl}
(\!|e : V|\!)^{\text{init}} & : & S \\
(\!|e : V|\!)^{\text{step}} & : & \Gamma \to S \times [0,1]\ \texttt{array} \to S \times V \times [0,\infty)
\end{array}
$$

$$
\begin{array}{lcl}
(\!|e|\!)^{\text{init}} & = & [\![e]\!]^{\text{init}} \qquad \text{if } e \text{ is deterministic} \\
(\!|e|\!)^{\text{step}}_\gamma\,(m,[]) & = & \mathit{let}\ m', v = [\![e]\!]^{\text{step}}_\gamma\,(m)\ \mathit{in}\ m', v, 1 \\
\\
(\!|\texttt{sample}(e)|\!)^{\text{init}} & = & [\![e]\!]^{\text{init}} \\
(\!|\texttt{sample}(e)|\!)^{\text{step}}_\gamma\,(m,[r]) & = & \mathit{let}\ m', \mu = [\![e]\!]^{\text{step}}_\gamma\,(m)\ \mathit{in}\ m', \mathit{icdf}_\mu(r), 1 \\
\\
(\!|\texttt{factor}(e)|\!)^{\text{init}} & = & [\![e]\!]^{\text{init}} \\
(\!|\texttt{factor}(e)|\!)^{\text{step}}_\gamma\,(m,[]) & = & \mathit{let}\ m', v = [\![e]\!]^{\text{step}}_\gamma\,(m)\ \mathit{in}\ m', (), v \\
\\
(\!|f(e)|\!)^{\text{init}} & = & (\!|e|\!)^{\text{init}}, (\!|e_f|\!)^{\text{init}} \qquad \text{where } \texttt{proba}\ f\ x = e_f \\
(\!|f(e)|\!)^{\text{step}}_\gamma\,((m_e, m_f),[r_e : r_f]) & = & \mathit{let}\ m'_e, v_e, w_e = (\!|e|\!)^{\text{step}}_\gamma\,(m_e, r_e)\ \mathit{in} \\
 & & \mathit{let}\ m'_f, v, w_f = (\!|e_f|\!)_{[x \leftarrow v_e]}\,(m_f, r_f)\ \mathit{in} \\
 & & (m'_f, m'_e), v, w_e \times w_f \\
\\
(\!|e\ \texttt{where rec}\ E|\!)^{\text{init}} & = & (\!|e|\!)^{\text{init}}, (\!|E|\!)^{\text{init}} \\
(\!|e\ \texttt{where rec}\ E|\!)^{\text{step}}_\gamma\,((m,M),[r_e : r_E]) & = & \mathit{let}\ F(\rho) = \left(\mathit{let}\ M', \rho, w = (\!|E|\!)_{\gamma+\rho}\,(M, r_E)\ \mathit{in}\ \rho\right)\ \mathit{in} \\
 & & \mathit{let}\ \rho = \mathit{fix}(F)\ \mathit{in} \\
 & & \mathit{let}\ M', \rho, W = (\!|E|\!)^{\text{step}}_{\gamma+\rho}\,(M, r_E)\ \mathit{in} \\
 & & \mathit{let}\ m', v, w = (\!|e|\!)^{\text{step}}_{\gamma+\rho}\,(m, r_e)\ \mathit{in} \\
 & & (m', M'), v, w \times W
\end{array}
$$

Fig. 7. Probabilistic operational semantics of expressions (full version in Figure 17 of the appendix).

The semantics of probabilistic equations is presented in Figure 8. Given an environment, a state, and an array of random elements, the transition function $(\!|E|\!)^{\text{step}} : \Gamma \to S \times [0,1]\ \texttt{array} \to S \times \Gamma \times [0,\infty)$ returns a tuple (next state, environment, weight). The only difference with the deterministic case is that the transition functions now take the random elements for the sub-expressions as arguments and accumulates the weights.

Example. Consider the `coin` node defined in Section 3.1. The number of random variables is 1 because there is only one parameter: p. For an input expression e with initial state m_e, the initial state of the expression `coin(e)` is:

$$
m = \overbrace{m_e}^{\texttt{e}}, (\overbrace{()}^{\texttt{p}}, ((\mathit{nil}, \overbrace{((), ())}^{\texttt{0}}), \overbrace{()}^{\texttt{1}}, \overbrace{((), ())}^{\texttt{last p}}))
$$

4.3 Inference

Notations. In the following, $\mathit{Meas}(U)$ is the set of measures on a measurable space U, δ_x is the Dirac delta measure ($\delta_x(U) = 1$ if $x \in U$, 0 otherwise), we note $\int_U \mu(dx)\ g(x)$ the integral of a measurable function g w.r.t. the measure μ.

$$
\begin{array}{lcl}
(\!|E|\!)^{\text{init}} & : & S \\
(\!|E|\!)^{\text{step}} & : & \Gamma \to S \times [0,1]\ \texttt{array} \to S \times \Gamma \times [0,\infty)
\end{array}
$$

$$
\begin{array}{lcl}
(\!|x = e|\!)^{\text{init}} & = & (\!|e|\!)^{\text{init}} \\
(\!|x = e|\!)^{\text{step}}_{\gamma}\ (m, r) & = & \textit{let } m', v, w = (\!|e|\!)^{\text{step}}_{\gamma}\ (m, r) \textit{ in } m', [x \leftarrow v], w \\
\\
(\!|\texttt{init } x = e|\!)^{\text{init}} & = & \textit{nil}, (\!|e|\!)^{\text{init}} \\
(\!|\texttt{init } x = e|\!)^{\text{step}}_{\gamma}\ ((\textit{nil}, m_0), r) & = & \textit{let } m', i, w = (\!|e|\!)^{\text{step}}_{\gamma}\ (m_0, r) \textit{ in } (\gamma(x), m_0), [x.\texttt{last} \leftarrow i], w \\
(\!|\texttt{init } x = e|\!)^{\text{step}}_{\gamma}\ ((v, m_0), r) & = & (\gamma(x), m_0), [x.\texttt{last} \leftarrow v], 1 \\
\\
(\!|E_1 \texttt{ and } E_2|\!)^{\text{init}} & = & (\!|E_1|\!)^{\text{init}}, (\!|E_2|\!)^{\text{init}} \\
(\!|E_1 \texttt{ and } E_2|\!)^{\text{step}}_{\gamma}\ ((M_1, M_2), [r_1 : r_2]) & = & \textit{let } M_1', \rho_1, w_1 = (\!|E_1|\!)^{\text{step}}_{\gamma}\ (M_1, r_1) \textit{ in} \\
 & & \textit{let } M_2', \rho_2, w_2 = (\!|E_2|\!)^{\text{step}}_{\gamma}\ (M_2, r_2) \textit{ in} \\
 & & (M_1', M_2'), \rho_1 + \rho_2, w_1 \times w_2
\end{array}
$$

Fig. 8. Probabilistic operational semantics of equations.

The Lebesgues measure over $[0,1]^p$ is denoted λ^p and by abuse of notation, we note $\int_{[0,1]^p} g(x)dx$ the integral of g w.r.t to the measure λ^p. If $f : U \to V$ is a measurable function and $\mu \in \mathit{Meas}(V)$, then the pushforward of μ along f is the measure $f_*\mu \in \mathit{Meas}(U)$ defined as $f_*\mu = \mu \circ f^{-1}$. With these notations, the change of variable formula is: $\int g(y) f_*\mu(dy) = \int g(f(x))\mu(dx)$, for any measurable function $g : V \to W$.

Given a probabilistic model e of type V with a state of type S, the deterministic expression `infer(e)` computes a stream of distributions. At each time step, given a measure over possible states, the transition function returns a measure over possible next states σ, and a measure over return values μ.

$$
\begin{array}{lcl}
[\![\texttt{infer(e)}]\!]^{\text{init}} & : & \mathit{Meas}(S) \\
[\![\texttt{infer(e)}]\!]^{\text{step}} & : & \Gamma \to \mathit{Meas}(S) \to \mathit{Meas}(S) \times \mathit{Meas}(V)
\end{array}
$$

The initial state of `infer(e)` is a Dirac delta measure over the initial state of the model e. 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 1) integrates this function along all possible values of the state, 2) normalizes it ($\top$ denotes the entire space), and 3) splits the result into a distribution of next states σ and a distribution of values μ.

$$
\begin{array}{lcl}
[\![\texttt{infer(e)}]\!]^{\text{init}} & = & \delta_{(\!|e|\!)^{\text{init}}} \\
\\
[\![\texttt{infer(e)}]\!]^{\text{step}}_{\gamma}\ (\sigma) & = & \textit{let } \psi(m) = \displaystyle\int_{[0,1]^{|e|}} \textit{let } m', v, w = (\!|e|\!)^{\text{step}}_{\gamma}\ (m, r) \textit{ in } w \times \delta_{(m', v)}\ dr \textit{ in} \\
 & & \textit{let } \nu = \displaystyle\int \sigma(dm)\ \psi(m) \textit{ in} \\
 & & \textit{let } \overline{\nu} = \nu / \nu(\top) \textit{ in} \\
 & & \pi_{1*}(\overline{\nu}), \pi_{2*}(\overline{\nu})
\end{array}
\tag{3}
$$

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.

The stream of distributions corresponding to `infer`(e) is obtained by iteratively integrating the transition function along the distribution of states.

$$(\!(e)\!)^{\text{init}} = \delta_{m_0}) \xrightarrow{\int \sigma_0(dm)\psi_0(m)} \begin{matrix}\sigma_1\\ \mu_1\end{matrix} \xrightarrow{\int \sigma_1(dm)\psi_1(m)} \begin{matrix}\sigma_2\\ \mu_2\end{matrix} \xrightarrow{\int \sigma_2(dm)\psi_2(m)} \begin{matrix}\sigma_3\\ \mu_3\end{matrix} \to \dots$$

4.4 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 can exhibit a bisimulation [50] that relates the states and ensure the equality of output values.

Definition 4.2. Deterministic expressions are equivalent if there is a bisimulation, i.e., a relation on states $\mathcal{P} \subseteq S \times S$ such that:

- $(\llbracket e_1 \rrbracket^{\text{init}}, \llbracket e_2 \rrbracket^{\text{init}}) \in \mathcal{P}$
- $\forall \gamma$, if $(m_1, m_2) \in \mathcal{P}$, then $(m_1', m_2') \in \mathcal{P}$ and $v_1 = v_2$ where $m_1', v_1 = \llbracket e_1 \rrbracket_\gamma^{\text{step}}(m_1)$ and $m_2', v_2 = \llbracket e_2 \rrbracket_\gamma^{\text{step}}(m_2)$.

A bisimulation on states can be lifted to a bisimulation on measures of states to characterize equivalence between probabilistic processes [9, 47].

Definition 4.3. Let $\mathcal{P} \subseteq S \times S$ be a relation on states. The lifted relation $\mathcal{P}^+$ is the relation on measures of states defined as $(\sigma_1, \sigma_2) \in \mathcal{P}^+$, when for any measurable subset $\mathcal{P}' \subseteq \mathcal{P}$, $\sigma_1(\pi_1 \mathcal{P}') = \sigma_2(\pi_2 \mathcal{P}')$.

Two probabilistic expressions are equivalent if starting from bisimilar states, integrating at each steps the pairs (value, weight) computed by the probabilistic operational semantics, the output measures are equal and the measures on states are bisimilar.

Definition 4.4. Two probabilistic expressions e_1 and e_2 are equivalent if there is a bisimulation $\mathcal{P} \subseteq S \times S$ such that:

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

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

Using this definition, we can show that two probabilistic expressions are equivalent if they describe the same stream of measures, i.e., if at each step, integrating the transition function along the distributions of state in both expressions yields the same measures.

Proposition 4.5. *Let e_1 and e_2 be two equivalent expressions with probabilistic bisimulation $\mathcal{P}$. If $(\sigma_1, \sigma_2) \in \mathcal{P}^+$, then with the notations of Definition 4.4, $(\sigma_1'', \sigma_2'') \in \mathcal{P}^+$ and $\mu_1' = \mu_2'$ where*

$$\begin{aligned} \sigma_1'', \mu_1' &= \int \sigma_1(dm_1)\ \sigma_1'(m_1),\ \int \sigma_1(dm_1)\ \mu_1(m_1) \\ \sigma_2'', \mu_2' &= \int \sigma_2(dm_2)\ \sigma_2'(m_2),\ \int \sigma_2(dm_2)\ \mu_2(m_2) \end{aligned}$$

Proof. Let σ and σ' be the measures on $\mathcal{P}$ such that for any measurable $\mathcal{P}' \subseteq \mathcal{P}$, $\sigma(\mathcal{P}') = \sigma_1(\pi_1\mathcal{P}') = \sigma_2(\pi_2\mathcal{P}')$ and $\sigma'(m)(\mathcal{P}') = \sigma_1'(m_1)(\pi_1\mathcal{P}') = \sigma_2'(m_2)(\pi_2\mathcal{P}')$, where $m = (m_1, m_2)$. By the change of variable formula, for $i \in \{1, 2\}$,

$$\sigma_i''(\pi_i\mathcal{P}') = \int \pi_{i*}(\sigma)(dm_i)\ \sigma_i'(m_i)(\pi_i\mathcal{P}') = \int \sigma(dm)\ \sigma_i'(\pi_i(m))(\pi_i\mathcal{P}') = \int \sigma(dm)\ \sigma'(m)(\mathcal{P}')$$

Similarly, let $\mu(m) = \mu_1(m_1) = \mu_2(m_2)$. Then, for $i \in \{1, 2\}$,

$$\mu_i' = \int \pi_{i*}(\sigma)(dm_i)\ \mu_i(m_i) = \int \sigma(dm)\ \mu_i(\pi_i(m)) = \int \sigma(dm)\ \mu(m)$$

□

For a given context, each triplet (state, value, weight) is a function of the previous state and of the random elements. If we can relate the states of e_1 with the states of e_2 and the random elements of e_1 with the random elements of e_2, then program equivalence can be reduced to the comparison of the stream of pairs (value, weight). To relate two distributions, we use a *coupling*: a joint measure over the product space such that each marginal corresponds to one of the measure. We adapt the presentation of coupling for discrete probabilistic systems of [3] and follow the co-algebraic formulation of [29] to get a characterization of probabilistic equivalence by coupling for our operational semantics.

Proposition 4.6 (Operational coupling equivalence). *Probabilistic expressions e_1 and e_2 are equivalent if there is a pair of measurable relations $\mathcal{P} \subseteq S \times S$ and $\mathcal{R} \subseteq [0,1]^{|e_1|} \times [0,1]^{|e_2|}$ such that:*

- $\pi_1\mathcal{R} = [0,1]^{|e_1|}$ *and* $\pi_2\mathcal{R} = [0,1]^{|e_2|}$
- *There is a coupling Λ of the Lebesgue measures over $[0,1]^{|e_1|}$, and $[0,1]^{|e_2|}$:* $\Lambda(\mathcal{R}) = 1$, $\pi_{1*}\Lambda = \lambda^{|e_1|}$, *and* $\pi_{2*}\Lambda = \lambda^{|e_2|}$
- $(\!(e_1)\!)^{init}, (\!(e_2)\!)^{init}) \in \mathcal{P}$
- $\forall \gamma$, *if* $(m_1, m_2) \in \mathcal{P}$ *and* $(r_1, r_2) \in \mathcal{R}$, *then* $(m_1', m_2') \in \mathcal{P}$, $v_1 = v_2$ *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. The proof is adapted from [29, Lemma 5.5]. Assume $(m_1, m_2) \in \mathcal{P}$. Let us define a measure on $(S \times V) \times (S \times V)$ as $C = \int_{\mathcal{R}} w \times \delta_{(m_1', v),(m_2', v)}\ \Lambda(dr_1, dr_2)$, where $w = w_1 = w_2$ and $v = v_1 = v_2$. It is a coupling of $\psi_1(m_1)$ and $\psi_2(m_2)$. Indeed, for any $i \in \{1, 2\}$,

$$\pi_{i*}C = \int_{\mathcal{R}} w \times \delta_{(m_i', v)}\ \Lambda(dr_1, dr_2) = \int_{[0,1]^{|e_i|}} w_i \times \delta_{(m_i', v_i)}\ \lambda^{|e_i|}(dr_i) = \psi_i(m_i)$$

We prove that $\mu_1 = \mu_2$ by applying the change of variable formula along $\pi_i : [0,1]^{|e_1|} \times [0,1]^{|e_2|} \to [0,1]^{|e_i|}$,

$$\mu_i = \int_{[0,1]^{|e_i|}} w_i \times \delta_{v_i}\ \lambda^{|e_i|}(dr_i) = \int_{\pi_i\mathcal{R}} w \times \delta_v\ \pi_{i*}\Lambda(dr_i) = \int_{\mathcal{R}} w \times \delta_v\ \Lambda(dr_1, dr_2).$$

Let us prove that $(\sigma'_1, \sigma'_2) \in \mathcal{P}^+$. For any $\mathcal{P}' \subseteq \mathcal{P}$, let us set $Q = \{(m'_1, v), (m'_2, v) \mid (m'_1, m'_2) \in \mathcal{P}'\}$, we prove that $\sigma'_1(\pi_1\mathcal{P}') = \sigma'_2(\pi_2\mathcal{P}')$ by applying the definition of the pushforward measure along $\pi_i\pi_1$, noticing that $\pi_i\mathcal{P}' = \pi_1\pi_i Q$,

$$\sigma'_i(\pi_i\mathcal{P}') = \pi_{1*}\psi_i(m_i)(\pi_i\mathcal{P}') = \pi_{1*}\pi_{i*}C(\pi_1\pi_i Q) = \pi_{i*}C(\pi_i Q) = C(Q).$$

□

Finding a relation and a coupling 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 relation on random elements is a permutation, 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, and each expression has one random variable ($|\, e_1 \,| = |\, e_2 \,| = 1$):

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

We define the following:

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

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

$$(\!|e_\ell|\!)^{\text{init}} = ((\!|e_1|\!)^{\text{init}}, (\!|e_2|\!)^{\text{init}})$$
$$(\!|e_\ell|\!)^{\text{step}}_\gamma\,((m_1, m_2), [r_1 : r_2]) = (m'_1, m'_2), v_y + v_x, 1$$

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:

$$(\!|e_r|\!)^{\text{init}} = (((), ((), ())), ((\!|e_2|\!)^{\text{init}}, (\!|e_1|\!)^{\text{init}}))$$
$$(\!|e_r|\!)^{\text{step}}_\gamma\,((((), ((), ())), (m_2, m_1)), [r_2 : r_1]) = (((), ((), ())), (m'_2, m'_1)), v_x + v_y, 1$$

With $\mathcal{P} = \{((m_1, m_2),\ (((), ((), ())), (m_2, m_1))) \mid (m_1, m_2) \in S \times S\}$
and $\mathcal{R} = \{([r_1 : r_2], [r_2 : r_1]) \mid r_1 \in [0,1], r_2 \in [0,1]\}$ we have:

- $\pi_1\mathcal{R} = [0,1]^2$ and $\pi_2\mathcal{R} = [0,1]^2$
- the Lebesgues measure on $[0,1]^4$, $\Lambda = \lambda^4$ satisfies $\pi_{1*}\Lambda = \lambda^2$ and $\pi_{2*}\Lambda = \lambda^2$
- $\mathcal{P}$ relates initial states $((\!|e_1|\!)^{\text{init}}, (\!|e_2|\!)^{\text{init}})$ and $(((), ((), ())), ((\!|e_2|\!)^{\text{init}}, (\!|e_1|\!)^{\text{init}}))$
- if $\mathcal{P}$ relates the current states (m_1, m_2) and $((((), ((), ())), (m_2, m_1))$ and the random elements are permuted, then $\mathcal{P}$ relates the next states (m'_1, m'_2) and $(((), ((), ())), (m'_2, m'_1))$, 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 4.6 to conclude that the two probabilistic expressions are equivalent.

5 DENOTATIONAL SEMANTICS

An alternative to the operational semantics where expressions are interpreted as state machines is to define a denotational semantics where expressions directly return streams of values [21]. This formalism has been successfully used in the Vélus project to prove an end-to-end dataflow synchronous compiler within the Rocq proof assistant [11–13, 15].

In this section, we first present a denotational semantics for the deterministic expressions of our language. We then define a probabilistic denotational semantics for probabilistic expressions and prove that this semantics is equivalent to the probabilistic operational semantics of Section 4.2, i.e., the `infer` operator yields the same stream of distributions.

$$H \vdash c \downarrow c \qquad H \vdash x \downarrow H(x) \qquad \frac{H \vdash e_1 \downarrow s_1 \qquad H \vdash e_2 \downarrow s_2}{H \vdash (e_1, e_2) \downarrow (s_1, s_2)} \qquad \frac{H \vdash e \downarrow s}{H \vdash op(e) \downarrow op(s)} \qquad \frac{H(x.\texttt{last}) = s}{H \vdash \texttt{last}\ x \downarrow s}$$

$$\frac{H \vdash e \downarrow s_e \qquad \texttt{node}\ f\ x = e_f \qquad [x \leftarrow s_e] \vdash e_f \downarrow s}{H \vdash f(e) \downarrow s} \qquad \frac{H + H_E \vdash E \qquad H + H_E \vdash e \downarrow s}{H \vdash e\ \texttt{where rec}\ E \downarrow s}$$

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

Fig. 9. Denotational semantics for deterministic expressions and equations (full version in Figure 20 of the appendix).

5.1 Deterministic denotational semantics

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.

In the denotational semantics, deterministic expressions compute streams of values. In a context H which maps variable names to stream of values, the semantics of a deterministic expression e returns a stream s: $H \vdash e \downarrow s$. The semantics of a set of equations E checks that the context H is compatible with all the equations: $H \vdash E$. The semantics of a set of equations thus defines a *relation* between the streams stored in the context. Compared to the operational semantics, the denotational semantics is not executable since the context must be guessed a priori and validated against the equations of the program.

Figure 9 presents the denotational 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 denotational semantics, contexts are un-ordered maps and scheduling equations does not change the semantics.

Causality. For some programs, the denotational semantics is not uniquely defined. For instance, the semantics of the expression `x where rec x = x` can be any stream of value (the equation is verified for any context containing `x`). Such programs are rejected by the causality analysis of the compiler (Definition 4.1).

5.2 Probabilistic denotational semantics

The key idea of the probabilistic denotational semantics is the same as in Section 4.2: 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.

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

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

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

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

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

Fig. 10. Denotational semantics for probabilistic expressions and equations (full version in Figure 21 of the appendix).

Figure 10 presents the semantics of probabilistic expressions and equations. In a context H which maps variable 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): $H, R \vdash e \Downarrow (s, w)$. 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: $H, R \vdash E : w_E$.

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 R to compute a stream of samples associated to the 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 argument 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 a set of local definitions the random streams are split between sub-expressions and the weight is the total weight of all sub-expressions.

5.3 Inference

As in the operational semantics, the `infer` operator is defined by integrating at each step the semantics of the model 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: $\overline{w} = \Pi\ w = w_0 \cdot (w_0 \times w_1) \cdot (w_0 \times w_1 \times w_2) \cdot \ldots$ 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 $\lambda_\omega^{|e|}$ is the uniform measure over the cube of random streams $([0,1]^\omega)^{|e|}$, then:

$$integ_{|e|}\ (\overline{w}_n \cdot \overline{ws})\ (v \cdot vs) \quad = \quad \left(let\ \mu = \int \overline{w}_n \times \delta_v\ \lambda_\omega^{|e|}(dR)\ in\ \mu/\mu(\top)\right) \cdot (integ\ \overline{ws}\ vs) \tag{4}$$

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 $\pi_{\leq k} : [0,1]^\omega \to [0,1]^k$ on the first k coordinates is the Lebesgue measure on $[0,1]^k$: $\lambda_{\leq k} = \pi_{\leq k *}(\lambda_\omega)$. For any measurable function $g : [0,1]^k \to V$ 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 $\lambda_\omega^{|e|}$ as the $|e|$-ary product measure of λ_ω, and lift the change of variable formula.

5.4 Program equivalence

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

Definition 5.1. Deterministic expressions e_1 and e_2 are equivalent if for any context H, $H \vdash e_1 \downarrow s$ and $H \vdash e_2 \downarrow s$.

Equivalence between probabilistic expressions is more complex. 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 denotational semantics.

Definition 5.2. Probabilistic expressions e_1 and e_2 are equivalent if for all contexts H, and $\forall k > 0$,

$$\int \overline{w_1}_k \times \delta_{s_{1k}}\ \lambda_\omega^{|e_1|}(dR_1) = \int \overline{w_2}_k \times \delta_{s_{2k}}\ \lambda_\omega^{|e_2|}(dR_2)$$

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

In the denotational 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 relate the random streams of e_1 with the random streams of e_2 with a coupling on uniform distributions, program equivalence can be reduced to the comparison of the streams of pairs (value, weight) computed by each expression.

Proposition 5.3 (Denotational coupling equivalence). *Probabilistic expressions e_1 and e_2 are equivalent if there is $\mathcal{R} \subseteq ([0,1]^\omega)^{|e_1|} \times ([0,1]^\omega)^{|e_2|}$ measurable such that,*

- *$\pi_1\mathcal{R} = [0,1]^{|e_1|}$ and $\pi_2\mathcal{R} = [0,1]^{|e_2|}$*
- *there is a coupling Λ_ω on $([0,1]^\omega)^{|e_1|} \times ([0,1]^\omega)^{|e_2|}$, such that $\Lambda_\omega(\mathcal{R}) = 1$, $\pi_{1*}\Lambda_\omega = \lambda_\omega^{|e_1|}$ and $\pi_{2*}\Lambda_\omega = \lambda_\omega^{|e_2|}$*
- *for all contexts H and arrays of random streams $(R_1, R_2) \in \mathcal{R}$, $s_1 = s_2$ and $w_1 = w_2$, where $H, R_1 \vdash e_1 \Downarrow (s_1, w_1)$ and $H, R_2 \vdash e_2 \Downarrow (s_2, w_2)$.*

Proof. With $w_k = w_{1k} = w_{2k}$ and $s_k = s_{1k} = s_{2k}$, for all $k > 0$,

$$\int \overline{w_1}_k \times \delta_{s_{1k}}\ \lambda_\omega^{|e_1|}(dR_1) = \int \overline{w}_k \times \delta_{s_k}\ \Lambda_\omega(dR) = \int \overline{w_2}_k \times \delta_{s_{2k}}\ \lambda_\omega^{|e_2|}(dR_2)$$

□

Finding such a coupling is in general difficult. As in Section 4.4, a useful simple case is when the two programs involve the same random variables in different orders. Then, the coupling permutes the random streams.

The denotational 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 $H \vdash e_i \downarrow \mu_i$, then we define $s_i = icdf_{\mu_i}(R_i)$. Then, the derivation tree for the lhs expression is:

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

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

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

Since both programs compute the same stream of pairs (value, weight) and the relation permuting random streams $\mathcal{R} = \{([R_1 : R_2], [R_2 : R_1]) \mid R_1 \in [0,1]^\omega, R_2 \in [0,1]^\omega\}$ is the support of the coupling $\Lambda_\omega = \lambda_\omega^4$ of λ_ω^2 and λ_ω^2, the two programs are equivalent.

6 SEMANTICS COMPARISON

In this section we show that the operational and the denotational semantics coincide for causal programs, i.e., they both define the same streams. We then prove that these new semantics coincide with the original kernel semantics for the restricted set of schedulable programs that can be interpreted by the original semantics.

6.1 Adequacy between the operational and denotational semantics

For a probabilistic expression e, we first relate the operational semantics of Section 4.2 and the denotational semantics of Section 5.2. 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 (Definition 4.1), the probabilistic operational semantics and the probabilistic denotational semantics coincide.

Proposition 6.1. *For a correct model e (well-typed, causal, and without runtime error), the probabilistic denotational semantics is well defined and matches the trace of the probabilistic operational semantics, i.e., for all contexts H, R, $H, R \vdash e \Downarrow (s, w)$ is well defined and $\forall k \geq 0$, $(m_{k+1}, s_{k+1}, w_{k+1}) = (\!|e|\!)^{step}_{H_{k+1}}(m_k, R_{k+1})$ with $m_0 = (\!|e|\!)^{init}$. By construction, for all $k > 0$ the value s_k and the weight w_k only depend on $H_{\leq k}$ and $R_{\leq k}$.*

Proof Sketch. Probabilistic constructs aside, the kernel language of Figure 2 is a subset of the Vélus language defined in [11]. 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 10, the distribution of random streams in sub-expressions is performed by the compilation function. We can then apply the correctness theorem of [14, Theorem 5] : *if a program is correct and without runtime error, the denotational semantics is well defined (there is a valid environment for all equations sets) and matches the trace of the compiled state machine.*

The execution of the compiled state machine is deterministic and corresponds to the operational semantics of the normalized scheduled program which does not require any fixpoint computation. Since, the normalization and scheduling passes preserve the operational semantics [22], the execution of the state machine also corresponds to the operational semantics of the original unscheduled program. □

We can now state the adequacy theorem, i.e., the `infer` operator yields the same stream of distributions in the operational semantics and in the denotational semantics.

Theorem 6.2 (Adequacy). *For a correct model e (well-typed, causal, and without runtime errors), for all context H such that* $H \vdash \texttt{infer}(e) \downarrow \mu$*, the operational execution trace yields the same stream of distributions, that is,* $\sigma_0 = [\![\texttt{infer}(e)]\!]^{init}$ *and* $\forall k \geq 0, \sigma_{k+1}, \mu_{k+1} = [\![\texttt{infer}(e)]\!]^{step}_{H_{k+1}}(\sigma_k)$.

Proof. We show $\forall k > 0$:

$$\sigma_k \propto \int_{([0,1]^k)^{|e|}} \overline{w_k} \times \delta_{m_k}\ \lambda_{\leq k}^{|e|}(dR_{\leq k})$$
$$\mu_k \propto \int_{([0,1]^k)^{|e|}} \overline{w_k} \times \delta_{s_k}\ \lambda_{\leq k}^{|e|}(dR_{\leq k})$$

By Proposition 6.1, for all $k > 0$, the values s_k, and the weights w_k computed by the operational semantics corresponds to the stream (s, w) of the denotational semantics. From the definition of ψ in Equation (3) in Section 4.3 and Fubini's theorem we have:[6]

$$\begin{aligned}
\nu_{k+1} &= \int \sigma_k(dm)\ \psi(m) \\
&\propto \int_{([0,1]^k)^{|e|}} \overline{w_k} \times \psi(m_k)\ \lambda_{\leq k}^{|e|}(dR_{\leq k}) \\
&\propto \int_{([0,1]^k)^{|e|}} \int_{[0,1]^{|e|}} \overline{w_k} \times w_{k+1} \times \delta_{m_{k+1}, v_{k+1}}\ \lambda^{|e|}(dR_{k+1})\ \lambda_{\leq k}^{|e|}(dR_{\leq k}) \\
&\propto \int_{([0,1]^{k+1})^{|e|}} \overline{w_{k+1}} \times \delta_{m_{k+1}, v_{k+1}}\ \lambda_{\leq k+1}^{|e|}(dR_{\leq k+1})
\end{aligned}$$

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:

$$\mu_k \propto \int \overline{w_k} \times \delta_{s_k}\ \lambda_{\omega}^{|e|}(dR)$$

which corresponds to Equation (4) in Section 5.3 and concludes the proof. □

[6]The symbol $\propto$ means proportional to.

$$
\begin{array}{lcl}
\{\!|e:V|\!\}^{\text{init}} & : & S \\
\{\!|e:V|\!\}^{\text{step}} & : & \Gamma \to S \to \mathit{Meas}(S \times V)
\end{array}
$$

$$
\begin{array}{lcl}
\{\!|e|\!\}^{\text{init}} & = & [\![e]\!]^{\text{init}} \\
\{\!|e|\!\}^{\text{step}}_{\gamma}\,(m) & = & \mathit{let}\ m', v = [\![e]\!]^{\text{step}}_{\gamma}\,(m)\ \mathit{in}\ \delta_{m',v} \quad \text{if } e \text{ is deterministic} \\
\\
\{\!|\texttt{sample}(e)|\!\}^{\text{init}} & = & [\![e]\!]^{\text{init}} \\
\{\!|\texttt{sample}(e)|\!\}^{\text{step}}_{\gamma}\,(m) & = & \mathit{let}\ m', \mu = [\![e]\!]^{\text{step}}_{\gamma}\,(m)\ \mathit{in}\ \int \mu(dv)\ \delta_{m',v} \\
\\
\{\!|\texttt{factor}(e)|\!\}^{\text{init}} & = & [\![e]\!]^{\text{init}} \\
\{\!|\texttt{factor}(e)|\!\}^{\text{step}}_{\gamma}\,(m) & = & \mathit{let}\ m', v = [\![e]\!]^{\text{step}}_{\gamma}\,(m)\ \mathit{in}\ v \times \delta_{m',()} \\
\\
\left\{\!\!\left|\begin{array}{l} e\ \texttt{where rec init}\ x = c \\ \quad \texttt{and}\ x = e_x \\ \quad \texttt{and}\ y = e_y \end{array}\right|\!\!\right\}^{\text{init}} & = & c, \left(\{\!|e|\!\}^{\text{init}}, \{\!|e_x|\!\}^{\text{init}}, \{\!|e_y|\!\}^{\text{init}}\right) \\
\\
\left\{\!\!\left|\begin{array}{l} e\ \texttt{where rec init}\ x = c \\ \quad \texttt{and}\ x = e_x \\ \quad \texttt{and}\ y = e_y \end{array}\right|\!\!\right\}^{\text{step}}_{\gamma}\,(p_x, (m, m_x, m_y)) & = & \displaystyle\int \{\!|e_x|\!\}^{\text{step}}_{\gamma+[x.\texttt{last}\leftarrow p_x]}\,(m_x)(dm'_x, dv_x) \\
& & \displaystyle\int \{\!|e_y|\!\}^{\text{step}}_{\gamma+[x.\texttt{last}\leftarrow p_x, x\leftarrow v_x]}\,(m_y)(dm'_y, dv_y) \\
& & \displaystyle\int \{\!|e|\!\}^{\text{step}}_{\gamma+[x.\texttt{last}\leftarrow p_x, x\leftarrow v_x, y\leftarrow v_y]}\,(m)(dm', d_v) \\
& & \delta_{(v_x,(m',m'_x,m'_y)),v}
\end{array}
$$

Fig. 11. A simplified excerpt of the original ProbZelus probabilistic kernel operational semantics [5].

6.2 Correspondance between the original and the operational semantics

In this section we show that for scheduled programs where local declarations are all ordered according to data dependencies, the original kernel semantics coincide with the operational semantics, and thus with the denotational semantics.

Kernel semantics. The original ProbZelus semantics [5, Section 3.1] is a co-iterative semantics similar to the operational semantics presented in Section 4 where expressions are interpreted as state machines. The key difference is that probabilistic state machines directly computes a stream of unnormalized measures. Given a current state, the transition function $\{\!|e|\!\}^{\text{step}}$ of type $\Gamma \to S \to \mathit{Meas}(S \times V)$ returns a measure over pairs (next state, value).

Figure 11 shows a simplified excerpt of the semantics of probabilistic expressions from [5]. In a probabilistic context, a deterministic expression is interpreted as the Dirac delta 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 illustrate local declarations, Figure 11 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 $x.\texttt{last}$, and integrates over all possible executions of the sub-expressions to compute the main expression.

Since the original ProbZelus semantics focuses on a kernel language where local declarations are all *scheduled* the nested integrals are well-defined.

The `infer` operator is similar to its counterpart in Section 4.2. Given a probabilistic model e with a state of type S and a return value of type V, the state of `infer` is a measure over possible states of the model, and at each step, the transition function returns a measure over possible next states, and a measure over return values.

$$\begin{array}{lcl} [\![\texttt{infer}(e)]\!]^{\kappa\,\text{init}} & : & Meas(S) \\ [\![\texttt{infer}(e)]\!]^{\kappa\,\text{step}} & : & \Gamma \to Meas(S) \to Meas(S) \times Meas(V) \end{array}$$

Given the unnormalized measure defined by a model e, `infer`(e) performs the same operations as in the operational semantics (Equation (3) in Section 4.2): 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 then obtain a stream of distributions by iteratively integrating the transition function along the distribution of states.

$$\begin{array}{lcl} [\![\texttt{infer}(e)]\!]^{\kappa\,\text{init}} & = & \delta_{[\![e]\!]^{\text{init}}} \\ [\![\texttt{infer}(e)]\!]^{\kappa\,\text{step}}_{\gamma}(\sigma) & = & \mathit{let}\ \nu = \displaystyle\int \sigma(dm)\ \{\!|e|\!\}^{\text{step}}_{\gamma}(m)\ \mathit{in} \\ & & \mathit{let}\ \overline{\nu} = \nu/\nu(\top)\ \mathit{in} \\ & & (\pi_{1*}(\overline{\nu}), \pi_{2*}(\overline{\nu})) \end{array}$$

Correspondance. The definition of `infer` is very similar to its operational semantics counterpart where the function $\psi(m)$ in Equation (3) plays the role of the semantics of the model. We now show that these two notions coincide.

Proposition 6.3. *For all probabilistic expression e where all equations are scheduled, the kernel semantics corresponds to the operational semantics by integrating over random elements, i.e., for all environment γ and state m:*

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

Proof. The kernel semantics is only defined for a scheduled language. We first prove by induction on the structure of e that the kernel semantics coincides with the operational semantics where local declarations are ordered and can be computed incrementally without a fixpoint operator. For a set of scheduled equations, the ordered sequence of local declarations yields the same environment as the fixpoint operator (see Proposition A.1). Since all sets of equations are scheduled, the operational semantics (with a fixpoint operator) also coincides with the kernel semantics.

The case `sample`(μ) is a simple variable substitution $x = icdf_{\mu}(r)$. 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.

$$
\begin{aligned}
&\int \{\!|E_1|\!\}^{\text{step}}_{\gamma}(M_1)(dM_1', d\rho_1) \int \{\!|E_2|\!\}^{\text{step}}_{\gamma+\rho_1}(M_2)(dM_2', d\rho_2)\ \delta_{(M_1',M_2'),\rho_1+\rho_2} \\
&\quad = \int \left(\int_{[0,1]^{|e_1|}} \mathit{let}\ M_1', \rho_1, w_1 = (\!|E_1|\!)^{\text{step}}_{\gamma}(M_1, r_1)\ \mathit{in}\ w_1 \times \delta_{M_1,\rho_1}\ dr_1\right)(dM_1', d\rho_1) \\
&\qquad \int \left(\int_{[0,1]^{|e_2|}} \mathit{let}\ M_1', \rho_2, w_1 = (\!|E_2|\!)^{\text{step}}_{\gamma+\rho_1}(M_2, r_2)\ \mathit{in}\ w_2 \times \delta_{M_2,\rho_2}\ dr_2\right)(dM_2', d\rho_2) \\
&\qquad\quad \delta_{(M_1',M_2'),\rho_1+\rho_2} \\
&\quad = \int_{[0,1]^{|e_1|}} \int_{[0,1]^{|e_2|}} \int \left(\mathit{let}\ M_1', \rho_1, w_1 = (\!|E_1|\!)^{\text{step}}_{\gamma}(M_1, r_1)\ \mathit{in}\ w_1 \times \delta_{M_1,\rho_1}\right)(dM_1', d\rho_1) \\
&\qquad\qquad \int \left(\mathit{let}\ M_1', \rho_2, w_1 = (\!|E_2|\!)^{\text{step}}_{\gamma+\rho_1}(M_2, r_2)\ \mathit{in}\ w_2 \times \delta_{M_2,\rho_2}\right)(dM_2', d\rho_2)\ dr_1 dr_2 \\
&\qquad\qquad\quad \delta_{(M_1',M_2'),\rho_1+\rho_2} \\
&\quad = \int_{[0,1]^{|e_1|+|e_2|}} \mathit{let}\ M_1', \rho_1, w_1 = (\!|E_1|\!)^{\text{step}}_{\gamma}(M_1, r_1)\ \mathit{in} \\
&\qquad\qquad\qquad\quad \mathit{let}\ M_2', \rho_2, w_2 = (\!|E_2|\!)^{\text{step}}_{\gamma+\rho_1}(M_2, r_2)\ \mathit{in} \\
&\qquad\qquad\qquad\quad w_1 \times w_2 \times \delta_{(M_1',M_2'),\rho_1+\rho_2}\ dr_1 dr_2
\end{aligned}
$$

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 operational semantics and in the kernel based semantics.

Theorem 6.4 (Operational semantics correctness). *For every probabilistic model e where all equations sets are scheduled, for all distribution of states σ, and for all environment γ:*

$$
\begin{aligned}
[\![\texttt{infer}(e)]\!]^{\mathit{init}} &= [\![\texttt{infer}(e)]\!]^{\kappa\ \mathit{init}} \\
[\![\texttt{infer}(e)]\!]^{\mathit{step}}_{\gamma}(\sigma) &= [\![\texttt{infer}(e)]\!]^{\kappa\ \mathit{step}}_{\gamma}(\sigma)
\end{aligned}
$$

Proof. By construction, in the operational 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 semantics.

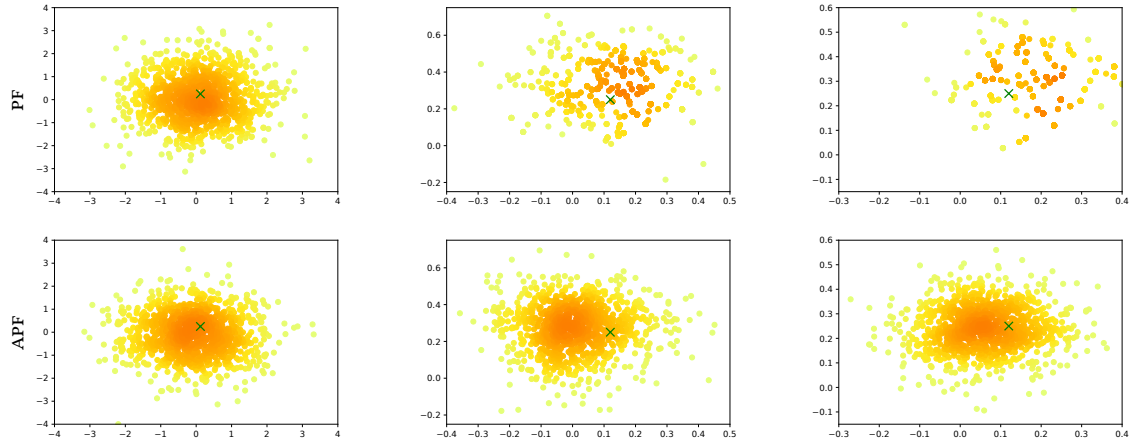
By Proposition 6.3 the un-normalized measure defined by the operational semantics matches the measure computed by the kernel semantics. Given this measure, the rest of the transition function of `infer` is the same in both cases. □

7 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 radar example of Section 2 where the current force is an unknown constant θ. We want to estimate the moving position (state parameter), and the current force (constant parameter). To estimate θ, the `current_force` is now a probabilistic model defined as follows (where the noise parameter `st` is a global constant):

```
proba current_force(x) = theta where
  rec init theta = sample(gaussian(zeros, st))
  and theta = last theta
```



(a) $t = 0$ (b) $t = 30$ (c) $t = 50$

Fig. 12. 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.

Filtering. To estimate state parameters, Sequential Monte Carlo (SMC) inference algorithms rely on filtering techniques [19, 30]. 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, 5, 6]. 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 12 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 [35]. The lower part of Figure 12 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

$$\textsc{cexp-c}\ \frac{}{C \vdash^c c} \qquad \textsc{cexp-var}\ \frac{x \in C}{C \vdash^c x} \qquad \textsc{cexp-pair}\ \frac{C \vdash^c e_1 \qquad C \vdash^c e_2}{C \vdash^c (e_1, e_2)} \qquad \textsc{cexp-op}\ \frac{C \vdash^c e}{C \vdash^c op(e)}$$

$$\textsc{cexp-where}\ \frac{C + dom(E) \vdash^c e \qquad C \vdash^c E : dom(E)}{C \vdash^c e \texttt{ where rec } E}$$

$$\textsc{ceq-init}\ \frac{}{C \vdash^c \texttt{init } x = e : \emptyset} \qquad \textsc{ceq-last}\ \frac{}{C \vdash^c x = \texttt{last } x : \{x\}} \qquad \textsc{ceq-cexp}\ \frac{C \vdash^c e}{C \vdash^c x = e : \{x\}} \qquad \textsc{ceq-exp}\ \frac{C \nvdash^c e}{C \vdash^c x = e : \emptyset}$$

$$\textsc{ceq-and}\ \frac{C + C_2 \vdash^c E_1 : C_1 \qquad C + C_1 \vdash^c E_2 : C_2}{C \vdash^c E_1 \texttt{ and } E_2 : C_1 + C_2}$$

Fig. 13. Constant expressions and equations.

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 denotational semantics of Section 5 to prove the correctness of this transformation, i.e., the transformation preserves the ideal semantics of the program.

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

```
let current_force.prior = gaussian(zeros, st)
proba current_force.model(theta, x) = theta

let tracker_prior = current_force.prior
proba tracker.model(theta, y_obs) = x where
  rec x = motion(f)
  and f = current_force.model(theta, x)
  and () = observe(gaussian(g(x), 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)
```

7.1 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 $[], \emptyset \vdash prog : \Phi, C$ incrementally analyzes all the declarations to build the environment Φ which associates to each probabilistic node a type ϕ mapping 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 Figures 13 and 14.

Constants (CEQ-* and CEXP-* rules, Figure 13): The auxiliary judgement $C \vdash^c E : C'$ (CEQ-* rules) 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 previous value (x = `last` x) or if it is defined by a constant expression. The auxiliary judgement $C \vdash^c e$ (CEXP-* rules)

$$\textsc{decl-cst}\ \frac{C \vdash^c e}{\Phi, C \vdash \texttt{let}\ x = e : \Phi, C + \{x\}} \qquad \textsc{decl-proba}\ \frac{C \vdash e : \phi}{\Phi, C \vdash \texttt{proba}\ f\ x = e : \Phi + [f \leftarrow \phi], C}$$

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

$$\textsc{exp-where}\ \frac{C \vdash e : \phi_e \qquad C \vdash^c E : D \qquad C, D \vdash E : \phi_E}{C \vdash e\ \texttt{where rec}\ E : \phi_e + \phi_E}$$

$$\textsc{exp-sample}\ \frac{x \in D \qquad C \vdash^c e}{C, D \vdash \texttt{init}\ x = \texttt{sample}(e) : [x \leftarrow e]} \qquad \textsc{eq-one}\ \frac{C \vdash e : \phi}{C, D \vdash x = e : \phi}$$

$$\textsc{eq-and}\ \frac{C, D \vdash E_1 : \phi_1 \qquad C, D \vdash E_2 : \phi_2}{C, D \vdash E_1\ \texttt{and}\ E_2 : \phi_1 + \phi_2}$$

Fig. 14. Extract constant parameters and associated prior distributions (full type system in Appendix B.2).

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 (DECL-* rules, Figure 14): 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 (EXP-* rules, Figure 14): The auxiliary judgement $C \vdash e : \phi$ 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.

Equations (EQ-* rules, Figure 14): 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 `current_force` and is propagated through the node `tracker` that calls `current_force`. The final environment is:

$$\Phi = [\texttt{current\_force} \leftarrow [\texttt{theta} \leftarrow \texttt{gaussian}(\ldots)], \texttt{tracker} \leftarrow [\theta \leftarrow \texttt{current\_force.prior}]]$$

7.2 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 15 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.

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

Fig. 15. APF compilation (full definition in Appendix B.3).

A model $\texttt{proba}\ f\ x = e$ 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, $\texttt{let}\ f.\texttt{prior} = im(\phi)$, and 2) a new model that takes the constant parameters $dom(\phi)$ as additional arguments, $\texttt{proba}\ f.\texttt{model}\ (dom(\phi), x) = C_\phi(e)$.

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 $f.\texttt{model}$ 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 $f.\texttt{prior}$.

Finally, the call to $\texttt{infer}(f(e))$ is replaced by a call to $\texttt{APF.infer}(f.\texttt{model}, f.\texttt{prior}, e)$.

7.3 Correctness

We use the denotational semantics defined in Section 5.2 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 $\texttt{infer}(f(e))$ and $\texttt{APF.infer}(f.\texttt{model}, f.\texttt{prior}, e)$ then corresponds to the case $f_\theta(e)$ in Figure 15.

Definition 7.1. For a model f that is compiled into $f.\texttt{prior}$ and $f.\texttt{model}$, the ideal semantics of `APF.infer` externalizes the definition of the constant parameters: $\texttt{APF.infer}(f.\texttt{model}, f.\texttt{prior}, e)$ 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 7.2. *For all probabilistic functions f such that $\Phi(f) = \phi$, for all expressions e in the body of f such that $C \vdash e : \phi_e$, for all contexts H, R, there is a permutation relating R and $[R' : R^p]$ such that*

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

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., $H_f(\vec{p}) = icdf_{\vec{\mu}}(R_0^p)$.

Lemma 7.3. *For all equations E in the body of f such that $C, D \vdash E : \phi_E$, for all contexts H, R, there is a permutation relating R and $[R' : R^p]$ such that*

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

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 apply Proposition 5.3 by giving a permutation relating R and $R' + R^p$ inducing the coupling, and show that a semantics derivation for e in a context H, R is equivalent to a semantics derivation for $C_\phi(e)$ in the context $H + H_f, R'$. 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* $x \in dom(\phi)$. We use the permutation relating $[R_x]$ $[]$: $[R_x]$ and with $v_x = icdf_\mu(R_{x0})$

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

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

$$\frac{H_f(x) = v_x \quad \overline{H + H_f, [] \vdash \emptyset : 1}}{H + H_f, [] \vdash C_\phi(\texttt{init}\ x = \texttt{sample}(d)\ \texttt{and}\ x = \texttt{last}\ x) : 1}$$

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 two permutations relating R_e with $[R'_e : R_e^p]$ and relating R_g with $[R'_g, R_g^p]$. 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 $\vec{v_p} = icdf_{\vec{\mu_g}}(R_{g\,0}^p)$.

$$\frac{\dfrac{H + H_f, R'_e \vdash C_\phi(e) \Downarrow (s_e, w_e)}{H, R_e \vdash e \Downarrow (s_e, w_e)} \quad \dfrac{[x \leftarrow s_e] + [\vec{p_g} \leftarrow \vec{v_p}], R'_g \vdash C_{\phi_g}(e_g) \Downarrow (s, w)}{[x \leftarrow s_e], R_g \vdash e_g \Downarrow (s, w)}}{H, [R_e : R_g] \vdash g_\theta(e) \Downarrow (s, w_e \times w)}$$

On the other end, by construction we have $\texttt{proba}\ g.\texttt{model}\ (\vec{p_g}, x) = C_{\phi_g}(e_g)$ and there are two cases. If $\theta \in dom(\phi)$, then the constant parameters are already in the context and $H_f(\theta) = \vec{v_p}$. Using the permutation relating $[R_e : R_g]$ with $[R'_e : R'_g] : [R^p_e : R^p_g]$ we have:

$$\dfrac{\dfrac{\overline{H + H_f, [] \vdash \theta \Downarrow (\vec{v_p}, 1)} \quad H + H_f, R'_e \vdash C_\phi(e) \Downarrow (s_e, w_e)}{H + H_f, R'_e \vdash (\theta, C_\phi(e)) \Downarrow ((\vec{v_p}, s_e), w_e)} \quad [x \leftarrow s_e, \vec{p_g} \leftarrow \vec{v_p}], R'_g \vdash C_{\phi_g}(e_g) \Downarrow (s, w)}{\dfrac{H + H_f, [R'_e, R'_g] \vdash g.\texttt{model}(\theta, C_\phi(e)) \Downarrow (s, w_e \times w)}{H + H_f, [R'_e, R'_g] \vdash C_\phi(g_\theta(e)) \Downarrow (s, w_e \times w)}}$$

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

$$\dfrac{\dfrac{\dfrac{\dots}{H + H_f + [\theta \leftarrow \vec{v_p}], [R'_e, R'_g] \vdash g.\texttt{model}(\theta, C_\phi(e)) \Downarrow (s, w)} \quad \dfrac{\dfrac{g.\texttt{prior} = \vec{\mu_g}}{H + H_f, R^p_g \vdash \texttt{sample}(g.\texttt{prior}) \Downarrow (\vec{v_p}, 1)}}{H + H_f + [\theta \leftarrow \vec{v_p}], R^p_g \vdash \texttt{init}\ \theta = \texttt{sample}(g.\texttt{prior}) : 1}}{H + H_f, [R'_e, R'_g, R^p_g] \vdash g.\texttt{model}(\theta, C_\phi(e))\ \texttt{where rec init}\ \theta = \texttt{sample}(g.\texttt{prior}) \Downarrow (s, w)}}{H + H_f, [R'_e, R'_g, R^p_g] \vdash C_\phi(g_\theta(e)) \Downarrow (s, w)}$$

□

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

Theorem 7.4 (APF compilation). *For all probabilistic nodes f, for all contexts H,*

$$H \vdash \texttt{infer}(f(e)) \downarrow d \iff H \vdash \texttt{APF.infer}(f.\texttt{model}, f.\texttt{prior}, e) \downarrow d$$

Proof. From Definition 7.1, we need to show that for all random streams R:

$$H, R \vdash f_\theta(e) \Downarrow (s, w) \iff H + [\theta \leftarrow \vec{v_p}], R' \vdash f.\texttt{model}(\theta, e) \Downarrow (s, w)$$

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

8 RELATED WORK

Probabilistic Semantics. In the seminal work [45], 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 [27]. This setting has been extended to continuous distributions with models based on positive cones [26, 34], a variation on Banach spaces with positive scalars [51]. 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 [33].

Quasi-Borel spaces (QBS) define an alternative semantics of higher-order probabilistic programs with conditioning [41, 53] 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 [28]. Our new semantics rely on a similar representation: probabilistic expressions are interpreted by pushingforward a uniform measure over $[0,1]^{|e|}$ along a measurable function.

Our semantics are also closely related to the sampling semantics of [10] 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 (operational and denotational) 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 [23, 55, 56] 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 [17] 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. In the original ProbZelus semantics, the interpretation of local declarations yields nested integrals that are only well defined if equations are ordered according to data dependencies (Figure 11 in Section 6.2). Our new semantics relaxes this assumptions using a fixpoint operator.

Existing works define a fixpoint operator to interpret loops and recursions in probabilistic lambda-calculi [34, 44, 54]. 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. An alternative is to lift the deterministic operational semantics of Section 4 to measures, i.e., on measure over $V_\bot = V + \{\bot\}$ with the least element $\bot = \delta_{[x \leftarrow \bot]_{x \in X}}$. However, using this CPO the semantics of `sample(`e`)` is re-computed at each fixpoint iteration which breaks the correlation between mutually recursive equations. Therefore, we cannot directly define a schedule agnostic kernel semantics.

Example. Consider the following example: `x = y and y = sample(`μ`)`. In the kernel semantics, local declarations are interpreted as integration. The semantics is thus the fixpoint of the following function F where the argument ρ is a measure over environments.

$$F(\rho) \;=\; \int \rho(dr) \int \delta_{r(\mathsf{y})}(dx) \int \mu(dy)\; \delta_{[\mathsf{x} \leftarrow x,\; \mathsf{y} \leftarrow y]}$$

If the least element ρ_0 is the null measure, then $\rho_1 = F(\rho_0)$ is also the null measure. If $\rho_0 = \delta_{[\mathsf{x} \leftarrow \bot,\; \mathsf{y} \leftarrow \bot]}$ we get :

$$\begin{aligned}
\rho_0 &= \delta_{[\mathsf{x} \leftarrow \bot, \mathsf{y} \leftarrow \bot]} \\
\rho_1 &= \textstyle\int \mu(dy_1)\; \delta_{[\mathsf{x} \leftarrow \bot, \mathsf{y} \leftarrow y_1]} \\
\rho_2 &= \textstyle\int \mu(dy_1) \int \mu(dy_2)\; \delta_{[\mathsf{x} \leftarrow y_1, \mathsf{y} \leftarrow y_2]} \\
\rho_3 &= \textstyle\int \mu(dy_2) \int \mu(dy_3)\; \delta_{[\mathsf{x} \leftarrow y_2, \mathsf{y} \leftarrow y_3]}
\end{aligned}$$

The fixpoint is reached after 3 iterations, but the correlation between x and y is lost. The semantics differs from the scheduled version y = `sample`(μ) `and` x = y which yields the expected measure: $\int \mu(dy)\ \delta_{[\mathtt{x}\leftarrow y,\mathtt{y}\leftarrow y]}$.

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 [34, 54]. 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 operational semantics. We leave for future investigations these connections.

Program Equivalence. Probabilistic bisimulation has been introduced for testing equivalence of (discrete) probabilistic systems [47] and generalized to study Labelled Markov Process (continuous systems) [31]. Probabilistic coupling is a classic proof technique for probabilistic bisimulation, as shown in [29] using a categorical framework and a co-algebraic presentation of probabilistic systems. Our presentation of probabilistic bisimulation and coupling for state machines follows this line of research.

Probabilistic coupling has been extensively used for discrete systems. In [3], an algorithm, based on couplings, has been introduced to effectively construct probabilistic bisimulation for discrete systems. Coupling has been applied to the formal verification of distribution equivalence, convergence, and stochastic domination [4, 42]. Probabilistic coupling has also been used to define a probabilistic bisimulation for a (higher-order) lambda-calculus with continuous probabilistic choice [46].

A logical relation is proposed in [23, 55, 56] to reason about contextual equivalence for probabilistic programs. In particular [55] 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 7.2 (APF Correctness). We thus show that we can apply similar techniques with our denotational 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.

9 CONCLUSION

In this paper we proposed two semantics for a reactive probabilistic programming languages: an operational semantics and a denotational 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 operational semantics manipulates state machines and equivalence reasoning requires the description of bisimulations on states. The denotational 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 denotational semantics to prove the correctness of the transformation.

Acknowledgements. The authors are grateful to the following people for many discussions and helpful feedback: Timothy Bourke, Grégoire Bussone, Raphaëlle Crubillé, Thomas Ehrhard, Guillaume Geoffroy, Adrien Guatto, Patrick Hoscheit, Paul Jeanmaire, Dexter Kozen, Basile Pesin, Gordon Plotkin, and Marc Pouzet. This work was supported by the project ReaLiSe, Émergence Ville de Paris 2021-2025.

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] Christel Baier, Bettina Engelen, and Mila E. Majster-Cederbaum. 2000. Deciding Bisimilarity and Similarity for Probabilistic Processes. *J. Comput. Syst. Sci.* 60, 1 (2000), 187–231.

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

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

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

[7] 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.

[8] 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.

[9] Richard Blute, Josée Desharnais, Abbas Edalat, and Prakash Panangaden. 1997. Bisimulation for Labelled Markov Processes. In *LICS*.

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

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

[12] 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.

[13] 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.

[14] Timothy Bourke, Paul Jeanmaire, and Marc Pouzet. 2025. Functional Stream Semantics for a Synchronous Block-Diagram Compiler. In *LICS*.

[15] 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.

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

[17] 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.

[18] Paul Caspi and Marc Pouzet. 1998. A Co-iterative Characterization of Synchronous Stream Functions. In *CMCS*.

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

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

[21] Jean-Louis Colaço and Marc Pouzet. 2003. Clocks as First Class Abstract Types. In *EMSOFT*.

[22] 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*.

[23] Ryan Culpepper and Andrew Cobb. 2017. Contextual Equivalence for Probabilistic Programs with Continuous Random Variables and Scoring. In *ESOP*.

[24] Pascal Cuoq and Marc Pouzet. 2001. Modular Causality in a Synchronous Stream Language. In *ESOP*.

[25] 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*.

[26] 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.

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

[28] 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.

[29] Erik P. de Vink and Jan J. M. M. Rutten. 1997. Bisimulation for Probabilistic Transition Systems: A Coalgebraic Approach. In *ICALP*.

[30] 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.

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

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

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

[34] 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.

[35] 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*.

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

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

[38] 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.

[39] 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.

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

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

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

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

[44] Claire Jones and Gordon D. Plotkin. 1989. A Probabilistic Powerdomain of Evaluations. In *LICS*.

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

[46] Ugo Dal Lago and Francesco Gavazzo. 2019. On Bisimilarity in Lambda Calculi with Continuous Probabilistic Choice. In *MFPS*.

[47] Kim Guldstrand Larsen and Arne Skou. 1989. Bisimulation Through Probabilistic Testing. In *POPL*.

[48] Wonyeol Lee, Hangyeol Yu, Xavier Rival, and Hongseok Yang. 2020. Towards verified stochastic variational inference for probabilistic programs. *Proc. ACM Program. Lang.* 4, POPL (2020), 16:1–16:33.

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

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

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

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

[53] 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*.

[54] 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.

[55] 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.

[56] 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.

$$
\begin{array}{lll}
|\, e \,| & = 0 & \text{if } e \text{ is deterministic} \\
|\, \texttt{sample}(e) \,| & = 1 & \\
|\, \texttt{factor}(e) \,| & = 0 & \\
|\, f(e) \,| & = |\, e \,| + |\, e_f \,| & \text{where } \texttt{proba}\ f\ x = e_f \\
|\, \texttt{present}\ e\ \rightarrow\ e_1\ \texttt{else}\ e_2 \,| & = |\, e_1 \,| + |\, e_2 \,| & \\
|\, \texttt{reset}\ e_1\ \texttt{every}\ e_2 \,| & = |\, e_1 \,| & \\
|\, e\ \texttt{where rec}\ E \,| & = |\, e \,| + |\, E \,| &
\end{array}
$$

$$
\begin{array}{lll}
|\, x = e \,| & = |\, e \,| & \\
|\, \texttt{init}\ x = e \,| & = |\, e \,| & \\
|\, E_1\ \texttt{and}\ E_2 \,| & = |\, E_1 \,| + |\, E_2 \,| &
\end{array}
$$

Fig. 16. Dimension of probabilistic expressions and equations.

A SEMANTICS

A.1 Operational semantics

Model dimension. The dimension of a probabilistic model is defined as the number of random variables in the model, i.e., the number of calls to the `sample` operator. Since loops and recursive calls are not allowed in the language of Figure 2, the dimension of a model can be statically computed with the function $|\, e : V \,| : \mathbb{N}$ defined in Figure 16.

The operational semantics is presented in Section 4. Figures 17 and 18 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 counts 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.

Scheduling. To compare the new denotational semantics with the kernel 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 denotational semantics except for the two following rules:

$$
\begin{array}{lll}
(\!|E_1\ \texttt{and}\ E_2|\!)^{\text{s step}}_{\gamma}\ ((M_1, M_2), [r_1 : r_2]) & = & \mathit{let}\ M_1', \rho_1, w_1 = (\!|E_1|\!)^{\text{s step}}_{\gamma}\ (M_1, r_1)\ \mathit{in} \\
 & & \mathit{let}\ M_2', \rho_2, w_2 = (\!|E_2|\!)^{\text{s step}}_{\gamma+\rho_1}\ (M_2, r_2)\ \mathit{in} \\
 & & (M_1', M_2'), \rho_1 + \rho_2, w_1 \times w_2
\end{array}
$$

$$
\begin{array}{lll}
(\!|e\ \texttt{where rec}\ E|\!)^{\text{s step}}_{\gamma}\ ((m, M), [r_e : r_E]) & = & \mathit{let}\ M', \rho, W = (\!|E|\!)^{\text{s step}}_{\gamma}\ (M, r_E)\ \mathit{in} \\
 & & \mathit{let}\ m', v, w = (\!|e|\!)^{\text{s step}}_{\gamma+\rho}\ (m, r_e)\ \mathit{in} \\
 & & (m', M'), v, w \times W
\end{array}
$$

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.

$$
\begin{array}{lll}
(\!|e|\!)^{\text{init}} & = & [\![e]\!]^{\text{init}} \qquad \text{if } e \text{ is deterministic} \\
(\!|e|\!)^{\text{step}}_{\gamma}\ (m, []) & = & \mathit{let}\ m', v = [\![e]\!]^{\text{step}}_{\gamma}\ (m)\ \mathit{in}\ m', v, 1 \\
\\
(\!|\texttt{sample}(e)|\!)^{\text{init}} & = & [\![e]\!]^{\text{init}} \\
(\!|\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}} & = & [\![e]\!]^{\text{init}} \\
(\!|\texttt{factor}(e)|\!)^{\text{step}}_{\gamma}\ (m, []) & = & \mathit{let}\ m', v = [\![e]\!]^{\text{step}}_{\gamma}\ (m)\ \mathit{in}\ m', (), v \\
\\
(\!|f(e)|\!)^{\text{init}} & = & (\!|e|\!)^{\text{init}}, \left(\!\left|e_f\right|\!\right)^{\text{init}} \qquad \text{where } \texttt{proba}\ f\ x = e_f \\
(\!|f(e)|\!)^{\text{step}}_{\gamma}\ ((m_e, m_f), [r_e : r_f]) & = & \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 = \left(\!\left|e_f\right|\!\right)_{[x \leftarrow v_e]}\ (m_f, r_f)\ \mathit{in} \\
 & & (m'_f, m'_e), v, w_e \times w_f \\
\\
(\!|\texttt{present}\ e\ \rightarrow\ e_1\ \texttt{else}\ e_2|\!)^{\text{init}} & = & [\![e]\!]^{\text{init}}, (\!|e_1|\!)^{\text{init}}, (\!|e_2|\!)^{\text{init}} \\
(\!|\texttt{present}\ e\ \rightarrow\ e_1\ \texttt{else}\ e_2|\!)^{\text{step}}_{\gamma}\ ((m, m_1, m_2), [r_1 : r_2]) & = & \mathit{let}\ m', v = [\![e]\!]^{\text{step}}_{\gamma}\ (m)\ \mathit{in} \\
 & & \mathit{if}\ v\ \mathit{then}\ \mathit{let}\ (m'_1, v_1, w) = (\!|e_1|\!)^{\text{step}}_{\gamma}\ (m_1, r_1) \\
 & & \qquad \mathit{in}\ (m', m'_1, m_2), v_1, w \\
 & & \quad \mathit{else}\ \ \mathit{let}\ (m'_2, v_2, w) = (\!|e_2|\!)^{\text{step}}_{\gamma}\ (m_2, r_2) \\
 & & \qquad \mathit{in}\ (m', m_1, m'_2), v_2, w \\
\\
(\!|\texttt{reset}\ e_1\ \texttt{every}\ e_2|\!)^{\text{init}} & = & (\!|e_1|\!)^{\text{init}}, (\!|e_1|\!)^{\text{init}}, [\![e_2]\!]^{\text{init}} \\
(\!|\texttt{reset}\ e_1\ \texttt{every}\ e_2|\!)^{\text{step}}_{\gamma}\ ((m_0, m_1, m_2), w, r) & = & \mathit{let}\ m'_2, v_2 = [\![e_2]\!]^{\text{step}}_{\gamma}\ (m_2)\ \mathit{in} \\
 & & \mathit{let}\ m'_1, v_1, w = (\!|e_1|\!)^{\text{step}}_{\gamma}\ (\ \mathit{if}\ v_2\ \mathit{then}\ m_0\ \mathit{else}\ m_1, r)\ \mathit{in} \\
 & & (m_0, m'_1, m'_2), v_1, w \\
\\
(\!|e\ \texttt{where rec}\ E|\!)^{\text{init}} & = & (\!|e|\!)^{\text{init}}, (\!|E|\!)^{\text{init}} \\
(\!|e\ \texttt{where rec}\ E|\!)^{\text{step}}_{\gamma}\ ((m, M), [r_e : r_E]) & = & \mathit{let}\ F(\rho) = \left(\mathit{let}\ M', \rho, w = (\!|E|\!)_{\gamma+\rho}\ (M, r_E)\ \mathit{in}\ \rho\right)\ \mathit{in} \\
 & & \mathit{let}\ \rho = \mathit{fix}(F)\ \mathit{in} \\
 & & \mathit{let}\ M', \rho, W = (\!|E|\!)^{\text{step}}_{\gamma+\rho}\ (M, r_E)\ \mathit{in} \\
 & & \mathit{let}\ m', v, w = (\!|e|\!)^{\text{step}}_{\gamma+\rho}\ (m, r_e)\ \mathit{in} \\
 & & (m', M'), v, w \times W
\end{array}
$$

Fig. 17. Operational semantics for probabilistic expressions.

Proposition A.1. *For an expression where all equations are scheduled, the scheduled semantics is equal to the operational semantics with a fixpoint.*

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

Lemma A.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:*

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

$$
\begin{array}{lcl}
(\!|x = e|\!)^{\text{init}} & = & (\!|e|\!)^{\text{init}} \\
(\!|x = e|\!)^{\text{step}}_{\gamma}\ (m, r) & = & \textit{let } m', v, w = (\!|e|\!)^{\text{step}}_{\gamma}\ (m, r)\ \textit{in } m', [x \leftarrow v], w \\
\\
(\!|\texttt{init}\ x = e|\!)^{\text{init}} & = & \textit{nil}, (\!|e|\!)^{\text{init}} \\
(\!|\texttt{init}\ x = e|\!)^{\text{step}}_{\gamma}\ ((\textit{nil}, m_0), r) & = & \textit{let } m', i, w = (\!|e|\!)^{\text{step}}_{\gamma}\ (m_0, r)\ \textit{in } (\gamma(x), m_0), [x.\texttt{last} \leftarrow i], w \\
(\!|\texttt{init}\ x = e|\!)^{\text{step}}_{\gamma}\ ((v, m_0), r) & = & (\gamma(x), m_0), [x.\texttt{last} \leftarrow v], 1 \\
\\
(\!|E_1\ \texttt{and}\ E_2|\!)^{\text{init}} & = & (\!|E_1|\!)^{\text{init}}, (\!|E_2|\!)^{\text{init}} \\
(\!|E_1\ \texttt{and}\ E_2|\!)^{\text{step}}_{\gamma}\ ((M_1, M_2), [r_1 : r_2]) & = & \textit{let } M_1', \rho_1, w_1 = (\!|E_1|\!)^{\text{step}}_{\gamma}\ (M_1, r_1)\ \textit{in} \\
 & & \textit{let } M_2', \rho_2, w_2 = (\!|E_2|\!)^{\text{step}}_{\gamma}\ (M_2, r_2)\ \textit{in} \\
 & & (M_1', M_2'), \rho_1 + \rho_2, w_1 \times w_2
\end{array}
$$

Fig. 18. Operational semantics for equations.

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 $[r_1 : r_2]$:

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

□

A.2 Denotational semantics

Stream functions. The denotational semantics is presented in Section 5. The definition of this semantics relies on a few stream functions presented in Figure 19. 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.

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

Fig. 19. Stream functions for the denotational semantics.

$$H \vdash c \downarrow c \qquad H \vdash x \downarrow H(x) \qquad \frac{H \vdash e_1 \downarrow s_1 \qquad H \vdash e_2 \downarrow s_2}{H \vdash (e_1, e_2) \downarrow (s_1, s_2)} \qquad \frac{H \vdash e \downarrow s}{H \vdash op(e) \downarrow op(s)} \qquad \frac{H(x.\texttt{last}) = s}{H \vdash \texttt{last}\ x \downarrow s}$$

$$\frac{H \vdash e \downarrow s_e \qquad \texttt{node}\ f\ x = e_f \qquad [x \leftarrow s_e] \vdash e_f \downarrow s}{H \vdash f(e) \downarrow s} \qquad \frac{H + H_E \vdash E \qquad H + H_E \vdash e \downarrow s}{H \vdash e\ \texttt{where rec}\ E \downarrow s}$$

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

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

$$\frac{H \vdash E_1 \qquad H \vdash E_2}{H \vdash E_1\ \texttt{and}\ E_2}$$

Fig. 20. Deterministic denotational semantics.

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 $x \in dom(H)$, $(tl\ H)(x) = tl\ (H(x))$.

Denotational semantics. The full deterministic and probabilistic denotational semantics including the rules for `present` and `reset` are given in Figures 20 and 21. The semantics of the control structure `present` $e\ \to\ e_1$ `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. $[(tl^{\,n}\ H) \vdash e_1 \downarrow s_n]_{n \in \mathbb{N}}$ represents the stream of streams where s_n

$$H, [] \vdash c \Downarrow (c, 1) \qquad H, [] \vdash x \Downarrow (H(x), 1) \qquad \frac{H, R_e \vdash e \downarrow (s_e, w_e) \qquad \texttt{proba}\ f\ x = e_f \qquad [x \leftarrow s_e], R_f \vdash e_f \Downarrow (s, w)}{H, [R_e : R_f] \vdash f(e) \Downarrow (s, w \times w_e)}$$

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

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

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

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

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

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

Fig. 21. Probabilistic denotational semantics.

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, 5, 6] belong to the family of SMC algorithms. These methods rely on a set of independent simulations, called *particles*. Each particle returns an 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 $1 \leq i \leq N$ corresponds to a possible value of the parameters (*i.e.*, random variables) p_t^i of the model. We begin by sampling a new set of particles

Data: probablisitic model `model`, observation y_t, and previous result μ_{t-1}.
Result: μ_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 μ_t

Algorithm 1: Particle Filter.

Data: probabilistic model `model`, observation y_t, and previous result μ_{t-1}.
Result: μ_t an approximation of the distributions of state parameter x_t and constant parameter θ.

for *each particle* $i = 1$ **to** N **do**
  $x^i_{t-1}, \Theta^i_{t-1} =$ `sample()`
  $\theta^i =$ `sample()`
  $x^i_t, w^i_t =$ `model(`$y_t \mid \theta^i, x^i_{t-1}$`)`
  $\Theta^i_t = \mathit{Udpate}(\Theta^i_{t-1}, \lambda\theta.$ `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 μ_t

Algorithm 2: *Assumed Parameter Filter* [35].

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 the update of 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 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 $x^i_{t-1} \to x^i_t$. 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 $w^i_t / \sum_{i=1}^N w^i_t$.

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

$$\frac{\Phi, C \vdash d_1 : \Phi_1, C_1 \qquad \Phi_1, C_1 \vdash d_2 : \Phi', C'}{\Phi, C \vdash d_1\ d_2 : \Phi', C'}$$

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

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

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

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

Fig. 22. Extract constant parameters and associated prior distributions.

$$\frac{}{C \vdash^c c} \qquad \frac{x \in C}{C \vdash^c x} \qquad \frac{C \vdash^c e_1 \qquad C \vdash^c e_2}{C \vdash^c (e_1, e_2)} \qquad \frac{C \vdash^c e}{C \vdash^c op(e)} \qquad \frac{C + dom(E) \vdash^c e \qquad C \vdash^c E : dom(E)}{C \vdash^c e\ \texttt{where rec}\ E}$$

$$\frac{}{C \vdash^c \texttt{init}\ x = e : \emptyset} \qquad \frac{}{C \vdash^c x = \texttt{last}\ x : \{x\}} \qquad \frac{C \vdash^c e}{C \vdash^c x = e : \{x\}} \qquad \frac{C \not\vdash^c e}{C \vdash^c x = e : \emptyset}$$

$$\frac{C + C_2 \vdash^c E_1 : C_1 \qquad C + C_1 \vdash^c E_2 : C_2}{C \vdash^c E_1\ \texttt{and}\ E_2 : C_1 + C_2}$$

Fig. 23. Constant expressions and equations.

B.2 Static Analysis

The full type system is given in Figures 22 and 23. The interesting cases are presented in Section 7.1.

B.3 Compilation

The complete compilation function to transform a ProbZelus model into a model compatible with `APF.infer` is given in Figure 24. Most cases simply call the compilation functions on all sub-expressions. The interesting cases are presented in Section 7.2.

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

Fig. 24. APF Compilation.