Parallelizable Feynman-Kac models for universal probabilistic programming

Michele Boreale and Luisa Collodi

Università degli Studi di Firenze, Italy
Dipartimento di Statistica, Informatica, Applicazioni "G. Parenti"
`{michele.boreale,luisa.collodi}@unifi.it`

Abstract. We study provably correct and efficient instantiations of the Sequential Monte Carlo (SMC) inference scheme in the context of formal operational semantics of Probabilistic Programs (PPs). We focus on *universal* PPs featuring sampling from arbitrary measures and conditioning/reweighting in unbounded loops. We first equip Probabilistic Program Graphs (PPGs), an automata-theoretic description format of PPs, with an expectation-based semantics over infinite execution traces, which also incorporates trace weights. We then prove a finite approximation theorem that provides bounds to this semantics based on expectations taken over finite, fixed-length traces. This enables us to frame our semantics within a Feynman-Kac (FK) model, and ensures the consistency of the Particle Filtering (PF) algorithm, an instance of SMC, with respect to our semantics. Building on these results, we introduce VPF, a vectorized version of the PF algorithm tailored to PPGs and our semantics. Experiments conducted with a proof-of-concept implementation of VPF show very promising results compared to state-of-the-art PP inference tools.

Keywords: Probabilistic programming, operational semantics, SIMD parallelism, Sequential Monte Carlo.

1 Introduction

Probabilistic Programming Languages (PPLs) [26, 7] offer a systematic approach to define arbitrarily complicated probabilistic models. One is typically interested in performing *inference* on these models, given observed data; for example, finding the posterior distribution of the program's variables conditioned on the observed data. Here, in the context of formal operational semantics of Probabilistic Programs, we study provably correct and parallelizable instantiations of the Sequential Monte Carlo (SMC) inference scheme.

In terms of formal semantics of PPLs, the denotational approach introduced by Kozen [32] offers a solid mathematical foundation. However, when it comes to practical algorithms for PPL-based inference, the landscape appears somewhat fragmented. On one hand, *symbolic* and *static analysis* techniques, see e.g. [43, 23, 38, 8, 10, 42], yield results with correctness guarantees firmly grounded in the semantics of PPLs but often struggle with scalability. On the other hand, practical languages and inference algorithms predominantly leverage Monte Carlo (MC) *sampling* techniques (MCMC, SMC), which are more scalable but often lack a clear connection to formal semantics [25, 17, 12]. Notable exceptions to this situation include works such as [40, 51, 34, 13, 33], which are discussed in the related work section further below.

Establishing the consistency of an inference algorithm with respect to a PPL's formal semantics is not merely a theoretical pursuit. In the context of *universal* PPLs [24], integration of unbounded loops and conditioning with MC sampling, which requires truncating computations at a finite time, presents significant challenges [10]. Additionally, the interplay between continuous and discrete distributions in these PPLs can lead to complications, potentially causing existing sampling-based algorithms to yield incorrect results [51].

In the present work, we establish a precise connection between *Probabilistic Program Graphs (PPGs)*, a general automata-theoretic description format of PPs, and *Feynman-Kac (FK)* models, a formalism for state-based probabilistic processes and observations defined over a finite time horizon [19, Ch.5]. This connection enables us to prove the consistency for PPGs of the *Particle Filtering (PF)* inference algorithm, one of the incarnations of Sequential Monte Carlo approach [19, Ch.10]. In establishing this connection, we adopt a decisively operational perspective.

In a PPG (Section 3), computation (essentially, sampling) progresses in successive stages specified by the direct edges of a graph (transitions), with nodes serving as *checkpoints* between stages for *conditioning* on observed data or more generally updating computation weights. The operational semantics of PPGs is formalized in terms of Markov kernels and score functions. Building on this, we introduce a measure-theoretic, infinite-trace semantics (Section 4, with the necessary measure theory reviewed in Section 2). A finite approximation theorem then allows us to relate this trace semantics precisely to a finite-time horizon FK model (Section 5). PF is known to be *consistent* for FK models asymptotically: as the number N of simulated instances (*particles*) tends to infinity, the distribution of these particles converges to the measure defined by the FK model [19, Ch.11]. Therefore, consistency of PF for PPGs will automatically follow.

Our approach yields additional insights. First, the finite approximation theorem holds for a class of *prefix-closed functions* defined on infinite traces: these are the functions where the output only depends on a finite initial segment of the input argument. They can be viewed as a generalization of the prefix-closed sets involved in the definition of Safety properties in model checking [5]. The finite approximation theorem implies that the expectation of a prefix-closed function, defined on the probability space of infinite traces, can be approximated by the expectation of functions defined over truncated traces, with respect to a measure defined on a suitable FK model. As expectation in a FK model can be effectively estimated, via PF or other algorithms, our finite approximation result lays a sound basis for the statistical model checking of PPs. Second, the automata-theoretic operational semantics of PPGs translates into a *vectorized* implementation of PF, leveraging the fine-grained, SIMD parallelism exisisting at the level of particles. Specifically, the transition function and the score functions are applied simultaneously to the entire vector of N simulated particles at each step. This is practically significant, as modern CPUs and programming languages offer extensive support for vectorization, that may lead to dramatic speedups. We demonstrate this with a prototype vectorized implementation of a PPG-based PF algorithm using TensorFlow [1], called VPF. Experiments comparing VPF with state-of-the-art PPLs on challenging examples from the literature show very promising results (Section 6). Concluding remarks are provided in the final section (Section 7). Omitted proofs and additional technical material have been reported in an extended version available online [14].

In summary, our main contributions are as follows: (1) A clean semantics for PPGs based on expectation taken over infinite-trace, which incorporates conditioning/reweighting; (2) a finite approximation theorem linking this semantics to finite traces and FK models, thereby establishing the consistency of PF for PPGs; (3) a vectorized version of the PF algorithm based on PPGs.

Related work The book [7] is an introduction to and survey of recent literature on PPLs. A more concise introduction can be found in the review article [26]. With few notable exceptions, most work on the semantics of PPL follows the denotational approach initiated by Kozen [32]. This includes, among others, the works of Borgström, Gordon et al. [16], Staton et al. [45, 46] and Scibior et al. [44]. In this context, a general goal here is devising denotational semantics driven methods to combine and reason on densities, such as in [11, 27, 28, 48]. This goal is quite orthogonal to what we do here; in particular, we do not require that a PP induces a density on the probability space of infinite traces.

Relevant to our approach is a series of works by Lunden et al. on SMC inference applied to PPLs. In [34], for a lambda-calculus enriched with an explicit resample primitive, consistency of PF is shown to hold, under certain restrictions, independently of the placements of the resample's in the code. Operationally, their functional approach is very different from our automata-theoretic one. In particular, they handle suspension and resumption of particles in correspondence of resampling via an implicit use of *continuations*, in the style of webPPL [25] and other PPLs. The combination of functional style and continuations does not naturally lend itself to vectorization. For instance, ensuring that all particles are *aligned*, that is are at a resample point of their execution, is an issue that can impact negatively on performance or accuracy. On the contrary, in our automata-theoretic model, placement of resamples and alignment are not issues: resampling always happens after each (vectorized) transition step, so all particles are automatically aligned. Note that in PPGs a transition can group together complicated, conditioning free computations; in any case, consistency of PF is guaranteed. In a subsequent work [35, 36], Lunden al. study concrete implementation issues of SMC. In [35], they consider *PPL Control-Flow Graphs* (PCFGs), a structure intended as a target for the compilation of high-level PPLs, such as their CorePPL. The PCFG model is very similar in spirit to PPGs, however, it lacks a formal semantics. Lunden et al. also offer an implementation of this framework, designed to take advantage of the potential parallelism existing at the level of particles. We compare our implementation with theirs in Section 6.

PCFGs are also considered as the basic operational model in a series of works by K. Chatterjee et al., see e.g. [18] and references therein. The operational semantics they consider is similar to ours, but they do not consider an expected-based semantics, where we incorporate weights. More generally, in their works emphasis is on verification, like obtaining certified bounds on the termination probability of a given program. Issues related to conditioning and to consistency of sampling algorithms are not considered. Wang et al. [49] also consider a model very similar to PCFGs, but their focus is on exact/symbolic techniques (see further below).

Adiyta et al. prove consistency of Markov Chain Monte Carlo (MCMC) for their PPL R2 [40], which is based on a big-step sampling semantics that considers finite execution paths. No approximation results bridging finite and infinite traces hence unbounded loops, is provided. It is also unclear if a big-step semantics would effectively translate into a SIMD-parallel algorithm. Wu et al. [51] provide the PPL Blog with a rigorous measure-theoretic semantics, formulated in terms of Bayesian Networks, and a very efficient implementation of the PF algorithm tailored to such networks. Again, they do not offer results for unbounded loops. In our previous work [13], we have considered a measure theoretic semantics for a PPL with unbounded loops, and provided a finite approximation result and a SIMD-parallel implementation, with guarantees, of what is in effect a *rejection sampling* algorithm. Rejection may be effective for limited forms of conditioning; but it rapidly becomes wasteful and ineffective as conditioning becomes more demanding, so to speak: e.g. when it is repeated in a loop, or the observed data have a low likelihood in the model. Finally, SMCP3 [33] provides a rich measure-theoretic framework for extending the practical Gen language [20] with expressive proposal distributions.

A rich area in the field of PPL focuses on symbolic, exact techniques [43, 23, 38, 8, 10, 42, 29] aiming to obtain termination certificates, or certified bounds on termination probability of PPs, or even exact representations the posterior distribution; see also [6, 9, 47, 3, 31, 49] for some recent works in this direction. Our goal and methodology, as already stressed, are rather different, as we focus on inference via sampling and the ensuing consistency issues. While certificates in the traditional sense are impossible to obtain in our framework, we offer a principled methodology of statistical inference that is much more scalable.

2 Preliminaries on measure theory

We review a few basic concepts from measure theory following closely the presentation in the first two chapters of [2], which is a reference for whatever is not explicitly described below. Given a nonempty set Ω, a *sigma-field* $\mathcal{F}$ on Ω is a collection of subsets of Ω that contains Ω, and is closed under complement and under countable disjoint union. The pair $(\Omega, \mathcal{F})$ is called a *measurable space*. A (total) function $f : \Omega_1 \to \Omega_2$ is *measurable* w.r.t. the sigma-fields $(\Omega_1, \mathcal{F}_1)$ and $(\Omega_2, \mathcal{F}_2)$ if whenever $A \in \mathcal{F}_2$ then $f^{-1}(A) \in \mathcal{F}_1$. We let $\overline{\mathbb{R}} = \mathbb{R} \cup \{-\infty, +\infty\}$ be the set of extended reals, assuming the standard arithmetic for $\pm\infty$ (cf. [2, Sect.1.5.2]), and $\overline{\mathbb{R}}^+$ the set of nonnegative reals including $+\infty$. The *Borel sigma-field* $\mathcal{F}$ on $\Omega = \overline{\mathbb{R}}^m$ is the minimal sigma-field that contains all rectangles of the form $[a_1, b_1] \times \cdots \times [a_n, b_n]$, with $a_i, b_i \in \overline{\mathbb{R}}$. An important case of measurable spaces $(\Omega, \mathcal{F})$ is when $\Omega = \overline{\mathbb{R}}^m$ for some $m \geq 1$ and $\mathcal{F}$ is the Borel sigma-field over Ω. Throughout the paper, *"measurable" means "Borel measurable"*, both for sets and for functions. On functions, Borel measurability is preserved by composition and other elementary operations on functions; continuous real functions are Borel measurable. We will let $\mathcal{F}_k$ denote the Borel sigma-field over $\overline{\mathbb{R}}^k$ $(k \geq 1)$ when we want to be specific about the dimension of the space.

A *measure* over a measurable space $(\Omega, \mathcal{F})$ is a function $\mu : \mathcal{F} \to \overline{\mathbb{R}}^+$ that is countably additive, that is $\mu(\cup_{j\geq 1} A_j) = \sum_{j\geq 1} \mu(A_j)$ whenever A_j's are pairwise disjoint sets in $\mathcal{F}$. The *Lebesgue integral* of a Borel measurable function f w.r.t. a measure μ [2, Ch.1.5], both defined over a measure space $(\Omega, \mathcal{F})$, is denoted by $\int_\Omega \mu(d\omega) f(\omega)$, with the subscript Ω omitted when clear from the context. When μ is the standard Lebesgue measure, we may omit μ and write the integral as $\int_\Omega d\omega f(\omega)$. For $A \in \mathcal{F}$, $\int_A \mu(d\omega) f(\omega)$ denotes $\int_\Omega \mu(d\omega) f(\omega) 1_A(\omega)$, where $1_A(\cdot)$ is the indicator function of the set A. We let δ_v denote Dirac's measure concentrated on v: for each set A in an appropriate sigma-field, $\delta_v(A) = 1$ if $v \in A$, $\delta_v(A) = 0$ otherwise. Otherwise said, $\delta_v(A) = 1_A(v)$. Another measure that arises (in connections with discrete distributions) is the counting measure, $\mu_C(A) := |A|$. In particular, for a nonnegative f, we have the equality $\int_A \mu_C(d\omega) f(\omega) = \sum_{\omega\in A} f(\omega)$. A *probability measure* is a measure μ defined on $\mathcal{F}$ such that $\int \mu(du) = 1$. For a given nonnegative measurable function f defined over Ω, its *expectation* w.r.t. a probability measure ν is just its integral: $\mathrm{E}_\nu[f] = \int \nu(d\omega) f(\omega)$. The following definition is central.

Definition 1 (Markov kernel). *Let $(\Omega_1, \mathcal{F}_1)$ and $(\Omega_2, \mathcal{F}_2)$ be measurable spaces. A function $K : \Omega_1 \times \mathcal{F}_2 \longrightarrow \overline{\mathbb{R}}^+$ is a* Markov kernel *from Ω_1 to Ω_2 if it satisfies the following properties:*

1. *for each $\omega \in \Omega_1$, the function $K(\omega, \cdot) : \mathcal{F}_2 \to \overline{\mathbb{R}}^+$ is a probability measure on $(\Omega_2, \mathcal{F}_2)$;*
2. *for each $A \in \mathcal{F}_2$, the function $K(\cdot, A) : \Omega_1 \to \overline{\mathbb{R}}^+$ is measurable.*

Notationally, we will most often write $K(\omega, A)$ as $K(\omega)(A)$. The following is a standard result about the construction of finite product of measures over a product space[1] $\Omega^t = \Omega \times \cdots \times \Omega$ (t times) for $t \geq 1$ an integer. It is customary to denote the measure μ^t defined by the theorem also as $\mu^1 \otimes K_2 \otimes \cdots \otimes K_t$.

Theorem 1 (product of measures, [2],Th.2.6.7). *Let $t \geq 1$ be an integer. Let μ^1 be a probability measure on Ω and $K_2, ..., K_t$ be $t-1$ (not necessarily distinct) Markov kernels from Ω to Ω. Then there is a unique probability measure μ^t defined on $(\Omega^t, \mathcal{F}^t)$ such that for every $A_1 \times \cdots \times A_t \in \mathcal{F}^t$ we have: $\mu^t(A_1 \times \cdots \times A_t) = \int_{A_1} \mu^1(d\omega_1) \int_{A_2} K_2(\omega_1)(d\omega_2) \cdots \int_{A_t} K_t(\omega_{t-1})(d\omega_t)$.*

3 Probabilistic programs

We first introduce a general formalism for specifying programs, in the form of certain graphs that can be regarded as symbolic finite automata. For this formalism, we introduce an operational semantics in terms of Markov kernels.

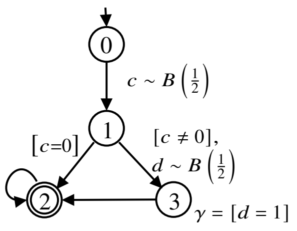
Probabilistic Program Graphs In defining probabilistic programs, we will rely on a repertoire of basic distributions: continuous, discrete and mixed distributions will be allowed. A crucial point for expressiveness is that a measure may depend on *parameters*, whose value at runtime is determined by the state of the program. To ensure that the resulting programs define measurable functions (on a suitable space), it is important that the dependence between the parameters and the measure be in turn of measurable type. We will formalize this in terms of Markov kernels. Additionally, we will consider score functions, a generalization of 0/1-valued predicates. Formally, we will consider the two families of functions defined below. In the definitions, we will let $m \geq 1$ denote a fixed integer, representing the number of *variables* in the program, conventionally referred to as $x_1, ..., x_m$. We will let v range over $\overline{\mathbb{R}}^m$, the content of the program variables in a given state, or *store*.

- *Parametric measures*: Markov kernels $\zeta : \overline{\mathbb{R}}^m \times \mathcal{F}_m \longrightarrow [0, 1]$.
- *Score functions*: measurable functions $\gamma : \overline{\mathbb{R}}^m \longrightarrow [0, 1]$. A *predicate* is a special case of a score function $\varphi : \overline{\mathbb{R}}^m \longrightarrow \{0, 1\}$. An Iverson bracket style notation will be often employed, e.g.: $[x_1 \geq 1]$ is the predicate that on input v yields 1 if $v_1 \geq 1$, 0 otherwise.

For a parametric measure ζ and a store $v \in \overline{\mathbb{R}}^m$, $\zeta(v)$ is a distribution, that can be used to sample a new store $v' \in \overline{\mathbb{R}}^m$ depending on the current program store v. Analytically, ζ may be expressed by, for instance, chaining together sampling of individual components of the store. This can be done by relying on *parametric densities*: measurable functions $\rho : \overline{\mathbb{R}}^m \times \overline{\mathbb{R}} \longrightarrow \overline{\mathbb{R}}^+$ such that, for a designated measure μ_ρ, the function $(v, A) \mapsto \int_A \mu_\rho(dr)\, \rho(v, r)$ $(A \in \mathcal{F}_m)$ is a Markov kernel from $\overline{\mathbb{R}}^m$ to $\overline{\mathbb{R}}$. This is explained via the following example.

[1] We shall freely identify language-theoretic *words* with *tuples*, hence use the notations $A_1 \cdot A_2 \cdot \cdots \cdot A_k$ and $A_1 \times A_2 \times \cdots \times A_k$ interchangeably. This convention will also apply to infinite words (cf. Section 4).

```
c~B(1/2);
if (c==0) skip
else {
d~B(1/2);
observe(d==1);
skip
}
```


0
$c \sim B\left(\frac{1}{2}\right)$
1
$[c=0]$
$[c \neq 0]$,
$d \sim B\left(\frac{1}{2}\right)$
2
3
$\gamma = [d = 1]$

```
d~U(0,2); r~U(0,1);
x:=-1; y:=1;
while(|x-y|≥.1) {
x~N(x,d); y~N(x,r);
observe(|x-y|≤3);
}
```

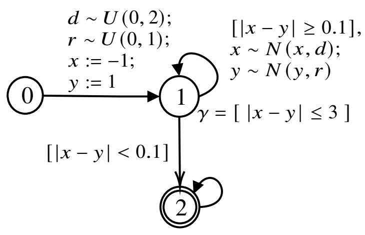


Fig. 1: **Left**. The PPG of Example 2 and a corresponding pseudo-code. The nil node (2) is distinguished with a double border. Constant 1 predicates and score functions, and the identity function are not displayed in transitions. The score function γ decorates node 3, that is $\mathbf{sc}(3) = \gamma$. **Right**. The PPG for the drunk man and mouse random walk of Example 3 and a corresponding pseudo-code. The score function γ decorates node 1, that is $\mathbf{sc}(1) = \gamma$.

Example 1. Fix $m = 2$. Consider the Markov kernel defined as follows, for each $x_1, x_2 \in \overline{\mathbb{R}}$ and $A \in \mathcal{F}_2$

$$\zeta(x_1,x_2)(A) := \int \mu_1(dr_1)\ \Big(\rho_1(x_1,x_2,r_1) \cdot \int \mu_2(dr_2)\rho_2(r_1,x_2,r_2)1_A(r_1,r_2)\Big) \tag{1}$$

where: $\mu_1 = \mu_C$ is the counting measure; $\rho_1(x_1,x_2,r) = \frac{1}{2}1_{\{x_1\}}(r) + \frac{1}{2}1_{\{x_2\}}(r)$ is the density of a discrete distribution on $\{x_1,x_2\}$; $\mu_2 = \mu_L$ is the ordinary Lebesgue measure; $\rho_1(x_1,x_2,r) = N(x_1,x_2,r) := \frac{1}{|x_2|\sqrt{2\pi}}\exp(-\frac{1}{2}(\frac{r-x_1}{|x_2|})^2)$ is the density of the Normal distribution of mean x_1 and standard deviation[2] $|x_2|$. The function ζ is a parametric measure: concretely, it corresponds to first sampling uniformly r_1 from the the set $\{x_1,x_2\}$, then sampling r_2 from the Normal distribution of mean r_1 and s.d. $|x_2|$ (if $|x_2|$ is positive and finite, otherwise from a default distribution). Rather than via (1), we will describe ζ via the following more handy notation:

$$r_1 \sim \rho_1(x_1,x_2)\ ;\ r_2 \sim \rho_2(r_1,x_2)$$

(or listed top-down). Note that the sampling order from left to right is relevant here.

In fact, as far as the formal framework of PPGs introduced below is concerned, how the parametric measures ζ's are analytically described is irrelevant. From the practical point of view, it is important we know how to (efficiently) sample from the measure $\zeta(v)$, for any v, in order for the inference algorithms to be actually implemented (see Section 5). In concrete terms, $\zeta(v)$ might represent the (possibly unknown) distribution of the outputs in $\overline{\mathbb{R}}^m$ returned by a piece of code, when invoked with input v. Another important special case of parametric measure is the following. For any $v = (v_1,...,v_m) \in \overline{\mathbb{R}}^m$, $r \in \overline{\mathbb{R}}$ and $1 \leq i \leq m$, let $v[r@i] := (v_1,...,r,...,v_m)$ denote the tuple where v_i has been replaced by r. Consider the parametric measure $\zeta(v) = \delta_{v[g(v)@i]}$, where $g : \overline{\mathbb{R}}^m \to \overline{\mathbb{R}}$ is a measurable function. In programming terms, this corresponds to the deterministic *assignment* of the value $g(v)$ to the variable x_i. We will describe this ζ as: $x_i := g(x_1,...,x_m)$.

In the definition of PPG below, one may think of the computation (sampling) taking place in successive stages on the edges (transitions) of the graph, with nodes serving as *checkpoints* (a term we have borrowed from [34]) between stages for conditioning on observed data — or, more generally, re-weighting the score assigned to a computation. The edges also account for the control flow among the different stages via predicates computed on the store of the source nodes.

Definition 2 (PPG). *Fix $m \geq 1$. A* Probabilistic Program Graph (PPG) *on $\overline{\mathbb{R}}^m$ is 4-tuple* $\mathbf{G} = (\mathcal{P}, E, \mathsf{nil}, \mathbf{sc})$ *satisfying the following.*

- $\mathcal{P} = \{S_1,...,S_k\}$ *is a finite, nonempty set of* program checkpoints *(*programs, *for short).*
- *E is a finite, nonempty set of* transitions *of the form (S,φ,ζ,S'), where: $S,S' \in \mathcal{P}$ are called the* source *and* target *program checkpoint, respectively; $\varphi : \overline{\mathbb{R}}^m \to \{0,1\}$ is a predicate; and $\zeta : \overline{\mathbb{R}}^m \times \mathcal{F}_m \to [0,1]$ is a parametric measure.*

[2] With the proviso that, when $x_2 = 0$ or $|x_1|, |x_2| = +\infty$, $N(x_1,x_2,r)$ denotes an arbitrarily fixed, default probability density.

– $\mathsf{nil} \in \mathcal{P}$ *is a distinguished* terminated *program checkpoint, such that* $(\mathsf{nil}, 1, \mathrm{id}, \mathsf{nil})$ *(*id = *identity) is the only transition in* E *with* nil *as source.*
– $\mathbf{sc}$ *is a mapping from* $\mathcal{P}$ *to the set of score functions, s.t.* $\mathbf{sc}(\mathsf{nil})$ *is the constant 1.*

Additionally, denoting by E_S *the set of transitions in* E *with* S *as a source checkpoint, the following* consistency *condition is assumed: for each* $S \in \mathcal{P}$*, the function* $\sum_{(S,\varphi,\zeta,S')\in E_S} \varphi$ *is the constant 1.*

We first illustrate Definition 2 with a simple example. This will also serve to illustrate the finite approximation theorem later on in this section.

Example 2. Consider the PPG in Fig. 1, left. Here we have $m = 2$ and $B(p)$ is the Bernoulli distribution with success probability p. On the left, a more conventional pseudo-code notation for the resulting program. We will not pursue a systematic formal translation from this program notation to PPGs, though.

The following example illustrate the use of scoring functions inside loops. It is a bit contrived, but close to the structure of more significant scenarios, such as the aircraft tracking example of [51], cf. Section 6.

Example 3 (Of mice and drunk men). Consider the following variation on the classical drunk man's random walk. On a street, a drunk man and a mouse perform independent random walks starting at conventional positions -1 and 1 respectively. Initially, each of them samples a value from a uniform distribution, to be used as a standard deviation (s.d.) of the length of subsequent steps: the drunk man samples d from $(0, 2)$, the mouse r from $(0, 1)$. Then, at each discrete time step, they independently sample their own next position from a Normal distribution centered at the current position (x man, y mouse), with the s.d. (d man, r mouse) chosen at the beginning. The process is stopped as soon as the man and the mouse meet, which we take to mean the distance between them is $< 1/10$.

The man and the mouse' actions are independent. On the other hand, it has been suggested that in certain urban areas a man is never more than 3m away from a mouse [37]. Let us take this information at face value, and incorporate it in our model: we let $\mathbf{sc}(\cdot)$ associate the appropriate checkpoint in the graph with the score function $\gamma := [|x - y| \leq 3]$ — actually a predicate written in Iverson bracket notation. The resulting PPG, $\mathbf{G} = (\mathcal{P}, E, \mathsf{nil}, \mathbf{sc})$, has $m = 4$, three checkpoints and the transition structure described in Figure 1, right. A pseudo-code description is also showed.

Operational semantics of PPGs For any given PPG $\mathbf{G}$, we will define a Markov kernel $\kappa_{\mathbf{G}}(\cdot, \cdot)$ that describes its operational semantics. From now on, we will consider one arbitrarily fixed PPG, $\mathbf{G} = (\mathcal{P}, E, \mathsf{nil}, \mathbf{sc})$ and just drop the subscript $_{\mathbf{G}}$ from the notation. Let us also remark that the scoring function $\mathbf{sc}(\cdot)$ will play no role in the definition of the Markov kernel — it will come into play in the trace based semantics of Section 4.

Some additional notational shorthand is in order. First, we identify $\mathcal{P}$ with the finite set of naturals $\{0, ..., |\mathcal{P}| - 1\}$. With this convention, we have that $\overline{\mathbb{R}}^m \times \mathcal{P} \subseteq \overline{\mathbb{R}}^{m+1}$. Henceforth, we define our state space and sigma-field as follows:

$$\Omega := \overline{\mathbb{R}}^{m+1} \qquad \mathcal{F} := \text{Borel sigma-field over } \overline{\mathbb{R}}^{m+1}.$$

We keep the symbol $\mathcal{F}_k$ for the Borel sigma-field over $\overline{\mathbb{R}}^k$, for any $k \geq 1$. For any $S \in \mathcal{P}$ and $A \in \mathcal{F}$, we let $A_S := \{v \in \overline{\mathbb{R}}^m : (v, S) \in A\}$ be the *section* of A at S. Note that $A_S \in \mathcal{F}_m$, as sections of measurable sets are measurable, see [2, Th.2.6.2,proof(1)].

Definition 3 (PPG Markov kernel). *The function* $\kappa : \Omega \times \mathcal{F} \longrightarrow \mathbb{R}^+$ *is defined as follows, for each* $\omega \in \Omega$ *and* $A \in \mathcal{F}$*:*

$$\kappa(\omega)(A) := \begin{cases} \delta_\omega(A) & \text{if } \omega \notin \overline{\mathbb{R}}^m \times \mathcal{P} \\ \sum_{(S,\varphi,\zeta,S')\in E_S} \varphi(v) \cdot \zeta(v)\,(A_{S'}) & \text{if } \omega = (v, S) \in \overline{\mathbb{R}}^m \times \mathcal{P}. \end{cases} \tag{2}$$

Lemma 1. *The function* κ *is a Markov kernel from* Ω *to* Ω*.*

4 Trace semantics and finite approximation for PPGs

Trace semantics In what follows, we fix an arbitrary a PPG, $\mathbf{G} = (\mathcal{P}, E, \mathsf{nil}, \mathbf{sc})$ and let κ denote the induced Markov kernel, as per Definition 3. For any $t \geq 1$, we call Ω^t the set of *paths of length t*. Consider now the set of paths of infinite length, Ω^∞, that is the set of infinite sequences $\tilde{\omega} = (\omega_1, \omega_2, ...)$ with $\omega_i \in \Omega$. For any $\omega^t \in \Omega^t$ and $\tilde{\omega} \in \Omega^\infty$, we identify the pair $(\omega^t, \tilde{\omega})$ with the element of Ω^∞ in which the prefix ω^t is followed by $\tilde{\omega}$. For $t \geq 1$ and a measurable $B_t \subseteq \Omega^t$, we let $\mathfrak{c}(B_t) := B_t \cdot \Omega^\infty \subseteq \Omega^\infty$ be the *measurable cylinder* generated by B_t. We let C be the minimal sigma-field over Ω^∞ generated by all measurable cylinders. Under the same assumptions of Theorem 1 on the measure μ^1 and on the kernels $K_2, K_3, ...$ there exists a unique measure μ^∞ on C such that for each $t \geq 1$ and each measurable cylinder $\mathfrak{c}(B_t)$, it holds that $\mu^\infty(\mathfrak{c}(B_t)) = \mu^t(B_t)$: see [2, Th.2.7.2], also known as the *Ionescu-Tulcea theorem*. In the definition below, we let $0 = (0, ..., 0)$ (m times) and consider $\delta_{(0,S)}$, the Dirac's measure on Ω that concentrates all the probability mass in $(0, S)$.

Definition 4 (probability measure induced by S). *Let $S \in \mathcal{P}$. For each integer $t \geq 1$, we let μ_S^t be the probability measure over Ω^t uniquely defined by Theorem 1(a) by letting $\mu^1 = \delta_{(0,S)}$ and $K_2 = \cdots = K_t = \kappa$. We let μ_S^∞ be the unique probability measure on C induced by μ_1 and $K_2 = \cdots = K_t = \cdots = \kappa$, as determined by the Ionescu-Tulcea theorem.*

In other words, $\mu_S^t = \delta_{(0,S)} \otimes \kappa \otimes \cdots \otimes \kappa$ ($t - 1$ times κ). By convention, if $t = 1$, $\mu_S^t = \delta_{(0,S)}$. The measure μ_S^∞ can be informally interpreted as the limit of the measures μ_S^t and represents the semantics of S.

The following is a general lemma useful to connect measure over sets of infinite and finite traces. In its statement, we let μ^∞ denote a generic measure on the cylindrical sigma-field, obtained as an infinite product of kernels in the sense of the Ionescu-Tulcea theorem, and by μ^t the corresponding finite product measures. We shall make use of the following notation. For $\tilde{\omega} = (\omega_1, \omega_2, ...) \in \Omega^\infty$, we let $\tilde{\omega}_{1:t} := (\omega_1, ..., \omega_t) \in \Omega^t$. For $h : \Omega^t \to \overline{\mathbb{R}}^+$ a nonnegative function, we let $\tilde{h} : \Omega^\infty \to \overline{\mathbb{R}}^+$ be defined as follows for each $\tilde{\omega} \in \Omega^\infty$:

$$\tilde{h}(\tilde{\omega}) := h(\tilde{\omega}_{1:t}) \,. \tag{3}$$

Lemma 2. *Let $h : \Omega^t \to \overline{\mathbb{R}}^+$ a nonnegative measurable function. Then $\tilde{h}$ as defined in (3) is measurable. Moreover, for each measurable cylinder $\mathfrak{c}(B_t) \subseteq \Omega^\infty$ ($B_t \subseteq \Omega^t$), we have $\int_{\mathfrak{c}(B_t)} \mu^\infty(d\tilde{\omega})\tilde{h}(\tilde{\omega}) = \int_{B_t} \mu^t(d\omega^t)h(\omega^t)$.*

Recall that the *support* of an (extended) real valued function f is the set $\mathrm{supp}(f) := \{z : f(z) \neq 0\}$. In what follows, *we shall concentrate on nonnegative measurable functions f* to avoid unnecessary complications with the existence of integrals. General functions can be dealt with by the usual trick of decomposing f as $f = f^+ - f^-$, where $f^+ = \max(0, f)$ and $f^- = -\min(0, f)$, and then dealing separately with f^+ and f^-. Let us introduce a *combined score function* $\mathrm{sc} : \Omega \to [0, 1]$ as follows, for each $\omega = (v, S)$:

$$\mathrm{sc}(\omega) := \begin{cases} \mathbf{sc}(S)(v) & \text{if } \omega = (v, S) \in \overline{\mathbb{R}}^m \times \mathcal{P} \\ 1 & \text{otherwise.} \end{cases} \tag{4}$$

The function $\mathrm{sc}(\cdot)$ is extended to a *weight function* on infinite traces, $\mathrm{w} : \Omega^\infty \to [0, 1]$ by letting[3], for any $\tilde{\omega} = (\omega_1, \omega_2, ...)$:

$$\mathrm{w}(\tilde{\omega}) := \Pi_{j \geq 1} \mathrm{sc}(\omega_j) \,. \tag{5}$$

For each $t \geq 1$, we define the weight function truncated at time t, $\mathrm{w}_t : \Omega^t \to [0, 1]$, by $\mathrm{w}_t(\omega^t) := \Pi_{j=1}^t \mathrm{sc}(\omega_j)$. Both w and w_t ($t \geq 1$) are measurable functions on the respective domains. We arrive at the definition of the semantics of programs. We consider the ratio of the unnormalized semantics ($[S]f$) to the weight of all traces, terminated or not ($[S]\mathrm{w}$). In the special case when the score functions represent conditioning, this choice corresponds to quotienting over the probability of *non failed* traces. In PPL, quotienting over non failed states is somewhat standard: see e.g. the discussion in [30, Section 8.3.2].

[3] Note that $\mathrm{w}(\tilde{\omega})$ is well-defined because $0 \leq \mathrm{sc}(\omega_j) \leq 1$ for each $j \geq 0$, hence the series of partial products is monotonically nonincreasing.

Definition 5 (trace semantics). *Let f be a nonnegative measurable function defined on Ω^∞. We let the* unnormalized semantics *of S and f be $[S]f := \mathrm{E}_{\mu_S^\infty}[f]$ $(= \int \mu_S^\infty(d\tilde{\omega})f(\tilde{\omega}))$. We let*

$$[\![S]\!]f := \frac{[S]f \cdot \mathrm{w}}{[S]\mathrm{w}} \qquad (6)$$

provided the denominator above is > 0; otherwise $[\![S]\!]f$ is undefined.

Finite approximation We are mainly interested in $[\![S]\!]f$ in cases where the value of f is, informally speaking, determined by a finite prefix of its argument: we call these functions *prefix-closed*, and will define them further below. We first have to introduce prefix-closed languages[4], for which some notation on languages of finite and infinite words is useful. Given two words $w, w' \in \Omega^*$, we write $w \prec w'$ if w is a prefix of w', i.e. there exists a word $w'' \in \Omega^*$ such that $ww'' = w'$; otherwise we write $w \nprec w'$. For $L, L' \subseteq \Omega^*$, we write $L \nprec L'$ if for any $w \in L$ and $w' \in L'$ we have $w' \nprec w'$. A sequence of languages $L_0, L_1, ...$ such that for each j, $L_j \subseteq \Omega^j$ (with $\Omega^0 := \{\epsilon\}$, the empty sequence) is said to be *prefix-free* if for each $i \neq j$, $L_i \nprec L_j$. Note that if $L_0 \neq \emptyset$ then $L_j = \emptyset$ for $j \geq 1$. For the sake of uniform notation, in what follows we convene that $\omega^0 := \epsilon$ and $\mathfrak{c}(\{\epsilon\}) := \Omega^\infty$. We say $A \subseteq \Omega^\infty$ is a *prefix closed set* if there is a prefix-free sequence of languages $L_0, L_1, ...$ such that $A = \cup_{j=0}^{\infty}\mathfrak{c}(L_j)$; we call L_j a *j-branch* of A, and refer to $L_0, L_1, ...$ collectively as *branches of A*. For any $t \geq 1$, we define the following subsets of Ω^t:

$$L^{\leq t} := \cup_{j=0}^{t} L_j \cdot \Omega^{t-j}, \qquad L^{>t} := \{\omega^t \ : \ \text{there is } t' > t \text{ and } \omega_{t'} \in L_{t'} \text{ s.t. } \omega^t \prec \omega^{t'}\}\,.$$

Informally speaking, $L^{\leq t}$ is the set of paths of length t that will become members of A however we extend them to infinite words. $L^{>t}$ is the set of paths of length t for which some infinite extensions, but not all, are in A — they are so to speak "undecided". Of special interest is the prefix-free sequence of languages defined below.

Definition 6 (termination). *Let* $\mathsf{T} := \overline{\mathbb{R}}^m \times \{\mathsf{nil}\}$ *be the set of* terminated *states. We let $T_j \subseteq \Omega^j$ $(j \geq 0)$ be the set of finite sequences that* terminate at time j*, that is: $T_0 := \emptyset$ and $T_j := (\mathsf{T}^c)^{j-1} \cdot \mathsf{T}$, for $j \geq 1$. We let $T_\mathrm{f} := \cup_{t\geq 0}\mathfrak{c}(T_t) \subseteq \Omega^\infty$ denote the set of infinite sequences that* terminate in finite time.

Note that $\{T_j : j \geq 0\}$ forms a prefix-free sequence, that $T^{\leq t} \subseteq \Omega^t$ is the set of all paths of length t that terminate within time t, while $\mathfrak{c}(T_t) \subseteq \Omega^\infty$ is the set of infinite execution paths with termination at time t.

The next definition introduces prefix-closed functions. These are functions f with a prefix-free support, condition (a), additionally satisfying two extra conditions. Condition (b) just states that the value of f on its support is determined by a finite prefix of the input sequence. Condition (c), T-respectfulness, means that a trace that terminates at time j ($\omega^j \in T_j$) cannot lead to $\mathrm{supp}(f)$ at a later time ($\omega^j \notin L^{>j}$). This is a consistency condition, formalizing that the value of f does not depend on, so to speak, what happens *after* termination.

Definition 7 (prefix-closed function). *Let $f : \Omega^\infty \to \overline{\mathbb{R}}^+$ be a nonnegative measurable function and $(L_0, L_1, ...)$ be a prefix-closed sequence. We say f is a* prefix-closed function *with branches $L_0, L_1, ...$ if the following conditions are satisfied.*

(a) $\mathrm{supp}(f)$ *is prefix-free with branches L_j $(j \geq 0)$.*
(b) for each $j \geq 0$ and $\omega^j \in L_j$, f is constant on $\mathfrak{c}(\{\omega^j\})$.
(c) $\mathrm{supp}(f)$ *is* T-respectful*: for each $j \geq 0$, $L^{>j} \cap T_j = \emptyset$.*

Note that there may be different prefix-free sequences w.r.t. which f is prefix-closed.

Example 4. The indicator function 1_{T_f} is clearly a prefix closed, measurable function with $\mathrm{supp}(1_{T_\mathrm{f}}) = T_\mathrm{f}$ and branches $L_j = T_j$. For more interesting examples, consider the PPG in Example 3 and the following functions.

[4] In the context of model checking, these languages arise as complements of Safety properties; see e.g. [5, Def.3.22].

– $f_1(\tilde{\omega}) = j$ if $\omega_j \in \mathsf{T}$ is the first terminated state occurring in $\tilde{\omega}$, if such a ω_j exists; 0 otherwise.
– $f_2(\tilde{\omega}) = d$ if $\omega = (d, r, x, y, \mathsf{nil}) \in \mathsf{T}$ is the first terminated state occurring in $\tilde{\omega}$ and $d \in [0, 2]$, if such a ω_j exists; 0 otherwise.

f_1 returns the time the process terminates: $\mathrm{supp}(f_1) = T_{\mathrm{f}}$ has branches $L_j = T_j$ ($j \geq 0$). f_2 returns the value of d at termination. Here $\mathrm{supp}(f_2) = \{\tilde{\omega} \in T_{\mathrm{f}}$: the first terminated state ω in $\tilde{\omega}$, if it exists, has $\omega(1) = d \in (0, 2]\}$, and $L_0 = \emptyset$, $L_j = (\mathsf{T}^{\mathrm{c}})^{j-1} \cdot (\mathsf{T} \cap ((0, 2] \times \overline{\mathbb{R}}^4))$ ($j \geq 1$).

We will now study how to consistently approximate infinite computations (μ_S^∞ semantics) with finite ones (μ_S^t semantics). This will lead to the main result of this section (Theorem 2). As a first step, let us introduce an appropriate notion of finite approximation for functions f defined on the infinite product space Ω^∞. Fix an arbitrary element $\star \in \Omega$. For each $f : \Omega^\infty \to \overline{\mathbb{R}}^+$ and $t \geq 1$, let us define the function $f_t : \Omega^t \to \overline{\mathbb{R}}^+$ by

$$f_t(\omega^t) := f(\omega^t, \star^\infty) .$$

The intuition here is that, for a prefix-closed function f, the function f_t approximates correctly f for all finite paths in the L_j-branches of f, for $j \leq t$. Consider for instance the function $f = f_1$ in Example 4. On $L^{\leq t}$, the approximation f_t gives the correct value w.r.t. f in a precise sense: $f_t(\omega^t) = f(\omega^t, \star^\infty) = f(\omega^t, \tilde{\omega}')$ whatever $\star$ and $\tilde{\omega}'$. On the other hand, for finite paths $\omega^t \in L^{>t}$, f_t may not approximate f correctly: we may have $f_t(\omega^t) = f(\omega^t, \star^\infty) \neq f(\omega^t, \tilde{\omega}')$ depending on the specific $\star$ and $\tilde{\omega}'$. The catch is, as t grows large, the set $L^{>t}$ will become thinner and thinner — at least under reasonable assumptions on the measure μ_S^∞.

It is not difficult to check that, for any t, f_t is measurable over Ω^t.The next result shows how to approximate $[[S]]f$ with quantities defined *only in terms of* f_t, w_t *and* μ_S^t, which is the basis for the sampling-based inference algorithm in the next section. Formally, for $t \geq 1$ and a measurable function $h : \Omega^t \to \overline{\mathbb{R}}^+$, we let

$$[S]^t h := \mathrm{E}_{\mu_S^t}[h] \ (= \textstyle\int \mu_S^t(d\omega^t) h(\omega^t)) .$$

The finite approximation theorem for $[[S]]f$ relies on the following approximation lemma for the unnormalized semantics. The intuitive content of the lemma is as follows. Consider a prefix closed function f with branches $L_0, L_1, \ldots$. For any time t, it is not difficult to see that $\mathfrak{c}(L^{\leq t} \cap T^{\leq t}) \subseteq \mathrm{supp}(f) \subseteq \mathfrak{c}(L^{\leq t} \cap T^{\leq t}) \cup (\mathfrak{c}(T^{\leq t}))^{\mathrm{c}}$ (the last inclusion involves T-respectfulness). As f_t approximates correctly f on $L^{\leq t}$ one sees that the first inclusion leads to the lower bound $[S]^t f_t \cdot 1_{L^{\leq t} \cap T^{\leq t}} \cdot \mathrm{w}_t \leq [S] f \mathrm{w}$. As for the upper bound, the intuition is that, over $(\mathfrak{c}(T^{\leq t}))^{\mathrm{c}}$, f is upper-bounded by M.

Lemma 3. *Let $t \geq 1$ and let $f \leq M$ be a prefix-closed function with branches L_j ($j \geq 0$). Then*

$$[S]^t f_t \cdot 1_{L^{\leq t} \cap T^{\leq t}} \cdot \mathrm{w}_t \leq [S] f \cdot \mathrm{w} \leq [S]^t f_t \cdot 1_{L^{\leq t} \cap T^{\leq t}} \cdot \mathrm{w}_t + M \cdot [S]^t (1 - 1_{T^{\leq t}}) \cdot \mathrm{w}_t . \tag{7}$$

The proof of the main result follows by applying the above lemma to the numerators and denominators of the expressions involved in (8). In the formulation of the upper bound, we find it convenient to introduce a 'correction factor' $\alpha_t \geq 1$, the ratio of the weight of *all* traces to *terminated* traces at time t. We premise a technical lemma on convergence of integrals.

Lemma 4. *Let $S \in \mathcal{P}$. As $t \to +\infty$, we have $[S]^t 1_{\mathsf{T}^{\leq t}} \cdot \mathrm{w}_t \longrightarrow [S] 1_{\mathsf{T}_{\mathrm{f}}} \cdot \mathrm{w}$. Moreover, the sequence $[S]^t 1_{\mathsf{T}^{\leq t}} \cdot \mathrm{w}_t$ ($t \geq 1$) is monotonically nondecreasing.*

Theorem 2 (finite approximation). *Consider $S \in \mathcal{P}$ and $t \geq 1$ such that $[S]^t 1_{T^{\leq t}} \cdot \mathrm{w}_t > 0$. Then for any prefix-closed function f with branches $L_0, L_1, \ldots$ we have that $[[S]]f$ is well defined. Moreover, given an upper bound $f \leq M$ ($M \in \overline{\mathbb{R}}^+$), for each t large enough and $\alpha_t := \frac{[S]^t \mathrm{w}_t}{[S]^t 1_{T^{\leq t}} \cdot \mathrm{w}_t}$ we have:*

$$\frac{[S]^t f_t \cdot 1_{L^{\leq t} \cap T^{\leq t}} \cdot \mathrm{w}_t}{[S]^t \mathrm{w}_t} \leq [[S]] f \leq \frac{[S]^t f_t \cdot 1_{L^{\leq t} \cap T^{\leq t}} \cdot \mathrm{w}_t}{[S]^t \mathrm{w}_t} \alpha_t + M \cdot (\alpha_t - 1) . \tag{8}$$

Proof. We first show that $[[S]]f$ is well defined, that is that $[S]\mathrm{w} > 0$. Indeed, from Lemma 4, and from $[S]^t 1_{\mathsf{T}^{\le t}} \cdot \mathrm{w}_t > 0$ for at least one t, we get $[S]1_{\mathsf{T}_\mathrm{f}} \cdot \mathrm{w} > 0$; since $1_{\mathsf{T}_\mathrm{f}} \cdot \mathrm{w} \le \mathrm{w}$, we get $[S]1_{\mathsf{T}_\mathrm{f}} \cdot \mathrm{w} \le [S]\mathrm{w}$, hence the wanted statement.

Now consider $[[S]]f = \dfrac{[S]f \cdot \mathrm{w}}{[S]\mathrm{w}}$, for f like in the hypothesis, and the inequalities in (8). Consider the following bounds for the numerator and denominator of this fraction.

$$[S]^t f_t \cdot 1_{L^{\le t} \cap T^{\le t}} \cdot \mathrm{w}_t \le [S]f \cdot \mathrm{w} \le [S]^t f_t \cdot 1_{L^{\le t} \cap T^{\le t}} \cdot \mathrm{w}_t + M([S]^t \mathrm{w}_t - [S]^t 1_{T^{\le t}} \cdot \mathrm{w}_t) \tag{9}$$

$$[S]^t 1_{T^{\le t}} \cdot \mathrm{w}_t \le [S]\mathrm{w} \le [S]^t \mathrm{w}_t\,. \tag{10}$$

The bounds in (9) are just those in Lemma 3, with the term $M \cdot (\cdots)$ written in an equivalent form. As to (10), first apply the bounds of Lemma 3 to the constant function $f = 1$. Note that this f is measurable, and is trivially prefix closed for the prefix-free sequence of languages $L_0 = \{\epsilon\}$ and $L_j = \emptyset$ for $j > 0$. As a consequence, for $t \ge 1$, over Ω^t we have $L^{\le t} = \Omega^t$, hence $1_{L^{\le t} \cap T^{\le t}} = 1_{T^{\le t}}$. Moreover $f_t = M = 1$ identically. From these facts, it is immediate to see that the bounds (7) of Lemma 3 specialize to (10). From the above established bounds (9) and (10) for the numerator and denominator of $[[S]]f = \frac{[S]f \cdot \mathrm{w}}{[S]\mathrm{w}}$, it follows that

$$\frac{[S]^t f_t \cdot 1_{L^{\le t} \cap T^{\le t}} \cdot \mathrm{w}_t}{[S]^t \mathrm{w}_t} \le [[S]]f \le \frac{[S]^t f_t \cdot 1_{L^{\le t} \cap T^{\le t}} \cdot \mathrm{w}_t}{[S]^t 1_{T^{\le t}} \cdot \mathrm{w}_t} + M \cdot \left(\frac{[S]^t \mathrm{w}_t}{[S]^t 1_{T^{\le t}} \cdot \mathrm{w}_t} - 1 \right). \tag{11}$$

Now, multiplying and dividing the first term of the above upper bound by $[S]^t \mathrm{w}_t$, positive by hypothesis, and recalling the definition of α_t, the wanted (8) follows.

When f is an indicator function, $f = 1_A$, we can of course take $M = 1$ in the theorem above. We first illustrate the above result with a simple example.

Example 5. Consider the PPG of Example 2 (Fig. 1, left). We ask what is the expected value of c upon termination of this program. Formally, we consider the program checkpoint $S = 0$, and the function f on traces that returns the value of c on the first terminated state, if any, and 0 elsewhere. This f is clearly prefix-closed with branches $L_j \subseteq T_j$. We apply Theorem 2 to $[[S]]f$. Fixing the time $t = 4$, we can calculate easily the quantities involved in the approximation of $[[S]]f$ in (8). In doing so, we must consider the finitely many paths of length t of nonzero probability and weight (there only two of them), their weights and the value of c on their final state when terminated[5].

$$[S]^t f_t \cdot 1_{T^{\le t}} \cdot \mathrm{w}_t = 0 \cdot \tfrac{1}{2} + 1 \cdot \tfrac{1}{2} \cdot \tfrac{1}{2} = \tfrac{1}{4} \qquad [S]^t \mathrm{w}_t = \tfrac{1}{2} + \tfrac{1}{2} \cdot \tfrac{1}{2} = \tfrac{3}{4}$$
$$[S]^t 1_{T^{\le t}} \cdot \mathrm{w}_t = \tfrac{1}{2} + \tfrac{1}{2} \cdot \tfrac{1}{2} = \tfrac{3}{4} \qquad \alpha_t = 1\,.$$

Then, with $M = 1$, the lower and upper bounds in (8) coincide and yield $[[S]]f = \frac{1}{3}$. If we remove conditioning on node 4, then all the paths of length t have weight 1, and a similar calculation yields $[[S]]f = \frac{1}{2}$.

In more complicated cases, we may not be able to calculate exactly the quantities involved in (8), but only to estimate them via sampling. To this purpose, we will introduce Feynman-Kac models and the Particle Filtering algorithm in the next section. For now, we content ourselves with the following example.

Example 6. Consider the PPG of Example 3 and $f = f_2$ from Example 4. Take $S = 0$. Then $[[S]]f$ is the posterior expectation of the value of d, the drunk man's standard deviation. We can compute upper and lower bounds on $[[S]]f$ using (8). Let us fix $t = 60$. By sampling from μ_S^t, we can compute separately the following estimates for each of the expected values involved in (8): $[S]^t f_t \cdot 1_{L^{\le t} \cap T^{\le t}} \cdot \mathrm{w}_t = 0.396$, $[S]^t \mathrm{w}_t = 0.987$ and $\alpha_t = 1.497$. Combining these estimates as in (8), with $M = 2$, we get the bounds: $0.402 \le [[S]]f \le 1.596$. This relatively large interval can be narrowed down by considering higher values of t, hence α_t closer to 1. A more efficient and accurate method to compute the bounds in (8) will be introduced in Section 5, the Particle Filtering algorithm.

The theorem below confirms that the bounds established above are asymptotically tight, at least under the assumption that the program $S \in \mathcal{P}$ terminates with probability 1. In this case, in fact, the probability mass outside $\mathsf{T}^{\le t}$ tends to 0, which leads the lower and the upper bound in (8) to coincide. Moreover, we get a simpler formula in the special case when termination is guaranteed to happen within a fixed time limit; for instance, in the case of acyclic[6] PPGs.

[5] Here, we also use the fact that $f_t \cdot 1_{T^{\le t}} = f_t \cdot 1_{L^{\le t} \cap T^{\le t}}$, a consequence of $L_j \subseteq T_j$ for all js.

[6] Or, more accurately, PPGs where the only loop is the self-loop on the nil state.

Theorem 3 (tightness). *Assume the same hypotheses as in Theorem 2. Further assume that $\mu_S^\infty(T_\mathrm{f}) = 1$. Then both the lower and the upper bounds in* (8) *tend to $[\![S]\!]f$ as $t \to +\infty$. In particular, if for some $t \geq 1$ we have $[S]^t 1_{T^{\leq t}} = 1$, then*

$$[\![S]\!]f = \frac{[S]^t f_t \cdot \mathrm{w}_t}{[S]^t \mathrm{w}_t}\,. \tag{12}$$

Example 7. For the PPG of Example 2 one has $\mu_S^\infty(T_\mathrm{f}) = 1$. As already seen in Example 2, lower and upper bounds coincide for $t \geq 4$.

A practically relevant class of closed prefix functions are those where the result $f(\tilde{\omega})$ only depends on computing a function h, defined on Ω, on the first terminated state, if any, of the sequence $\tilde{\omega}$. This way h is *lifted* to Ω^∞. This case covers all the examples seen so far. We formally introduce lifting below. Recall that for $t \geq 1$, $T_t = (\mathsf{T}^\mathrm{c})^{t-1} \cdot \mathsf{T}$.

Definition 8 (lifting). *Let $h : \Omega \to \overline{\mathbb{R}}^+$ a nonnegative measurable function such that* $\mathrm{supp}(h) \subseteq \mathsf{T}$. *The* lifting *of h is the measurable function $\check{h} : \Omega^\infty \to \overline{\mathbb{R}}^+$ defined as follows for each $\tilde{\omega} = (\omega_1, \omega_2, \ldots)$: $\check{h}(\tilde{\omega}) := \sum_{t\geq 1} 1_{\mathrm{c}(T_t)}(\tilde{\omega}) \cdot h(\omega_t)$.*

Clearly, any $\check{h}$ is prefix closed with branches $L_0 = \emptyset$ and $L_j = (\mathsf{T}^\mathrm{c})^{j-1} \cdot \mathrm{supp}(h) \subseteq T_j$ for $j \geq 1$. In particular, $\mathrm{supp}(\check{h}) \subseteq T_\mathrm{f}$. As an example, the indicator function for the set of paths that eventually terminate, $\check{h} = 1_{T_\mathrm{f}}$, is clearly the lifting of $h = 1_\mathsf{T}$; the functions f_1, f_2 in Example 4 can also be obtained by lifting (details omitted). More complicated functions, that also look at sequences of events before termination, can often be encoded as lifted functions at the cost of introducing in the program extra variables to keep track of those events in the store.

5 Feynman-Kac models

In the field of Sequential Monte Carlo methods, Feynman-Kac (FK) models [19, Ch.9] are characterized by the use of *potential* functions. A potential in a Feynman-Kac model is a function that assigns a weight $G_t(x)$ to a *particle* (instance of a random process) in state x at time t. This weight represents how plausible or fit x is at time t based on some observable or conditioning. In other words, G_t modifies the *importance* of particles as the system evolves. For instance, in a model for tracking an object, the potential function could depend on the distance between the predicted particle position and the actual observed position. Particles closer to the observed position get higher weights.

FK models and probabilistic program semantics We first introduce FK models in a general context. Our formulation follows closely [19, Ch.9]. Throughout this and the next section, we let $t \geq 1$ be an arbitrary fixed integer.

Definition 9 (Feynman-Kac models). *A* Feynman-Kac (FK) model *is a tuple* FK $= (\mathcal{X}, t, \mu^1, \{K_i\}_{i=2}^t, \{G_i\}_{i=1}^t)$*, where $\mathcal{X} = \overline{\mathbb{R}}^\ell$ for some $\ell \geq 1$, μ^1 is a probability measure on $\mathcal{X}$ and, for $i = 2, \ldots, t$: K_i is a Markov kernel from $\mathcal{X}$ to $\mathcal{X}$, and $G_i : \mathcal{X} \to \overline{\mathbb{R}}^+$ is a measurable function.*

Let μ^t denote the unique product measure on $\mathcal{X}^t$ induced by $\mu_1, K_2, \ldots, K_t$ as per Theorem 1. Let $G := \Pi_{i=1}^t G_i$. Provided $0 < \mathrm{E}_{\mu^t}[G] < +\infty$*, the* Feynman-Kac measure *induced by* FK *is defined by the following, for every measurable $A \subseteq \mathcal{X}^t$:*

$$\phi_{\mathrm{FK}}(A) := \frac{\mathrm{E}_{\mu^t}[1_A \cdot G]}{\mathrm{E}_{\mu^t}[G]}\,. \tag{13}$$

We will refer to G in the above definition as the *global potential*. Equality (13) easily generalizes to expectations taken according to ϕ_{FK}. That is, for any measurable nonnegative function g on $\mathcal{X}^t$, we can easily show that:

$$\mathrm{E}_{\phi_{\mathrm{FK}}}[g] = \frac{\mathrm{E}_{\mu^t}[g \cdot G]}{\mathrm{E}_{\mu^t}[G]}\,. \tag{14}$$

In what follows, we will suppress the subscript $_{\text{FK}}$ from ϕ_{FK} in the notation, when no confusion arises. Comparing (14) against the definition (6) suggests that the global potential G should play in FK models a role analogous to the weight function w in probabilistic programs. Note however that there is a major technical difference between the two, because FK models are only defined for a finite time horizon model given by t. A reconciliation between the two is possible thanks to the finite approximation theorem seen in the last section; this will be elaborated further below (see Theorem 4).

We will be particularly interested in the *t-th marginal* of ϕ, that is the probability measure on $\mathcal{X}$ defined as ($A \subseteq \mathcal{X}$ measurable):

$$\phi_t(A) := \phi(\mathcal{X}^{t-1} \times A) = \mathrm{E}_\phi[1_{\mathcal{X}^{t-1}\times A}] \,. \tag{15}$$

The measure ϕ_t is called *filtering* distribution (at time t), and can be effectively be computed via the Particle Filtering algorithm described in the next subsection.

Now let $\mathbf{G} = (\mathcal{P}, E, \mathsf{nil}, \mathbf{sc})$ be an arbitrary fixed PPG. Comparing (14) against e.g. the lower bound in (8) suggests considering the following FK model associated with $\mathbf{G}$ and a checkpoint S.

Definition 10 **(FK$_S$ model).** *Let $t \geq 1$ be an integer and S a program checkpoint of* $\mathbf{G}$. *We define* FK$_S$ *as the FK model where: $\mathcal{X} = \Omega$, $\mu^1 = \delta_{(0,S)}$, $K_i = \kappa$ ($i = 2, ..., t$) and $G_i =$ sc ($i = 1, ..., t$). We let ϕ_S denote the measure on Ω^t induced by* FK$_S$.

We now restrict our attention to functions f that are the lifting of a nonnegative h defined on Ω. Let $\phi_{S,t}$ denote the filtering distributions of ϕ_S at time t obtained by (15). In the following theorem we express the bounds in (8) in terms of the measure $\phi_{S,t}$. The whole point and interest of this result is that the bounds are expressed directly as expectations; these are moreover taken w.r.t. a *1-dimensional* filtering distribution ($\phi_{S,t}$), rather than a t-dimensional one (μ^t_S). Importantly, there are well-known algorithms to estimate expectations under a filtering distribution, as we will see in the next subsection.

Theorem 4 **(filtering distributions and lifted functions).** *Under the same assumptions of Theorem 2, further assume that f is the lifting of h. Then $\alpha_t = \mathrm{E}_{\phi_{S,t}}[1_{\mathrm{T}}]^{-1}$ and*

$$\beta_L := \mathrm{E}_{\phi_{S,t}}[h] \leq [[S]]f \leq \mathrm{E}_{\phi_{S,t}}[h] \cdot \alpha_t + M \cdot (\alpha_t - 1) =: \beta_U \,. \tag{16}$$

Example 8. Consider again the PPG of Example 2. We can re-compute $[[S]]f$ relying on Theorem 4. Fix $t = 4$. We first compute the filtering distribution ϕ_t on $\mathcal{X} = \overline{\mathbb{R}}^3$ relying on its definition (15). Similarly to what we did in Example 5, we consider the nonzero-weight, nonzero-probability traces of length four. Then we project onto the final (fourth) state, and compute the weights of the resulting triples (c, d, S), then normalize. There are only two triples (c, d, S) of nonzero probability:

$$\phi_t(0,0,2) = \frac{2}{3} \qquad\qquad \phi_t(1,1,2) = \frac{1}{3} \,.$$

The function f considered in Example 5 is the lifting of the function $h(c, d, S) = c \cdot [S = 2]$ defined on $\mathcal{X} = \overline{\mathbb{R}}^3$. We apply Theorem 4 and get $\beta_L = \mathrm{E}_{\phi_t}[h] = \frac{1}{3} \leq [[S]]f$. Moreover $\mathrm{E}_{\phi_t}[1_{\mathrm{T}}] = 1$, hence $\alpha_t = 1$ according to Theorem 4. Hence $\beta_L = \beta_U = [[S]]f = \frac{1}{3}$. This coincides with what found in examples 5 and 7.

We can apply the above theorem to the functions described in Example 4 and to other computationally challenging cases: we will do so in Section 6, after introducing in the next section the Particle Filtering algorithm.

The Particle Filtering algorithm From a computational point of view, our interest in FK models lies in the fact that they allow for a simple, unified presentation of a class of efficient inference algorithms, known as *Particle Filtering (PF)* [19, 21, 50]. For the sake of presentation, we only introduce here the basic version, *Bootstrap* PF, following closely[7] [19, Ch.11]. Fix a generic FK model, FK $= (\mathcal{X}, t, \mu^1, \{K_i\}_{i=2}^t, \{G_i\}_{i=1}^t)$. Fix $N \geq 1$, the number of *particles*, that is instances of the random process represented by the K_i's, we

[7] Additional details in [14].

Algorithm 1 A generic PF algorithm

```
   Input: FK = $(\mathcal{X}, t, \mu^1, \{K_k\}_{k=2}^t, \{G_k\}_{k=1}^t)$, a FK model; $N \geq 1$, number of particles.
   Output: $X_t^{1:N} \in \mathcal{X}^N$, $W_t^{1:N} \in \mathbb{R}^{+N}$.
1: $X_1^{(j)} \sim \mu^1$                          } $(j = 1, ..., N)$      ▹ state initialisation
2: $W_1^{(j)} := G_1(X_1^{(j)})$                   }                       ▹ weight initialisation
3: for $k = 2, ..., t$ do
4:     $r_{1:N} \sim R(W_{k-1}^{1:N})$                                     ▹ resampling
5:     $X_k^{(j)} \sim K_k(X_{k-1}^{(r_j)})$       } $(j = 1, ..., N)$      ▹ state update
6:     $W_k^{(j)} := G_k(X_k^{(j)})$               }                       ▹ weight update
7: end for
8: return $(X_t, W_t)$
```

want to simulate. Let $W = W^{1:N} = (W^{(1)}, ..., W^{(N)})$ be a tuple of N real nonnegative random variables, the *weights*. Denote by $\widehat{W}$ the normalized version of W, that is $\widehat{W}^{(i)} = W^{(i)}/(\sum_{j=1}^N W^{(j)})$. A *resampling scheme* for (N, W) is a N-tuple of random variables $R = (R_1, ..., R_N)$ taking values on $1..N$ and depending on W, such that, for each $1 \leq i \leq N$, one has: $\mathrm{E}[\sum_{j=1}^N 1_{R^{(j)}=i} |W] = N \cdot \widehat{W}^{(i)}$. In other words, each index $i \in 1..N$ on average is selected in R a number of times proportional to its weight in W. We shall write $R(W)$ to indicate that R depends on a given weight vector W. Various resampling schemes have been proposed in the literature, among which the simplest is perhaps *multimomial resampling*; see e.g. [19, Ch.9] and references therein. Algorithm 1 presents a generic PF algorithm. Resampling here takes place at step 4: its purpose is to give more importance to particles with higher weight, when extracting the next generation of N particles, while discarding particles with lower weight.

The justification and usefulness of this algorithm is that, under mild assumptions, for any measurable function h defined on $\mathcal{X}$, expectation under ϕ_t, the filtering distribution on $\mathcal{X}$ at time t, in the limit can be expressed a weighted sum with weights $\widehat{W}_t^{(j)}$:

$$\sum_{j=1}^{N} \widehat{W}_t^{(j)} \cdot h(X_t^{(j)}) \longrightarrow \mathrm{E}_{\phi_t}[h] \quad \text{a.s. as } N \longrightarrow +\infty\,. \tag{17}$$

The practical implication here is that we can estimate quite effectively the expectations involved in (16), for $\phi_t = \phi_{S,t}$, as weighted sums like in (17). Note that in the above consistency statement t is held fixed — it is one of the parameter of the FK model — while the number of particles N tends to $+\infty$.

6 Implementation and experimental validation

6.1 Implementation

The PPG model is naturally amenable to a vectorized implementation of PF that leverages the fine-grained, SIMD parallelism existing at the level of particles. At every iteration, the state of the N particles, $\omega^N = (\omega_1, ..., \omega_N)$ with $\omega_i = (v_i, z_i) \in \overline{\mathbb{R}}^{m+1}$, will be stored using a pair of arrays (V, Z) of shape $N \times m$ and $N \times 1$, respectively. The weight vector is stored using another array W of shape $N \times 1$. We rely on vectorization of operations: for a function $f : \overline{\mathbb{R}}^k \to \overline{\mathbb{R}}$ and a $N \times k$ array U, $f(U)$ will denote the $N \times 1$ array obtained by applying f to each row of U. In particular, we denote by $(Z = s)$ (for any $s \in \mathbb{N}$) the $N \times 1$ array obtained applying element-wise the indicator function $1_{\{s\}}$ to Z element-wise, and by $\varphi(V)$ the $N \times 1$ array obtained by applying the predicate φ to V to the row-wise. For U a $N \times k$ array and W a $N \times 1$ array, $U * W$ denotes the $N \times k$ array obtained by multiplying the the jth row of U by the jth element of W, for $j = 1, ..., N$: when W is a $0/1$ vector, this is an instance of *boolean masking*. Abstracting the vectorization primitives of modern CPUs and programming languages, we model the assignments of a vector to an array variable as a single instruction, written $U := Z$. The usual rules for broadcasting scalars to vectors apply, so e.g. $V := S$ for $S \in \overline{\mathbb{R}}$ means filling V with S. Likewise, for ζ a parametric distribution, $U \sim \zeta(V)$ means sampling N times independently from $\zeta(v_1), ..., \zeta(v_N)$, and assigning the resulting matrix to U: this too counts as a single instruction.

Algorithm 2 VPF, a Vectorized PF algorithm for PPGs.

```
**Input**: $\mathbf{G} = (\mathcal{P}, E, \mathsf{nil}, \mathbf{sc})$, a PPG; $S \in \mathcal{P}$, initial program checkpoint; $t \geq 1$, time horizon; $N \geq 1$, number of particles.
**Output**: $V \in \overline{\mathbb{R}}^{m\times N}$, $Z, W \in \overline{\mathbb{R}}^{1\times N}$.
 1: $V := S$ ; $Z := S$                                              ▷ state initialisation
 2: $W := \gamma_S(Z)$                                               ▷ weight initialisation
 3: for $t-1$ times do
 4:     $(V,Z) := \text{Resampling}((V,Z), W)$                       ▷ resampling
 5:     for $(s,\varphi,\zeta,s') \in E$ do
 6:         $M_{s,\varphi} := \varphi(V) * (Z = s)$                  ▷ mask computation
 7:     end for
 8:     $V \sim \sum_{(s,\varphi,\zeta,s')\in E} \zeta(V) * M_{s,\varphi}$ ; $Z := \sum_{(s,\varphi,\zeta,s')\in E} s' \cdot M_{s,\varphi}$    ▷ state update
 9:     $W := \sum_{s\in\mathcal{P}} \gamma_s(V) * (Z = s)$          ▷ weight update
10: end for
11: return $(V,Z,W)$
```

Based on the above idealized model of vectorized computation, we present VPF, a vectorized version of the PF algorithm for PPGs, as Algorithm 2. Here it is assumed that $\mathcal{P} \subseteq \mathbb{N}$, while $\mathbf{sc}(s) = \gamma_s$. On line 4, Resampling$(\cdot)$ denotes the result of applying a generic resampling algorithm based on weights W to the current particles' state, represented by the pair of vectors (V, Z). With respect to the generic PF Algorithm 1, here in the returned output, (V, Z) corresponds to X_t and W to W_t. Note that there are no loops where the number of iterations depends on N; the **for** loop in lines 5–7 only scans the transitions set E, whose size is independent of N. Line 8 is just a vectorized implementation of sampling from the Markov kernel function in (13). Line 9 is a vectorized implementation of the combined score function (4). In the actual TensorFlow implementation, the sums in lines 8 and 9 are encoded via boolean masking and vectorized operations.

6.2 Experimental validation

We illustrate some experimental results obtained with a proof-of-concept TensorFlow-based [1] implementation of Algorithm 2. We still refer to this implementation as VPF. We have considered a number of challenging probabilistic programs that feature conditioning inside loops. For all these programs, we will estimate $[[S]]f$, for given functions f, relying on the bounds provided by Theorem 4 in terms of expectations w.r.t. filtering distributions. Such expectations will be estimated via VPF. We also compare VPF with two state-of-the-art PPLs, webPPL [25] and CorePPL [35]. webPPL is a popular PPL supporting several inference algorithms, including SMC, where resampling is handled via continuation passing. We have chosen to consider CorePPL as it supports a very efficient implementation of PF. In [35], a comparison of CorePPL with webPPL, Pyro [12] and other PPLs in terms of performance shows the superiority of CorePPL SMC-based inference across a number of benchmarks. As discussed in the Introduction, CorePPL's implementation is based on a compilation into an intermediate format, conceptually similar to our PPGs[8].

Models For our experiments we have considered the following programs: *Aircraft tracking* (AT, [51]), *Drunk man and mouse* (DMM, Example 3), *Hare and tortoise* (HT, e.g. [4]), *Bounded retransmission protocol* (BRP, [31]), *Non-i.i.d. loops* (NIID, e.g. [31]), the *ZeroConf* protocol (ZC, [9]), and two variations of *Random Walks*, RW1 ([13], Example 2) and RW2 in the following. In particular, AT is a model where a single aircraft is tracked in a 2D space using noisy measurements from six radars. HT simulates a race between a hare and a tortoise on a discrete line. BRP models a scenario where multiple packets are transmitted over a lossy channel. NIID describes a process that keeps tossing two fair coins until both show tails. ZC is an idealized version of the network connection protocol by the same name. RW1, RW2 are random walks with Gaussian steps. The pseudo-code of these models is reported in Appendix B.

[8] Direct compilation of CorePPL to GPU via the intermediate-level format RootPPL is also supported. However, the results we have obtained with RootPPL are generally worse in terms of execution time, and not presented here. Our PC configuration is as follows. OS: Windows 10; CPU: 2.8 GHz Intel Core i7; GPU: Nvidia T500, driver v. 522.06; TF: v. 2.10.1; CUDA Toolkit v. 11.8; cuDNN SDK v. 8.6.0.

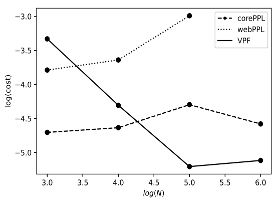
These programs feature conditioning/scoring inside loops. In particular, DMM, HT and NIID feature unbounded loops: for these three programs, in the case of VPF we have truncated the execution after $k = 1000, 100, 100$ iterations, respectively, and set the time parameter t of Theorem 4 accordingly, which allows us to deduce bounds on the value of $[[S]]f$.[9] For the other tools, we just consider the truncated estimate returned at the end of k iterations. AT, BRP, ZC, RW1 and RW2 feature bounded loops, but are nevertheless quite challenging. In particular, AT features multiple conditioning inside a for-loop, sampling from a mix of continuous and discrete distributions, and noisy observations. Below, we discuss the obtained experimental results in terms of accuracy, performance, scalability.

Accuracy We report in Table 1 (Appendix) the execution time, the estimated expected value and the Effective Sample Size (ESS, a measure of diversity of particles, the higher the better; see [14]) for VPF, CorePPL and webPPL, as the number N of particles increases. At least for $N \geq 10^5$, the tools tend generally to return similar estimates of the expected value, which we take as an empirical evidence of accuracy. Additional insight into accuracy is obtained by directly comparing the results of VPF with those of webPPL-rejection (when available), which is an exact inference algorithm. The expected values estimated by webPPL-rejection are consistently in line to those of VPF. In terms of ESS, the difference across the tools is significant. Except for model RW1, VPF yields ESS that are higher or comparable to those of the other tools.

Performance For larger values of N VPF generally outperforms the other considered tools in terms of execution time. The difference is especially noticeable for $N = 10^6$. A graphical representation of the data in Table 1 is provided in Figure 2, with scatterplots showing the ratio of execution times ($time_{\text{other-tool}}/time_{\text{VPF}}$) on a log scale. In the case of WebPPL, nearly all data points lie above the x-axis, indicating superiority of VPF. In the case of CorePPL, for $N = 10^5$ the data points are quite uniformly distributed across the x-axis, indicating basically a tie. For $N = 10^6$, we have a majority of points above the x-axis, indicating again superiority of VPF.

A closer look in the $N = 10^6$ case reveals that the only programs where CorePPL beats VPF are RW1 and ZC. This is most likely due to the low probability of conditioning in these programs; for instance in RW1 just a single final conditioning is performed. As in CorePPL resampling is only performed following a conditioning, this may explain its lower execution times in these cases. To further investigate this issue, we consider RW2, where the probability of conditioning is governed by a parameter $\lambda \in [0, 1]$, and run it for different values of λ. The obtained results are showed in Figure 3. We observe that for both CorePPL and WebPPL execution time tends to increase as the probability λ of conditioning increases; on the contrary, the execution time of VPF appears to be insensitive to λ. This suggests that VPF has a definite advantage over tools with explicit resample, on models with heavy conditioning.

Scalability The plot on the right shows the behaviour the *average unit cost (per particle)* of VPF, CorePPL and WebPPL across all the models we analyzed for $N = 10^3, ..., 10^6$ on a log-scale. Here, for each N the average unit cost (in seconds) is $(t_1 + t_2 + .. + t_k)/(N \cdot k)$, with t_i the execution time of the i-th example. Consistently with Figure 3, we can observe that the cost of VPF decreases as the number of samples increases, whereas the cost of the other tools remains constant or increases (webPPL).



7 Conclusion

We study correct and efficient implementations of Sequential Monte Carlo inference algorithms for universal probabilistic programs. Building on a clean trace-based operational semantics for PPGs, we prove a finite approximation theorem that allows us to establish a precise relation with FK models, and consistency of the PF algorithm for our semantics. Preliminary experiments conducted with VPF, a vectorized version of PF tailored to PPGs, show very promising results when compared to state-of-the-art tools for inference in PPs.

[9] For the precise definition of f in each case, see Appendix B.

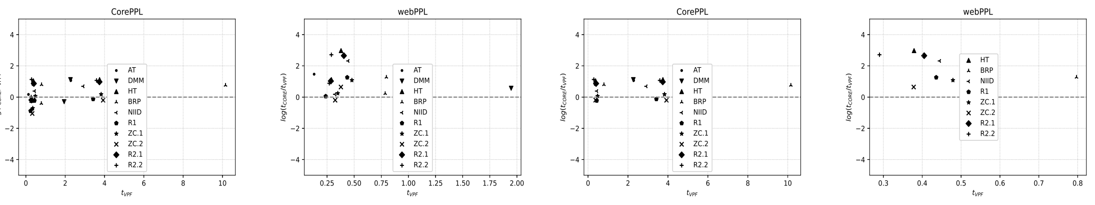


Fig. 2: For $N = 10^5, 10^6$, scatterplots of the log-ratios of execution times, $\log_{10}(\text{time}_{tool}/\text{time}_{\text{VPF}})$, based on the data points of Table 1. From left to right: $N = 10^5$, $tool$ = CorePPL; $N = 10^5$, $tool$ = WebPPL-smc; $N = 10^6$, $tool$ = CorePPL; $N = 10^6$, $tool$ = WebPPL-smc. For $N = 10^6$, the vast majority of the data points lie above the x-axis, indicating substantially better performance of VPF across different examples.

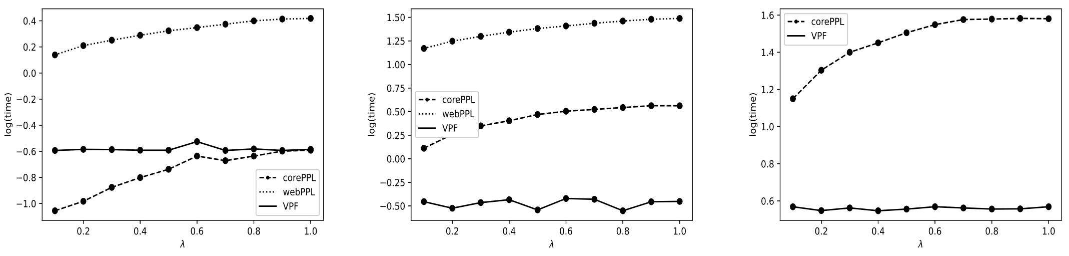


Fig. 3: Execution times (in seconds) for the RW2 program, as a function of the probability λ of conditioning on external data for $N = 10^4$ (left), $N = 10^5$ (center) and $N = 10^6$ (right). webPPL missing from the right-most plot due to time-out. Execution times of VPF are basically insensitive to λ.

References

1. Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, Xiaoqiang Zheng. TensorFlow: A System for Large-Scale Machine Learning. *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI '16). arXiv:1605.08695*, 2016.
2. Robert B. Ash. *Real Analysis and Probability.* Academic Press, Inc., New York, NY, USA, 1972.
3. Martin Avanzini, Georg Moser, and Michael Schaper. Automated Expected Value Analysis of Recursive Programs. *Proc. ACM Program. Lang.*, 7, PLDI, 2023.
4. Alexander Bagnall, Gordon Stewart, Anindya Banerjee. Formally Verified Samplers from Probabilistic Programs with Loops and Conditioning. *Proc. ACM Program. Lang.*, 7, 1–24, 2023.
5. Christel Baier and Joost-Pieter Katoen. *Principles of model checking.* MIT press, 2008.
6. Jialu Bao, Nitesh Trivedi, Drashti Pathak, Justin Hsu, Subhajit Roy. Data-driven invariant learning for probabilistic programs. *In: CAV. Lecture Notes in Computer Science*, vol. 13371, pp. 33–54. Springer, 2022.
7. Gilles Barthe, Joost-Pieter Katoen, and Alexandra Silva. *Foundations of Probabilistic Programming.* Cambridge University Press, 2020.
8. Ezio Bartocci, Laura Kovács, and Miroslav Stankovic. Mora-automatic generation of moment-based invariants. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 492–498,2020.
9. Kevin Batz, Mingshuai Chen, Sebastian Junges, Benjamin Lucien Kaminski, Joost-Pieter Katoen, Christoph Matheja. Probabilistic Program Verification via Inductive Synthesis of Inductive Invariants. In: Sankaranarayanan, S., Sharygina, N. (eds) *Tools and Algorithms for the Construction and Analysis of Systems. TACAS 2023. Lecture Notes in Computer Science*, vol 13994. Springer, 2023.
10. Raven Beutner, Luke Ong, Fabian Zaiser. Guaranteed bounds for posterior inference in universal probabilistic programming. *Proceedings of the 43rd ACM SIGPLAN international conference on programming language design and implementation, (PLDI 22)*, 2022.
11. Sooraj Bhat, Johannes Borgström, Andrew D. Gordon, and Claudio Russo. Deriving probability density functions from probabilistic functional programs. *Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2013)*, 2013.

12. Eli Bingham, Jonathan P Chen, Martin Jankowiak, Fritz Obermeyer, Neeraj Pradhan, Theofanis Karaletsos, Rohit Singh, Paul Szerlip, Paul Horsfall, and Noah D Goodman. Pyro: Deep universal probabilistic programming. *The Journal of Machine Learning Research* 20, 1, 973–978, 2019.
13. Michele Boreale and Luisa Collodi. Guaranteed inference for probabilistic programs: a parallelisable, small-step operational approach. In *VMCAI 2024*, *Lecture Notes in Computer Science*, Vol. 14500, Springer, 2023.
14. Michele Boreale and Luisa Collodi. Full version of the present paper. Available from `https://github.com/Luisa-unifi/Vectorized-Particle-Filtering`, April 2025.
15. Michele Boreale and Luisa Collodi. Code for the experiments described in the present paper. `https://github.com/Luisa-unifi/Vectorized-Particle-Filtering`, April 2025.
16. Johannes Borgström, Ugo Dal Lago, Andrew D. Gordon, Marcin Szymczak. A lambda-calculus foundation for universal probabilistic programming. *ACM SIGPLAN Notices* 51.9: 33-46, 2016.
17. Bob Carpenter, Andrew Gelman, Matthew D. Hoffman, Daniel Lee, Ben Goodrich, Michael Betancourt, Marcus Brubaker, Jiqiang Guo, Peter Li, and Allen Riddell. Stan: A probabilistic programming language. *Journal of Statistical Software* 76(1), 2017. 10.18637/jss.v076.i01
18. Krishnendu Chatterjee, Amir K. Goharshady, Tobias Meggendorfer, Đorde Žikelic. Sound and Complete Certificates for Quantitative Termination Analysis of Probabilistic Programs. In: Shoham, S., Vizel, Y. (eds) *Computer Aided Verification. CAV 2022. Lecture Notes in Computer Science*, vol 13371. Springer, 2022.
19. Nicolas Chopin, Omiros Papaspiliopoulos. *An Introduction to Sequential Monte Carlo*. Springer Nature Switzerland AG, 2021
20. Marco F. Cusumano-Towner, Feras A. Saad, Alexander K. Lew, Vikash K. Mansinghka. Gen: A General-purpose Probabilistic Programming System with Programmable Inference. *In Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pp. 221–236. New York, NY, USA, 2019.
21. Pierre Del Moral, Arnaud Doucet, Ajay Jasra. Sequential Monte Carlo Samplers. *Journal of the Royal Statistical Society. Series B (Statistical Methodology)*, Vol. 68, No. 3, pp. 411-436, 2006.
22. Víctor Elvira, Luca Martino, Christian P. Robert. Rethinking the Effective Sample Size. *arXiv:1809.04129*, 2018.
23. Timon Gehr, Sasa Misailovic, and Martin T. Vechev. Psi: Exact symbolic inference for probabilistic programs. *Proceedings of the 28th International Conference in Computer Aided Verification (CAV 2016), Toronto*, page 62–83, 2016.
24. Noah D. Goodman, Vikash Mansinghka, Daniel Roy, Keith Bonawitz, and Joshua Tenenbaum. Church: a language for generative models. *Proc. Uncertainty in Artificial Intelligence*. 2008.
25. Noah D. Goodman and Andreas Stuhlmüller. The design and implementation of probabilistic programming languages. Retrieved 2023-8-31 from `http://dippl.org`.
26. Andrew D. Gordon, Thomas A. Henzinger, Aditya V. Nori, and Sriram K. Rajamani. Probabilistic programming. *Proceedings of Future of Software Engineering Proceedings (FOSE2014)*, pages 167–181, 2014.
27. Maria I. Gorinova, Andrew D. Gordon, Charles Sutton. Probabilistic programming with densities in SlicStan: efficient, flexible, and deterministic.*Proceedings of the ACM on Programming Languages 3, POPL*. 1-30, 2019.
28. Maria I. Gorinova, Andrew D. Gordon, Charles Sutton, Matthijs Vákár. Conditional independence by typing. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 44.1: 1-54, 2021.
29. Michikazu Hirata, Yasuhiko Minamide, Tetsuya Sato. Semantic Foundations of Higher-Order Probabilistic Programs in Isabelle/HOL. In *14th International Conference on Interactive Theorem Proving (ITP)*, 2023. *Leibniz International Proceedings in Informatics (LIPIcs)*, Volume 268, pp. 18:1-18:18, Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2023.
30. Benjamin L. Kaminski. *Advanced weakest precondition calculi for probabilistic programs*. Doctoral thesis (Ph.D), RWTH Aachen University, 2019.
31. Lutz Klinkenberg, Christian Blumenthal, Mingshuai Chen, Darion Haase, and Joost-Pieter Katoen. Exact Bayesian Inference for Loopy Probabilistic Programs using Generating Functions. In *Proceedings of the ACM on Programming Languages (OPSLA)*, Volume 8: 127,923-953, 2024.
32. Dexter Kozen. Semantics of probabilistic programs. *Journal of Computer and System Sciences*, 22(3):328–350, 1981.
33. Alexander K. Lew, George Matheos, Tan Zhi-Xuan, Matin Ghavamizadeh, Nishad Gothoskar, Stuart Russell, Vikash K. Mansinghka. SMCP3: Sequential Monte Carlo with Probabilistic Program Proposals. *Proceedings of The 26th International Conference on Artificial Intelligence and Statistics (AISTATS)*. PMLR 206:7061-7088, 2023.
34. Daniel Lundén, Johannes Borgström, David Broman. Correctness of sequential Monte Carlo inference for probabilistic programming languages. *Programming Languages and Systems*, pp. 404–431. Springer International Publishing, Cham, 2021.

35. Daniel Lundén, Joey Ohman, Jan Kudlicka, Viktor Senderov, Fredrik Ronquist, and David Broman. Compiling universal probabilistic programming languages with efficient parallel sequential Monte Carlo inference. In *ESOP*, pages 29–56, 2022.
36. Daniel Lundén, David Broman, Fredrik Ronquist, Lawrence M. Murray. Automatic alignment of sequential Monte Carlo inference in higher-order probabilistic programs. *ESOP*, pages 535-563, 2023.
37. Tshepo Mokoena. Urban Myths: Are you never more than 6ft from a rat in a city? The Guardian online, 2014. `https://www.theguardian.com/cities/2014/feb/13/urban-myths-6ft-from-a-rat`.
38. Praveen Narayanan, Jacques Carette, Chungchieh Shan Wren Romano, and Robert Zinkov. Probabilistic inference by program transformation in Hakaru (system description). *Proceedings of the 13th International Symposium on Functional and Logic Programming (FLOPS 2016)*, pages 62–79, 2016.
39. Praveen Narayanan, Chung-chieh Shan. Symbolic disintegration with a variety of base measures. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 42 (2), 1-60, 2020.
40. Aditya V. Nori, Chung-Kil Hur, Sriram K. Rajamani, and Selva Samuel. R2: An efficient mcmc sampler for probabilistic programs. *Proceedings of the Twenty-Eighth AAAI Conference on Artificial Intelligence*, page 2476–2482, July 2014.
41. Alexey Radul, Boris Alexeev. The Base Measure Problem and its Solution. *Proceedings of The 24th International Conference on Artificial Intelligence and Statistics (AISTATS)*, PMLR 130:3583-3591, 2021.
42. Francesca Randone, Luca Bortolussi, Emilio Incerto, Mirco Tribastone. Inference of Probabilistic Programs with Moment-Matching Gaussian Mixtures. *Proceedings of the ACM on Programming Languages (POPL)*:1882-1912, 2024.
43. Sriram Sankaranarayanan, Aleksandar Chakarov, and Sumit Gulwani. Static analysis for probabilistic programs: Inferring whole program properties from finitely many paths. *Proceedings of the 34th ACM SIGPLAN conference on Programming language design and implementation.*, 2013.
44. Adam Scibior, Ohad Kammar, Matthijs Vákár, Sam Staton, Hongseok Yang, Yufei Cai, Klaus Ostermann, Sean K. Moss, Chris Heunen, Zoubin Ghahramani. Denotational validation of higher-order Bayesian inference. *Proceedings of the ACM on Programming Languages 2. POPL.* 1-29, 2017.
45. Sam Staton. Commutative semantics for probabilistic programming. *Proceedings of the 26th European Symposium on Programming (ESOP2017), Uppsala, Sweden*, 2017.
46. Sam Staton, Frank Wood, Hongseok Yang, Chris Heunen, and Ohad Kammar. Semantics for probabilistic programming: higher-order functions, continuous distributions, and soft constraints. *Proceedings of the 31st Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*, 2016.
47. Zachary Susag, Sumit Lahiri, Justin Hsu, and Subhajit Roy. Symbolic execution for randomized programs. *Proc. ACM Program. Lang., 6, OOPSLA*, 1583–1612, 2022.
48. Joseph Tassarotti, Jean-Baptiste Tristan. Verified density compilation for a probabilistic programming language. *Proceedings of the ACM on Programming Languages 7, PLDI*. 615-637, 2023.
49. Peixin Wang, Tengshun Yang, Hongfei Fu, Guanyan Li, C.-H. Luke Ong. Static Posterior Inference of Bayesian Probabilistic Programming via Polynomial Solving. *Proceedings of the ACM on Programming Languages*, Volume 8, Issue *PLDI'24*, 202, 1361 - 1386, 2024.
50. Frank Wood, Jan Willem Meent, Vikash Mansinghka. A New Approach to Probabilistic Programming Inference. *Proceedings of the Seventeenth International Conference on Artificial Intelligence and Statistics*, PMLR 33:1024-1032, 2014.
51. Yi Wu, Siddharth Srivastava, Nicholas Hay, Simon Du, Stuart Russell. Discrete-Continuous Mixtures in Probabilistic Programming: Generalized Semantics and Inference Algorithms. *Proceedings of the 35th International Conference on Machine Learning*, PMLR 80:5343-5352, 2018.

A Table 1

		AT			DMM			HT			BRP			NIID		
		VPF	**CorePPL**	**webPPL-smc**	**VPF**	**CorePPL**	**webPPL-smc**	**VPF**	**CorePPL**	**webPPL-smc**	**VPF**	**CorePPL**	**webPPL-smc**	**VPF**	**CorePPL**	**webPPL-smc**
$N = 10^3$	*time*	**0.009**	0.014	0.190	2.348	**0.050**	0.474	0.274	**0.015**	0.152	0.676	**0.021**	0.155	0.240	**0.010**	0.061
	EV	6.805	6.955	6.696	0.478±0.110	0.501	0.427	32.834	33.683	32.368	0.018	0.016	0.023	3.594	2.694	3.473
	ESS	**1000**	**1000**	999	**994.6**	726.9	9.73	**955.0**	758.9	951.1	**1000**	**1000**	974.5	**1000**	846.6	726.9
$N = 10^4$	*time*	**0.131**	0.194	3.842	1.947	**0.988**	7.190	0.290	**0.180**	3.839	0.786	**0.309**	1.328	0.323	**0.058**	0.490
	EV	6.817	6.967	6.760	0.539±0.115	0.498	0.481	32.725	33.474	32.702	0.029	0.025	0.024	3.364	2.766	3.417
	ESS	$\mathbf{10^4}$	$\mathbf{10^4}$	*9975*	**9984.5**	7797.4	69.417	**9445.0**	7692.9	9476.2	$\mathbf{10^4}$	$\mathbf{10^4}$	9745.8	$\mathbf{10^4}$	8555.6	7560.5
$N = 10^5$	*time*	**0.354**	2.252	-	**2.268**	29.936	-	**0.379**	4.225	361.792	**0.797**	5.010	15.038	**0.445**	1.083	92.419
	EV	6.818	6.970	-	0.537±0.120	0.507	-	33.128	33.545	32.560	0.024	0.025	0.026	3.467	2.772	3.430
	ESS	$\mathbf{10^5}$	9.9e$\mathbf{10^5}$	-	$\approx \mathbf{10^5}$	7.6e10^4	-	$\approx \mathbf{10^5}$	7.7e10^4	$\approx \mathbf{10^5}$	$\mathbf{10^5}$	$\mathbf{10^5}$	9.7e10^4	$\mathbf{10^5}$	8.5e10^4	7.6e10^4
$N = 10^6$	*time*	**2.286**	26.481	-	**38.977**	-	-	**3.749**	49.493	-	**10.155**	58.448	-	**2.916**	14.323	-
	EV	6.834	6.980	-	0.541±0.111	-	-	33.432	33.606	-	0.024	0.025	-	3.413	2.774	-
	ESS	$\mathbf{10^6}$	9.9e10^5	-	$\approx \mathbf{10^6}$	-	-	$\approx \mathbf{10^6}$	7.7e10^5	-	$\mathbf{10^6}$	$\mathbf{10^6}$	-	$\mathbf{10^6}$	8.5e10^5	-
webPPL-rej	*EV*	-			0.494			32.683			0.023			3.414		

		RW1			ZC.1			ZC.2			RW2.1			RW2.2		
		VPF	**CorePPL**	**webPPL**	**VPF**	**CorePPL**	**webPPL**	**VPF**	**CorePPL**	**webPPL**	**VPF**	**CorePPL**	**webPPL**	**VPF**	**CorePPL**	**webPPL**
$N = 10^3$	*time*	0.192	**0.009**	0.045	0.206	**0.018**	0.083	0.262	**0.016**	0.049	0.231	**0.024**	0.232	0.232	**0.021**	0.187
	EV	0.323	0.324	0.343	0.212	0.142	0.250	0.514	0.477	0.483	1.046	0.642	0.912	0.677	0.750	1.092
	ESS	537.0	**1000**	46.739	**1000**	**1000**	392.2	**1000**	**1000**	245.2	**992.0**	780.0	479.9	**999.0**	997.0	133.9
$N = 10^4$	*time*	0.238	**0.031**	0.269	0.349	**0.068**	0.610	0.325	**0.029**	0.207	0.285	**0.186**	3.043	**0.271**	0.275	2.081
	EV	0.334	0.328	0.336	0.242	0.129	0.232	0.483	0.474	0.478	1.367	0.856	1.066	0.982	0.929	1.083
	ESS	5163.0	**10000**	562.795	**10000**	**10000**	4263.0	**10000**	**10000**	2446.8	**9967.0**	7529.0	4026.6	**9701.0**	9350.9	582.5
$N = 10^5$	*time*	0.436	**0.260**	7.956	**0.479**	0.558	5.778	0.378	**0.243**	1.673	**0.405**	3.003	181.802	**0.290**	3.887	149.831
	EV	0.337	0.328	0.332	0.174	0.131	0.233	0.493	0.479	0.479	1.009	0.998	0.982	1.040	0.982	1.037
	ESS	5.1e10^4	$\mathbf{10^5}$	5.7e10^3	$\mathbf{10^5}$	$\mathbf{10^5}$	4.2e10^4	$\mathbf{10^5}$	$\mathbf{10^5}$	2.4e10^4	$\approx \mathbf{10^5}$	7.3e10^4	4.7e10^4	$\approx \mathbf{10^5}$	9.5e10^4	2.0e10^3
$N = 10^6$	*time*	3.422	**2.569**	-	**3.829**	5.790	-	3.928	**2.485**	-	**3.742**	35.271	-	**3.595**	42.134	-
	EV	0.329	0.329	-	0.245	0.130	-	0.479	0.480	-	1.011	1.001	-	1.023	1.002	-
	ESS	5.2e10^5	$\mathbf{10^6}$	-	$\mathbf{10^6}$	$\mathbf{10^6}$	-	$\mathbf{10^6}$	$\mathbf{10^6}$	-	$\approx \mathbf{10^6}$	7.3e10^5	-	$\approx \mathbf{10^6}$	9.4e10^5	-
webPPL-rej	*EV*	0.332			0.235			0.479			1.022			1.061		

Table 1: Execution time ($time$) in seconds, estimated expected value (EV) and effective sample size ($ESS := (\sum_{i=1}^{N} W_i)^2/(\sum_{i=1}^{N} W_i^2)$; the higher the better, see e.g. [22]) as the number of particles (N) increases, for VPF, CorePPL and webPPL, when applied on Aircraft tracking (AT), Drunk man and mouse (DMM), Hare and tortoise (HT), Bounded retransmission protocol (BRP), Non-i.i.d. loops (NIID), ZeroConf (ZC.1, ZC.2) and Random Walks (RW1 and RW2.1, RW2.2). For VPF, with reference to Theorem 4: for the bounded loops AT, BRP, RW1, RW2.1, RW2.2, ZC.1 and ZC.2, we have $EV = \beta_L = \beta_U$ (as $\alpha_t = 1$); for HT and NIID, we only provide β_L, as β_U is vacuous ($M = +\infty$). For DMM we give the midpoint of the interval $[\beta_L, \beta_U]$ ± its half-width. Best results for $time$ and ESS for each example and value of N are marked in **boldface**. Everywhere, '–' means no result due to out-of-memory or timeout (500s). The results for DDM, especially for smaller values of N, exhibit a significant empirical variance: those reported in the table are obtained by averaging over 10 runs of each algorithm. Generally, there is an agrement across the tools about the estimates EV: an exception is NIID, where CorePPL returns values significantly different from the other tools' and from the exact value $\frac{24}{7} = 3.428\cdots$, cf. [31]. Also, for DMM the EV estimates returned by CorePPL and webPPL appear to be consistently lower than the midpoint of the interval returned by VPF.

B Programs pseudo-code

We consider the following probabilistic models described for example in [26]. For convenience, the programs are described in the language of [26], which is based on sequential composition, but they are easy to translate into our language. The `return` statement at the end of each program describes the function f considered in the estimation of $[[S]]f$; e.g. the DMM example, `return r` means f is the lifting of the function $(d, r, x, y, S) \mapsto r$.

```
while(time<=8){
   float[] radius;
   float obs-dist;
   if(time==0){
      x = Gaussian(2,1);
      y = Gaussian(-1.5,1);
   }else{
      x = Gaussian(x,2);
      y = Gaussian(y,2);}
    for i in (0,6]{
       d= compute-distance(i,x,y);
       if(d>radius[i]){
            flag=Bernoulli(0.999);
            if (flag==true){
               obs-dist=radius[i];
            }else{
            obs-dist=radius[i]+0.001*trunc-gauss(0,radius[i]));
            }
        }else{
            obs-dist=d+0.1*trunc-gauss(0,radius[i]);
        }
    obs-dist1 = Gaussian(obs-dist,0.01);
    observe(obs-dist1==...); //evidence numbers omitted
}
time=time+1;
}
return x
```

(a) Aircraft Tracking (AT).

```
d=uniform(0,2);
r=uniform(0,1);
x=-1;
y=1;
while(|x-y|<1/10){
     x=Gaussian(x,d);
     y=Gaussian(y,r);
    observe(|x-y|<=3);
}
return r;
```

(b) Drunk Man and Mouse (DMM).

```
initialPos=uniform(0,10);
tortoise=initialPos;
hare=0;
n=0;
while(hare<tortoise){
    n=n+1;
    tortoise=tortoise+1
    flag=Bernoulli (2/5);
    if (flag==true){
        hare=hare+Gaussian(4,2);
     }
    observe(|hare-tortoise|<=10);
}
observe((n>=20));
return hare;
```

(c) Hare and Tortoise (HT).

```
initialPos=uniform(0,10);
s=100;
f=0;
t=0;
n=0;
while(s>=0 && f<=4 && t<=280){
   t=t+1;
   flag=Bernoulli(0.2);
   if (flag==1){
       f=f+1;
       n=n+1;
       observe((s<=80));
   }else{
       f=0;
       s=s-1;
   }
return s>0;
```

(d) Bounded Retransmission Protocol (BRP).

```
a0=1;
b0=1;
c0=1;
d0=1;
n=0;
while((a0==true||b0==true)){
     a1=Bernoulli(0.5);
     b1=Bernoulli(0.5);
     observe(c0==a1 || d0==b1);
     c1=a1;
     d1=b1;
     n=n+1;
}
return n
```

(e) Non-i.i.d. loops (NIID).

```
r=uniform(0,1);
y=0;
n=0;
while(|y|<1 && (n<=100)){
    y=Gaussian(y,2r);
n=n+1;
}
observe(n>=3);
return r;
```

(f) Random Walk (RW1).

```
p=uniform(0,1);
start=1;
curprobe,established=0;
while(curprobe < 100 && established <=0 && start <= 1){
    if(start = 1){
        flag=Bernoulli(p)
        if (flag==false){
            established=1;
        }
         start=0;
     }else{
         flag=Bernoulli(lambda)
         if (flag==true){
             curprobe := curprobe + 1;
         }else{
             observe(curprobe>=20);
             start=1;
             curprobe=0;
         }
     }
}
return p;
```

(g) ZeroConf (ZC.1: $\lambda = 0.99$, ZC.2: $\lambda = 0.5$).

```
var=uniform(0,7);
y=1;
prob=0.5;
i=0;
while(i <= 100){
    oldy=y;
    y=Gaussian(oldy,2*var);
    flag=Bernoulli(lambda);
    if(flag){
        observe(|y-oldy|<2)};
      i=i+1;}
return y;
```

(h) Random Walk (RW2.1: $\lambda = 0.5$, RW2.2: $\lambda = 0.9999$).