# Universal Probabilistic Programming Language Compilation with Parallel Efficient Sequential Monte Carlo Inference

Daniel Lundén(✉)[1], Joey Öhman[2], Jan Kudlicka[3], Viktor Senderov[4], Fredrik Ronquist[4], and David Broman[1]

[1] KTH Royal Institute of Technology, Sweden, `dlunde@kth.se`
[2] AI Sweden, Sweden
[3] BI Norwegian Business School, Norway
[4] Swedish Museum of Natural History, Sweden

**Abstract.** Probabilistic programming languages (PPLs) allow for natural encoding of arbitrary inference problems, and PPL implementations can provide automatic general-purpose inference for these problems. However, constructing inference implementations that are efficient enough is challenging for many real-world problems. Often, this is due to PPLs not fully exploiting available parallelization and optimization opportunities. For example, handling of probabilistic *checkpoints* in PPLs through the use of continuation-passing style transformations or non-preemptive multitasking—as is done in many popular PPLs—often disallows compilation to low-level languages required for high-performance platforms such as graphics processing units (GPUs). As a solution to this checkpoint problem, we introduce the concept of *PPL control-flow graphs* (PCFGs), providing a simple and efficient approach that can be used for handling checkpoints in such languages. We use this approach to implement *RootPPL*: a low-level PPL built on CUDA and C++ with OpenMP, providing highly efficient and massively parallel SMC inference. We also introduce a general method of *compiling* universal high-level PPLs to PCFGs, and illustrate its application when compiling *Miking CorePPL*—a high-level universal PPL—to RootPPL. This is the first time a universal PPL has been compiled to GPUs with SMC inference. Both RootPPL and the CorePPL compiler are evaluated through a set of real-world experiments in the domains of phylogenetics and epidemiology, demonstrating up to 6× speedups over state-of-the-art PPLs implementing SMC inference.

## 1 Introduction

*Probabilistic programming languages* (PPLs) are languages in which a wide range of statistical problems can be expressed. Furthermore, these languages provide *inference algorithms* as part of their implementations. This allows language users to focus solely on encoding their statistical problems, which the language implementation then solve automatically. Many such languages exist, and are applied in, e.g., statistics, machine learning, and artificial intelligence. Some example PPLs are WebPPL [21], Birch [28], Anglican [36], and Pyro [11].

However, implementing efficient PPL inference algorithms is challenging for many real-world problems. Most often, *universal*[5] PPLs implement general-purpose inference algorithms—most commonly sequential Monte Carlo (SMC) methods [15], Markov chain Monte Carlo (MCMC) methods [19], Hamiltonian Monte Carlo (HMC) methods [13], variational inference (VI) [35], or a combination of these. In some cases, poor efficiency may be due to an inference algorithm that is not well suited to the particular PPL program. However, in other cases, the PPL implementations do not fully exploit opportunities for parallelization and optimization on the available hardware. Unfortunately, doing this is often difficult without also introducing complexity for end users of PPLs.

A critical performance consideration is handling of probabilistic *checkpoints* [33] in PPLs. Checkpoints are locations in probabilistic programs where inference algorithms must pause execution, for example to resample in SMC inference, or to record locations where MCMC inference can explore alternative execution paths (for example, at definition points of random variables). The most common approach to checkpoints—used in universal PPLs such as WebPPL [21], Anglican [36], and Birch [28]—is to have them implicitly at a subset of PPL-specific language constructs. This means that it must be possible to pause and resume execution arbitrarily, often leading to solutions involving continuation-passing style (CPS) transformations [10, 21, 36] or non-preemptive multitasking [28] (e.g., coroutines or exceptions). Essentially, these methods allow for saving and restoring the execution context at arbitrary points in programs. Such methods are most often not available in languages such as C and CUDA [1] that are used for high-performance platforms such as graphics processing units (GPUs), making compilation of PPLs to these languages and platforms challenging. Some approaches for running PPLs on GPUs do exist, however. LibBi [26] runs on GPUs with SMC inference, but is not universal. Pyro [11] is universal and runs on GPUs, but currently not in combination with SMC. In this paper, we compile a universal PPL and run it with SMC on the GPU for the first time.

First, we present an alternative solution to CPS and non-preemptive multitasking, in which probabilistic programs are written as a set of code blocks connected in what we term a *PPL control-flow graph* (PCFG). PPL checkpoints are restricted to only occur at tail position in these blocks, and communication between blocks is only allowed through an explicit PCFG *state*. This makes pausing and resuming executions straightforward: it is simply a matter of stopping after executing a block, and then resuming by running the next block. The next block is determined by a variable in the PCFG state which can be modified from within the blocks, which allows for loops and branching, and gives the same expressive power as other universal PPLs. We implement the above approach

[5] A term first coined in Goodman et al. [20]. No precise definition exists, but in principle, it indicates a PPL in which probabilistic operations can be performed at any point during execution. In particular, the number of random variables cannot always be decided statically.

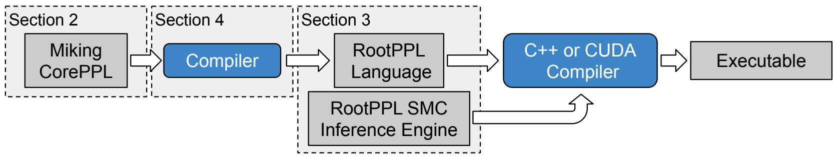


Fig. 1: The CorePPL and RootPPL toolchain. Solid rectangular components (gray) represent programs, and rounded components (blue) translations. The dashed rectangles indicate in which sections the components are presented.

in *RootPPL*: a low-level universal PPL framework built using C++ and CUDA with highly efficient and parallel SMC inference. RootPPL consists of both an inference engine, as well as a simple macro-based PPL.

A problem with RootPPL is that it is low-level, and therefore difficult to write programs in. In particular, sending data between blocks through the PCFG state can quickly get involved for more complex models. To solve this, we develop a general technique for *compiling* high-level universal PPLs to PCFGs. The key idea is to decompose functions in the high-level language to a set of PCFG blocks, such that checkpoints in the original function are always located at tail position in blocks. As a result of the decomposition, a part of the call stack needs to be stored in the PCFG state. The compiler adds code for handling this call stack explicitly in the PCFG blocks. We illustrate the compilation technique by introducing a high-level source language, *Miking CorePPL*, and compiling it to RootPPL. Fig. 1 illustrates the overall toolchain.

In summary, we make the following contributions.

– We introduce PCFGs, used in the implmentation of RootPPL: a low-level universal PPL with highly efficient and parallel SMC inference (Section 3).
– We develop an approach for compiling high-level universal PPLs to PCFGs, and use it to compile Miking CorePPL to RootPPL. In particular, we propose an algorithm for decomposing high-level functions to PCFG blocks (Section 4).

Furthermore, we introduce Miking CorePPL in Section 2, and evaluate the performance of RootPPL and the CorePPL-to-RootPPL compiler in Section 5 on real-world models from phylogenetics and epidemiology, achieving up to 6× speedups over state-of-the-art.

## 2 Miking CorePPL

In this section, we introduce the Miking CorePPL language, used as source language for the compiler in Section 4. We begin by discussing design considerations (Section 2.1), and then present the syntax and semantics (Section 2.2).

## 2.1 Design Considerations

Miking CorePPL (or CorePPL for short) is intended as an *intermediate representation* (IR) PPL, similar to IRs used by LLVM [7] and GCC [2]. This potentially allows for reusing CorePPL, both for domain-specific high-level PPLs, and PPL compiler back-ends. Because of this, CorePPL needs to be expressive enough to allow easy translation from various domain-specific PPLs, as well as simple enough for effective use as a shared IR for compilers. Therefore, we base CorePPL on the lambda calculus, extended with standard data types and constructs.

We must also consider which PPL-specific constructs to include. Critically, most PPLs include constructs for defining random variables and for likelihood updating [22]. CorePPL includes such constructs, including first-class probability distributions, in order to match the expressive power of existing PPLs.

## 2.2 Syntax and Semantics

CorePPL is built on top of the *Miking* framework [12]: a meta-language system for creating domain-specific and general-purpose languages. This allows us to reuse many existing Miking language components and transformations, both when building the CorePPL language itself, and also when building other PPLs on top of it. More precisely, CorePPL extends *Miking Core*—a core functional programming language in Miking—with PPL constructs.

A CorePPL program $\mathbf{t}$ is inductively defined by

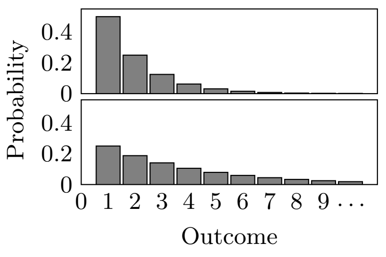
$$\begin{aligned}\mathbf{t} ::= {} & x \mid \texttt{lam}\ x\texttt{.}\ \mathbf{t} \mid \mathbf{t}_1\ \mathbf{t}_2 \mid \texttt{let}\ x\ \texttt{=}\ \mathbf{t}_1\ \texttt{in}\ \mathbf{t}_2 \mid C\ \mathbf{t} \mid c \\ & \mid \texttt{recursive}\ [\texttt{let}\ x\ \texttt{=}\ \mathbf{t}]\ \texttt{in} \\ & \mid \texttt{match}\ \mathbf{t}_1\ \texttt{with}\ p\ \texttt{then}\ \mathbf{t}_2\ \texttt{else}\ \mathbf{t}_3 \mid \texttt{[}\mathbf{t}_1\texttt{,}\ \mathbf{t}_2\texttt{,}\ \ldots\texttt{,}\ \mathbf{t}_n\texttt{]} \\ & \mid \texttt{\{}l_1\ \texttt{=}\ \mathbf{t}_1\texttt{,}\ l_2\ \texttt{=}\ \mathbf{t}_2\texttt{,}\ \ldots\texttt{,}\ l_3\ \texttt{=}\ \mathbf{t}_3\texttt{\}} \\ & \mid \texttt{assume}\ \mathbf{t} \mid \texttt{weight}\ \mathbf{t} \mid \texttt{observe}\ \mathbf{t}_1\ \mathbf{t}_2 \mid D\ \mathbf{t}_1\ \mathbf{t}_2\ \ldots\ \mathbf{t}_{|D|}\end{aligned} \tag{1}$$

where the metavariable $x$ ranges over a set of variable names; $C$ over a set of data constructor names; $p$ over a set of patterns; $l$ over a set of record labels; and $c$ over various literals, such as integers, floating-point numbers, booleans, and strings, as well as over various built-in functions in prefix form such as `addi` (adds integers). The notation $[\texttt{let}\ x = \mathbf{t}]$ indicates a sequence of mutually recursive `let` bindings. The metavariable $D$ ranges over a set of probability distribution names, with $|D|$ indicating the number of parameters for a distribution $D$. For example, for the normal distribution, $|\mathcal{N}| = 2$. In addition to (1), we will also use the standard syntactic sugar `;` to indicate sequencing, as well as `if` $\mathbf{t}_1$ `then` $\mathbf{t}_2$ `else` $\mathbf{t}_3$ for `match` $\mathbf{t}_1$ `with true then` $\mathbf{t}_2$ `else` $\mathbf{t}_3$.

To first give some intuition for CorePPL, and PPLs in general, consider the program in Fig. 2a. The program encodes a variation of the geometric distribution, for which the result is the number of times a coin is flipped until the result is tails. The core of the program is the recursive function `geometric`, defined using a function over the probability of heads for the coin, $p$. This function is initially called at line 8 with the argument 0.5, indicating a fair coin. On line

```
1 recursive let geometric = lam p.
2   let x = assume (Bernoulli p) in
3   if x then
4     weight (log 1.5);
5     addi 1 (geometric p)
6   else 1
7 in geometric 0.5
```

(a)



(b)

Fig. 2: A toy example encoding a skewed geometric distribution, illustrating CorePPL. The CorePPL program is given in (a), and the corresponding distribution is illustrated in (b). The upper part of (b) shows the distribution for (a) with line 4 omitted, and the lower part of (b) shows it with line 4 included.

2, we define the random variable `x` to have a Bernoulli distribution (i.e., a single coin flip) using the `assume` construct (often known as *sample* in PPLs with sampling-based inference). If the random variable is `false` (tails), we stop and return the result 1. If the random variable is instead `true` (heads), we keep flipping the coin by recursively calling `geometric`, and add 1 to the result of this. To illustrate likelihood updating, we make a contrived modification to the standard geometric distribution by adding `weight (log 1.5)` on line 4. This *weights* the execution by a factor of 1.5 each time the result is heads (note that CorePPL uses logarithmic weights for numerical stability). As a result, the unnormalized probability of seeing $n$ coin flips, including the final tails, is $0.5^n \cdot 1.5^{n-1}$. The difference compared to the standard geometric distribution is illustrated in Fig. 2b. The `weight` construct is also commonly known as *factor* or *score* in other PPLs.

What separates PPLs from ordinary programming languages is the ability to modify the likelihood of execution paths, akin to the use of `weight` in Fig. 2a. Most often, this is used to *condition* a probabilistic model on observed data. For this purpose, CorePPL includes an explicit `observe` construct, which allows for modifying the likelihood based on observed data assumed to be generated from a given probability distribution. For instance, `observe 0.3 (Normal 0 1)` updates the likelihood with $f_{\mathcal{N}(0,1)}(0.3)$ (note that this can equivalently be expressed through `weight`), where $f_{\mathcal{N}(0,1)}$ is the probability density function of the standard normal distribution. This conditioning can be related to Bayes' theorem: the random variables defined in a program define a prior distribution (e.g., the upper part of Fig. 2b), the use of `weight` or `observe` a likelihood function, and the inference algorithm of the PPL infers the posterior distribution (e.g., the lower part of Fig. 2b)

Besides simple data types such as integers, CorePPL includes sequences, recursive variants, and records, similar to other functional languages. For example, `[1, 2, 3]` defines a sequence of length 3, `{a = false, b = 1.2}` a record with labels `a` and `b`, and `Leaf {age = 1.0}` a variant data type with the constructor name `Leaf`, containing a singleton record with the label `age`. These data types, as well as pattern matching over them, are non-trivial language features that are handled by the CorePPL compiler in Section 4.3. Furthermore, the CorePPL models in Section 5 also make frequent use of these features.

CorePPL supports pattern matching through the `match` construct. For example, `match a with Leaf {age = f} then f else 0.0` checks if `a` is a `Leaf`, and returns its age if so, or `0.0` otherwise. Here, `f` is a pattern variable that is bound to the value of the `age` element of `a` in the `then` branch of the `match`.

The type system and pattern matching features in Miking, and consequently CorePPL, are not directly related to the key contributions in this paper. Therefore, they are not discussed in further detail.

We will consider CorePPL again in Section 4 for the compilation to PCFGs.

## 3 PCFGS and RootPPL

In this section, we introduce the new PCFG concept (Section 3.1), and show how SMC can be applied over these (Section 3.2). Finally, we present the RootPPL framework (Section 3.3), based on the PCFG concept and SMC inference.

### 3.1 PPL Control-Flow Graphs

In order to handle checkpoints efficiently without CPS or non-preemptive multitasking (see Section 1), we introduce *PPL control-flow graphs* (PCFGs). In contrast to traditional PPLs, where checkpoints are most often implicit, we make them explicit and central in the PCFG framework. The main benefit of this approach is that handling of checkpoints in inference algorithms is greatly simplified, which in particular allows for implementing the framework in low-level languages. However, the explicit checkpoint approach make PCFGs rather low-level, and they are mainly intended as a target when compiling from high-level PPLs. We introduce such a compiler in Section 4.

Formally, we define a PCFG as a 6-tuple $(B, S, \mathit{sim}, b_0, b_{\text{stop}}, \mathcal{L})$. The first component $B$ is a set of *basic blocks*, inspired by basic blocks used as a part of control-flow analysis in traditional compilers [9]. In concrete settings, the blocks in $B$ are pieces of code that together make up a complete probabilistic program. Unlike basic blocks used in traditional compilers, these pieces of code are allowed to contain branches internally. The second component $S$ is a set of *states*, representing collections of information that flow between basic blocks. In concrete settings, this state often contains local variables that must be passed between blocks, and an accumulated likelihood. The blocks and states form the domain of the function $\mathit{sim} : B \times S \to B \times S \times \{\text{false}, \text{true}\}$. This function performs computation specific for the given block over the given state, and outputs a *successor* block indicating what to execute next, an updated state, and a boolean indicating whether or not there is a checkpoint at the end of the executed block.

To illustrate this formalization, consider the PCFG in Fig. 3a for which $B = \{b_0, b_1, \ldots, b_4, b_{\text{stop}}\}$. The block $b_0$ is present in every PCFG, and represents

$$\begin{tikzcd}
 & & b_2 \arrow[loop right, -open triangle 45] \arrow[d, shift right, -open triangle 45] \arrow[dr, -open triangle 45] & & \\
b_0 \arrow[r] & b_1 \arrow[ur, -open triangle 45] & & b_4 \arrow[r] & b_{\text{stop}} \\
 & & b_3 \arrow[u, shift right] & &
\end{tikzcd}$$

(a)

$sim(b_0, s_0) \mapsto (b_1, s_1, \text{false})$
$sim(b_1, s_1) \mapsto (b_2, s_2, \text{true})$
$sim(b_2, s_2) \mapsto (b_4, s_3, \text{true})$
$sim(b_4, s_3) \mapsto (b_{\text{stop}}, s_4, \text{false})$

(b)

Fig. 3: A PCFG illustration. Figure (a) shows an example PCFG. The arrows denote the possible flows of control between the blocks, with regular arrows denoting checkpoint transitions, and arrows with open tips non-checkpoint transitions. Figure (b) shows a possible execution sequence with *sim* for (a).

**Algorithm 1** A standard SMC algorithm applied to PCFGs.

**Input:** A PCFG $(B, S, sim, b_0, b_{\text{stop}}, \mathcal{L})$. A set of initial states $\{s_n\}_{n=1}^N$.
**Output:** An updated set of states $\{s_n\}_{n=1}^N$.

1. **Initialization:** For each $1 \leq n \leq N$, let $a_n := b_0$ and $c_n :=$ false.
2. **Propagation:** If all $a_n = b_{\text{stop}}$, terminate and output $\{s_n\}_{n=1}^N$. If not, for each $1 \leq n \leq N$ where $c_n =$ false, let $(a_n, s_n, c_n) := sim(a_n, s_n)$. If all $c_n =$ true, go to 3. If not, repeat 2.
3. **Resampling:** For each $1 \leq n \leq N$, let $p_n := \mathcal{L}(s_n)/\sum_{i=1}^N \mathcal{L}(s_i)$. For each $1 \leq n \leq N$, draw a new index $i$ from $\{i\}_{i=1}^N$ with probabilities $\{p_i\}_{i=1}^N$. Let $(s'_n, b'_n) := (s_i, b_i)$. Finally, for each $1 \leq n \leq N$, let $(s_n, b_n, c_n) := (s'_n, b'_n, \text{false})$. Go to 2.

its entry point. Similarly, the block $b_{\text{stop}}$ is a special block indicating termination, which must be reachable from all other blocks. For some initial state $s_0 \in S$, Fig. 3b illustrates a a possible execution sequence starting at $b_0$ in Fig. 3a before terminating at $b_{\text{stop}}$. The structure of a PCFG restricts checkpoints to *only* occur at the end of basic blocks, and also confines communication between blocks to the state. These restrictions greatly simplify inference algorithm implementations. More precisely, rather than relying on CPS or non-preemptive multitasking, the inference algorithm can simply run a block $b$ with *sim*, handle the checkpoint, and then run the successor block indicated by the output of *sim*.

## 3.2 SMC and PCFGs

To prepare for introducing RootPPL in Section 3.3, we here present how SMC inference can be applied to PCFGs. A more general pedagogical introduction to SMC can be found in, e.g., Naesseth et al. [29]. At a high level, SMC inference works by simulating many instances—known as *particles* in SMC literature—of a PCFG program concurrently, occasionally *resampling* the different particles based on their current likelihoods. In CorePPL, for example, such likelihoods are determined by `weight` and `observe`. Resampling allows the downstream simulation to focus on particles with higher likelihood.

In order to apply SMC inference over PCFGs, we need some way of determining the likelihood of the SMC particles. For this, we use the final component of the PCFG definition, $\mathcal{L} : S \to \mathbb{R}_{\geq 0}$, which is a function mapping states to a likelihood (a non-negative real number). Concretely, this likelihood is most often stored directly in the state as a real number, and $\mathcal{L}$ simply extracts it.

An SMC algorithm over PCFGs is defined in Algorithm 1. It takes a PCFG as input, together with a set of $N$ states $\{s_n\}_{n=1}^{N}$ which represent the SMC particles. Step 1 in the algorithm sets up variables $a_n$ and $c_n$, indicating for each particle its current block and whether or not a checkpoint has occurred in it. In step 2, all particles that have not yet reached a checkpoint are simulated using *sim*. This step is repeated until all particles have reached a checkpoint. Then, the algorithm moves on to step 3—*resampling*. Here we use the likelihood function $\mathcal{L}$ to compute the relative likelihoods of all particles, and then resample them based on this. That is, we sample $N$ particles from the existing $N$ particles (with replacement) based on the relative likelihoods. After resampling, we return to step 2. If all particles have reached the termination block $b_{\text{stop}}$, the algorithm terminates and returns the current states.

Importantly, note in Algorithm 1 that the input states are *not* required to be identical. For example, in concrete settings, each state should have a unique seed used for generating random numbers (e.g., for `assume` in CorePPL). Note that non-identical initial states in Algorithm 1 imply that different particles may traverse the blocks in $B$ differently, and reach checkpoints at different times. Although this means that different particles can be at different blocks concurrently, the SMC algorithm is still correct [24]. This is an essential property of PCFGs that allows for the encoding of universal probabilistic programs in PCFG-based PPLs. Furthermore, it implies that some particles may reach $b_{\text{stop}}$ earlier than others. To solve this, we require in Algorithm 1 that $sim(b_{\text{stop}}, s) = (b_{\text{stop}}, s, \text{true})$ holds for all states $s$. That is, particles that have finished also participate in resampling, and cannot cause step 2 to loop infinitely.

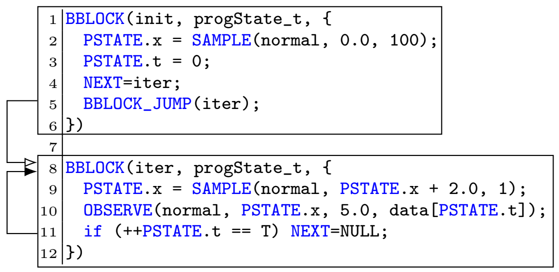
Next, we describe our implementation of PCFGs with SMC—RootPPL.

## 3.3 RootPPL

We make use of the PCFG framework when implementing RootPPL: a new low-level PPL framework built on top of CUDA C++ and C++, intended for highly optimized and massively parallel SMC inference on general-purpose GPUs. RootPPL consists of two major components: a macro-based C++ PPL for encoding probabilistic models, and an SMC inference engine.

The macro-based language has two purposes: to support compiling the same program to either CPU or GPU, and to simplify the encoding of models for programmers. As a result, the macros are designed to hide all hardware details from the programmer. To illustrate this macro-based PPL, consider the example RootPPL program in Fig. 4a. This program encodes a simple state-space model for an object moving along an axis in $\mathbb{R}$, given by

$$X_0 \sim \mathcal{N}(0, 100), \quad X_t \sim \mathcal{N}(x_{t-1} + 2, 1), \quad Y_t \sim \mathcal{N}(x_t, 5), \quad 1 \leq t \leq T. \qquad (2)$$



```
 1 BBLOCK(init, progState_t, {
 2   PSTATE.x = SAMPLE(normal, 0.0, 100);
 3   PSTATE.t = 0;
 4   NEXT=iter;
 5   BBLOCK_JUMP(iter);
 6 })
 7
 8 BBLOCK(iter, progState_t, {
 9   PSTATE.x = SAMPLE(normal, PSTATE.x + 2.0, 1);
10   OBSERVE(normal, PSTATE.x, 5.0, data[PSTATE.t]);
11   if (++PSTATE.t == T) NEXT=NULL;
12 })
```

(a) RootPPL program

```
struct progState_t {
  double x;
  int t;
};
```

(b) Program state

Fig. 4: Part (a) illustrates a RootPPL program implementing the state-space model from (2). Details are provided in the text. The type `progState_t` defines the RootPPL program state used, and is defined in (b).

Here, $X_0$ is the initial position, $X_t$ the subsequent positions, and $Y_t$ a set of noisy observations of the object position. The inference goal is to determine the distribution of $X_T$ (the final position of the object) conditioned on all $Y_t$.

In Fig. 4a this is implemented in two basic blocks, introduced with the `BBLOCK` macro in RootPPL. The first block `init` draws $X_0$ using the `SAMPLE` macro (equivalent to `assume` in CorePPL) on line 2, and stores the drawn value in the *program state* variable `x` through the `PSTATE` macro. This program state is the RootPPL instantiation of the PCFG state introduced in Section 3.1. Another program state variable, `t` (corresponding to the index $t$ in the model), is initialized on line 3. As preparation for iterating over the `iter` block, we set the `NEXT` construct to `iter` at line 4. Finally, the block exits by making a direct non-checkpoint transition to `iter` using the `BBLOCK_JUMP` macro at line 5.

In `iter`, we sample $X_1$ at line 9 and write the result to `x` (overwriting the previous $X_0$, which is no longer needed). This is followed by a likelihood update on line 10 using the `OBSERVE` macro (equivalent to `observe` in CorePPL), corresponding to observing $Y_1$ in the model. All $Y_t$ are accessed through the `data` array, which is a shared global constant, avoiding memory duplication in the program state. Finally, at line 11, we check if the time `T` (a shared global constant for $T$) has been reached. If this is the case, `NEXT` is set to `NULL`, indicating termination. This is equivalent to moving to $b_{\text{stop}}$ in the PCFG formalization. Otherwise, `NEXT` keeps its value set at line 4, and jumps to the beginning of the `iter` block. By not using `BBLOCK_JUMP`, as is done in the first block on line 5, we here allow `iter` to return to the inference engine in between iterations—indicating checkpoint transitions. In RootPPL, this means that SMC inference will resample the instances before returning to `iter` for the next iteration.

For each RootPPL program, its program state is defined by the programmer as an arbitrary C++ struct, and this struct type is passed as an argument (`progState_t` in Fig. 4a) to the definition of each basic block. As we have seen, the variables in the struct can subsequently be accessed through the `PSTATE` macro. The program state for the example program in Fig. 4a is illustrated in

Fig. 4b. As described in Section 3.1, this program state is the *only* possible means to pass data from one basic block to another in RootPPL.

This minimal example does not illustrate all language features (e.g., `WEIGHT`) Further details on the RootPPL language are available at GitHub [4].

The second part of the RootPPL framework is the SMC inference engine. To achieve high performance, it is crucial to take advantage of the highly parallel nature of SMC and available hardware for parallelization. For this purpose, RootPPL supports compilation to either plain C++ on a single CPU core, C++ with CPU parallelism through OpenMP [3], and to CUDA C++ [1] with massive parallelism on the GPU.

The main inference loop in RootPPL is given below (cf. Algorithm 1).

1. Initialize random seeds.
2. Execute the basic block indicated by `NEXT` for all particles. This may execute a chain of blocks with non-checkpoint transitions in between them (using the `BBLOCK_JUMP` macro) before returning to the inference engine.
3. If all particles have terminated (i.e., `NEXT = NULL`), stop.
4. Perform resampling over all particles, and go to 2.

The random seeds in step 1 are initialized differently depending on the compile target. For plain C++ on a single core, one seed is shared between all particles, because they are in any case executed sequentially. However, for OpenMP and CUDA, the parallel execution requires that each thread is assigned a unique seed which is then shared for all particles running on that thread. Furthermore, for CUDA, these seeds are placed in thread-local CUDA memory for each particle to minimize memory overhead when using `SAMPLE` (which is performance critical). In addition, when compiling to CUDA, the initialization of seeds is done in parallel using a CUDA compute kernel.

In step 2, again depending on the compile target, the particles are executed either sequentially, in parallel using OpenMP threads, or in parallel using a CUDA compute kernel. This is followed by a termination check in step 3. In this step, the first particle is first checked for termination. If it has not terminated, we directly move to the resampling step. If it has terminated, we must iteratively check other particles to either find a particle that has not terminated, or to conclude that all particles have terminated and stop the inference. This approach both allows for particles terminating at different times, and also introduces minimal overhead for the case when all particles terminate at the same time (which is quite common). When all particles terminate at the same time, it is enough to check the first particle in all iterations of step 3 except the last.

The resampling step is the most difficult one to parallelize efficiently. This is because of the normalizing sum (e.g., $\sum_{i=1}^{N} \mathcal{L}(s_i)$ in Algorithm 1) that must be computed in order to determine resampling probabilities. We use systematic resampling for single core and OpenMP, and parallel systematic resampling for CUDA, as described in Murray et al. [27] (we do not use in-place propagation). For CUDA, the normalizing sum is computed in parallel via the Thrust library [8].

Another important consideration for the inference engine is memory allocation. In particular, the memory allocated for `NEXT`, the likelihood, and the `PSTATE` for each particle, is laid out as separate arrays in memory, rather than one big array of structs. This is known as memory coalescing, and avoids strided memory accesses in global memory. This is the preferred memory layout for parallel operations, and is particularly important for CUDA. Another memory consideration is the copying of particle data in memory required as part of resampling, when particles can be both discarded and duplicated. For this, we use a custom aligned memory transfer in CUDA, because the standard `memcpy` implementation in CUDA proved to be a bottleneck. On a single core, and with OpenMP, `memcpy` can be used without issue. Additionally, we perform a specific optimization when copying the program state used in the CorePPL compiler. This program state consists of a possibly large stack (with user-definable size) together with a stack pointer, and we ensure that the part of the stack located beyond the stack pointer (which is not used) is not copied. This is a critical optimization for the CorePPL compiler, because most programs do not have a deep call stack at checkpoints.

Other things supported in RootPPL is the estimation of *normalizing constants* for encoded models, as well as adaptive resampling based on the current *effective sample size* (ESS). These are standard concepts in SMC inference, and further details can be found in, e.g., Naesseth et al. [29].

In the next section, we will use RootPPL as the target language for the CorePPL compiler.

## 4 Compiling to PCFGs

In this section, we introduce the ideas for compiling high-level universal PPLs to PCFGs. First, we present the key transformation—*function decomposition* into basic blocks—using a toy example (Section 4.1) as well as a formal algorithm (Section 4.2). We also give a high-level overview of the CorePPL-to-RootPPL compiler implementing the above ideas (Section 4.3), and discuss its strengths and limitations (Section 4.4).

### 4.1 Function Decomposition Example

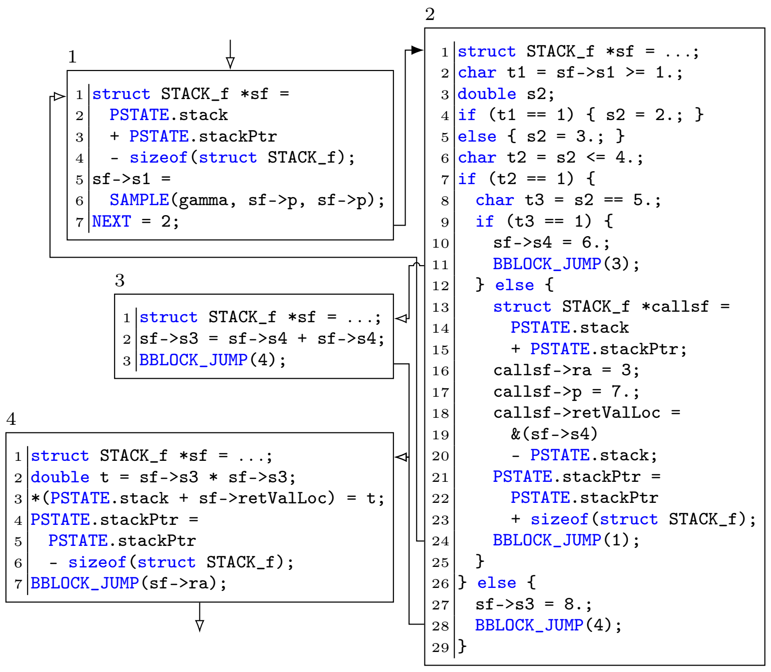
The major challenge when compiling high-level PPLs is how to implement pausing and resuming at checkpoints in order to temporarily yield control to an inference algorithm. As discussed in Section 3.1, this is especially difficult when compiling to low-level languages due to runtime limitations. We solve this problem by compiling to the PCFGs introduced in Section 3, which are designed specifically for implementation in low-level target languages. A challenge with this approach is that checkpoints can occur at arbitrary locations in high-level probabilistic programs, whereas in PCFGs, checkpoints must always occur at tail position in basic blocks. We solve this by *decomposing* functions in the source language into a set of basic blocks. Our approach is similar to how functions are

```
recursive let f: Float -> Float =
 lam p.
  let s1 = assume (Gamma p p) in
  resample;
  let s2 =
    if geqf s1 1. then 2.
    else 3. in
  let s3 =
    if leqf s2 4. then
      let s4 =
        if eqf s2 5. then 6.
        else f 7. in
      addf s4 s4
    else 8. in
  mulf s3 s3
in
```

(a) Source CorePPL program.

```
recursive let f: Float -> Float =
 lam p.
  let s1 = assume (Gamma p p) in
  resample;                          1
  let t1 = geqf s1 1. in
  let s2 = if t1 then 2. else 3. in
  let t2 = leqf s2 4. in
  let s3 =
    if t2 then
      let t3 = eqf s2 5. in
      let s4 =                         2
        if t3 then 6. else f 7. in
      addf s4 s4 3
    else 8. in
  mulf s3 s3 4
in
```

(b) Intermediate ANF representation.



Block 1:

```
struct STACK_f *sf =
  PSTATE.stack
  + PSTATE.stackPtr
  - sizeof(struct STACK_f);
sf->s1 =
  SAMPLE(gamma, sf->p, sf->p);
NEXT = 2;
```

Block 2:

```
struct STACK_f *sf = ...;
char t1 = sf->s1 >= 1.;
double s2;
if (t1 == 1) { s2 = 2.; }
else { s2 = 3.; }
char t2 = s2 <= 4.;
if (t2 == 1) {
  char t3 = s2 == 5.;
  if (t3 == 1) {
    sf->s4 = 6.;
    BBLOCK_JUMP(3);
  } else {
    struct STACK_f *callsf =
      PSTATE.stack
      + PSTATE.stackPtr;
    callsf->ra = 3;
    callsf->p = 7.;
    callsf->retValLoc =
      &(sf->s4)
      - PSTATE.stack;
    PSTATE.stackPtr =
      PSTATE.stackPtr
      + sizeof(struct STACK_f);
    BBLOCK_JUMP(1);
  }
} else {
  sf->s3 = 8.;
  BBLOCK_JUMP(4);
}
```

Block 3:

```
struct STACK_f *sf = ...;
sf->s3 = sf->s4 + sf->s4;
BBLOCK_JUMP(4);
```

Block 4:

```
struct STACK_f *sf = ...;
double t = sf->s3 * sf->s3;
*(PSTATE.stack + sf->retValLoc) = t;
PSTATE.stackPtr =
  PSTATE.stackPtr
  - sizeof(struct STACK_f);
BBLOCK_JUMP(sf->ra);
```

(c) Compiled RootPPL PCFG illustration. Some RootPPL constructs are omitted or slightly modified for readability. In particular, the BBLOCK construct used in Fig. 4a is omitted. Instead, the blocks are illustrated visually as nodes in a graph, numbered by indices. The arrows indicate control flow between the blocks, with the incoming arrow to block 1 representing the call to `f`, and the outgoing arrow from block 4 representing the return from `f`.

Fig. 5: Compilation of a CorePPL program (a) to a RootPPL PCFG (c). The program in (b) illustrates an intermediate ANF representation of (a), and also indicates the parts of the program corresponding to the blocks in (c). Further details are provided in the text.

decomposed into basic blocks in standard compilers such as GCC [2] and LLVM [7] (see, e.g., Aho et al. [9]). The difference is that we only decompose *as needed*, based on where checkpoints occur. In particular, we do *not* decompose functions, and parts of functions, in which checkpoints are guaranteed not to occur. This allows for more optimizations by the underlying compiler (e.g., NVCC or GCC for RootPPL).

To first give an intuition for the decomposition, consider the toy CorePPL function in Fig. 5a, and the resulting compilation to a RootPPL PCFG in Fig. 5c. For this example, we introduce an explicit *inference hint* and SMC checkpoint `resample` in CorePPL, indicating where SMC should pause executions in order to resample. This is the sole checkpoint considered in this example (and the CorePPL compiler), but the method can be applied in general to arbitrary checkpoints. The first step in the decomposition is to translate the program into A-normal form (ANF) [16], which is illustrated in Fig. 5b. ANF is commonly used in compilers, and ensures that non-trivial expressions (e.g., function applications and checkpoints) are always name-bound. For CorePPL, ANF guarantees that the body of each `let` expression, or expression in tail position, is either trivial, contains at most one function application, or is an `if` expression with a trivial condition, resulting in simplified decomposition. We will use the program in Fig. 5b as the target for decomposition in the following. Note that variables introduced by ANF start with a `t` in Fig. 5b, while the original variables from Fig. 5a start with an `s`.

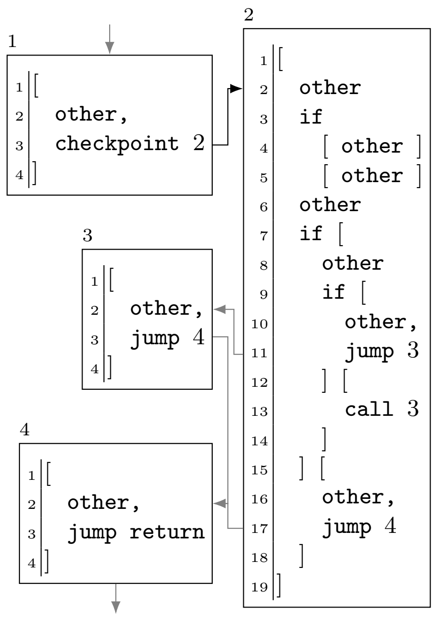
The goal with the decomposition is to ensure that we *immediately* return control to the inference engine at checkpoints. In the PCFG framework, the only way this can be fulfilled is by ensuring that checkpoints are located at tail position in basic blocks. The first example of this is the `resample` checkpoint at line 4 in Fig. 5b, causing a split into the blocks 1 and 2 in the compiled RootPPL PCFG in Fig. 5c. Note that in block 1, `NEXT` is set to 2 at line 7 before returning, indicating that the inference engine should resume execution at block 2 when the checkpoint has been handled, also illustrated by a control-flow checkpoint arrow in Fig. 5c. Note the stack frame pointer `sf` in block 1 for this invocation of `f`, which points to a location in an explicit call stack in the RootPPL program state `PSTATE`. This is required as a result of compiling to PCFGs—*any* data that must be passed between basic blocks (e.g., a call stack) *has* to be put in the program state. The stack frame pointer `sf` is defined in the same way at the top of all blocks for the decomposed function `f` in Fig. 5c, but is replaced with ... in blocks other than the first for brevity.

It is not enough to split into blocks at explicit checkpoints. Consider, for example, the recursive call to `f` in the `else` branch on line 12 in Fig. 5b. During this function call, at least one checkpoint (`resample`) will be encountered, causing at least one block split within the function, meaning that all data required by `f` must be put in an explicit stack frame and stored in the program state. It would otherwise be lost in between the basic blocks of `f`. In particular, the block return address `ra` is stored in the stack frame, indicating which block to return to at the end of the function call. In the case of the call to `f` at line 12 in Fig. 5b, we must return to line 13. Therefore, we must ensure that line 13 is located at the beginning of its own basic block in Fig. 5c (block 3). In general, all calls to decomposed functions (i.e., functions that may, directly or indirectly, encounter a checkpoint) must be located at tail position in their basic blocks. Besides line 13 in Fig. 5b, note that this also means that line 15 in Fig. 5b cannot be part of block 2. In fact, it cannot be part of block 3 either, because it may be executed independently of line 13 in Fig. 5b if the `else` branch of the `if` at line 9 in Fig. 5b is taken. As a result, it must be put in its own block (block 4 in Fig. 5c). The decomposition of function applications and `if` expressions is similar to how standard compilers decompose sequences of machine instructions into basic blocks (sequences of instructions without any internal jumps or branches) [9]. The difference, however, is that we do not split into blocks at *all* `if` expressions and function calls. For example, the `if` at line 6 in Fig. 5b is guaranteed not to include a checkpoint, and can therefore be left untouched (lines 4–5 in Fig. 5c). Similarly, the call to `geqf` at line 5 in Fig 5b is guaranteed not to encounter any checkpoints. Conservatively determining which functions are guaranteed not to encounter any checkpoints can be done through static analysis. For example, such a static analysis phase is part of the CorePPL compiler, described in Section 4.3.

Let us now take a closer look at the call stack handling in Fig. 5c. The following description is specific for RootPPL, but similar solutions must be applied if compiling to other target languages utilizing PCFGs. First, the program state `PSTATE` consists of a byte array `stack`, and a pointer to the top of this stack named `stackPtr`. This stack pointer is incremented and decremented when stack frames are added and removed, respectively, at function calls and returns. The type `STACK_f` represents the stack frame for the function `f` (such a stack frame type must be determined and set up for each function we decompose), and contains its block return address `ra`, its parameter `p` (functions with multiple parameters have one entry for each parameter), and an address `retValLoc` at which its return value must be written. Additionally, it contains the local variables `s1`, `s3`, and `s4` that travel across the blocks in `f`. Note, however, that local variables used only within a single block do not need to go in the stack frame (e.g., `t1` and `s2`), and can instead be handled directly by the underlying target language (e.g., CUDA for RootPPL). The recursive call to `f` at line 12 in Fig. 5b is illustrated at lines 13–24 in block 2 in Fig. 5c. Here, we allocate a new complete stack frame `callsf`, and initialize `ra`, `p`, and `retValLoc`. Allocating the complete stack frame prior to the function call is different compared to most ordinary compilers, where the part of the stack frame containing local variables is most often allocated at the start of the called function. This allows for making the allocation size dependent on, e.g., function arguments. Here, all stack frame sizes are instead known at compile time. When the stack frame has been set up, the stack pointer is incremented at lines 21–23, and control is passed to the recursive invocation of `f` by using `BBLOCK_JUMP` at line 24. Inversely, function return is illustrated in block 4 on lines 3–7. First, the return value is set, and

```
 1 [
 2   other,
 3   checkpoint,
 4   other,
 5   if [other] [other],
 6   other,
 7   if [
 8     other,
 9     if
10       [other]
11       [call],
12     other,
13   ] [other],
14   other
15 ]
```

(a) The program from Fig. 5b translated to type [**stmt**].



```
1
[
  other,
  checkpoint 2
]
```

```
2
[
  other
  if
    [ other ]
    [ other ]
  other
  if [
    other
    if [
      other,
      jump 3
    ] [
      call 3
    ]
  ] [
    other,
    jump 4
  ]
]
```

```
3
[
  other,
  jump 4
]
```

```
4
[
  other,
  jump return
]
```

(b) Decomposition of (a) into [**tstmt**] basic blocks.

Fig. 6: Illustrating Algorithm 2 on the example from Fig. 5.

second, the stack pointer is decremented. Finally, the return block is retrieved from the stack frame, and control is passed to this block at line 7.

## 4.2 Function Decomposition Algorithm

We now turn to a formal description of the decomposition algorithm. To avoid going into specifics of the underlying target language, and in particular the call stack handling, we take a more abstract view of function bodies, and regard them as lists of statements of the form

$$\textbf{stmt} ::= \texttt{checkpoint} \mid \texttt{call} \mid \texttt{if}\ [\textbf{stmt}]\ [\textbf{stmt}] \mid \texttt{other}. \qquad (3)$$

Here, the [**stmt**] syntax indicates a list of **stmt**s. Thus, the `if` construct inductively contains two lists of **stmt**s.

The representation **stmt** is best illustrated through an example. Consider again the program in Fig. 5b, and its mapping to **stmt**s in Fig. 6a. Due to ANF, we can view the body of `f` as a sequence of `let` bindings and operations separated by `;`, each performing a single operation of some kind (e.g., a checkpoint or a function application). Each such operation is mapped to a **stmt** in Fig. 6a. The `resample` checkpoint at line 4 in Fig. 5b is mapped to a `checkpoint` at line 3 in Fig. 6a, and the application of `f` at line 12 is mapped to a `call` at line 11. Other applications, such as `geqf` and `leqf`, are, however, guaranteed to not encounter any checkpoints. Therefore, they are mapped to `other`s, and *not* `call`s. The three `if`s at lines 6, 9, and 12, are mapped to `if`s. Note that the `if` conditions in Fig. 5b are always lifted to a separate `let` as a result of ANF, and are therefore not part of the `if` in **stmt**. All remaining operations are treated equally when decomposing, and are therefore simply mapped to `other`s.

While the illustration above only shows how to map a CorePPL function body to **stmt**s, the representation is general. For example, in the CorePPL compiler described in Section 4.3, the decomposition is actually performed *after* translation to C, and not at the CorePPL stage. This is simply because, at the CorePPL level, there is no notion of basic blocks. It is therefore more natural to perform this translation closer to RootPPL.

We now turn to the full decomposition algorithm over lists of **stmt**s, given in Algorithm 2. The target language representation is a small extension of **stmt**, adding transitions between $\mathbb{N}$-indexed basic blocks. It is given by

$$\begin{aligned}\textbf{tstmt} ::= {} & \texttt{checkpoint}\ \textbf{next} \mid \texttt{call}\ \textbf{next} \\ & \mid \texttt{if}\ [\textbf{tstmt}]\ [\textbf{tstmt}] \mid \texttt{jump}\ \textbf{next} \mid \texttt{other}.\end{aligned} \tag{4}$$

In particular, we annotate `checkpoint`s and `call`s with the type **next**, given by **next** $::=$ `return` $\mid n$, where $n \in \mathbb{N}$. For `checkpoint`s, the **next** indicates which block to jump to after the checkpoint has been handled, and for `call`s, it indicates the block to *return to* (e.g., the value set for `ra` in Fig 5c) at the end of the function invocation. We also include a `jump` in **tstmt** for directly jumping to another block (corresponding to `BBLOCK_JUMP` in Fig. 5c). The `return` case of **next** indicates that the next block is given by the return address for the current function call. For example, `BBLOCK_JUMP(sf->ra)` is equivalent to `jump return`.

Fig. 6b shows the result of applying Algorithm 2 on the [**stmt**] in Fig. 6a. Note that the block structure in Fig. 6b mirrors that of Fig. 5c. The entry point in Algorithm 2 is the function DECOMPOSE, which accepts a [**stmt**] as input, and produces a map from indices to [**tstmt**] as output (e.g., Fig 6b). The core of Algorithm 2 is the function REC, which recursively constructs the basic blocks. It is called from DECOMPOSE, and makes use of the function INITNEXT. The accumulator is the triple $(\mathsf{block}, \mathsf{blocks}, \mathsf{next})$ of type **acc** $::= [\textbf{stmt}] \times (\mathbb{N} \to [\textbf{stmt}]) \times \textbf{next}_+$, where $\mathsf{block}$ is the current block being constructed, $\mathsf{blocks}$ are all blocks constructed so far, and $\mathsf{next}$ indicates the action to take at tail position in the current block. The type $\textbf{next}_+$ is defined as $\textbf{next}_+ ::= \textbf{next} \mid$ `none`. When reaching the end of a block, a value `none` for next means do nothing, a value `return` indicates that the next block is the return block for the current function invocation, and a natural number $n$ means that the next block has index $n$.

We will now walk through the translation of Fig. 6a to Fig. 6b. The accumulator is set to $([], \varnothing,$ `return`$)$ at line 2 in Algorithm 2 just before the initial call to REC, indicating that the current block is empty, that we have accumulated no complete blocks so far, and that we must use the return block address when reaching the end of the current block. In the first call to REC, the `other` at line 2 in Fig. 6a causes the case at line 23 in Algorithm 2 to be triggered, which simply

**Algorithm 2** A functional-style algorithm for function decomposition into basic blocks. Tuples are denoted with comma-separated expressions within parentheses, and sequences with comma-separated items within square brackets. Type annotation is denoted with the : character, the cons operator with :: characters, and sequence concatenation with ++. The non-pure function newIndex returns a unique number from $\mathbb{N}$ at every call.

```
function DECOMPOSE srcs: [stmt] → (ℕ → [tstmt]) =
  let (block, blocks, _) = REC ([], ∅, return) srcs in
  blocks ∪ (newIndex (), block)

function INITNEXT next: next₊ → next =
  match next with none → newIndex () | _ → next

function REC (block, blocks, next) srcs: acc → [stmt] → acc =
  match srcs with
  | [] → match next with
    | none → (block, blocks, next)
    | n | return → (block ++ [jump next], blocks, next)
  | src :: srcs → match src with
    | checkpoint | call → match srcs with
      | [] →
        let next = INITNEXT next in
        (block ++ [src next], blocks, next)
      | _ −>
        let index = newIndex () in
        let block = block ++ [src index] in
        let (nextBlock, blocks, next) = REC ([], blocks, INITNEXT next) srcs in
        (block, blocks ∪ (index, nextBlock), next)
    | other → REC (block ++ [other], blocks, next) srcs
    | if thn els → match srcs with
      | [] →
        let (thn, thnBlocks, thnNext) = REC ([], blocks, next) thn in
        let (els, elsBlocks, elsNext) = REC ([], thnBlocks, thnNext) els in
        let thn = if next ≠ elsNext ∧ thnNext = none
          then thn ++ [jump elsNext] else thn in
        (block ++ [if thn els], elsBlocks, elsNext)
      | _ →
        let (thn, thnBlocks, thnNext) = REC ([], blocks, none) thn in
        let (els, elsBlocks, elsNext) = REC ([], thnBlocks, thnNext) els in
        if elsNext = none then REC (block ++ [if thn els], elsBlocks, next) srcs
        else
          let thn = if thnNext = none then thn ++ [jump elsNext] else thn in
          let (nextBlock, blocks, next) =
            REC ([], elsBlocks, INITNEXT next) srcs in
          (block ++ [if thn els], blocks ∪ (elsNext, nextBlock), next)
```

accumulates the `other` in the current block. Next, the `checkpoint` triggers the case at line 14, followed by line 18, since the `checkpoint` is not at tail position. Here, at line 19, a new index is first created for the following block. The current block is then closed by tagging the `checkpoint` with the new index, resulting in block 1 in Fig. 6b. Next, the block following the `checkpoint` is recursively created at line 21. Finally, the recursively created block is added with the new index to the map of complete blocks (now also populated by the recursive call) and the updated accumulator triple is returned at line 22.

The complex part of Algorithm 2 involves handling of `if`s. In particular, cases where there are block splits within the branches must be handled with care. In our example, the first `if` at line 5 in Fig. 6a triggers the case at line 31, since it is not in tail position. To determine whether or not there is at least one split within the branches, we set next to `none` for the call on line 32. If a block is split during this call, INITNEXT will be applied on next, and thnNext at line 26 will be a natural number, indicating where the branch jumped to (either through a `jump`, `checkpoint`, or `call`) at tail position. However, if there is no split in the branch, the resulting thnNext remains `none`. This is the case for the `if` at line 5 in Fig. 6a, and `none` is passed to the recursive call at line 33 as well. Again, there is no split in the second branch, triggering the then case at line 34, and the `if` is simply accumulated in the same way as an `other`.

On the other hand, the `if`s at lines 7 and 9 in Fig. 6a do contain a split due to the `call` at line 11, resulting in blocks 2, 3, and 4, shown in Fig. 6b. The elsNext is a natural number for these `if`s, and the else case at line 35 is triggered. Here, particular care must be taken if there is only a split in the second branch of the `if`, and not the first. In that case, thnNext will be `none`, and unlike the second branch, no block jump is added to the end of this branch in the call at line 32. Therefore, we must instead add it at line 36. This is, e.g., how the `jump` at line 11 in block 2 in Fig. 6b is added. Note that an equivalent step to the above is not required for the second branch given that the split is only in the first branch, since the next from the first branch is passed to the recursive call for the second branch. After handling the `if` itself, we recursively create the new block following the `if` (note that we pass the next given as argument to REC here, and use INITNEXT on it to indicate a split has occurred), and give it the index that is created with INITNEXT in the call at line 32 or 33.

The case where `if` is at tail position, at line 25, is handled similarly to the case at line 31. The difference is that we do *not* pass `none` to the first branch, since there is nothing following the `if` which can be jumped to. Instead, we directly pass the current next to the first call at line 26.

In the blocks resulting from Algorithm 2, `call` and `checkpoint` only occurs in tail-position. The reason is that blocks are always closed directly after `call`s and `checkpoint`s. This is precisely the required property when compiling to PCFGs, as discussed in Section 4.1.

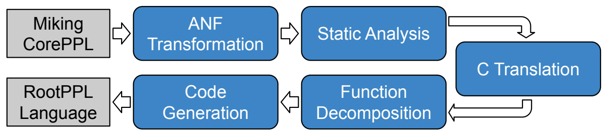


Fig. 7: The main components of the CorePPL-to-RootPPL compiler. Grey blocks are programs, and blue blocks are transformations or analyzes.

### 4.3 CorePPL-to-RootPPL Compiler

An overview of the CorePPL-to-RootPPL compiler components is given in Fig. 7. Besides the techniques described previously, an integral part of the compiler is the C translation step, which translates many of the CorePPL language features to C, including data type definitions and pattern matching. More precisely, CorePPL records and variants are translated to C structs and tagged unions, respectively, while pattern matching is compiled to C `if` statements.

A static analysis phase is also part of the compiler. This phase discovers what functions are guaranteed to not encounter any `resample`s. These functions do not need to be decomposed, and invocations can be handled directly by the C++ or CUDA compiler (and an explicit stack frame need not be set up). An example of such a function invocation is the `geqf s1 1.` at line 5 in Fig. 5b.

The final code generation stage shown in Fig. 7 adds RootPPL boilerplate code and emits a complete RootPPL program that can be provided as input to a C++ or CUDA compiler together with the RootPPL inference engine (see Fig. 1). The full CorePPL compiler implementation is hosted at GitHub [4], and consists of approximately 3000 lines of compiler code (a contribution of this paper). Note that the ANF, static analysis, and C translation steps are quite standard, with no new contributions.

An important final detail with respect to memory allocation in the compiler is the translation between relative and absolute addresses. This translation is illustrated in Fig. 5c. On line 3 in block 4, the `retValLoc` relative pointer is converted to an absolute pointer prior to dereferencing, and at lines 18-20 in block 2, the address of `s4` is translated to a relative address with respect to the start of the stack before being assigned to `retValLoc`. The reason for this is that at checkpoints in RootPPL, different SMC executions can be either moved or copied in memory as part of resampling. This means that absolute addresses cannot be used for referring to data on the `PSTATE` stack, and addresses relative to the start of the stack must be used instead.

### 4.4 Compiler Strengths and Limitations

The main strength of the CorePPL compiler, compared to using other PPL compilers and tools, is execution time of the compiled programs. In particular, the compilation from a universal PPL to CUDA is the first of its kind, and allows for utilizing GPUs for massively parallel SMC inference. There are, however, some limitations with the current compiler. Most importantly, the lack of standard garbage collectors in C++, and in particular CUDA, leads to restrictions for automatic data allocation and deallocation. Currently, only stack-based allocation is supported, which means that CorePPL programs that allocate and return complex data structures (including closures) from functions cannot be compiled. Therefore, first-class distributions are also not supported, and distributions are restricted to occur immediately at `assume`s, e.g. as the Bernoulli distribution in `assume (Bernoulli p)` in Fig. 2a. Exploring the use of garbage collectors, or other means for automatic memory management, is an interesting future research direction for CorePPL and RootPPL.

Furthermore, we disallow passing functions as arguments to other functions as it greatly complicates the static analysis required for determining `resample`-free functions. A potential solution is to use static analysis techniques such as 0-CFA [31].

## 5 Evaluation

In this section, we evaluate the performance of RootPPL and the CorePPL-to-RootPPL compiler. To do this, we compare them to state-of-the-art SMC PPL implementations on two probabilistic models: a constant rate birth-death (CRBD) model from evolutionary biology (Sections 5.1 and 5.3), and a vector-borne disease model from epidemiology (Section 5.2). SMC has been shown to handle these models particularly well [32, 25], and they are therefore good candidates for this evaluation. Comparison with other types of inference algorithms is beyond the scope of this paper.

In addition to CorePPL (compiled to RootPPL) and RootPPL (hand-tuned), we implement the models above in a set of state-of-the-art PPLs with SMC inference: Birch [28], WebPPL [21], and Pyro [11]. For each PPL, we make a good faith effort to implement the two models as efficiently as possible, given the available language features. RootPPL is compiled for single-core and multicore with OpenMP (GCC 7.5.0), as well as for CUDA 11.4. Birch 1.634 is compiled with the same version of GCC, and also makes use of OpenMP. WebPPL 0.9.15 is used with Node.js 14.17.6. Pyro 1.7.0 is used with PyTorch 1.9.0 and CUDA 10.2. Additionally, we use Numba 0.54.0—a just-in-time (JIT) compiler for Python—to improve the performance of Pyro for the experiment in Section 5.1.

To aid the comparison between languages both in the text, and in Fig. 8, Fig. 9, and Fig. 10, we use the (S), (M), and (G) symbols behind PPLs to indicate if they run on single-core, multicore, or GPU, respectively. In spite of the CUDA dependency for Pyro, we did not observe any GPU usage during Pyro SMC runs. SMC is a minor inference algorithm in Pyro, with variational inference instead being the main focus. This may explain this lack of GPU support for SMC. As a consequence, we classify SMC in Pyro as (M), and not (G).

We ran all experiments on a machine with a 12-core (24 threads) Intel Xeon Gold 6136 CPU, 64 GB of memory, and an NVIDIA TITAN RTX GPU with 24 GB of memory and 4608 CUDA cores.

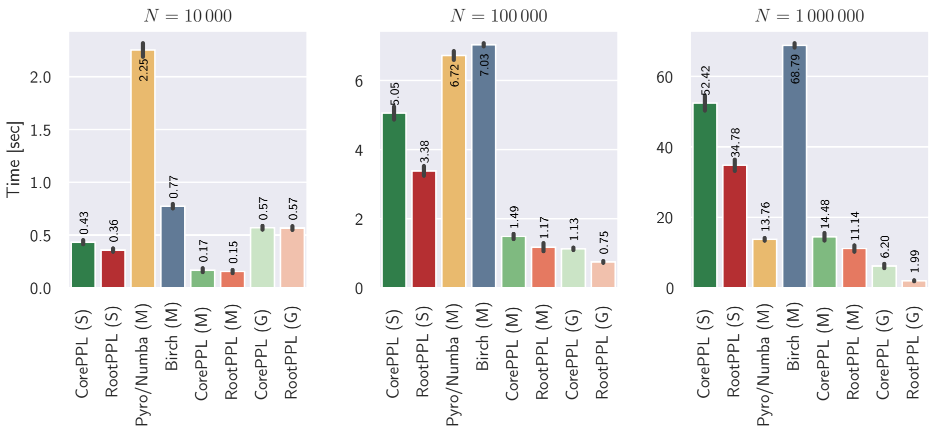


Fig. 8: Execution times for the CRBD experiment, for different numbers of particles $N$. One standard deviation for execution times is given with a black vertical line at the top of each bar. PPLs with an (S) runs on a single core, (M) on multicore, and (G) on the GPU.

## 5.1 Experiment: Constant-Rate Birth Death

In this experiment, we consider the non-trivial CRBD model described in Ronquist et al. [32]. This model encodes the posterior distributions of the rates with which new evolutionary lineages arise (birth rate) and die out (death rate), conditioned on the input of a fixed evolutionary tree (phylogeny). We set the phylogeny to the dated Alcedinidae tree (Kingfisher birds) referenced in Ronquist et al. [32], and introduced in Jetz et al. [23]. A notable feature of this model is that it contains recursive tree constructions, which can only be expressed in universal PPLs. The CorePPL implementation of this model can be found in Appendix B (118 lines of code).

For the experiment, the measure is execution time. In order to ensure fairness, we disabled variance-reducing techniques such as delayed sampling [25] and ESS-triggered resampling in all PPLs where available. As a consequence, all implementations use precisely the same SMC inference algorithm. We checked this, as well as the correctness of the implementations, by considering the output normalizing constant estimates in all runs (see Appendix A). The variance and mean of these estimates were comparable for all PPLs.

The results of the experiment are shown in Fig. 8 for three different numbers of SMC particles: 10 000, 100 000, and 1 000 000. We ran the PPL implementations for 100 iterations (a number determined by available time and hardware) for each number of SMC particles. The exception to this is WebPPL (S) and Pyro (M), which we ran only for 10 000 particles due to very high execution times. For 10 000 particles, averaged over the 100 iterations, WebPPL (S) ran for 55 seconds (standard deviation 0.63 seconds), and Pyro (M) for 250 seconds (standard deviation 28 seconds). For this reason, WebPPL (S) and Pyro (M) are also omitted from Fig. 8. Pyro relies heavily upon vectorization through PyTorch, and the expensive operations in the CRBD model are recursive and stochastic tree constructions, which are difficult to vectorize. This explains the particularly abnormal execution times for Pyro (M).

For this experiment, RootPPL is clearly the best alternative (in all categories). The difference compared to CorePPL can be explained by hand-tuned details in the RootPPL model. In particular, the RootPPL model uses efficient array encodings of the observed tree, and also precomputes the recursion order over this tree and encodes it as an iterative procedure. CorePPL instead compiles the tree as a tagged union type with pointers to subtrees in each node, and traverses it through recursion. Automatically discovering this transformation from trees to arrays and recursion to iteration is non-trivial and not considered here, but could have potential for future work.

In order to improve the performance of Pyro, we also applied Numba to parallelize the recursive tree construction in the model. This is a more fine-grained parallelism compared to the natural SMC particle parallelism, and resulted in an order-of-magnitude performance boost over Pyro (M). In fact, unlike CorePPL, RootPPL, and Birch, the execution times for Pyro/Numba (M) seems to grow sub-linearly when going from 100 000 to 1 000 000 particles, as this only increases mean execution time from 6.72 seconds to 13.76. We conjecture that this is related to the different type of parallelism introduced with Numba, in combination with its JIT compilation. Looking at adding such parallelism to RootPPL and CorePPL is therefore an interesting direction for future work.

## 5.2 Experiment: Vector-Borne Disease

Next, we consider the vector-borne disease model from Funk et al. [17], which is also studied further in Murray et al. [25]. This epidemiological model encodes a dengue outbreak in Micronesia, and includes the spread of disease between mosquito and human populations. The inference is over the number of susceptible, exposed, infectious, and recovered (SEIR) individuals in the populations at discrete time steps (days), and the observations are daily numbers of reported new cases at health centers (the data is available in Funk et al. [17]). The CorePPL implementation of this model can be found in Appendix B (140 lines of code).

The experiment setup is identical to the CRBD experiment in Section 5.1, but with fewer SMC particles due to more demanding computations in the model. The results are shown in Fig. 9. We omit WebPPL (S) completely here due to high execution times. Pyro (M), is included, however, since the simple non-stochastic control-flow in this model allows for much better vectorization compared to CRBD. Pyro/Numba (M) is excluded, as parallelization with Numba on top of the parallelization in Pyro is not possible for this model.

This time, CorePPL is the best option, by a small margin, over RootPPL. This can be explained by how RootPPL preallocates memory, which is dynamically allocated in CorePPL. This results in slightly more memory being moved/-copied during resampling for this model in RootPPL.

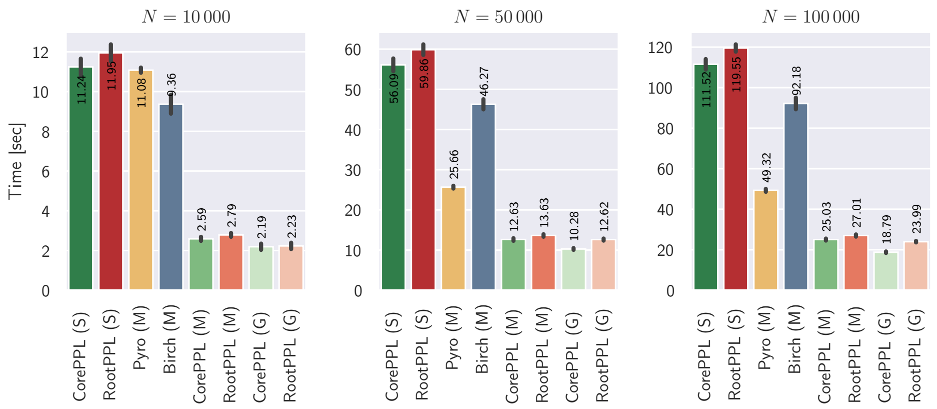


Fig. 9: Execution times for the Vector-Borne Disease experiment, for different numbers of particles $N$. One standard deviation for execution times is given with a black vertical line at the top of each bar. PPLs with an (S) runs on a single core, (M) on multicore, and (G) on the GPU.

The difference between GPU and CPU for CorePPL and RootPPL is not as significant as in Fig. 8. This can be explained by the lower numbers of SMC particles used for this experiment, or by RootPPL using different implementations for binomial distribution sampling on the CPU and GPU. Specifically, the GPU uses a custom, and less efficient version, because the C++ standard library binomial sampling implementation is not available in CUDA. Because binomial sampling is the most expensive operation in this model, this has the potential for improving GPU performance further.

## 5.3 Experiment: CRBD with Variance-Reducing Techniques

In this final experiment, we consider the CRBD model from Section 5.1 again, but now with delayed sampling and ESS-triggered resampling allowed. Also, we now instead consider a different, more challenging, phylogeny of Tyrant flycatchers [32, 23].

The results are shown in Fig. 10. Other than the changes above, the setup is identical to Section 5.1. We added static delayed sampling manually to all models, to ensure fairness. Note, however, that automatic and dynamic delayed sampling, as introduced in Murray et al. [25], is also natively supported in Birch (but introduces some unfair overhead). CorePPL is omitted here, as adding efficient delayed sampling to the model is rendered more difficult by the lack of support for mutable data structures. Mutable data structures could, however, potentially be added to CorePPL and the CorePPL-to-RootPPL compiler. We leave this for future work. Based on the experiment in Section 5.1, WebPPL (S) and Pyro (M) are also not considered here.

The results offer no surprise over Fig 8, and RootPPL is again the best alternative. Note the increased execution times here compared to Fig 8. This

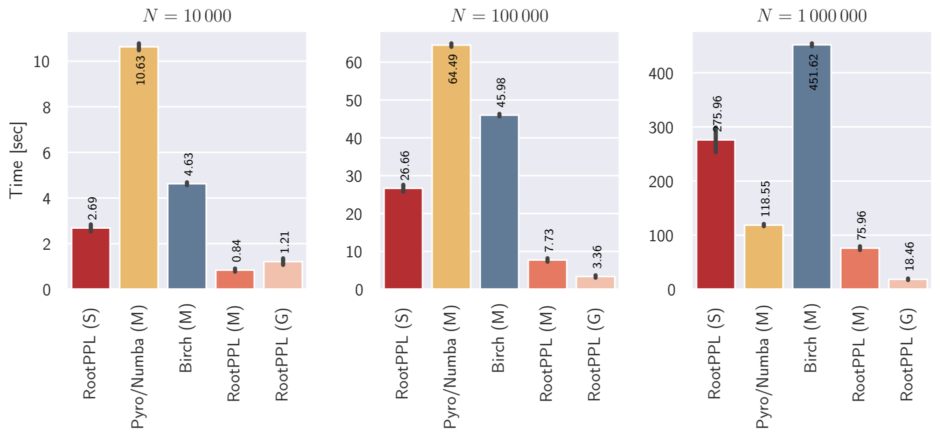


Fig. 10: Execution times for the CRBD experiment with variance-reducing techniques, for different numbers of particles $N$. One standard deviation for execution times is given with a black vertical line at the top of each bar. PPLs with an (S) runs on a single core, (M) on multicore, and (G) on the GPU. Note the $6\times$ speedup of RootPPL (M) over Birch (M) for $N = 100\,000$.

is because of the more challenging phylogeny and delayed sampling overhead (which is greatly compensated for by increase in inference accuracy).

## 6 Related Work

There are quite a few PPL implementations making use of SMC inference. Most closely related to the contributions in this paper is Birch [28]. Similarly to RootPPL, the target language for compilation is C++ and inference is done with SMC. However, while performance is one of the main goals with Birch, some overhead is inevitably introduced by supporting various quality-of-life C++ features—in particular object-oriented features including classes and class inheritance. In RootPPL, such features are not supported, in favor of performance. Similarly to RootPPL, CPU parallelism in Birch is supported through the use of OpenMP. Compilation to GPUs is, however, currently not supported in Birch.

The PCFG concept can also be related to Birch. In Birch, models for SMC inference are written as a method `simulate` that is called iteratively. Resampling is *only* performed in between calls to this method. Furthermore, passing data between calls to `simulate` is done through particle variables in an object defined as part of the model. Clearly, this fits the abstract definition of a PCFG state. Furthermore, PCFG basic blocks can be viewed as a natural generalization of the Birch `simulate` method, conceptually allowing for many `simulate` methods with arbitrary control-flow in between them. As with PCFG blocks, the explicit `simulate` function used in Birch can, potentially, make it more difficult to express models for programmers. This is not a problem when using our approach for compiling to PCFGs, as the block decomposition is then done automatically.

Besides Birch, parallelism for SMC inference in PPLs is surprisingly absent in previous work. The predecessor of Birch, LibBi [26], is an exception to this, and implements highly performant SMC inference through SIMD instructions, OpenMP, and CUDA. However, in contrast with RootPPL and CorePPL, the LibBi modeling language is not universal. In other words, many probabilistic models cannot be expressed in LibBi (which inspired the development of Birch).

Pyro [11] is a relatively recent PPL mainly focused on stochastic variational inference, which also supports MCMC and SMC. Parallelism is supported through vectorization using PyTorch [6] tensors, and is more explicit compared to the natural SMC particle parallelism considered in this paper. For fine-tuning models, the explicit parallelism can, however, be more flexible and allow for further optimizations. We saw this, for instance, when combining Pyro with Numba [5] for the experiments in Section 5.

Other universal PPLs implementing SMC inference include WebPPL [21] and Anglican [36]. These languages are embedded in JavaScript, and Clojure, respectively, and implement a number inference algorithms (including SMC) through CPS transformations. The focus is on ease of modeling through functional-style constructs supported by complex runtimes (V8 for JavaScript and the JVM for Clojure), as well as supporting a large number of different inference algorithms. Parallelism for SMC is not directly supported. This is different compared to CorePPL and RootPPL, where the focus is parallelism and performance.

There are also many other probabilistic programming tools, libraries, and languages available, for instance Gen [14], Turing [18], Hakaru [30], Stan [13], and Edward [34]. Generally, these either focus on assisting users in manually constructing inference algorithms tailored for their specific models, or on providing efficient inference for a restricted set of models.

## 7 Conclusion

In this paper, we have introduced the concept of PCFGs and a general method for compiling universal PPLs to PCFGs. We illustrated these contributions further through the RootPPL implementation and the CorePPL compiler. This is the first time a universal PPL has been compiled to GPU with SMC inference. Furthermore, the evaluation showed that CorePPL and RootPPL not only is able to deal with real-world SMC inference problems, but also outperforms the current state-of-the art with up to $6\times$ speedups for challenging models (and even more when comparing across both CPU and GPU). This gives strong empirical support for the usefulness of the contributions.

Possible improvements upon this work include the exploration of more complex CUDA and C++ runtimes for RootPPL, e.g. runtimes with automatic memory management through garbage collection. Additionally, high-performance implementations similar to RootPPL for other inference methods (e.g. MCMC) are highly relevant for many probabilistic models—for instance various models from phylogenetics [32]. We leave these topics for future work.

# Bibliography

[1] CUDA Toolkit | NVIDIA Developer. https://developer.nvidia.com/cuda-toolkit (2021), accessed: 2021-09-20

[2] GCC, the GNU Compiler Collection - GNU Project. https://gcc.gnu.org/ (2021), accessed: 2021-09-20

[3] Home - OpenMP. https://www.openmp.org/ (2021), accessed: 2021-09-20

[4] Miking DPPL. https://github.com/miking-lang/miking-dppl (2021), accessed: 2021-12-01

[5] Numba: A High Performance Python Compiler. http://numba.pydata.org/ (2021), accessed: 2021-10-11

[6] PyTorch. https://pytorch.org/ (2021), accessed: 2021-10-11

[7] The LLVM Compiler Infrastructure Project. https://llvm.org/ (2021), accessed: 2021-09-20

[8] Thrust - Parallel Algorithms Library. https://thrust.github.io/ (2021), accessed: 2021-09-24

[9] Aho, A.V., Lam, M.S., Sethi, R., Ullman, J.D.: Compilers: principles, techniques and tools. Addison-Wesley (2006)

[10] Appel, A.W.: Compiling with Continuations. Cambridge University Press (1991)

[11] Bingham, E., Chen, J.P., Jankowiak, M., Obermeyer, F., Pradhan, N., Karaletsos, T., Singh, R., Szerlip, P., Horsfall, P., Goodman, N.D.: Pyro: Deep universal probabilistic programming. Journal of Machine Learning Research 20(28), 1–6 (2019)

[12] Broman, D.: A vision of miking: Interactive programmatic modeling, sound language composition, and self-learning compilation. In: Proceedings of the 12th ACM SIGPLAN International Conference on Software Language Engineering. p. 55–60. SLE 2019, Association for Computing Machinery, New York, NY, USA (2019)

[13] Carpenter, B., Gelman, A., Hoffman, M., Lee, D., Goodrich, B., Betancourt, M., Brubaker, M., Guo, J., Li, P., Riddell, A.: Stan: A probabilistic programming language. Journal of Statistical Software, Articles 76(1), 1–32 (2017)

[14] Cusumano-Towner, M.F., Saad, F.A., Lew, A.K., Mansinghka, V.K.: Gen: A general-purpose probabilistic programming system with programmable inference. In: Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 221–236. PLDI 2019, ACM, New York, NY, USA (2019)

[15] Doucet, A., de Freitas, N., Gordon, N.: Sequential Monte Carlo Methods in Practice. Information Science and Statistics, Springer New York (2001)

[16] Flanagan, C., Sabry, A., Duba, B.F., Felleisen, M.: The essence of compiling with continuations. In: Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation. p. 237–247. PLDI '93, Association for Computing Machinery, New York, NY, USA (1993)

[17] Funk, S., Kucharski, A.J., Camacho, A., Eggo, R.M., Yakob, L., Murray, L.M., Edmunds, W.J.: Comparative analysis of dengue and zika outbreaks reveals differences by setting and virus. PLOS Neglected Tropical Diseases 10(12), 1–16 (12 2016)

[18] Ge, H., Xu, K., Ghahramani, Z.: Turing: a language for flexible probabilistic inference. In: International Conference on Artificial Intelligence and Statistics, AISTATS 2018, 9-11 April 2018, Playa Blanca, Lanzarote, Canary Islands, Spain. pp. 1682–1690 (2018)

[19] Gilks, W., Richardson, S., Spiegelhalter, D.: Markov Chain Monte Carlo in Practice. Chapman & Hall/CRC Interdisciplinary Statistics, Taylor & Francis (1995)

[20] Goodman, N.D., Mansinghka, V.K., Roy, D., Bonawitz, K., Tenenbaum, J.B.: Church: A language for generative models. In: Proceedings of the Twenty-Fourth Conference on Uncertainty in Artificial Intelligence. pp. 220–229. AUAI Press (2008)

[21] Goodman, N.D., Stuhlmüller, A.: The design and implementation of probabilistic programming languages. http://dippl.org (2014), accessed: 2020-07-09

[22] Gordon, A.D., Henzinger, T.A., Nori, A.V., Rajamani, S.K.: Probabilistic programming. In: Future of Software Engineering Proceedings. p. 167–181. FOSE 2014, Association for Computing Machinery, New York, NY, USA (2014)

[23] Jetz, W., Thomas, G.H., Joy, J.B., Hartmann, K., Mooers, A.O.: The global diversity of birds in space and time. Nature 491(7424), 444–448 (Nov 2012)

[24] Lundén, D., Borgström, J., Broman, D.: Correctness of sequential monte carlo inference for probabilistic programming languages. In: Yoshida, N. (ed.) Programming Languages and Systems. pp. 404–431. Springer International Publishing, Cham (2021)

[25] Murray, L., Lundén, D., Kudlicka, J., Broman, D., Schön, T.: Delayed sampling and automatic rao-blackwellization of probabilistic programs. In: Proceedings of the Twenty-First International Conference on Artificial Intelligence and Statistics. vol. 84, pp. 1037–1046. PMLR (2018)

[26] Murray, L.M.: Bayesian state-space modelling on high-performance hardware using libbi. arXiv e-prints p. arXiv:1306.3277 (2013)

[27] Murray, L.M., Lee, A., Jacob, P.E.: Parallel resampling in the particle filter. Journal of Computational and Graphical Statistics 25(3), 789–805 (2016)

[28] Murray, L.M., Schön, T.B.: Automated learning with a probabilistic programming language: Birch. Annual Reviews in Control 46, 29–43 (2018)

[29] Naesseth, C., Lindsten, F., Schön, T.: Elements of Sequential Monte Carlo. Foundations and Trends in Machine Learning Series, Now Publishers (2019)

[30] Narayanan, P., Carette, J., Romano, W., Shan, C., Zinkov, R.: Probabilistic inference by program transformation in hakaru (system description). In: International Symposium on Functional and Logic Programming - 13th International Symposium, FLOPS 2016, Kochi, Japan, March 4-6, 2016, Proceedings. pp. 62–79. Springer (2016)

[31] Nielson, F., Nielson, H.R., Hankin, C.: Principles of Program Analysis. Springer-Verlag (1999)

[32] Ronquist, F., Kudlicka, J., Senderov, V., Borgström, J., Lartillot, N., Lundén, D., Murray, L., Schön, T.B., Broman, D.: Universal probabilistic programming offers a powerful approach to statistical phylogenetics. Communications Biology 4(1), 244 (Feb 2021)

[33] Tolpin, D., van de Meent, J.W., Yang, H., Wood, F.: Design and implementation of probabilistic programming language anglican. In: Proceedings of the 28th Symposium on the Implementation and Application of Functional Programming Languages. IFL 2016, Association for Computing Machinery, New York, NY, USA (2016)

[34] Tran, D., Kucukelbir, A., Dieng, A.B., Rudolph, M., Liang, D., Blei, D.M.: Edward: A library for probabilistic modeling, inference, and criticism. arXiv e-prints p. arXiv:1610.09787 (2016)

[35] Wainwright, M.J., Jordan, M.I.: Graphical models, exponential families, and variational inference. Foundations and Trends in Machine Learning 1(1–2), 1–305 (2008)

[36] Wood, F., Meent, J.W., Mansinghka, V.: A new approach to probabilistic programming inference. In: Proceedings of the Seventeenth International Conference on Artificial Intelligence and Statistics. vol. 33, pp. 1024–1032. PMLR (2014)

## A Normalizing Constant Experiment Estimates

This section contains normalizing constant estimates produced during the experiments in Section 5, and can be used to justify the equivalence and correctness of the implementations in the various PPLs. The estimates are shown in Fig. 11, Fig. 12, and Fig. 13.

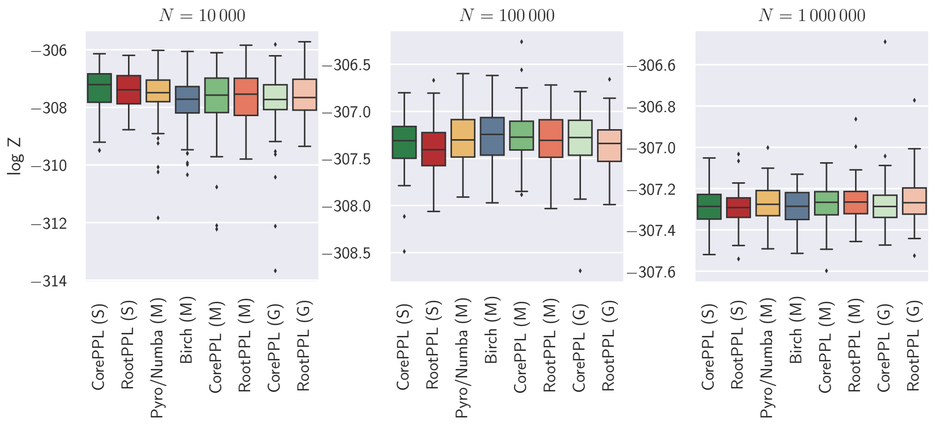


Fig. 11: Box plots of the (log) normalizing constant estimates $Z$ for the CRBD experiment in Section 5.1. Mirrors the structure of Fig. 8.

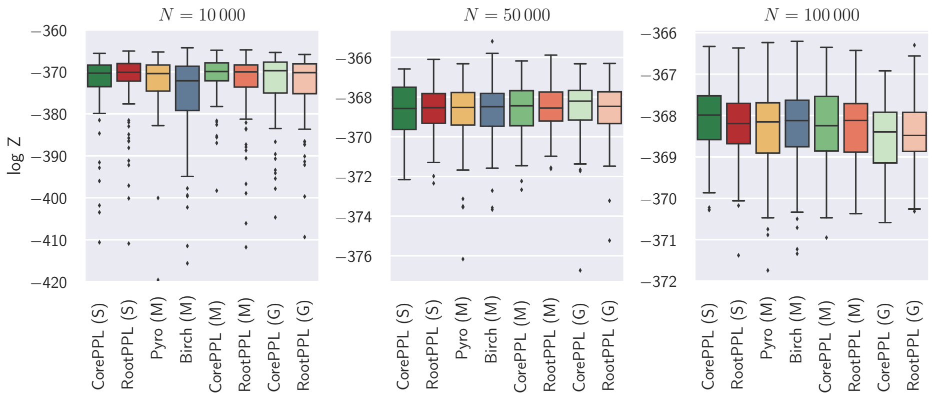


Fig. 12: Box plots of the (log) normalizing constant estimates $Z$ for the SSM experiment in Section 5.2. Mirrors the structure of Fig. 9.

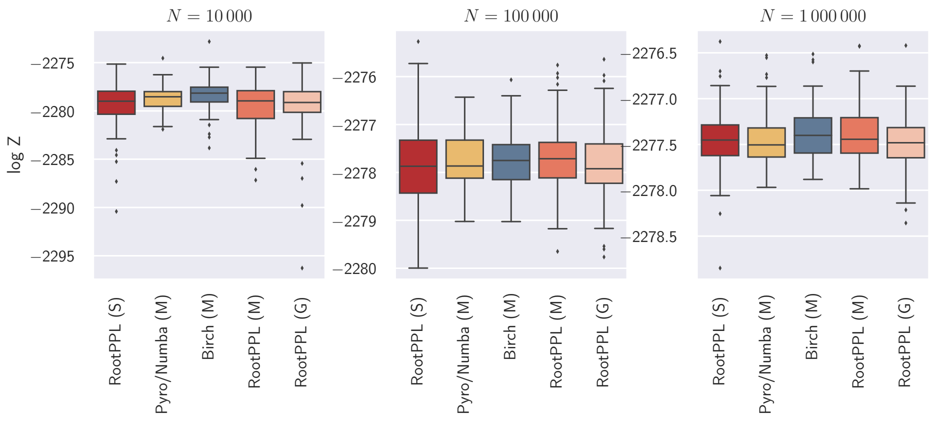


Fig. 13: Box plots of the (log) normalizing constant estimates $Z$ for the CRBD experiment with variance-reducing techniques in Section 5.3. Mirrors the structure of Fig. 10.

## B CorePPL Experiment Source Code

This section contains the models used for CorePPL in the experiments presented in Section 5.1 and Section 5.2. These are presented in Listing 1 and Listing 2, respectively.

Listing 1: The CRBD model used for Section 5.1

```
-----------------------------------------------
-- The Constant-Rate Birth-Death (CRBD) model --
-----------------------------------------------

-- The prelude includes a few PPL helper functions
include "pplprelude.mc"

-- The tree.mc file defines the general tree structure
include "tree.mc"

-- The tree-instance.mc file includes the actual tree and the rho constant
include "tree-instance.mc"

mexpr

-- CRBD goes undetected, including iterations. Mutually recursive functions.
recursive
  let iter: Int -> Float -> Float -> Float -> Float -> Float -> Bool =
    lam n: Int.
    lam startTime: Float.
    lam branchLength: Float.
    lam lambda: Float.
    lam mu: Float.
    lam rho: Float.
      if eqi n 0 then
        true
      else
        let eventTime = assume (Uniform (subf startTime branchLength) startTime) in
        if crbdGoesUndetected eventTime lambda mu rho then
          iter (subi n 1) startTime branchLength lambda mu rho
      else
        false

  let crbdGoesUndetected: Float -> Float -> Float -> Float -> Bool =
    lam startTime: Float.
    lam lambda: Float.
    lam mu: Float.
    lam rho: Float.
      let duration = assume (Exponential mu) in
      let cond =
        -- 'and' does not use short-circuiting: using 'if' as below is more
        -- efficient
        if (gtf duration startTime) then
          (eqBool (assume (Bernoulli rho)) true)
        else false
      in
      if cond then
        false
      else
        let branchLength = if ltf duration startTime then duration else startTime in
        let n = assume (Poisson (mulf lambda branchLength)) in
        iter n startTime branchLength lambda mu rho
in

-- Simulation of branch
recursive
let simBranch: Int -> Float -> Float -> Float -> Float -> Float -> Float =
  lam n: Int.
  lam startTime: Float.
  lam stopTime: Float.
  lam lambda: Float.
  lam mu: Float.
  lam rho: Float.
    if eqi n 0 then 0.
    else
      let currentTime = assume (Uniform stopTime startTime) in
      if crbdGoesUndetected currentTime lambda mu rho then
        let v = simBranch (subi n 1) startTime stopTime lambda mu rho in
        addf v (log 2.)
      else
        negf inf
in

-- Simulating along the tree structure
recursive
let simTree: Tree -> Tree -> Float -> Float -> Float -> () =
  lam tree: Tree.
  lam parent: Tree.
  lam lambda: Float.
  lam mu: Float.
  lam rho: Float.
    let lnProb1 = mulf (negf mu) (subf (getAge parent) (getAge tree)) in
    let lnProb2 = match tree with Node _ then log lambda else log rho in

    let startTime = getAge parent in
    let stopTime = getAge tree in
    let n = assume (Poisson (mulf lambda (subf startTime stopTime))) in
    let lnProb3 = simBranch n startTime stopTime lambda mu rho in

    weight (addf lnProb1 (addf lnProb2 lnProb3));
    resample;

    match tree with Node { left = left, right = right } then
      simTree left tree lambda mu rho;
      simTree right tree lambda mu rho
    else ()
in

```

```
-- Priors
let lambda = assume (Gamma 1.0 1.0) in
let mu = assume (Gamma 1.0 0.5) in

-- Adjust for normalizing constant
let numLeaves = countLeaves tree in
let corrFactor =
  subf (mulf (subf (int2float numLeaves) 1.) (log 2.)) (lnFactorial numLeaves) in
weight corrFactor;
resample;

-- Start of the simulation along the two branches
(match tree with Node { left = left, right = right } then
  simTree left tree lambda mu rho;
  simTree right tree lambda mu rho
else ());

lambda

-- Returns the posterior for the lambda
```

Listing 2: The SSM model used for Section 5.2

```
---------------------------------------------------------------------------
-- The SEIR model from https://docs.birch.sh/examples/VectorBorneDisease/ --
---------------------------------------------------------------------------

-- Include pow and exp
include "math.mc"

-- Include data
include "data.mc"

mexpr

-- Human parameters
let hNu: Float = 0. in
let hMu: Float = 1. in
let hLambda: Float = assume (Beta 1. 1.) in
let hDelta: Float =
  assume (Beta (addf 1. (divf 2. 4.4)) (subf 3. (divf 2. 4.4))) in
let hGamma: Float =
  assume (Beta (addf 1. (divf 2. 4.5)) (subf 3. (divf 2. 4.5))) in

-- Mosquito parameters
let mNu: Float = divf 1. 7. in
let mMu: Float = divf 6. 7. in
let mLambda: Float = assume (Beta 1. 1.) in
let mDelta: Float =
  assume (Beta (addf 1. (divf 2. 6.5)) (subf 3. (divf 2. 6.5))) in
let mGamma: Float = 0. in

-- Other parameters
let rho: Float = assume (Beta 1. 1.) in
let z: Int = 0 in

-- Human SEIR component
let n: Int = 7370 in
let hI: Int = addi 1 (assume (Poisson 5.0)) in
let hE: Int = assume (Poisson 5.0) in
let hR: Int =
  floorfi (assume (Uniform 0. (int2float (addi 1 (subi (subi n hI) hE))))) in
let hS: Int = subi (subi (subi n hE) hI) hR in

-- Human initial deltas
let hDeltaS: Int = 0 in
let hDeltaE: Int = hE in
let hDeltaI: Int = hI in
let hDeltaR: Int = 0 in

-- Mosquito SEIR component
let u: Float = assume (Uniform (negf 1.) 2.) in
let mS: Int = floorfi (mulf (int2float n) (pow 10. u)) in
let mE: Int = 0 in
let mI: Int = 0 in
let mR: Int = 0 in

-- Mosquito initial deltas
let mDeltaS: Int = 0 in
let mDeltaE: Int = 0 in
let mDeltaI: Int = 0 in
let mDeltaR: Int = 0 in

-- Conditioning function
let condition: Int -> Int -> Int -> Int =
  lam t: Int. lam zP: Int. lam hDeltaI: Int.
    let z: Int = addi zP hDeltaI in
    let y: Int = get ys t in
    let z: Int = if neqi (negi 1) y then
      observe y (Binomial z rho); 0
      else z in
    resample;
    z
in

-- Initial conditioning
let z = condition 0 z hDeltaI in

-- Simulation function
recursive let simulate:
  Int -> Int -> Int -> Int -> Int -> Int -> Int -> Int -> Int -> Int -> ()
  =
  lam t: Int.
  lam hSP: Int.
  lam hEP: Int.
  lam hIP: Int.
  lam hRP: Int.
  lam mSP: Int.
  lam mEP: Int.
  lam mIP: Int.
  lam mRP: Int.
  lam zP: Int.

    -- Humans
    let hN: Int = addi (addi (addi hSP hEP) hIP) hRP in
    let hTau: Int =
      assume (Binomial hSP (subf 1. (exp (negf (divf
                                          (int2float mIP)
                                          (int2float hN)))))) in
    let hDeltaE: Int = assume (Binomial hTau hLambda) in
    let hDeltaI: Int = assume (Binomial hEP hDelta) in
    let hDeltaR: Int = assume (Binomial hIP hGamma) in
    let hS: Int = subi hSP hDeltaE in
    let hE: Int = subi (addi hEP hDeltaE) hDeltaI in
    let hI: Int = subi (addi hIP hDeltaI) hDeltaR in
    let hR: Int = addi hRP hDeltaR in

    -- Mosquitos
    let mTau: Int =
      assume (Binomial mSP (subf 1. (exp (negf (divf
                                         (int2float hIP)
                                         (int2float hN)))))) in
    let mN: Int = addi (addi (addi mSP mEP) mIP) mRP in
    let mDeltaE: Int = assume (Binomial mTau mLambda) in
    let mDeltaI: Int = assume (Binomial mEP mDelta) in
    let mDeltaR: Int = assume (Binomial mIP mGamma) in
    let mS: Int = subi mSP mDeltaE in
    let mE: Int = subi (addi mEP mDeltaE) mDeltaI in
    let mI: Int = subi (addi mIP mDeltaI) mDeltaR in
    let mR: Int = addi mRP mDeltaR in

    let mS: Int = assume (Binomial mS mMu) in
    let mE: Int = assume (Binomial mE mMu) in
    let mI: Int = assume (Binomial mI mMu) in
    let mR: Int = assume (Binomial mR mMu) in

    let mDeltaS: Int = assume (Binomial mN mNu) in
    let mS: Int = addi mS mDeltaS in

    let z: Int = condition t zP hDeltaI in

    -- Recurse
    let tNext: Int = addi t 1 in
    if eqi (length ys) t then
      -- We do not return anything here, but simply run the model for
      -- estimating the normalizing constant.
      ()
    else
      simulate tNext hS hE hI hR mS mE mI mR z
in

-- Initiate recursion
simulate 1 hS hE hI hR mS mE mI mR z
```