Compiling Universal Probabilistic Programming Languages with Efficient Parallel Sequential Monte Carlo Inference*

Daniel Lundén[1](✉), Joey Öhman[2], Jan Kudlicka[3], Viktor Senderov[4], Fredrik Ronquist[4,5], and David Broman[1]

[1] EECS and Digital Futures, KTH Royal Institute of Technology, Stockholm, Sweden, {dlunde,dbro}@kth.se
[2] AI Sweden, Stockholm, Sweden, joey.ohman@ai.se
[3] Department of Data Science and Analytics, BI Norwegian Business School, Oslo, Norway, jan.kudlicka@bi.no
[4] Department of Bioinformatics and Genetics, Swedish Museum of Natural History, Stockholm, Sweden, {viktor.senderov,fredrik.ronquist}@nrm.se
[5] Department of Zoology, Stockholm University

Abstract. Probabilistic programming languages (PPLs) allow users to encode arbitrary inference problems, and PPL implementations provide general-purpose automatic 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 probabilistic *checkpoints* in PPLs through 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 GPUs. To solve the checkpoint problem, we introduce the concept of *PPL control-flow graphs* (PCFGs)—a simple and efficient approach to checkpoints in low-level 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. The approach is the first to compile a universal PPL to GPUs with SMC inference. We evaluate RootPPL and the CorePPL compiler 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.

Keywords: Probabilistic Programming Languages · Compilers · Sequential Monte Carlo · GPU Compilation

* This project is financially supported by the Swedish Foundation for Strategic Research (FFL15-0032 and RIT15-0012), the European Union's Horizon 2020 research and innovation program under the Marie Skłodowska-Curie grant agreement PhyPPL (No 898120), and the Swedish Research Council (grant number 2018-04620).

I. Sergey (Ed.): ESOP 2022, LNCS 13240, pp. 29–56, 2022.
https://doi.org/10.1007/978-3-030-99336-8_2

1 Introduction

Probabilistic programming languages (PPLs) allow for encoding a wide range of statistical inference problems and provide *inference algorithms* as part of their implementations. Specifically, PPLs allow language users to focus solely on encoding their statistical problems, which the language implementation then solves automatically. Many such languages exist and are applied in, e.g., statistics, machine learning, and artificial intelligence. Some example PPLs are WebPPL [20], Birch [32], Anglican [40], and Pyro [10].

However, implementing efficient PPL inference algorithms is challenging for many real-world problems. Most often, *universal*[6] PPLs implement general-purpose inference algorithms—most commonly sequential Monte Carlo (SMC) methods [14], Markov chain Monte Carlo (MCMC) methods [18], Hamiltonian Monte Carlo (HMC) methods [12], variational inference (VI) [39], or a combination of these. In some cases, poor efficiency may be due to an inference algorithm 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 tricky without introducing complexity for end-users of PPLs.

A critical performance consideration is handling probabilistic *checkpoints* [37] in PPLs. Checkpoints are locations in probabilistic programs where inference algorithms must interject, for example, to resample in SMC inference or record random draw locations where MCMC inference can explore alternative execution paths. The most common approach to checkpoints—used in universal PPLs such as WebPPL [20], Anglican [40], and Birch [32]—is to associate them with PPL-specific language constructs. In general, PPL users can place these constructs without restriction, and inference algorithms interject through continuation-passing style (CPS) transformations [9,20,40] or non-preemptive multitasking [32] (e.g., coroutines) that enable pausing and resuming executions. These solutions are often not available in languages such as C and CUDA [1] used for high-performance platforms such as graphics processing units (GPUs), making compiling PPLs to these languages and platforms challenging. Some approaches for running PPLs on GPUs do exist, however. LibBi [29] runs on GPUs with SMC inference but is not universal. Stan [12] and AugurV2 [22] partially run MCMC inference on GPUs but have limited expressive power. Pyro [10] runs on GPUs, but currently not in combination with SMC. In this paper, we compile a universal PPL and run it with SMC on GPUs for the first time.

A more straightforward approach to checkpoints, used for SMC in Birch [32] and Pyro [10], is to encode models with a `step` function called iteratively. Checkpoints then occur each time `step` returns. This paper presents a new approach to checkpoint handling, generalizing the `step` function approach. We write probabilistic programs as a set of code blocks connected in what we term a *PPL*

[6] A term due to Goodman et al. [19]. No precise definition exists, but in principle, a universal PPL program can perform probabilistic operations at any point. In particular, it is not always possible to statically determine the number of random variables.

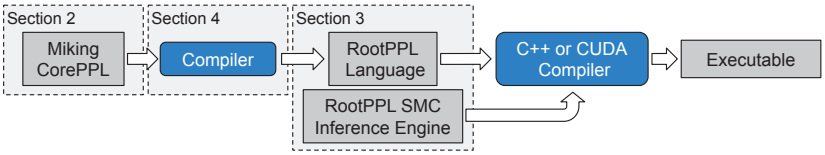


Fig. 1: The CorePPL and RootPPL toolchain. Solid rectangular components (gray) represent programs and rounded components (blue) translations. The dashed rectangles indicate paper sections.

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*. As a result, pausing and resuming executions is straightforward: it is simply a matter of stopping after executing a block and then resuming by running the next block. A variable in the PCFG state, set from within the blocks, determines the next block. This variable allows for loops and branching and gives the same expressive power as other universal PPLs. We implement the above approach 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 and a simple macro-based PPL.

A problem with RootPPL is that it is low-level and, therefore, challenging to write programs in. In particular, sending data between blocks through the PCFG state can quickly get difficult 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 always occur at tail position in blocks. As a result of the decomposition, the PCFG state must store a part of the call stack. 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, a framework for checkpoint handling in PPLs, and use it to implement 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 give 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 compiler in Section 5 on real-world models from phylogenetics and epidemiology, achieving up to 6× speedups over the state-of-the-art. An artifact accompanying this paper supports the evaluation [26]. An extended version of this article is also available [27]. A † symbol in the text indicates more information is available in the extended version.

2 Miking CorePPL

This section introduces the Miking CorePPL language, used as a source language for the compiler in Section 4. We discuss design considerations (Section 2.1) and present the syntax and semantics (Section 2.2).

2.1 Design Considerations

Miking CorePPL (or CorePPL for short) is an *intermediate representation* (IR) PPL, similar to IRs used by LLVM [6] and GCC [2]. This allows the reuse of CorePPL as a target for domain-specific high-level PPLs and PPL compiler back-ends. Consequently, CorePPL needs to be expressive enough to allow easy translation from various domain-specific PPLs and simple enough for practical 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 likelihood updating [21]. CorePPL includes such constructs, including first-class probability distributions, to match the expressive power of existing PPLs.

2.2 Syntax and Semantics

We build CorePPL on top of the *Miking* framework [11]: a meta-language system for creating domain-specific and general-purpose languages. This allows reusing many existing Miking language components and transformations when building the CorePPL language. More precisely, CorePPL extends *Miking Core*—a core functional programming language in Miking—with PPL constructs.

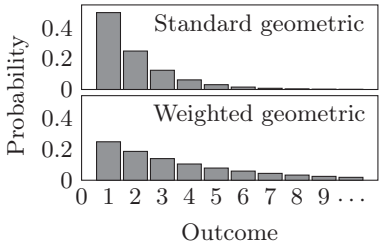
A CorePPL program $\mathbf{t}$ is inductively defined by

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

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 $\texttt{if}\ \mathbf{t}_1\ \texttt{then}\ \mathbf{t}_2\ \texttt{else}\ \mathbf{t}_3$ for $\texttt{match}\ \mathbf{t}_1\ \texttt{with true then}\ \mathbf{t}_2\ \texttt{else}\ \mathbf{t}_3$.

```
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. Part (a) gives the CorePPL program, and part (b) the corresponding distribution. 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.

Consider the simple but illustrative CorePPL 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 program's core is the recursive function `geometric`, defined using a function over the probability of heads for the coin, p. We initially call this function at line 7 with the argument 0.5, indicating a fair coin. On line 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 `true` (heads), we keep flipping the coin by a recursive call to `geometric` and add 1 to this result. To illustrate likelihood updating, we make a contrived modification to the standard geometric distribution by adding `weight (log 1.5)` on line 4. This construct *weights* the execution by a factor of 1.5 each time the result is heads. Note that CorePPL weight computations are in log-space for numerical stability (hence the `log 1.5` to factor by 1.5). Thus, the unnormalized probability of seeing n coin flips, including the final tails, is $0.5^n \cdot 1.5^{n-1}$—where 1.5^{n-1} is the factor introduced by the $n-1$ calls to `weight`. The difference compared to the standard geometric distribution is illustrated in Fig. 2b. The `weight` construct is also commonly named *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. We often use likelihood modification 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 originate 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 the `weight` and `observe` primitives a

likelihood function, and the inference algorithm of the PPL infers the posterior distribution (e.g., the lower part of Fig. 2b)

CorePPL includes sequences, recursive variants, records, and pattern matching, standard in 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 with the constructor name `Leaf`, containing a record with the label `age`. The `match` construct allows pattern matching. 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 data types and pattern matching features in Miking, and consequently CorePPL, are not directly related to the paper's key contributions. Therefore, we do not discuss them further. However, the CorePPL compiler in Section 4.3 supports the features, and the CorePPL models in Section 5 make frequent use of them. We consider CorePPL again in Section 4 when compiling to PCFGs.

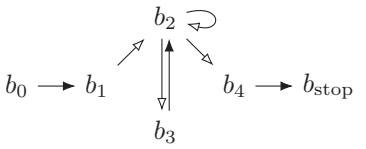
3 PPL control-flow graphs and RootPPL

This section introduces the new PCFG concept (Section 3.1) and shows how to apply SMC over these (Section 3.2). Finally, we present the PCFG and SMC-based RootPPL framework (Section 3.3).

3.1 PPL Control-Flow Graphs

In order to handle checkpoints efficiently without CPS or non-preemptive multitasking, 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 the handling of checkpoints in inference algorithms is greatly simplified, which allows for implementing the framework in low-level languages. However, the explicit checkpoint approach makes PCFGs relatively 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 $(B, S, sim, b_0, b_{\text{stop}}, \mathcal{L})$. The first component B is a set of *basic blocks* inspired by basic blocks used as a part of the control-flow analysis in traditional compilers [8]. In practice, the blocks in B are pieces of code that together make up a complete probabilistic program. Unlike basic blocks used in traditional compilers, we allow these pieces of code to contain branches internally. The second component S is a set of *states*, representing collections of information that flow between basic blocks. In practice, this state often contains local variables that live between blocks and an accumulated likelihood. The blocks and states form the domain of the function $sim : B \times S \to B \times S \times \{\text{false}, \text{true}\}$. This function performs computation specific for the given block over the given state and outputs a *successor* block indicating



(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. Part (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. Part (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 $a_n \coloneqq b_0$ and $c_n \coloneqq \text{false}$.
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 $c_n = \text{false}$, let $(a_n, s_n, c_n) \coloneqq sim(a_n, s_n)$. If all $c_n = \text{true}$, go to 3. If not, repeat 2.
3. **Resampling:** For each $1 \leq n \leq N$, let $p_n \coloneqq \mathcal{L}(s_n) / \sum_{i=1}^N \mathcal{L}(s_i)$. 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 $(s'_n, b'_n) \coloneqq (s_i, b_i)$. Finally, for each $1 \leq n \leq N$, let $(s_n, b_n, c_n) \coloneqq (s'_n, b'_n, \text{false})$. Go to 2.

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 its entry point. Similarly, the block b_{stop} is a unique block indicating termination, which must be reachable from all other blocks. For some initial state $s_0 \in S$, Fig. 3b illustrates a possible execution sequence starting at b_0 in Fig. 3a before terminating at b_{stop}. The structure of a PCFG restricts checkpoints to *only* occur at the end of basic blocks and 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 present how to apply SMC inference to PCFGs. The work by Naesseth et al. [33] contains a more general and pedagogical introduction to SMC. 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 a 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.

Algorithm 1 defines an SMC algorithm over PCFGs. 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. Step 2 simulates all particles that have not yet reached a checkpoint using *sim*. This step repeats until all particles have reached a checkpoint (this is a synchronization point for parallel implementations). Step 3 uses the likelihood function $\mathcal{L}$ to compute the relative likelihoods of all particles and then *resamples* 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_{stop}, the algorithm terminates and returns the current states.

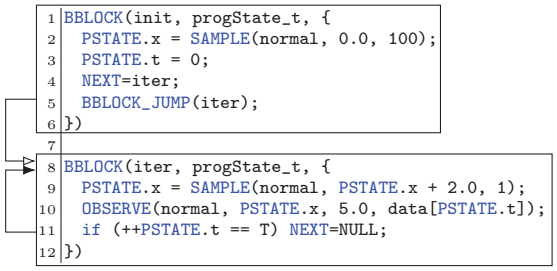
Note in Algorithm 1 that the input states are *not* required to be identical. For example, each state should have a unique seed used to generate random numbers (e.g., with `assume` in CorePPL). Non-identical initial states in Algorithm 1 imply that different particles may traverse the blocks in B differently 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 PCFG property is essential as it allows for the encoding of universal probabilistic programs in PCFG-based PPLs. Furthermore, it implies that some particles may reach b_{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 hide all hardware details from the programmer. To illustrate this macro-based PPL, consider the example RootPPL



```
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 encoding the state-space model in (2). The text provides details. We set `NEXT` at line 4 rather than in `iter` as an optimization. Part (b) defines the RootPPL program state type `progState_t`.

program in Fig. 4a. This program encodes a simple state-space model for an object moving along an axis in $\mathbb{R}$, given by

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

Here, X_0 is the initial position, X_t the following 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.

Fig. 4a implements (2) with 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). Line 10 updates the likelihood using the `OBSERVE` macro (equivalent to `observe` in CorePPL), corresponding to observing Y_1 in the model. We access all Y_t through the `data` array, a shared global constant, avoiding memory duplication in the program state. Finally, at line 11, we check if we are at time `T` (a shared global constant for T). If this is the case, `NEXT` is set to `NULL`, indicating termination. This is equivalent to moving to b_{stop} in the PCFG formalization. Otherwise, `NEXT` keeps its value set at line 4 and jumps to the beginning of the `iter` block. Not using `BBLOCK_JUMP` allows `iter` to return to the inference engine between iterations, indicating checkpoint transitions. In RootPPL, this means that SMC inference will resample the instances before returning to `iter` for the next iteration.

The programmer defines the RootPPL program state for each RootPPL program as an arbitrary C++ struct type and passes this type (e.g., `progState_t`

in Fig. 4a) to each basic block. The `PSTATE` macro accesses the variables in the struct. Fig. 4b illustrates the program state for the example program in Fig. 4a. 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 RootPPL 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. It is crucial to take advantage of the highly parallel nature of SMC and available hardware for parallelization to achieve high performance. For this purpose, RootPPL supports compilation to either C++ on single-core, C++ on multicore through OpenMP [3], and CUDA C++ [1] with massive parallelism on the GPU.

We present the main inference loop in RootPPL below (cf. Algorithm 1).

1. Initialize random seeds.
2. Execute the basic block indicated by `NEXT` for all particles. This execution may include a chain of blocks with non-checkpoint transitions 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. Resample 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 executed sequentially. However, for OpenMP and CUDA, the parallel execution requires that we assign each thread a unique seed shared between all particles running on it. 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, we initialize the seeds in parallel using a CUDA compute kernel.

Step 2 executes the particles sequentially, in parallel using OpenMP threads, or in parallel using a CUDA compute kernel. Step 3 then performs a termination check. First, we check if the first particle has terminated. If it has not terminated, we directly move to the resampling step. If it has terminated, we iteratively check other particles to either find a particle that has not terminated or conclude that all particles have terminated and stop the inference. This approach both allows for particles terminating at different times and introduces minimal overhead for the case when all particles terminate simultaneously (which is quite common). When all particles terminate simultaneously, 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. The reason is the normalizing sum (e.g., $\sum_{i=1}^{N} \mathcal{L}(s_i)$ in Algorithm 1) that we must compute 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. [31] (we do not use in-place propagation). We compute the normalizing sum in parallel via the Thrust library [7] for CUDA.

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 approach, known as memory coalescing, avoids strided memory accesses in global memory and is preferred for parallel operations, particularly for CUDA. Another memory consideration is particle duplication during resampling. For this, we use a custom aligned memory transfer in CUDA because the standard `memcpy` implementation in CUDA proved to be a bottleneck. With a single core and OpenMP, `memcpy` runs 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 not to copy the unused part of the stack located beyond the stack pointer. This is a critical optimization for the CorePPL compiler.

Other things supported in RootPPL are the estimation of *normalizing constants* for encoded models and adaptive resampling based on the current *effective sample size* (ESS). These are standard concepts in SMC inference. For more details, see, e.g., Naesseth et al. [33].

Next, we use RootPPL as the target language for the CorePPL compiler.

4 Compiling to PCFGs

This section introduces the ideas for compiling high-level universal PPLs to PCFGs. We present the key transformation—*function decomposition* into basic blocks—using a toy example (Section 4.1), a formal algorithm (Section 4.2), a high-level overview of the CorePPL-to-RootPPL compiler (Section 4.3), and the compilers strengths and limitations (Section 4.4).

4.1 Function Decomposition Example

The major challenge when compiling high-level PPLs is implementing pausing and resuming at checkpoints to yield control to an inference algorithm temporarily. Pausing and resuming in low-level languages is especially difficult due to runtime limitations. We solve this problem by compiling to the PCFGs introduced in Section 3, specifically designed 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 decomposed into basic blocks in standard compilers such as GCC [2] and LLVM [6] (see, e.g., Aho et al. [8]). 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).

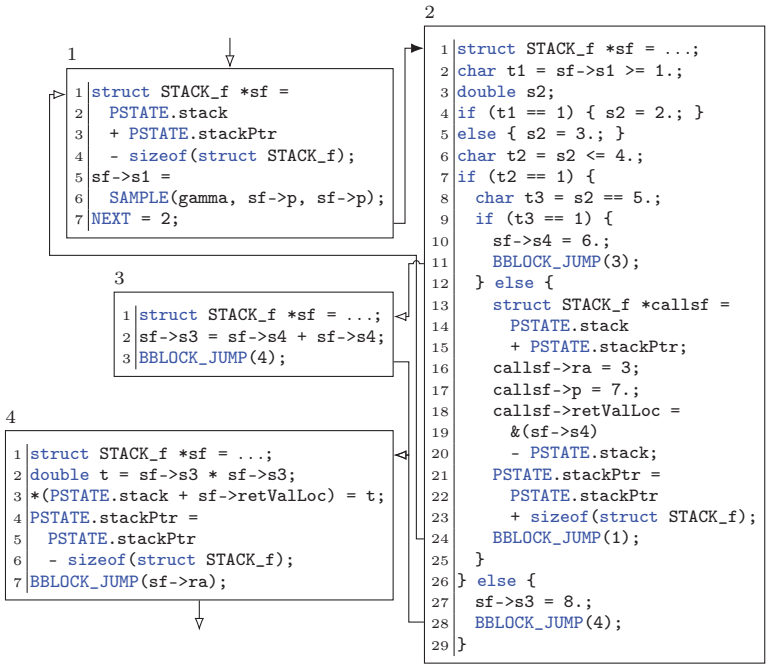
Consider the toy CorePPL function in Fig. 5a and the resulting compilation to a RootPPL PCFG in Fig. 5c. For this example, we introduce an explicit SMC checkpoint `resample` in CorePPL, indicating where SMC should pause

```
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                    1
  resample;
  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                                      2
      let t3 = eqf s2 5. in
      let s4 =
        if t3 then 6. else f 7. in
      addf s4 s4                                    3
    else 8. in
  mulf s3 s3                                        4
in
```

(b) Intermediate ANF representation.



(c) Compiled RootPPL PCFG illustration. Some RootPPL constructs are omitted or slightly modified for readability. In particular, we omit the `BBLOCK` construct used in Fig. 4a. Instead, we illustrate the blocks 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). Part (b) illustrates an intermediate ANF representation of (a) and also indicates the parts of the program corresponding to the blocks in (c). We provide further details in the text.

executions in order to resample. The `resample` construct is the sole checkpoint considered in this example (and the CorePPL compiler), but the method generally applies for arbitrary checkpoints. Optimally, the `resample` construct should be automatically inserted by the compiler [25]. However, we do not consider this problem in this paper and assume `resample`s are inserted prior to compilation. The first step in the decomposition is to translate the program into A-normal form (ANF) [15], 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 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 to fulfill this is to ensure that checkpoints occur at tail position in basic blocks. First, consider the `resample` checkpoint at line 4 in Fig. 5b, causing a split into 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 after handling the checkpoint, also illustrated by a closed arrow. 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`. We require such a call stack due to compiling to PCFGs—*any* data that lives between basic blocks (e.g., a call stack), such as `s1`, *must* be put in the program state. We define the stack frame pointer `sf` equivalently at the top of all blocks for the decomposed function `f` in Fig. 5c but replace the definition with ... in blocks other than the first for brevity.

It is not sufficient 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, we encounter at least one `resample`, resulting in 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. If not, we lose the data 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 place line 13 at the beginning of a basic block in Fig. 5c (block 3). In general, we must place all calls to decomposed functions (i.e., functions that may, directly or indirectly, encounter a checkpoint) at tail position in basic blocks. Besides line 13 in Fig. 5b, this also means that line 15 in Fig. 5b cannot be part of block 2. It cannot be part of block 3 either because it may be executed independently of line 13 in Fig. 5b if we take the `else` branch of the `if` at line 9 in Fig. 5b. Consequently, we must put it in a separate block (block 4 in Fig. 5c). The decomposition of function applications and `if` expressions is similar to how standard compilers decompose machine instructions into basic

blocks (sequences of instructions without any internal jumps or branches) [8]. 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 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. Such a static analysis phase is part of the CorePPL compiler, described in Section 4.3.

We 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`. We increment and decrement this stack pointer 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 we write its return value. 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 the underlying target language (e.g., CUDA for RootPPL) can instead handle them directly. Lines 13–24 in block 2 in Fig. 5c illustrate the recursive call to `f` at line 12 in Fig. 5b. 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 from most standard compilers, which most often allocate the part of the stack frame containing local variables at the start of the called function. This strategy allows for making the allocation size dependent on, e.g., function arguments. Here, we instead know all stack frame sizes at compile time. After setting up the stack frame, we increment the stack pointer at lines 21–23 and pass control to the recursive invocation of `f` by using `BBLOCK_JUMP` at line 24. Inversely, we illustrate function return in block 4 on lines 3–7. First, we set the return value, and second, we decrement the stack pointer. Finally, we retrieve the return block from the stack frame and pass control to this block at line 7.

4.2 Function Decomposition Algorithm

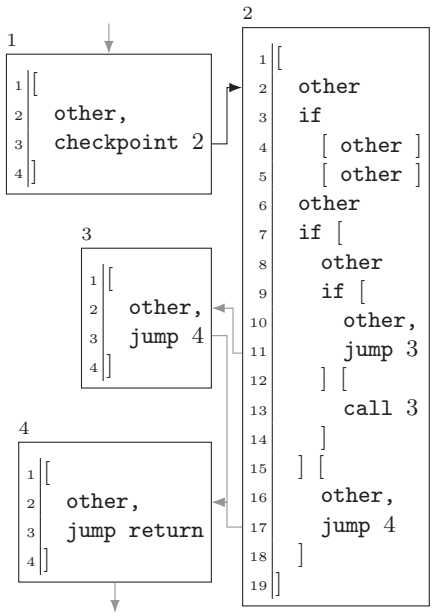
We now turn to a formal description of the decomposition algorithm. To avoid going into specifics of the underlying target language, and in particular the call stack handling, we take an abstract view of function bodies and regard them as lists of statements of the form

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

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

```
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**].



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

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

We illustrate the representation **stmt** through an example. Consider 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). We map each such operation to a **stmt** in Fig. 6a. The `resample` checkpoint at line 4 in Fig. 5b maps to a `checkpoint` at line 3 in Fig. 6a, and the application of `f` at line 12 maps to a `call` at line 11. However, other applications, such as `geqf` and `leqf`, are guaranteed not to encounter any checkpoints. Therefore, they map to `other`s, and *not* `call`s. The three `if`s at lines 6, 9, and 12 map to `if`s. Note that we always lift the `if` conditions in Fig. 5b to a separate `let` as a result of ANF, and they are therefore not part of the `if` representation in **stmt**. We map all remaining operations 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 (Section 4.3), the decomposition is performed *after* translation to C, and not at the CorePPL stage. The reason is that there are no basic blocks in CorePPL. 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**,

Algorithm 2 A functional-style algorithm for function decomposition into basic blocks. We denote tuples with comma-separated expressions within parentheses and sequences with comma-separated items within square brackets. We denote type annotation 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.

```
1  function DECOMPOSE srcs: [stmt] → (ℕ → [tstmt]) =
2    let (block, blocks, _) = REC ([], ∅, return) srcs in
3    blocks ∪ (newIndex (), block)
4
5  function INITNEXT next: next+ → next =
6    match next with none → newIndex () | _ → next
7
8  function REC (block, blocks, next) srcs: acc → [stmt] → acc =
9    match srcs with
10   | [] → match next with
11     | none → (block, blocks, next)
12     | n | return → (block ++ [jump next], blocks, next)
13   | src :: srcs → match src with
14     | checkpoint | call → match srcs with
15       | [] →
16         let next = INITNEXT next in
17         (block ++ [src next], blocks, next)
18       | _ ->
19         let index = newIndex () in
20         let block = block ++ [src index] in
21         let (nextBlock, blocks, next) = REC ([], blocks, INITNEXT next) srcs in
22         (block, blocks ∪ (index, nextBlock), next)
23     | other → REC (block ++ [other], blocks, next) srcs
24     | if thn els → match srcs with
25       | [] →
26         let (thn, thnBlocks, thnNext) = REC ([], blocks, next) thn in
27         let (els, elsBlocks, elsNext) = REC ([], thnBlocks, thnNext) els in
28         let thn = if next ≠ elsNext ∧ thnNext = none
29           then thn ++ [jump elsNext] else thn in
30         (block ++ [if thn els], elsBlocks, elsNext)
31       | _ →
32         let (thn, thnBlocks, thnNext) = REC ([], blocks, none) thn in
33         let (els, elsBlocks, elsNext) = REC ([], thnBlocks, thnNext) els in
34         if elsNext = none then REC (block ++ [if thn els], elsBlocks, next) srcs
35         else
36           let thn = if thnNext = none then thn ++ [jump elsNext] else thn in
37           let (nextBlock, blocks, next) =
38             REC ([], elsBlocks, INITNEXT next) srcs in
39           (block ++ [if thn els], blocks ∪ (elsNext, nextBlock), next)
```

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` | n, where $n \in \mathbb{N}$. For `checkpoint`s, the **next** indicates which block to jump to after handling the checkpoint, 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 return address gives the next block 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 (block, blocks, next) of type **acc** ::= [**stmt**] $\times$ ($\mathbb{N} \to$ [**stmt**]) $\times$ $\textbf{next}_+$, where block is the current block being constructed, blocks are all blocks constructed so far, and next indicates the action to take at tail position in the current block. The type $\textbf{next}_+$ is defined as $\textbf{next}_+$ ::= **next** | `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 now walk through the translation of Fig. 6a to Fig. 6b. We set the accumulator 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 triggers the case at line 23 in Algorithm 2, which 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. At line 19, we create a new index for the following block. We then close the current block by tagging the `checkpoint` with the new index, resulting in block 1 in Fig. 6b. Next, we recursively create the block following the `checkpoint` at line 21. Finally, we add the recursively created block with the new index to the map of complete blocks (now also populated by the recursive call) and return the updated accumulator triple at line 22.

The complex part of Algorithm 2 involves handling of `if`s. In particular, we must handle cases where there are block splits within the branches 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 32 will

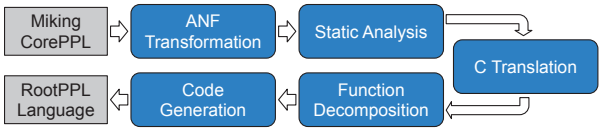


Fig. 7: The main components of the CorePPL-to-RootPPL compiler. Grey blocks are programs, and blue blocks are transformations or analyzes.

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`. There is no split in the first branch of 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 we accumulate the `if` in the same way as an `other`.

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, we must take particular care if there is only a split in the second branch of the `if` and not the first. In that case, `thnNext` is `none`, and unlike the second branch, we do not add a block jump to the end of this branch in the call at line 32. Therefore, we must instead add it at line 36. We add the `jump` at line 11 in block 2 in Fig. 6b in this way. Note that we do not require an equivalent step to the above for the second branch if the split is only in the first branch, since we pass the `next` from the first branch to the recursive call for the second branch. After handling the `if` itself, we recursively create the new block following the `if` at lines 37–38 (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 ELSNEXT at line 39.

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 we can jump 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 by construction. As discussed in Section 4.1, this is precisely the required property when compiling to PCFGs.

4.3 CorePPL-to-RootPPL Compiler

Fig. 7 gives an overview of the CorePPL-to-RootPPL compiler components. 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 simple static analysis phase discovering functions that are guaranteed not to encounter any `resample`s is also part of the compiler. It iterates through all

functions and marks a function as containing a resample if it either directly contains a `resample` or calls another function containing a `resample`. We do not need to decompose resample-free functions, and invocations can be handled directly by the C++ or CUDA compiler (and we do not need to set up an explicit stack frame). An example of such a function invocation is the `geqf s1 1.` at line 5 in Fig. 5b. We disallow passing functions as arguments to other functions as it complicates the analysis. A solution to allow passing functions as arguments is to use static analysis techniques such as 0-CFA [35] instead.

The code generation stage in Fig. 7 adds RootPPL boilerplate code and emits a complete RootPPL program that is provided as input to a C++ or CUDA compiler together with the RootPPL inference engine (see Fig. 1). The CorePPL compiler implementation is hosted at GitHub [4] and consists of approximately 3000 lines of 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 detail concerning memory allocation in the compiler is the translation between relative and absolute addresses. Fig. 5c illustrates this translation. On line 3 in block 4, we convert the `retValLoc` relative pointer 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`. This translation is needed because, at checkpoints in RootPPL, resampling copies and moves SMC executions in memory. Therefore, we cannot use absolute addresses to refer to data on the `PSTATE` stack and must instead use addresses relative to the start of the stack.

4.4 Compiler Strengths and Limitations

The main strength of the CorePPL compiler, compared to using other PPL compilers and tools, is the 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.

The compiler does, however, have some limitations. Most importantly, the lack of standard garbage collectors in C++ and CUDA leads to restrictions for automatic data allocation. Currently, we support only stack-based allocation, which means that CorePPL programs that allocate and return dynamically sized data structures (e.g., trees or linked lists) from functions are not supported. Consequently, the current compiler cannot handle probabilistic programs encoding distributions over such data structures (e.g., phylogenetic trees)—the distribution must be over fixed-size data types. However, as the evaluation in Section 5 suggests, practically significant universal probabilistic programs over fixed-sized data types are plentiful. In general, the compiler supports universal CorePPL programs including both stochastic branching and an unbound number of (stack-allocated) random variables. Automatic heap-based data allocation is a general challenge when compiling to GPUs and not specific to our approach. Exploring the use of garbage collectors or other means for automatic memory management on GPUs is an interesting direction for future research.

The compiler also lacks support for some features, which we foresee no substantial technical challenges in implementing in the near future. In particular, the compiler does not support first-class distributions—we restrict distributions to occur immediately at `assume`s (e.g., the Bernoulli distribution in `assume (Bernoulli p)` in Fig. 2a). Another possible feature is to add limited support for nested and higher-order functions.

5 Evaluation

This section evaluates RootPPL and the CorePPL-to-RootPPL compiler. The source code for all experiments is publicly available [26]. We compare RootPPL and CorePPL to state-of-the-art SMC PPL implementations on two 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). Previous work shows that SMC handles these models particularly well [36,28], and they are therefore good candidates for this evaluation. Comparison with other types of inference algorithms is a challenging problem and beyond the scope of this paper. For example, comparing SMC with variational inference (VI) is challenging as VI is approximate and SMC is asymptotically exact.

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 [32], WebPPL [20], and Pyro [10]. For each PPL, we implement the two models as efficiently as possible, given the available language features. We compile RootPPL with GCC 7.5.0 for single-core and multicore and with CUDA 11.4 for GPU. We compile Birch 1.634 with GCC 7.5.0. We use WebPPL 0.9.15 with Node.js 14.17.6. We use Pyro 1.7.0 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 Pyro performance for the Section 5.1 experiment.

To aid the comparison between languages both in the text and in the figures, we use the (S), (M), and (G) symbols suffixed to PPL names to indicate if they run on single-core, multicore, or GPU, respectively. Despite the CUDA dependency for Pyro, we did not observe any GPU usage during Pyro SMC runs. In Pyro, SMC is a minor inference algorithm, with variational inference instead being the main focus. This may explain this lack of GPU support for SMC. Consequently, 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.

5.1 Experiment: Constant-Rate Birth Death

In this experiment, we consider the non-trivial CRBD model described in Ronquist et al. [36]. 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 use the dated

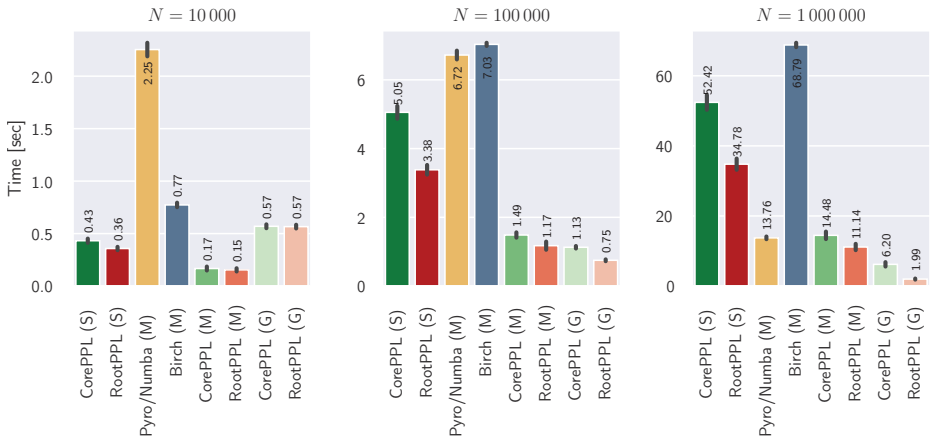


Fig. 8: Execution times for the CRBD experiment, for different numbers of particles N. The vertical line at the top of each bar indicates one standard deviation. PPLs with an (S) runs on a single core, (M) on multicore, and (G) on the GPU.

Alcedinidae phylogeny (Kingfisher birds) referenced in Ronquist et al. [36], and introduced in Jetz et al. [23]. A notable feature of this model is that it contains recursive tree constructions, which are only expressible in universal PPLs. The CorePPL implementation of this model consists of 118 lines of code†.

We measure execution time. To ensure fairness, we disabled variance-reducing techniques such as delayed sampling [28] and ESS-triggered resampling in all PPLs where available. Consequently, all implementations use precisely the same SMC inference algorithm. We checked this and the implementations' correctness by considering the output normalizing constant estimates in all runs†. 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 excessive execution times. For 10 000 particles, WebPPL (S) ran for 55 seconds (standard deviation 0.63 seconds), and Pyro (M) for 250 seconds (standard deviation 28 seconds). We omit WebPPL (S) and Pyro (M) 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).

RootPPL is the best alternative in all categories. We conjecture that the difference compared to CorePPL is due to hand-tuned details in the RootPPL model. The RootPPL model uses efficient array encodings of the observed tree, 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

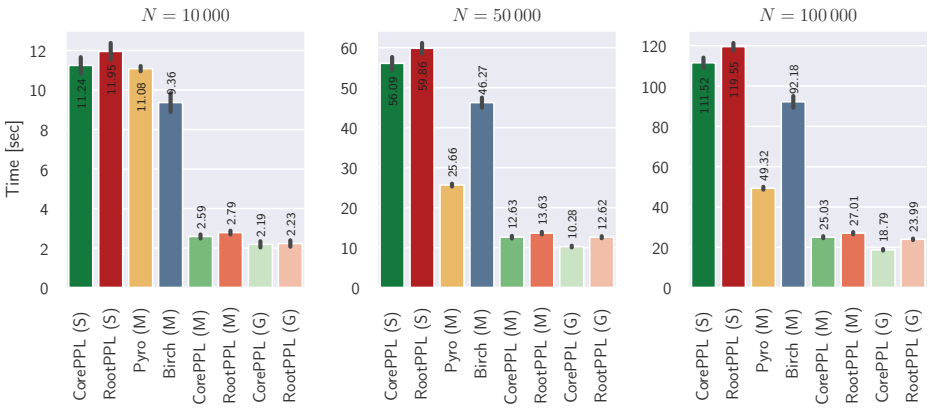


Fig. 9: Execution times for the Vector-Borne Disease experiment, for different numbers of particles N. The vertical line at the top of each bar indicates one standard deviation. PPLs with an (S) runs on a single core, (M) on multicore, and (G) on the GPU.

to subtrees in each node and traverses it via 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.

To improve the performance of Pyro, we also applied Numba to parallelize the recursive tree construction in the model manually. The parallelization we apply is more fine-grained than the natural SMC particle parallelism and resulted in an order-of-magnitude performance boost over Pyro (M). 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. Therefore, looking at adding such parallelism to RootPPL and CorePPL is an interesting direction for future work.

5.2 Experiment: Vector-Borne Disease

Next, we consider the vector-borne disease model from Funk et al. [16], which is also studied further in Murray et al. [28]. 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. [16]). The CorePPL implementation of this model consists of 140 lines of code[†].

The experiment setup is identical to Section 5.1 but with fewer SMC particles due to more demanding computations in the model. Fig. 9 shows the results. We

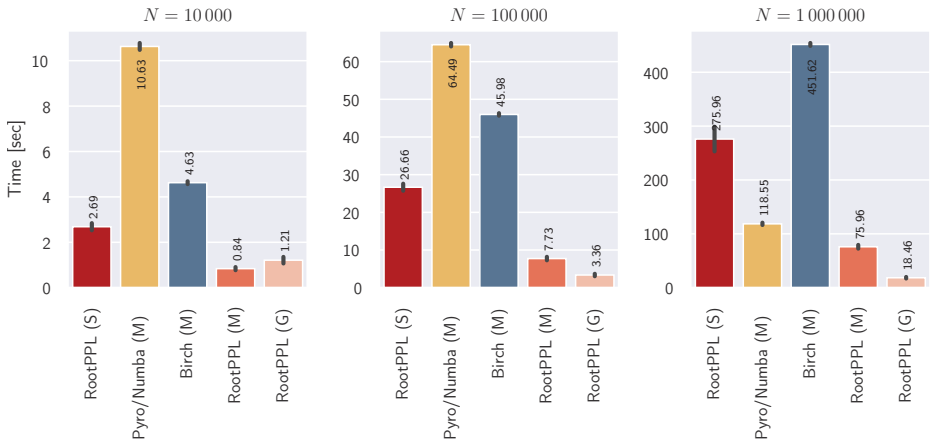


Fig. 10: Execution times for the CRBD experiment with variance-reducing techniques for different numbers of particles N. The vertical line at the top of each bar indicates one standard deviation. PPLs with an (S) runs on a single core, (M) on multicore, and (G) on the GPU. Note the 6× speedup of RootPPL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

omit WebPPL (S) entirely due to high execution times. However, we include Pyro (M) because the simple non-stochastic control-flow in this model allows much better vectorization than the CRBD model. The Numba optimization in Section 5.1 relied on the recursive structure of the model. We exclude Pyro/Numba (M) here, as such an optimization is not possible in this model.

This time, CorePPL is the best option, by a small margin, over RootPPL. We conjecture that this is due to how RootPPL preallocates memory, which is instead dynamically allocated in CorePPL. This results in copying slightly more memory during resampling for this model in RootPPL.

The difference between GPU and CPU for CorePPL and RootPPL is not as significant as in Fig. 8. We conjecture that this is due to the lower numbers of SMC particles used and RootPPL using different implementations for binomial distribution sampling on the CPU and GPU. 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 can improve GPU performance further.

5.3 Experiment: CRBD with Variance-Reducing Techniques

In this experiment, we again consider the CRBD model from Section 5.1, but with delayed sampling and ESS-triggered resampling allowed. Also, we now consider a different, more challenging phylogeny of Tyrant flycatchers [36,23].

Fig. 10 shows the results. 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. [28], 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 current lack of support for mutable data structures. 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 due to the more challenging phylogeny and delayed sampling overhead (which is greatly compensated by increased 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 [32]. Similarly to RootPPL, Birch implements SMC inference, and the target language for compilation is C++. However, while performance is one of the main goals with Birch, some overhead is inevitably introduced by supporting various quality-of-life C++ features—including automatic heap allocation [30] and object-oriented features. RootPPL does not support such features in favor of performance. Similarly to RootPPL, Birch supports CPU parallelism 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, users write models for SMC inference as a method `simulate` which the inference algorithm calls iteratively. Resampling *only* occurs between calls to this method. Furthermore, data is passed between calls to `simulate` through particle variables stored in an object defined as part of the model (similar to the PCFG state). We can view PCFG basic blocks as a natural generalization of the Birch `simulate` method, conceptually allowing for many `simulate` methods with arbitrary control-flow in between them. In particular, SMC particles can take *different* paths through the PCFG. As with PCFG blocks, the explicit `simulate` function used in Birch can potentially make it more challenging to express models for programmers. This is not a problem when using our approach of compiling into PCFGs, as we then do the block decomposition automatically.

Besides Birch, parallelism for SMC inference in PPLs is surprisingly absent in previous work. The predecessor of Birch, LibBi [29], 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, LibBi can not express many probabilistic models.

Pyro [10] is a PPL mainly focused on stochastic variational inference, supporting MCMC and SMC in addition. SMC in Pyro is similar to Birch in that models are constructed using an explicit `step` function (equivalent to `simulate` in Birch). In general, Pyro supports parallelism through vectorization using Py-

Torch [5] tensors, which is powerful but also restrictive. We saw this in Section 5.1, where we could not use Pyro tensors to parallelize the tree recursion.

Other universal PPLs implementing SMC inference include WebPPL [20] and Anglican [40]. These languages are embedded in JavaScript, and Clojure, respectively, and implement several 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) and supporting many different inference algorithms. Parallelism for SMC is not directly supported, which is different from CorePPL and RootPPL, where the focus is parallelism and performance.

Stan [12] and AugurV2 [22] support GPU parallelization of MCMC. Their modeling languages are, however, more restricted than CorePPL. Stan supports explicit parallelization of specific functions, and the AugurV2 compiler can compile to MCMC algorithms running partially in parallel on CUDA. This is quite different from the natural SMC parallelism in CorePPL and RootPPL.

There are also many other probabilistic programming tools, libraries, and languages available, for instance, Gen [13], Turing [17], Hakaru [34], and Edward [38]. 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

This paper 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 work compiling a universal PPL to GPU with SMC inference. Furthermore, the evaluation showed that CorePPL and RootPPL can deal with real-world SMC inference problems and outperform the current state-of-the-art with up to 6× speedups for challenging models (and even more when compared across 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 [36]. We leave these topics for future work.

Acknowledgments

We thank Lawrence Murray for his assistance with Birch; the anonymous reviewers at ESOP for their valuable comments; Gizem Çaylak for her valuable comments and contributions to CorePPL and Miking; Lars Hummelgren, Viktor Palmkvist, and Oscar Eriksson for their valuable comments and contributions to Miking; and finally all other Miking developers for their contributions to Miking.

References

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. PyTorch. https://pytorch.org/ (2021), accessed: 2021-10-11
6. The LLVM Compiler Infrastructure Project. https://llvm.org/ (2021), accessed: 2021-09-20
7. Thrust - Parallel Algorithms Library. https://thrust.github.io/ (2021), accessed: 2021-09-24
8. Aho, A.V., Lam, M.S., Sethi, R., Ullman, J.D.: Compilers: principles, techniques and tools. Addison-Wesley (2006)
9. Appel, A.W.: Compiling with Continuations. Cambridge University Press (1991)
10. 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)
11. 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, ACM, New York, NY, USA (2019)
12. 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)
13. 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)
14. Doucet, A., de Freitas, N., Gordon, N.: Sequential Monte Carlo Methods in Practice. Information Science and Statistics, Springer New York (2001)
15. 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 1993, ACM, New York, NY, USA (1993)
16. 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)
17. 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)
18. Gilks, W., Richardson, S., Spiegelhalter, D.: Markov Chain Monte Carlo in Practice. Chapman & Hall/CRC Interdisciplinary Statistics, Taylor & Francis (1995)

19. 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)
20. Goodman, N.D., Stuhlmüller, A.: The design and implementation of probabilistic programming languages. http://dippl.org (2014), accessed: 2020-07-09
21. 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, ACM, New York, NY, USA (2014)
22. Huang, D., Tristan, J.B., Morrisett, G.: Compiling markov chain monte carlo algorithms for probabilistic modeling. In: Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation. p. 111–125. PLDI 2017, ACM, New York, NY, USA (2017)
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: Programming Languages and Systems. pp. 404–431. Springer International Publishing, Cham (2021)
25. Lundén, D., Broman, D., Ronquist, F., Murray, L.M.: Automatic alignment of sequential Monte Carlo inference in higher-order probabilistic programs. arXiv e-prints p. arXiv:1812.07439 (2018)
26. Lundén, D., Öhman, J., Kudlicka, J., Senderov, V., Ronquist, F., Broman, D.: Artifact: Compiling Universal Probabilistic Programming Languages with Efficient Parallel Sequential Monte Carlo Inference (Jan 2022). https://doi.org/10.5281/zenodo.5914164
27. Lundén, D., Öhman, J., Kudlicka, J., Senderov, V., Ronquist, F., Broman, D.: Compiling universal probabilistic programming languages with efficient parallel sequential monte carlo inference. arXiv e-prints p. arXiv:2112.00364 (2022)
28. 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)
29. Murray, L.M.: Bayesian state-space modelling on high-performance hardware using LibBi. arXiv e-prints p. arXiv:1306.3277 (2013)
30. Murray, L.M.: Lazy object copy as a platform for population-based probabilistic programming. arXiv e-prints p. arXiv:2001.05293 (2020)
31. 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)
32. Murray, L.M., Schön, T.B.: Automated learning with a probabilistic programming language: Birch. Annual Reviews in Control **46**, 29–43 (2018)
33. Naesseth, C., Lindsten, F., Schön, T.: Elements of Sequential Monte Carlo. Foundations and Trends in Machine Learning Series, Now Publishers (2019)
34. 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)
35. Nielson, F., Nielson, H.R., Hankin, C.: Principles of Program Analysis. Springer-Verlag (1999)

36. 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)
37. 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, ACM, New York, NY, USA (2016)
38. 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)
39. 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)
40. 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)