Automatic Alignment in Higher-Order Probabilistic Programming Languages[⋆]

Daniel Lundén[1] (✉), Gizem Çaylak[1], Fredrik Ronquist[2,3], and David Broman[1]

[1] EECS and Digital Futures, KTH Royal Institute of Technology, Stockholm, Sweden, {dlunde,caylak,dbro}@kth.se
[2] Department of Bioinformatics and Genetics, Swedish Museum of Natural History, Stockholm, Sweden, fredrik.ronquist@nrm.se
[3] Department of Zoology, Stockholm University, Stockholm, Sweden

Abstract. Probabilistic Programming Languages (PPLs) allow users to encode statistical inference problems and automatically apply an *inference algorithm* to solve them. Popular inference algorithms for PPLs, such as sequential Monte Carlo (SMC) and Markov chain Monte Carlo (MCMC), are built around *checkpoints*—relevant events for the inference algorithm during the execution of a probabilistic program. Deciding the location of checkpoints is, in current PPLs, not done optimally. To solve this problem, we present a static analysis technique that automatically determines checkpoints in programs, relieving PPL users of this task. The analysis identifies a set of checkpoints that execute in the same order in every program run—they are *aligned*. We formalize alignment, prove the correctness of the analysis, and implement the analysis as part of the higher-order functional PPL Miking CorePPL. By utilizing the alignment analysis, we design two novel inference algorithm variants: *aligned SMC* and *aligned lightweight MCMC*. We show, through real-world experiments, that they significantly improve inference execution time and accuracy compared to standard PPL versions of SMC and MCMC.

Keywords: Probabilistic programming · Operational semantics · Static analysis.

1 Introduction

Probabilistic programming languages (PPLs) are languages used to encode statistical inference problems, common in research fields such as phylogenetics [39],

⋆ This project is financially supported by the Swedish Foundation for Strategic Research (FFL15-0032 and RIT15-0012), and also partially supported by the Wallenberg Al, Autonomous Systems and Software Program (WASP) funded by the Knut and Alice Wallenberg Foundation, and the Swedish Research Council (grants 2018-04620 and 2021-04830). The research has also been carried out as part of the Vinnova Competence Center for Trustworthy Edge Computing Systems and Applications at KTH Royal Institute of Technology.

The original version of this chapter was revised: Theorem 1 has been corrected. The correction to this chapter is available at https://doi.org/10.1007/978-3-031-30044-8_21

T. Wies (Ed.): ESOP 2023, LNCS 13990, pp. 535–563, 2023.
https://doi.org/10.1007/978-3-031-30044-8_20

computer vision [16], topic modeling [5], data cleaning [23], and cognitive science [15]. PPL implementations automatically solve encoded problems by applying an *inference algorithm*. In particular, automatic inference allows users to solve inference problems without having in-depth knowledge of inference algorithms and how to apply them. Some examples of PPLs are WebPPL [14], Birch [31], Anglican [48], Miking CorePPL [25], Turing [12], and Pyro [3].

Sequential Monte Carlo (SMC) and Markov chain Monte Carlo (MCMC) are general-purpose families of inference algorithms often used for PPL implementations. These algorithms share the concept of *checkpoints*: relevant execution events for the inference algorithm. For SMC, the checkpoints are *likelihood updates* [48,14] and determine the *resampling* of executions. Alternatively, users must sometimes manually annotate or write the probabilistic program in a certain way to make resampling explicit [25,31]. For MCMC, checkpoints are instead *random draws*, which allow the inference algorithm to manipulate these draws to construct a Markov chain over program executions [47,38]. When designing SMC and MCMC algorithms for *universal PPLs*[4], both the *placement* and *handling* of checkpoints are critical to making the inference both efficient and accurate.

For SMC, a standard inference approach is to resample at *all* likelihood updates [14,48]. This approach produces correct results asymptotically [24] but is highly problematic for certain models [39]. Such models require non-trivial and SMC-specific manual program rewrites to force good resampling locations and make SMC tractable. Overall, choosing the likelihood updates at which to resample significantly affects SMC execution time and accuracy.

For MCMC, a standard approach for inference in universal PPLs is *lightweight MCMC* [47], which constructs a Markov chain over random draws in programs. The key idea is to use an *addressing transformation* and a *runtime database* of random draws. Specifically, the database enables matching and reusing random draws between executions according to their *stack traces*, even if the random draws may or may not occur due to randomness during execution. However, the dynamic approach of looking up random draws in the database through their stack traces is expensive and introduces significant runtime overhead.

To overcome the SMC and MCMC problems in universal PPLs, we present a static analysis technique for higher-order functional PPLs that *automatically* determines checkpoints in a probabilistic program that always occur in the same order in every program execution—they are *aligned*. We formally define alignment, formalize the alignment analysis, and prove the soundness of the analysis with respect to the alignment definition. The novelty and challenge in developing the static analysis technique is to capture alignment properties through the identification of expressions in programs that may evaluate to *stochastic values* and expressions that may evaluate due to *stochastic branching*. Stochastic branching results from `if` expressions with stochastic values as conditions or function applications where the function itself is stochastic. Stochastic values and branches pose a significant challenge when proving the soundness of the analysis.

[4] A term coined by Goodman et al. [13]. Essentially, it means that the types and numbers of random variables cannot be determined statically.

We design two new inference algorithms that improve accuracy and execution time compared to current approaches. Unlike the standard SMC algorithm for PPLs [48,14], *aligned SMC* only resamples at aligned likelihood updates. Resampling only at aligned likelihood updates guarantees that each SMC execution resamples the same number of times, which makes expensive global termination checks redundant [25]. We evaluate aligned SMC on two diversification models from Ronquist et al. [39] and a state-space model for aircraft localization, demonstrating significantly improved inference accuracy and execution time compared to traditional SMC. Both models—constant rate birth-death (CRBD) and cladogenetic diversification rate shift (ClaDS)—are used in real-world settings and are of considerable interest to evolutionary biologists [33,28]. The documentations of both Anglican [48] and Turing [12] acknowledge the importance of alignment for SMC and state that all likelihood updates must be aligned. However, Turing and Anglican neither formalize nor enforce this property—it is up to the users to *manually* guarantee it, often requiring non-standard program rewrites [39].

We also design *aligned lightweight MCMC*, a new version of lightweight MCMC [47]. Aligned lightweight MCMC constructs a Markov chain over the program using the aligned random draws as *synchronization points* to match and reuse aligned random draws and a subset of unaligned draws between executions. Aligned lightweight MCMC does not require a runtime database of random draws and therefore reduces runtime overhead. We evaluate aligned lightweight MCMC for latent Dirichlet allocation (LDA) [5] and CRBD [39], demonstrating significantly reduced execution times and no decrease in inference accuracy. Furthermore, automatic alignment is orthogonal to and easily combines with the lightweight MCMC optimizations introduced by Ritchie et al. [38].

We implement the analysis, aligned SMC, and aligned lightweight MCMC in Miking CorePPL [25,7]. In addition to analyzing stochastic `if`-branching, the implementation analyzes stochastic branching at a standard pattern-matching construct. Compared to `if` expressions, the pattern-matching construct requires a more sophisticated analysis of the pattern and the value matched against it to determine if the pattern-matching causes a stochastic branch.

In summary, we make the following contributions.

– We invent and formalize alignment for PPLs. Aligned parts of a program occur in the same order in every execution (Section 4.1).
– We formalize and prove the soundness of a novel static analysis technique that determines stochastic value flow and stochastic branching, and in turn alignment, in higher-order probabilistic programs (Section 4.2).
– We design aligned SMC inference that only resamples at aligned likelihood updates, improving execution time and inference accuracy (Section 5.1).
– We design aligned lightweight MCMC inference that only reuses aligned random draws, improving execution time (Section 5.2).
– We implement the analysis and inference algorithms in Miking CorePPL. The implementation extends the alignment analysis to identify stochastic branching resulting from pattern matching (Section 6).

Section 7 describes the evaluation and discusses its results. The paper also has an accompanying artifact that supports the evaluation [26]. Section 8 discusses related work and Section 9 concludes. Next, Section 2 considers a simple motivating example to illustrate the key ideas. Section 3 introduces syntax and semantics for the calculus used to formalize the alignment analysis.

An extended version of the paper is also available at arXiv [27]. We use the symbol † in the text to indicate that more information (e.g., proofs) is available in the extended version.

2 A Motivating Example

This section presents a motivating example that illustrates the key alignment ideas in relation to aligned SMC (Section 2.1) and aligned lightweight MCMC (Section 2.2). We assume basic knowledge of probability theory. Knowledge of PPLs is helpful, but not a strict requirement. The book by van de Meent et al. [46] provides a good introduction to PPLs.

Probabilistic programs encode Bayesian statistical inference problems with two fundamental constructs: `assume` and `weight`. The `assume` construct defines random variables, which make execution nondeterministic. Intuitively, a probabilistic program then encodes a probability distribution over program executions (the prior distribution), and it is possible to sample from this distribution by executing the program with random sampling at `assume`s. The `weight` construct updates the *likelihood* of individual executions. Updating likelihoods for executions modifies the probability distribution induced by `assume`s, and the inference problem encoded by the program is to determine or approximate this modified distribution (the posterior distribution). The main purpose of `weight` in real-world models is to condition executions on observed data.[5]

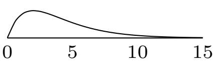
Consider the probabilistic program in Fig. 1a. The program is contrived and purposefully constructed to compactly illustrate alignment, but the real-world diversification models in Ronquist et al. [39] that we also consider in Section 7 inspired the program's general structure. The program defines (line 1) and returns (line 18) a Gamma-distributed random variable *rate*. Figure 1b illustrates the Gamma distribution. To modify the likelihood for values of *rate*, the program executes the *iter* function (line 10) three times, and the *survives* function (line 2) a random number of times n (line 13) within each *iter* call.

Conceptually, to infer the posterior distribution of the program, we execute the program infinitely many times. In each execution, we draw samples for the random variables defined at `assume`, and accumulate the likelihood at `weight`. The return value of the execution, weighted by the accumulated likelihood, represents one sample from the posterior distribution. Fig. 1c shows a histogram of such weighted samples of *rate* resulting from a large number of executions of Fig. 1a. The fundamental inference algorithm that produces such weighted samples is called *likelihood weighting* (a type of *importance sampling* [32]). We

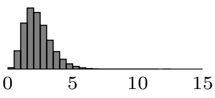
[5] A number of more specialized constructs for likelihood updating are also available in various PPLs, for example *observe* [48,14] and *condition* [14].

```
 1 let rate = assume Gamma(2, 2) in
 2 let rec survives = λn.
 3   if n = 0 then () else
 4     if assume Bernoulli(0.9) then
 5       weight 0.5;
 6       survives (n − 1)
 7     else
 8       weight 0
 9 in
10 let rec iter = λi.
11   if i = 0 then () else
12     weight rate;
13     let n = assume Poisson(rate) in
14     survives n;
15     iter (i − 1)
16 in
17 iter 3;
18 rate
```

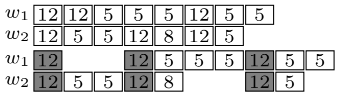
(a) Probabilistic program.



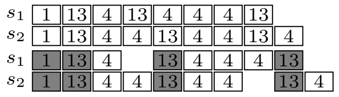
(b) Gamma(2, 2).



(c) Histogram.



(d) Aligning `weight`.



(e) Aligning `assume`.

Fig. 1: A simple example illustrating alignment. Fig. (a) gives a probabilistic program using functional-style PPL pseudocode. Fig. (b) illustrates the Gamma(2, 2) probability density function. Fig. (c) illustrates a histogram over weighted *rate* samples produced by running the program in (a) a large number of times. Fig. (d) shows two line number sequences w_1 and w_2 of `weight`s encountered in two program runs (top) and how to align them (bottom). Fig. (e) shows two line number sequences s_1 and s_2 of `assume`s encountered in two program runs (top) and how to align them (bottom).

see that, compared to the prior distribution for *rate* in Fig. 1b, the posterior is more sharply peaked due to the likelihood modifications.

2.1 Aligned SMC

Likelihood weighting can only handle the simplest of programs. In Fig. 1a, a problem with likelihood weighting is that we assign the weight 0 to many executions at line 8. These executions contribute nothing to the final distribution. SMC solves this by executing many program instances *concurrently* and occasionally *resampling* them (with replacement) based on their current likelihoods. Resampling discards executions with lower weights (in the worst case, 0) and replaces them with executions with higher weights. The most common approach in popular PPLs is to resample just after likelihood updates (i.e., calls to `weight`).

Resampling at all calls to `weight` in Fig. 1a is suboptimal. The best option is instead to *only* resample at line 12. This is because executions encounter lines 5 and 8 a *random* number of times due to the stochastic branch at line 3, while they encounter line 12 a fixed number of times. As a result of resampling at lines 5 and 8, executions become *unaligned*; in each resampling, executions can have reached either line 5, line 8, or line 12. On the other hand, if we resample only at line 12, all executions will always have reached line 12 for the same iteration of *iter* in every resampling. Intuitively, this is a sensible approach since, when resampling, executions have progressed the same distance through the program. We say that the `weight` at line 12 is *aligned*, and resampling only at aligned `weight`s results in our new inference approach called *aligned SMC*. Fig. 1d visualizes the `weight` alignment for two sample executions of Fig. 1a.

2.2 Aligned Lightweight MCMC

Another improvement over likelihood weighting is to construct a Markov chain over program executions. It is beneficial to propose new executions in the Markov chain by making small, rather than large, modifications to the previous execution. The lightweight MCMC [47] algorithm does this by redrawing a single random draw in the previous execution, and then reusing as many other random draws as possible. Random draws in the current and previous executions match through *stack traces*—the sequences of applications leading up to a random draw. Consider the random draw at line 13 in Fig. 1a. It is called exactly three times in every execution. If we identify applications and `assume`s by line numbers, we get the stack traces $[17, 13]$, $[17, 15, 13]$, and $[17, 15, 15, 13]$ for these three `assume`s in every execution. Consequently, lightweight MCMC can reuse these draws by storing them in a database indexed by stack traces.

The stack trace indexing in lightweight MCMC is overly complicated when reusing aligned random draws. Note that the `assume`s at lines 1 and 13 in Fig 1a are aligned, while the `assume` at line 4 is unaligned. Fig. 1e visualizes the `assume` alignment for two sample executions of Fig. 1a. Aligned random draws occur in the same same order in every execution, and are therefore trivial to match and reuse between executions through indexing by counting. The appeal with stack trace indexing is to additionally allow reusing a subset of *unaligned* draws.

A key insight in this paper is that aligned random draws can also act as *synchronization points* in the program to allow reusing unaligned draws *without* a stack trace database. After an aligned draw, we reuse unaligned draws occurring up until the next aligned draw, as long as they syntactically originate at the same `assume` as the corresponding unaligned draws in the previous execution. As soon as an unaligned draw does not originate from the same `assume` as in the previous execution, we redraw all remaining unaligned draws up until the next aligned draw. Instead of a trace-indexed database, this approach requires storing a list of unaligned draws (tagged with identifiers of the `assume`s at which they originated) for each execution segment in between aligned random draws. For example, for the execution s_1 in Fig. 1e, we store lists of unaligned Bernoulli random draws from line 4 for each execution segment in between the three aligned random draws at line 13. If a Poisson random draw n at line 13 does not change or decreases, we can reuse the stored unaligned Bernoulli draws up until the next Poisson random draw as *survives* executes n or fewer times. If the drawn n instead increases to n', we can again reuse all stored Bernoulli draws, but must supplement them with new Bernoulli draws to reach n' draws in total.

As we show in Section 7, using aligned draws as synchronization points works very well in practice and avoids the runtime overhead of the lightweight MCMC database. However, manually identifying aligned parts of programs and rewriting them so that inference can make use of alignment is, if even possible, tedious, error-prone, and impractical for large programs. This paper presents an automated approach to identifying aligned parts of programs. Combining static alignment analysis and using aligned random draws as synchronization points form the key ideas of the new algorithm that we call *aligned lightweight MCMC*.

3 Syntax and Semantics

In preparation for the alignment analysis in Section 4, we require an idealized base calculus capturing the key features of expressive PPLs. This section introduces such a calculus with a formal syntax (Section 3.1) and semantics (Section 3.2). We assume a basic understanding of the lambda calculus (see, e.g., Pierce [37] for a complete introduction). Section 6 further describes extending the idealized calculus and the analysis in Section 4 to a full-featured PPL.

3.1 Syntax

We use the untyped lambda calculus as the base for our calculus. We also add `let` expressions for convenience, and `if` expressions to allow intrinsic booleans to affect control flow. The calculus is a subset of the language used in Fig. 1a. We inductively define terms $\mathbf{t}$ and values $\mathbf{v}$ as follows.

Definition 1 (Terms and values).

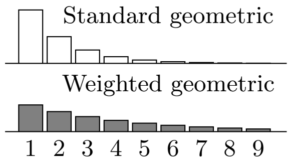
$$\begin{aligned}\mathbf{t} ::= {} & x \mid c \mid \lambda x.\ \mathbf{t} \mid \mathbf{t}\ \mathbf{t} \mid \texttt{let}\ x = \mathbf{t}\ \texttt{in}\ \mathbf{t} \qquad\qquad \mathbf{v} ::= c \mid \langle \lambda x.\ \mathbf{t}, \rho\rangle \\ & \mid \texttt{if}\ \mathbf{t}\ \texttt{then}\ \mathbf{t}\ \texttt{else}\ \mathbf{t} \mid \texttt{assume}\ \mathbf{t} \mid \texttt{weight}\ \mathbf{t} \\ & x, y \in X \quad \rho \in P \quad c \in C \quad \{\text{false}, \text{true}, ()\} \cup \mathbb{R} \cup D \subseteq C.\end{aligned} \tag{1}$$

X is a countable set of variable names, C a set of intrinsic values and operations, and $D \subset C$ a set of probability distributions. The set P contains all evaluation environments ρ, that is, partial functions mapping names in X to values $\mathbf{v}$. We use T and V to denote the set of all terms and values, respectively.

Values $\mathbf{v}$ are intrinsics or closures, where closures are abstractions with an environment binding free variables in the abstraction body. We require that C include booleans, the unit value $()$, and real numbers. The reason is that `weight` takes real numbers as argument and returns $()$ and that `if` expression conditions are booleans. Furthermore, probability distributions are often over booleans and real numbers. For example, we can include the normal distribution constructor $\mathcal{N} \in C$ that takes real numbers as arguments and produces normal distributions over real numbers. For example, $\mathcal{N}\ 0\ 1 \in D$, the standard normal distribution. We often write functions in C in infix position or with standard function application syntax for readability. For example, $1 + 2$ with $+ \in C$ means $+\ 1\ 2$, and $\mathcal{N}(0, 1)$ means $\mathcal{N}\ 0\ 1$. Additionally, we use the shorthand $\mathbf{t}_1; \mathbf{t}_2$ for `let _ =` $\mathbf{t}_1$ `in` $\mathbf{t}_2$, where _ is the do-not-care symbol. That is, $\mathbf{t}_1; \mathbf{t}_2$ evaluates $\mathbf{t}_1$ for side

```
1 let rec geometric = λ_.
2   let x = assume Bernoulli(0.5) in
3   if x then
4     weight 1.5;
5     1 + geometric ()
6   else 1
7 in geometric ()
```

(a) Probabilistic program $\mathbf{t}_{geo}$.



(b) Probability distributions.

Fig. 2: A probabilistic program $\mathbf{t}_{geo}$ [25], illustrating (1). Fig. (a) gives the program, and (b) the corresponding probability distributions. In (b), the y-axis gives the probability, and the x-axis gives the outcome (the number of coin flips). The upper part of (b) excludes the shaded `weight` at line 4 in (a).

effects only before evaluating $\mathbf{t}_2$. Finally, the untyped lambda calculus supports recursion through fixed-point combinators. We encapsulate this in the shorthand `let rec` f `=` $\lambda x.\mathbf{t}_1$ `in` $\mathbf{t}_2$ to conveniently define recursive functions.

The `assume` and `weight` constructs are PPL-specific. We define random variables from intrinsic probability distributions with `assume` (also known as *sample* in PPLs with sampling-based inference). For example, the term `let` x `=` `assume` $\mathcal{N}(0,1)$ `in` $\mathbf{t}$ defines x as a random variable with a standard normal distribution in $\mathbf{t}$. Boolean random variables combined with `if` expressions result in *stochastic branching*—causing the alignment problem. Lastly, `weight` (also known as *factor* or *score*) is a standard construct for likelihood updating (see, e.g., Borgström et al. [6]). Next, we illustrate and formalize a semantics for (1).

3.2 Semantics

Consider the small probabilistic program $\mathbf{t}_{geo} \in T$ in Fig. 2a. The program encodes the standard geometric distribution via a function *geometric*, which recursively flips a fair coin (a Bernoulli(0.5) distribution) at line 2 until the outcome is false (i.e., tails). At that point, the program returns the total number of coin flips, including the last tails flip. The upper part of Fig. 2b illustrates the result distribution for an infinite number of program runs with line 4 ignored.

To illustrate the effect of `weight`, consider $\mathbf{t}_{geo}$ with line 4 included. This `weight` modifies the likelihood with a factor 1.5 each time the flip outcome is true (or, heads). Intuitively, this emphasizes larger return values, illustrated in the lower part of Fig. 2b. Specifically, the (unnormalized) probability of seeing n coin flips is $0.5^n \cdot 1.5^{n-1}$, compared to 0.5^n for the unweighted version. The factor 1.5^{n-1} is the result of the calls to `weight`.

We now introduce a big-step operational semantics for single runs of programs $\mathbf{t}$. Such a semantics is essential to formalize the probability distributions encoded by probabilistic programs (e.g., Fig. 2b for Fig. 2a) and to prove the correctness of PPL inference algorithms. For example, Borgström et al. [6] define a PPL calculus and semantics similar to this paper and formally proves the correctness of an MCMC algorithm. Another example is Lundén et al. [24], who also define a

$$\frac{}{\rho \vdash x \ {}^{[]}{\Downarrow}^{1}_{[]}\ \rho(x)}(\textsc{Var}) \qquad \frac{}{\rho \vdash c \ {}^{[]}{\Downarrow}^{1}_{[]}\ c}(\textsc{Const}) \qquad \frac{}{\rho \vdash \lambda x.\mathbf{t} \ {}^{[]}{\Downarrow}^{1}_{[]}\ \langle \lambda x.\mathbf{t}, \rho \rangle}(\textsc{Lam})$$

$$\frac{\rho \vdash \mathbf{t}_1 \ {}^{s_1}{\Downarrow}^{w_1}_{l_1}\ \langle \lambda x.\mathbf{t}, \rho' \rangle \quad \rho \vdash \mathbf{t}_2 \ {}^{s_2}{\Downarrow}^{w_2}_{l_2}\ \mathbf{v}_2 \quad \rho', x \mapsto \mathbf{v}_2 \vdash \mathbf{t} \ {}^{s_3}{\Downarrow}^{w_3}_{l_3}\ \mathbf{v}}{\rho \vdash \mathbf{t}_1\ \mathbf{t}_2 \ {}^{s_1 \| s_2 \| s_3}{\Downarrow}^{w_1 \cdot w_2 \cdot w_3}_{l_1 \| l_2 \| l_3}\ \mathbf{v}}(\textsc{App})$$

$$\frac{\rho \vdash \mathbf{t}_1 \ {}^{s_1}{\Downarrow}^{w_1}_{l_1}\ c_1 \quad \rho \vdash \mathbf{t}_2 \ {}^{s_2}{\Downarrow}^{w_2}_{l_2}\ c_2}{\rho \vdash \mathbf{t}_1\ \mathbf{t}_2 \ {}^{s_1 \| s_2}{\Downarrow}^{w_1 \cdot w_2}_{l_1 \| l_2}\ \delta(c_1, c_2)}(\textsc{Const-App}) \quad \frac{\rho \vdash \mathbf{t} \ {}^{s}{\Downarrow}^{w}_{l}\ d \quad w' = f_d(c)}{\rho \vdash \texttt{assume}\ \mathbf{t} \ {}^{s \| [c]}{\Downarrow}^{w \cdot w'}_{l}\ c}(\textsc{Assume})$$

$$\frac{\rho \vdash \mathbf{t}_1 \ {}^{s_1}{\Downarrow}^{w_1}_{l_1}\ \mathbf{v}_1 \quad \rho, x \mapsto \mathbf{v}_1 \vdash \mathbf{t}_2 \ {}^{s_2}{\Downarrow}^{w_2}_{l_2}\ \mathbf{v}}{\rho \vdash \texttt{let}\ x = \mathbf{t}_1\ \texttt{in}\ \mathbf{t}_2 \ {}^{s_1 \| s_2}{\Downarrow}^{w_1 \cdot w_2}_{l_1 \| [x] \| l_2}\ \mathbf{v}}(\textsc{Let}) \quad \frac{\rho \vdash \mathbf{t} \ {}^{s}{\Downarrow}^{w}_{l}\ w'}{\rho \vdash \texttt{weight}\ \mathbf{t} \ {}^{s}{\Downarrow}^{w \cdot w'}_{l}\ ()}(\textsc{Weight})$$

$$\frac{\rho \vdash \mathbf{t}_1 \ {}^{s_1}{\Downarrow}^{w_1}_{l_1}\ \text{true} \quad \rho \vdash \mathbf{t}_2 \ {}^{s_2}{\Downarrow}^{w_2}_{l_2}\ \mathbf{v}_2}{\rho \vdash \texttt{if}\ \mathbf{t}_1\ \texttt{then}\ \mathbf{t}_2\ \texttt{else}\ \mathbf{t}_3 \ {}^{s_1 \| s_2}{\Downarrow}^{w_1 \cdot w_2}_{l_1 \| l_2}\ \mathbf{v}_2}(\textsc{If-True})$$

$$\frac{\rho \vdash \mathbf{t}_1 \ {}^{s_1}{\Downarrow}^{w_1}_{l_1}\ \text{false} \quad \rho \vdash \mathbf{t}_3 \ {}^{s_3}{\Downarrow}^{w_3}_{l_3}\ \mathbf{v}_3}{\rho \vdash \texttt{if}\ \mathbf{t}_1\ \texttt{then}\ \mathbf{t}_2\ \texttt{else}\ \mathbf{t}_3 \ {}^{s_1 \| s_3}{\Downarrow}^{w_1 \cdot w_3}_{l_1 \| l_3}\ \mathbf{v}_3}(\textsc{If-False})$$

Fig. 3: A big-step operational semantics for terms, formalizing single runs of programs $\mathbf{t} \in T$. The operation $\rho, x \mapsto \mathbf{v}$ produces a new environment extending ρ with a binding $\mathbf{v}$ for x. For each distribution $d \in D$, f_d is its *probability density* or *probability mass* function—encoding the relative probability of drawing particular values from the distribution. For example, $f_{\text{Bernoulli}(0.3)}(\text{true}) = 0.3$ and $f_{\text{Bernoulli}(0.3)}(\text{false}) = 1 - 0.3 = 0.7$. We use $\cdot$ to denote multiplication.

similar calculus and semantics and prove the correctness of PPL SMC algorithms. In particular, the correctness of our aligned SMC algorithm (Section 5.1) follows from this proof. The purpose of the semantics in this paper is to formalize alignment and prove the soundness of our analysis in Section 4. We use a big-step semantics as the finer granularity in a small-step semantics is redundant. We begin with a definition for intrinsics.

Definition 2 (Intrinsic functions). *For every $c \in C$, we attach an* arity *$|c| \in \mathbb{N}$. We define a partial function $\delta : C \times C \to C$ such that $\delta(c, c_1) = c_2$ is defined for $|c| > 0$. For all c, c_1, and c_2, such that $\delta(c, c_1) = c_2$, $|c_2| = |c| - 1$.*

Intrinsic functions are curried and produce intrinsic or intrinsic functions of one arity less through δ. For example, for $+ \in C$, we have $\delta(\delta(+, 1), 2) = 3$, $|+| = 2$, $|\delta(+, 1)| = 1$, and $|\delta(\delta(+, 1), 2)| = 0$. Next, randomness in our semantics is deterministic via a *trace* of random draws in the style of Kozen [22].

Definition 3 (Traces). *The set S of traces is the set such that, for all $s \in S$, s is a sequence of intrinsics from C with arity 0.*

In the following, we use the notation $[c_1, c_2, \ldots, c_n]$ for sequences and $\|$ for sequence concatenation. For example, $[c_1, c_2] \parallel [c_2, c_4] = [c_1, c_2, c_3, c_4]$. We also use subscripts to select elements in a sequence, e.g., $[c_1, c_2, c_3, c_4]_2 = c_2$. In practice, traces are often sequences of real numbers, e.g., $[1.1, 3.2, 8.4] \in S$.

Fig. 3 presents the semantics as a relation $\rho \vdash \mathbf{t} \; {}^{s}\!\Downarrow_{l}^{w} \; \mathbf{v}$ over $P \times T \times S \times \mathbb{R} \times L \times V$. L is the set of sequences over X, i.e., sequences of names. For example, $[x, y, z] \in L$, where $x, y, z \in X$. We use $l \in L$ to track the sequence of `let`-bindings during evaluation. For example, evaluating `let` x `= 1 in let` y `= 2 in` $x + y$ results in $l = [x, y]$. In Section 4, we use the sequence of encountered `let`-bindings to define alignment. For simplicity, from now on we assume that bound variables are always unique (i.e., variable shadowing is impossible).

It is helpful to think of ρ, $\mathbf{t}$, and s as the input to $\Downarrow$, and l, w and $\mathbf{v}$ as the output. In the environment ρ, $\mathbf{t}$, with trace s, evaluates to $\mathbf{v}$, encounters the sequence of `let` bindings l, and accumulates the weight w. The trace s is the sequence of all random draws, and each random draw in (Assume) consumes precisely one element of s. The rule (Let) tracks the sequence of bindings by adding x at the correct position in l. The number w is the likelihood of the execution—the probability density of all draws in the program, registered at (Assume), combined with direct likelihood modifications, registered at (Weight). The remaining aspects of the semantics are standard (see, e.g., Kahn [20]). To give an example of the semantics, we have $\varnothing \vdash \mathbf{t}_{geo} \; {}^{[\text{true},\text{true},\text{true},\text{false}]}\!\Downarrow_{[geometric,x,x,x,x]}^{0.5\cdot 1.5\cdot 0.5\cdot 1.5\cdot 0.5\cdot 1.5\cdot 0.5} \; 4$ for the particular execution of $\mathbf{t}_{geo}$ making three recursive calls. Next, we formalize and apply the alignment analysis to (1).

4 Alignment Analysis

This section presents the main contribution of this paper: automatic alignment in PPLs. Section 4.1 introduces A-normal form and gives a precise definition of alignment. Section 4.2 formalizes and proves the correctness of the alignment analysis. Lastly, Section 4.3 discusses a dynamic version of alignment.

4.1 A-Normal Form and Alignment

To reason about all subterms $\mathbf{t}'$ of a program $\mathbf{t}$ and to enable the analysis in Section 4.2, we need to uniquely label all subterms. A straightforward approach is to use variable names within the program itself as labels (remember that we assume bound variables are always unique). This leads us to the standard A-normal form (ANF) representation of programs [11].

Definition 4 (A-normal form).

$$\begin{aligned} \mathbf{t}_{\text{ANF}} &::= x \mid \texttt{let}\ x = \mathbf{t}'_{\text{ANF}}\ \texttt{in}\ \mathbf{t}_{\text{ANF}} \\ \mathbf{t}'_{\text{ANF}} &::= x \mid c \mid \lambda x.\ \mathbf{t}_{\text{ANF}} \mid x\ y \\ &\qquad \mid \texttt{if}\ x\ \texttt{then}\ \mathbf{t}_{\text{ANF}}\ \texttt{else}\ \mathbf{t}_{\text{ANF}} \mid \texttt{assume}\ x \mid \texttt{weight}\ x \end{aligned} \tag{2}$$

We use T_{ANF} to denote the set of all terms $\mathbf{t}_{\text{ANF}}$. Unlike $\mathbf{t} \in T$, $\mathbf{t}_{\text{ANF}} \in T_{\text{ANF}}$ enforces that a variable bound by a `let` labels each subterm in the program. Furthermore, we can automatically transform any program in T to a semantically equivalent T_{ANF} program, and $T_{\text{ANF}} \subset T$. Therefore, we assume in the remainder of the paper that all terms are in ANF.

Given the importance of alignment in universal PPLs, it is somewhat surprising that there are no previous attempts to give a formal definition of its meaning. Here, we give a first such formal definition, but before defining alignment, we require a way to restrict, or filter, sequences.

Definition 5 (Restriction of sequences). *For all $l \in L$ and $Y \subseteq X$, $l|_Y$ (the restriction of l to Y) is the subsequence of l with all elements not in Y removed.*

For example, $[x, y, z, y, x]|_{\{x,z\}} = [x, z, x]$. We now formally define alignment.

Definition 6 (Alignment). *For $\mathbf{t} \in T_{\text{ANF}}$, let $X_{\mathbf{t}}$ denote all variables that occur in $\mathbf{t}$. The sets $A_{\mathbf{t}} \in \mathcal{A}_{\mathbf{t}}$, $A_{\mathbf{t}} \subseteq X_{\mathbf{t}}$, are the largest sets such that, for arbitrary $\varnothing \vdash \mathbf{t} \ {}^{s_1}\!\Downarrow_{l_1}^{w_1} \mathbf{v}_1$ and $\varnothing \vdash \mathbf{t} \ {}^{s_2}\!\Downarrow_{l_2}^{w_2} \mathbf{v}_2$, $l_1|_{A_{\mathbf{t}}} = l_2|_{A_{\mathbf{t}}}$.*

For a given $A_{\mathbf{t}}$, the *aligned expressions*—expressions bound by a `let` to a variable name in $A_{\mathbf{t}}$—are those that occur in the same order in every execution, regardless of random draws. We seek the largest sets, as $A_{\mathbf{t}} = \varnothing$ is always a trivial solution. Assume we have a program with $X_{\mathbf{t}} = \{x, y, z\}$ and such that $l = [x, y, x, z, x]$ and $l = [x, y, x, z, x, y]$ are the only possible sequences of `let` bindings. Then, $A_{\mathbf{t}} = \{x, z\}$ is the only possibility. It is also possible to have multiple choices for $A_{\mathbf{t}}$. For example, if $l = [x, y, z]$ and $l = [x, z, y]$ are the only possibilities, then $\mathcal{A}_{\mathbf{t}} = \{\{x, z\}, \{x, y\}\}$. Next, assume that we transform the programs in Fig. 2a and Fig. 1a to ANF. The expression labeled by x in Fig. 2a is then clearly not aligned, as random draws determine how many times it executes (l could be, e.g., $[x, x]$ or $[x, x, x, x]$). Conversely, the expression n (line 13) in Fig. 1a is aligned, as its number and order of evaluations do not depend on any random draws.

Definition 6 is *context insensitive*: for a given $A_{\mathbf{t}}$, each x is either aligned or unaligned. One could also consider a context-sensitive definition of alignment in which x can be aligned in some contexts and unaligned in others. A context could, for example, be the sequence of function applications (i.e., the call stack) leading up to an expression. Considering different contexts for x is complicated and difficult to take full advantage of. We justify the choice of context-insensitive alignment with the real-world models in Section 7, neither of which requires a context-sensitive alignment.

With alignment defined, we now move on to the static alignment analysis.

4.2 Alignment Analysis

The basis for the alignment analysis is 0-CFA [34,42]—a static analysis framework for higher-order functional programs. The prefix 0 indicates that 0-CFA is context insensitive. There is also a set of analyses k-CFA [30] that adds increasing amounts (with $k \in \mathbb{N}$) of context sensitivity to 0-CFA. We could use such analyses with a context-sensitive version of Definition 6. However, the potential benefit of k-CFA is also offset by the worst-case exponential time complexity, already at $k = 1$. In contrast, the time complexity of 0-CFA is polynomial (cubic in the worst-case). The alignment analysis for the models in Section 7 runs instantaneously, justifying that the time complexity is not a problem in practice.

```
 1 let n1 = ¬ in let n2 = ¬ in
 2 let one = 1 in
 3 let half = 0.5 in let c = true in
 4 let f1 = λx1. let t1 = weight one in x1 in
 5 let f2 = λx2. let t2 = weight one in t2 in
 6 let f3 = λx3. let t3 = weight one in t3 in
 7 let f4 = λx4. let t4 = weight one in t4 in
 8 let bern = Bernoulli in
 9 let d1 = bern half in
10 let a1 = assume d1
11 let v1 = f1 one in
12 let v2 = n1 a1 in
13 let v3 = n2 c in
14 let f5 =
15   if a1 then let t5 = f4 one in f2
16   else f3
17 in
18 let v4 = f5 one in
19 let i1 =
20   if c then let t6 = f1 one in t6
21   else one
22 in i1
```

Fig. 4: A program $\mathbf{t}_{example} \in T_{\text{ANF}}$ illustrating the analysis.

The extensions to 0-CFA required to analyze alignment are non-trivial to design, but the resulting formalization is surprisingly simple. The challenge is instead to prove that the extensions correctly capture the alignment property from Definition 6. We extend 0-CFA to analyze stochastic values and alignment in programs $\mathbf{t} \in T_{\text{ANF}}$. As with most static analyses, our analysis is sound but conservative (i.e., sound but incomplete)—the analysis may mark aligned expressions of programs as unaligned, but not vice versa. That the analysis is conservative does not degrade the alignment analysis results for any model in Section 7, which justifies the approach. We divide the formal analysis into two algorithms. Algorithm 1 generates *constraints* for $\mathbf{t}$ that a valid analysis solution must satisfy. This section describes Algorithm 1 and the generated constraints. The second algorithm computes a solution that satisfies the generated constraints. We describe the algorithm at a high level, but omit a full formalization.[†]

For soundness of the analysis, we require $\langle \lambda x.\ \mathbf{t}, \rho \rangle \notin C$ (recall that C is the set of intrinsics). That is, closures are *not* in C. By Definition 3, this implies that closures are not in the sample space of probability distributions in D and that evaluating intrinsics never produces closures (this would unnecessarily complicate the analysis without any benefit).

In addition to standard 0-CFA constraints, Algorithm 1 generates new constraints for *stochastic values* and *unalignment*. We use the contrived but illustrative program in Fig. 4 as an example. Note that, while omitted from Fig. 4 for ease of presentation, the analysis also supports recursion introduced through `let rec`. Stochastic values are values in the program affected by random variables. Stochastic values initially originate at `assume` and then propagate through programs via function applications and `if` expressions. For example, a_1 (line 10) is stochastic because of `assume`. We subsequently use a_1 to define v_2 via n_1 (line 12), which is then also stochastic. Similarly, a_1 is the condition for the `if` resulting in f_5 (line 14), and the function f_5 is therefore also stochastic. When we apply f_5, it results in yet another stochastic value, v_4 (line 18). In conclusion, the stochastic values are a_1, v_2, f_5, and v_4.

Consider the flow of unalignment in Fig. 4. We mark expressions that may execute due to stochastic branching as unaligned. From our analysis of stochastic values, the program's only stochastic `if` condition is at line 15, and we determine that all expressions directly within the branches are unaligned. That is, the expression labeled by t_5 is unaligned. Furthermore, we apply the variable f_4 when defining t_5. Thus, *all* expressions in bodies of lambdas that flow to f_4 are unaligned. Here, it implies that t_4 is unaligned. Finally, we established that the function f_5 produced at line 15 is stochastic. Due to the application at line 18, all names bound by `lets` in bodies of lambdas that flow to f_5 are unaligned. Here, it implies that t_2 and t_3 are unaligned. In conclusion, the unaligned expressions are named by t_2, t_3, t_4, and t_5. For example, aligned SMC therefore resamples at the `weight` at t_1, but not at the `weights` at t_2, t_3, and t_4.

Consider the program in Fig. 1a again, and assume it is transformed to ANF. The alignment analysis must mark all names bound within the stochastic `if` at line 3 as unaligned because a stochastic value flows to its condition. In particular, the `weight` expressions at lines 5 and 8 are unaligned (and the `weight` at line 12 is aligned). Thus, aligned SMC resamples only at line 12.

To formalize the flow of stochastic values, we define *abstract values* $\mathbf{a} ::= \lambda x.y \mid \texttt{stoch} \mid \texttt{const}\ n$, where $x, y \in X$ and $n \in \mathbb{N}$. We use A to denote the set of all abstract values. The `stoch` abstract value is new and represents stochastic values. The $\lambda x.y$ and `const` n abstract values are standard and represent abstract closures and intrinsics, respectively. For each variable name x in the program, we define a set S_x containing abstract values that may occur at x. For example, in Fig. 4, we have $\texttt{stoch} \in S_{a_1}$, $(\lambda x_2.t_2) \in S_{f_2}$, and $(\texttt{const}\ 1) \in S_{n_1}$. The abstract value $\lambda x_2.t_2$ represents all closures originating at λx_2, and `const` 1 represents intrinsic functions in C of arity 1 (in our example, $\neg$). The body of the abstract lambda is the variable name labeling the body, not the body itself. For example, t_2 labels the body `let` t_2 `=` *one* `in` t_2 of λx_2. Due to ANF, all terms have a label, which the function NAME in Algorithm 1 formalizes.

We also define booleans $unaligned_x$ that state whether or not the expression labeled by x is unaligned. For example, we previously reasoned that $unaligned_x =$ true for $x \in \{t_2, t_3, t_4, t_5\}$ in Fig. 4. The alignment analysis aims to determine *minimal* sets S_x and boolean assignments of $unaligned_x$ for every program variable $x \in X$. A trivial solution is that all abstract values (there is a finite number of them in the program) flow to each program variable and that $unaligned_x =$ true for all $x \in X$. This solution is sound but useless. To compute a more precise solution, we follow the rules given by *constraints* $\mathbf{c} \in R$.[†]

We present the constraints through the GENERATECONSTRAINTS function in Algorithm 1 and for the example in Fig. 4. There are no constraints for variables that occur at the end of ANF `let` sequences (line 2 in Algorithm 1), and the case for `let` expressions (lines 3–36) instead produces all constraints. The cases for aliases (line 6), intrinsics (line 7), `assume` (line 35), and `weight` (line 36) are the most simple. Aliases of the form `let` x `=` y `in` $\mathbf{t}_2$ establish $S_y \subseteq S_x$. That is, all abstract values at y are also in x. Intrinsic operations results in a `const` abstract value. For example, the definition of n_1 at line 1 in Fig. 4 results in the constraint $\texttt{const}\ 1 \in S_{n_1}$. Applications of `assume` are the source of stochastic values. For example, the definition of a_1 at line 10 results in the constraint `stoch` $\in S_{a_1}$. Note that `assume` cannot produce any other abstract values, as we only

Algorithm 1 Constraint generation function for $\mathbf{t} \in T_{\text{ANF}}$. We denote the power set of a set E with $\mathcal{P}(E)$.

```
function GENERATECONSTRAINTS($\mathbf{t}$): $T_{\text{ANF}} \to \mathcal{P}(R)$ =
 1  match t with
 2  | $x \to \varnothing$
 3  | let $x = \mathbf{t}_1$ in $\mathbf{t}_2 \to$
 4  GENERATECONSTRAINTS($\mathbf{t}_2$) $\cup$
 5     match $\mathbf{t}_1$ with
 6     | $y \to \{S_y \subseteq S_x\}$
 7     | $c \to$ if $|c| > 0$ then $\{\texttt{const}\ |c| \in S_x\}$
 8           else $\varnothing$
 9     | $\lambda y.\ \mathbf{t}_y \to$ GENERATECONSTRAINTS($\mathbf{t}_y$)
10       $\cup\ \{\lambda y.\ \text{NAME}(\mathbf{t}_y) \in S_x\}$
11       $\cup\ \{unaligned_y \Rightarrow unaligned_n$
12           $\mid n \in \text{NAMES}(t_y)\}$
13     | $lhs\ rhs \to \{$
14        $\forall z \forall y\ \lambda z.y \in S_{lhs}$
15          $\Rightarrow (S_{rhs} \subseteq S_z) \wedge (S_y \subseteq S_x)$,
16        $\forall n\ (\texttt{const}\ n \in S_{lhs}) \wedge (n > 1)$
17          $\Rightarrow \texttt{const}\ n - 1 \in S_x$,
18        $\texttt{stoch} \in S_{lhs} \Rightarrow \texttt{stoch} \in S_x$,
19        $\texttt{const}\ \_ \in S_{lhs}$
20          $\Rightarrow (\texttt{stoch} \in S_{rhs} \Rightarrow \texttt{stoch} \in S_x)$,
21        $unaligned_x$
22          $\Rightarrow (\forall y\ \lambda y.\_ \in S_{lhs} \Rightarrow unaligned_y)$,
23        $\texttt{stoch} \in S_{lhs}$
24          $\Rightarrow (\forall y\ \lambda y.\_ \in S_{lhs} \Rightarrow unaligned_y)$
25     }
26    | if $y$ then $\mathbf{t}_t$ else $\mathbf{t}_e \to$
27      GENERATECONSTRAINTS($\mathbf{t}_t$)
28        $\cup$ GENERATECONSTRAINTS($\mathbf{t}_e$)
29        $\cup\ \{S_{\text{NAME}(\mathbf{t}_t)} \subseteq S_x, S_{\text{NAME}(\mathbf{t}_e)} \subseteq S_x$,
30            $\texttt{stoch} \in S_y \Rightarrow \texttt{stoch} \in S_x\}$
31        $\cup\ \{unaligned_x \Rightarrow unaligned_n$
32            $\mid n \in \text{NAMES}(\mathbf{t}_t) \cup \text{NAMES}(\mathbf{t}_e)\}$
33        $\cup\ \{\texttt{stoch} \in S_y \Rightarrow unaligned_n$
34            $\mid n \in \text{NAMES}(\mathbf{t}_t) \cup \text{NAMES}(\mathbf{t}_e)\}$
35    | assume $\_ \to \{\texttt{stoch} \in S_x\}$
36    | weight $\_ \to \varnothing$
37
38  function NAME($\mathbf{t}$): $T_{\text{ANF}} \to X$ =
39    match t with
40    | $x \to x$
41    | let $x = \mathbf{t}_1$ in $\mathbf{t}_2 \to$ NAME($\mathbf{t}_2$)
42
43  function NAMES($\mathbf{t}$): $T_{\text{ANF}} \to \mathcal{P}(X)$ =
44    match t with
45    | $x \to \varnothing$
46    | let $x = \_$ in $\mathbf{t}_2 \to \{x\} \cup$ NAMES($\mathbf{t}_2$)
47
48
49
50
```

allow distributions over intrinsics with arity 0 (see Definition 3). Finally, we use `weight` only for its side effect (likelihood updating), and therefore `weight`s do not produce any abstract values and consequently no constraints.

The cases for abstractions (line 9), applications (line 13), and `if`s (line 26) are more complex. The abstraction at line 4 in Fig. 4 generates (omitting the recursively generated constraints for the abstraction body $\mathbf{t}_y$) the constraints $\{\lambda x_1.x_1 \in S_{f_1}\} \cup \{unaligned_{x_1} \Rightarrow unaligned_{t_1}\}$. The first constraint is standard: the abstract lambda $\lambda x_1.x_1$ flows to S_{f_1}. The second constraint states that if the abstraction is unaligned, all expressions in its body (here, only t_1) are unaligned. We define the sets of expressions within abstraction bodies and `if` branches through the NAMES function in Algorithm 1 (line 43).

The application f_5 *one* at line 18 in Fig. 4 generates the constraints

$$\begin{aligned}
&\{\forall z \forall y\ \ \lambda z.y \in S_{f_5} \Rightarrow (S_{one} \subseteq S_z) \wedge (S_y \subseteq S_{v_4}),\\
&\ \forall n\ \ (\texttt{const}\ \ n \in S_{f_5}) \wedge (n > 1) \Rightarrow \texttt{const}\ \ n - 1 \in S_{v_4},\\
&\ \texttt{stoch} \in S_{f_5} \Rightarrow \texttt{stoch} \in S_{v_4},\\
&\ \texttt{const}\ \ \_ \in S_{f_5} \Rightarrow (\texttt{stoch} \in S_{one} \Rightarrow \texttt{stoch} \in S_{v_4}),\\
&\ unaligned_{v_4} \Rightarrow (\forall y\ \ \lambda y.\_ \in S_{f_5} \Rightarrow unaligned_y),\\
&\ \texttt{stoch} \in S_{f_5} \Rightarrow (\forall y\ \ \lambda y.\_ \in S_{lhs} \Rightarrow unaligned_y)\}
\end{aligned} \tag{3}$$

The first constraint is standard: if an abstract value $\lambda z.y$ flows to f_5, the abstract values of *one* (the right-hand side) flow to z. Furthermore, the result of the application, given by the body name y, must flow to the result v_4 of the application.

The second constraint is also relatively standard: if an intrinsic function of arity n is applied, it produces a const of arity $n-1$. The other constraints are new and specific for stochastic values and unalignment. The third constraint states that if the function is stochastic, the result is stochastic. The fourth constraint states that if we apply an intrinsic function to a stochastic argument, the result is stochastic. We could also make the analysis of intrinsic applications less conservative through intrinsic-specific constraints. The fifth and sixth constraints state that if the expression (labeled by v_4) is unaligned or the function is stochastic, all abstract lambdas that flow to the function are unaligned.

The `if` resulting in f_5 at line 14 in Fig. 4 generates (omitting the recursively generated constraints for the branches $\mathbf{t}_t$ and $\mathbf{t}_e$) the constraints

$$\begin{gathered}\{S_{\textsc{name}(f_2)} \subseteq S_{f_5}, S_{\textsc{name}(f_3)} \subseteq S_{f_5}, \texttt{stoch} \in S_{a_1} \Rightarrow \texttt{stoch} \in S_{f_5}\} \\ \cup \{\mathit{unaligned}_{f_5} \Rightarrow \mathit{unaligned}_{t_5}\} \cup \{\texttt{stoch} \in S_{a_1} \Rightarrow \mathit{unaligned}_{t_5}\}\end{gathered} \tag{4}$$

The first two constraints are standard and state that the result of the branches flows to the result of the `if` expression. The remaining constraints are new. The third constraint states that if the condition is stochastic, the result is stochastic. The last two constraints state that if the `if` is unaligned or if the condition is stochastic, all names in the branches (here, only t_5) are unaligned.

Given constraints for a program, we need to compute a solution satisfying all constraints. We do this by repeatedly iterating through all the constraints and propagating abstract values accordingly. We terminate when we reach a fixed point, i.e., when no constraint results in an update of either S_x or $\mathit{unaligned}_x$ for any x in the program. We extend the 0-CFA constraint propagation algorithm to also handle the constraints generated for tracking stochastic values and unalignment.[†] Specifically, the algorithm is a function ANALYZEALIGN: $T_{\text{ANF}} \to ((X \to \mathcal{P}(A)) \times \mathcal{P}(X))$ that returns a map associating each variable to a set of abstract values and a set of unaligned variables. In other words, ANALYZEALIGN computes a solution to S_x and $\mathit{unaligned}_x$ for each x in the analyzed program. For example, ANALYZEALIGN($\mathbf{t}_{example}$) results in

$$\begin{gathered} S_{n_1} = \{\texttt{const}\ 1\}\ \ S_{n_2} = \{\texttt{const}\ 1\}\ \ S_{f_1} = \{\lambda x_1.x_1\}\ \ S_{f_2} = \{\lambda x_2.t_2\} \\ S_{f_3} = \{\lambda x_3.t_3\}\ \ S_{f_4} = \{\lambda x_4.t_4\}\ \ S_{a_1} = \{\texttt{stoch}\}\ \ S_{v_2} = \{\texttt{stoch}\} \\ S_{f_5} = \{\lambda x_2.t_2, \lambda x_3.t_3, \texttt{stoch}\}\ \ S_{v_4} = \{\texttt{stoch}\}\ \ S_n = \varnothing \mid \text{other } n \in X \\ \mathit{unaligned}_n = \text{true} \mid n \in \{t_2, t_3, t_4, t_5\}\ \ \mathit{unaligned}_n = \text{false} \mid \text{other } n \in X. \end{gathered} \tag{5}$$

The example confirms our earlier intuition: an intrinsic ($\neg$) flows to n_1, `stoch` flows to a_1, f_5 is stochastic and originates at either $(\lambda x_2.t_2)$ or $(\lambda x_3.t_3)$, and the unaligned variables are t_2, t_3, t_4, and t_5. We now give soundness results.

Lemma 1 (0-CFA soundness). *For every* $\mathbf{t} \in T_{\text{ANF}}$*, the solution produced by* ANALYZEALIGN($\mathbf{t}$) *satisfies the constraints* GENERATECONSTRAINTS($\mathbf{t}$).

Proof. The well-known soundness of 0-CFA extends to the new alignment constraints. See, e.g., Nielson et al. [34, Chapter 3] and Shivers [42]. □

Theorem 1 (Alignment analysis soundness). *Assume* $\mathbf{t} \in T_{\text{ANF}}$*,* $\mathcal{A}_\mathbf{t}$ *from Definition 6, and an assignment to* S_x *and* $unaligned_x$ *for* $x \in X$ *according to* $\textsc{analyzeAlign}(\mathbf{t})$*. Let* $\widehat{A}_\mathbf{t} = \{x \mid \neg unaligned_x\}$ *and take arbitrary* $\varnothing \vdash \mathbf{t} \ {}^{s_1}{\Downarrow}^{w_1}_{l_1} \ \mathbf{v}_1$ *and* $\varnothing \vdash \mathbf{t} \ {}^{s_2}{\Downarrow}^{w_2}_{l_2} \ \mathbf{v}_2$*. Then,* $l_1|_{\widehat{A}_\mathbf{t}} = l_2|_{\widehat{A}_\mathbf{t}}$ *and consequently* $\widehat{A}_\mathbf{t} \subseteq A_\mathbf{t}$ *for at least one* $A_\mathbf{t} \in \mathcal{A}_\mathbf{t}$*.*

The proof[†] uses simultaneous structural induction over the derivations $\varnothing \vdash \mathbf{t} \ {}^{s_1}{\Downarrow}^{w_1}_{l_1} \ \mathbf{v}_1$ and $\varnothing \vdash \mathbf{t} \ {}^{s_2}{\Downarrow}^{w_2}_{l_2} \ \mathbf{v}_2$. At corresponding stochastic branches or stochastic function applications in the two derivations, a separate structural induction argument shows that, for the `let`-sequences l'_1 and l'_2 of the two stochastic subderivations, $l'_1|_{\widehat{A}_\mathbf{t}} = l'_2|_{\widehat{A}_\mathbf{t}} = []$. Combined, the two arguments give the result.

The result $\widehat{A}_\mathbf{t} \subseteq A_\mathbf{t}$ (cf. Definition 6) shows that the analysis is conservative.

4.3 Dynamic Alignment

An alternative to static alignment is *dynamic* alignment, which we explored in early stages when developing the alignment analysis. Dynamic alignment is fully context sensitive and amounts to introducing variables in programs that track (at runtime) when evaluation enters stochastic branching. To identify these stochastic branches, dynamic alignment also requires a runtime data structure that keeps track of the stochastic values. Similarly to k-CFA, dynamic alignment is potentially more precise than the 0-CFA approach. However, we discovered that dynamic alignment introduces significant runtime overhead. Again, we note that the models in Section 7 do not require a context-sensitive analysis, justifying the choice of 0-CFA over dynamic alignment and k-CFA.

5 Aligned SMC and MCMC

This section presents detailed algorithms for aligned SMC (Section 5.1) and aligned lightweight MCMC (Section 5.2). For a more pedagogical introduction to the algorithms, see Section 2. We assume a basic understanding of SMC and Metropolis–Hastings MCMC algorithms (see, e.g., Bishop [4]).

5.1 Aligned SMC

We saw in Section 2.1 that SMC operates by executing many instances of $\mathbf{t}$ concurrently, and resampling them at calls to `weight`. Critically, resampling requires that the inference algorithm can both suspend and resume executions. Here, we assume that we can create execution instances e of the probabilistic program $\mathbf{t}$, and that we can arbitrarily suspend and resume the instances. The technical details of suspension are beyond the scope of this paper. See Goodman and Stuhlmüller [14], Wood et al. [48], and Lundén et al. [25] for further details.

Algorithm 2 presents all steps for the aligned SMC inference algorithm. After running the alignment analysis and setting up the n execution instances, the algorithm iteratively executes and resamples the instances. Note that the algorithm resamples only at aligned weights (see Section 2.1).

Algorithm 2 Aligned SMC. The input is a program $\mathbf{t} \in T_{\text{ANF}}$ and the number of execution instances n.

```
1. Run the alignment analysis on $\mathbf{t}$, resulting in $\widehat{A}_{\mathbf{t}}$ (see Theorem 1).
2. Initiate $n$ execution instances $\{e_i \mid i \in \mathbb{N}, 1 \leq i \leq n\}$ of $\mathbf{t}$.
3. Execute all $e_i$ and suspend execution upon reaching an aligned weight (i.e., let $x$ = weight
   $w$ in t and $x \in \widehat{A}_{\mathbf{t}}$) or when the execution terminates naturally. The result is a new set of
   execution instances $e'_i$ with weights $w'_i$ accumulated from unaligned weights and the single final
   aligned weight during execution.
4. If all $e'_i = \mathbf{v}'_i$ (i.e., all executions have terminated and returned a value), terminate inference and
   return the set of weighted samples $(\mathbf{v}'_i, w'_i)$. The samples approximate the posterior probability
   distribution encoded by $\mathbf{t}$.
5. Resample the $e'_i$ according to their weights $w'_i$. The result is a new set of unweighted execution
   instances $e''_i$. Set $e_i \leftarrow e''_i$. Go to 3.
```

```
if assume Bernoulli(0.5) then
  weight 1; weight 10; true
else
  weight 10; weight 1; false
```

(a) Aligned better than unaligned.

```
if assume Bernoulli(0.1) then
  weight 9;
  if assume Bernoulli(0.5)
  then weight 1.5 else weight 0.5;
  true
else (weight 1; false)
```

(b) Unaligned better than aligned.

Fig. 5: Programs illustrating properties of aligned and unaligned SMC. Fig. (a) shows a program better suited for aligned SMC. Fig. (b) shows a program better suited for unaligned SMC.

We conjecture that aligned SMC is preferable over unaligned SMC for all practically relevant models, as the evaluation in Section 7 justifies. However, it is possible to construct contrived programs in which unaligned SMC has the advantage. Consider the programs in Fig. 5, both encoding Bernoulli(0.5) distributions in a contrived way using `weight`s. Fig. 5a takes one of two branches with equal probability. Unaligned SMC resamples at the first `weight`s in each branch, while aligned SMC does not because the branch is stochastic. Due to the difference in likelihood, many more `else` executions survive resampling compared to `then` executions. However, due to the final `weight`s in each branch, the branch likelihoods even out. That is, resampling at the first `weight`s is detrimental, and unaligned SMC performs worse than aligned SMC. Fig. 5b also takes one of two branches, but now with unequal probabilities. However, the two branches still have equal posterior probability due to the `weight`s. The nested if in the `then` branch does not modify the overall branch likelihood, but adds variance. Aligned SMC does not resample for any `weight` within the branches, as the branch is stochastic. Consequently, only 10% of the executions in aligned SMC take the `then` branch, while half of the executions take the `then` branch in unaligned SMC (after resampling at the first `weight`). Therefore, unaligned SMC better explores the `then` branch and reduces the variance due to the nested `if`, which results in overall better inference accuracy. We are not aware of any real model with the property in Fig. 5b. In practice, it seems best to always resample when using `weight` to condition on observed data. Such conditioning is, in practice, always done outside of stochastic branches, justifying the benefit of aligned SMC.

Algorithm 3 Aligned lightweight MCMC. The input is a program $\mathbf{t} \in T_{\text{ANF}}$, the number of steps n, and the global step probability $g > 0$.

```
1. Run the alignment analysis on $\mathbf{t}$, resulting in $\widehat{A}_{\mathbf{t}}$ (see Theorem 1).
2. Set $i \leftarrow 0$, $k \leftarrow 1$, and $l \leftarrow 1$. Call Run.
3. Set $i \leftarrow i + 1$. If $i = n$, terminate inference and return the samples $\{\mathbf{v}_j \mid j \in \mathbb{N}, 0 \leq j < n\}$.
   They approximate the probability distribution encoded by $\mathbf{t}$.
4. Uniformly draw an index $1 \leq j \leq |s_{i-1}|$ at random. Set $global \leftarrow$ true with probability $g$, and
   $global \leftarrow$ false otherwise. Set $w'_{-1} \leftarrow 1$, $w' \leftarrow 1$, $k \leftarrow 1$, $l \leftarrow 1$, and $reuse \leftarrow$ true. Call Run.
5. Compute the Metropolis–Hastings acceptance ratio $A = \min\left(1, \frac{w_i}{w_{i-1}} \frac{w'}{w'_{-1}}\right)$.
6. With probability $A$, accept $\mathbf{v}_i$ and go to 3. Otherwise, set $\mathbf{v}_i \leftarrow \mathbf{v}_{i-1}$, $w_i \leftarrow w_{i-1}$, $s_i \leftarrow s_{i-1}$,
   $p_i \leftarrow p_{i-1}$, $s'_i \leftarrow s'_{i-1}$, $p'_i \leftarrow p'_{i-1}$, and $n'_i \leftarrow n'_{i-1}$. Go to 3.

function RUN() = Run $\mathbf{t}$ and do the following:
  – Record the total weight $w_i$ accumulated from calls to weight.
  – Record the final value $\mathbf{v}_i$.
  – At unaligned terms let $c$ = assume $d$ in $\mathbf{t}$ ($c \notin \widehat{A}_{\mathbf{t}}$), do the following.
      1. If $reuse$ = false, $global$ = true, $n'_{i-1,k,l} \neq c$, or if $s'_{i-1,k,l}$ does not exist, sample a
         value $x$ from $d$ and set $reuse \leftarrow$ false. Otherwise, reuse the sample $x = s'_{i-1,k,l}$ and set
         $w'_{-1} \leftarrow w'_{-1} \cdot p'_{i-1,k,l}$ and $w' \leftarrow w' \cdot f_d(c)$.
      2. Set $s'_{i,k,l} \leftarrow x$, $p'_{i,k,l} \leftarrow f_d(x)$, and $n'_{i,k,l} \leftarrow c$.
      3. Set $l \leftarrow l + 1$. In the program, bind $c$ to the value $x$ and resume execution.
  – At aligned terms let $c$ = assume $d$ in $\mathbf{t}$ ($c \in \widehat{A}_{\mathbf{t}}$), do the following.
      1. If $j = k$, $global$ = true, or if $s_{i-1,k}$ does not exist, sample a value $x$ from $d$ normally.
         Otherwise, reuse the sample $x = s_{i-1,k}$. Set $w'_{-1} \leftarrow w'_{-1} \cdot p_{i-1,k}$ and $w' \leftarrow w' \cdot f_d(x)$.
      2. Set $s_{i,k} \leftarrow x$ and $p_{i,k} \leftarrow f_d(x)$.
      3. Set $k \leftarrow k + 1$, $l \leftarrow 1$, and $reuse \leftarrow$ true. In the program, bind $c$ to the value $x$ and resume
         execution.
```

5.2 Aligned Lightweight MCMC

Aligned lightweight MCMC is a version of lightweight MCMC [47], where the alignment analysis provides information about how to reuse random draws between executions. Algorithm 3, a Metropolis–Hastings algorithm in the context of PPLs, presents the details. Essentially, the algorithm executes the program repeatedly using the Run function, and redraws one aligned random draw in each step, while reusing all other aligned draws and as many unaligned draws as possible (illustrated in Section 2.2). It is possible to formally derive the Metropolis–Hastings acceptance ratio in step 5.† A key property in Algorithm 3 due to alignment (Definition 6) is that the length of s_i (and p_i) is constant, as executing $\mathbf{t}$ always results in the same number of aligned random draws.

In addition to redrawing only one aligned random draw, each step has a probability $g > 0$ of being *global*—meaning that inference redraws *every* random draw in the program. Occasional global steps fix problems related to slow mixing and ergodicity of lightweight MCMC identified by Kiselyov [21]. In a global step, the Metropolis–Hastings acceptance ratio reduces to $A = \min\left(1, \frac{w_i}{w_{i-1}}\right)$.

6 Implementation

We implement the alignment analysis (Section 4), aligned SMC (Section 5.1), and aligned lightweight MCMC (Section 5.2) for the functional PPL *Miking*

CorePPL [25], implemented as part of the *Miking* framework [7]. We implement the alignment analysis as a core component in the Miking CorePPL compiler, and then use the analysis when compiling to two Miking CorePPL backends: RootPPL and Miking Core. RootPPL is a low-level PPL with built-in highly efficient SMC inference [25], and we extend the CorePPL to RootPPL compiler introduced by Lundén et al. [25] to support aligned SMC inference. Furthermore, we implement aligned lightweight MCMC inference standalone as a translation from Miking CorePPL to Miking Core. Miking Core is the general-purpose programming language of the Miking framework, currently compiling to OCaml.

The idealized calculus in (1) does not capture all features of Miking CorePPL. In particular, the alignment analysis implementation must support records, variants, sequences, and pattern matching over these. Extending 0-CFA to such language features is not new, but it does introduce a critical challenge for the alignment analysis: identifying all possible stochastic branches. Determining stochastic `if`s is straightforward, as we simply check if `stoch` flows to the condition. However, complications arise when we add a `match` construct (and, in general, any type of branching construct). Consider the extension

$$\begin{aligned}&\mathbf{t} ::= \;\ldots\; \mid \texttt{match}\ \mathbf{t}\ \texttt{with}\ \mathbf{p}\ \texttt{then}\ \mathbf{t}\ \texttt{else}\ \mathbf{t} \mid \texttt{\{}k_1 = x_1\texttt{,}\ \ldots\texttt{,}\ k_n = x_n\texttt{\}}\\ &\mathbf{p} ::= x \mid \text{true} \mid \text{false} \mid \texttt{\{}k_1 = \mathbf{p}\texttt{,}\ \ldots\texttt{,}\ k_n = \mathbf{p}\texttt{\}}\\ &x, x_1, \ldots, x_n \in X \quad k_1, \ldots, k_n \in K \quad n \in \mathbb{N}\end{aligned} \tag{6}$$

of (1), adding records and simple pattern matching. K is a set of record keys. Assume we also extend the abstract values as $\mathbf{a} ::= \ldots \mid \texttt{\{}k_1 = X_1, \ldots, k_n = X_n\texttt{\}}$, where $X_1, \ldots, X_n \subseteq X$. That is, we add an abstract record tracking the names in the program that flow to its entries. Consider the program `match` t_1 `with {` $a = x_1$`,` $b = \text{false}$ `} then` t_2 `else` t_3. This `match` is, similar to `if`s, stochastic if `stoch` $\in S_{t_1}$. It is also, however, stochastic in other cases. Assume we have two program variables, x and y, such that `stoch` $\in S_x$ and `stoch` $\notin S_y$. Now, the `match` is stochastic if, e.g., $\texttt{\{}a = \{y\}\texttt{,}\ b = \{x\}\texttt{\}} \in S_{t_1}$, because the random value flowing from x to the pattern false may not match because of randomness. However, it is *not* stochastic if, instead, $S_{t_1} = \{\texttt{\{}a = \{x\}\texttt{,}\ b = \{y\}\texttt{\}}\}$. The randomness of x does *not* influence whether or not the branch is stochastic—the variable pattern x_1 for label a always matches.

Our alignment analysis implementation handles the intricacies of identifying stochastic `match` cases for nested record, variant, and sequence patterns. In total, the alignment analysis, aligned SMC, and aligned lightweight MCMC implementations consist of approximately 1000 lines of code directly contributed as part of this paper. The code is available on GitHub [2].

7 Evaluation

This section evaluates aligned SMC and aligned lightweight MCMC on a set of models encoded in Miking CorePPL: CRBD [33,39] in Sections 7.1 and 7.5, ClaDS [28,39] in Section 7.2, state-space aircraft localization in Section 7.3, and latent Dirichlet allocation in Section 7.4. CRBD and ClaDS are non-trivial models of considerable interest in evolutionary biology and phylogenetics [39]. Similarly, LDA is a non-trivial topic model [5]. Running the alignment analysis took approximately 5 ms–30 ms for all models considered in the experiment, justifying that the time complexity is not a problem in practice.

We compare aligned SMC with standard unaligned SMC [14], which is identical to Algorithm 2, except that it resamples at *every* call to `weight`.† We carefully checked that automatic alignment corresponds to previous manual alignments of each model. For all SMC experiments, we estimate the *normalizing constant* produced as a by-product of SMC inference rather than the complete posterior distributions. The normalizing constant, also known as marginal likelihood or model evidence, frequently appears in Bayesian inference and gives the probability of the observed data averaged over the prior. The normalizing constant is useful for model comparison as it measures how well different probabilistic models fit the data (a larger normalizing constant indicates a better fit).

We ran aligned and unaligned SMC with Miking CorePPL and the RootPPL backend configured for a single-core (compiled with GCC 7.5.0). Lundén et al. [25] shows that the RootPPL backend is significantly more efficient than other state-of-the-art PPL SMC implementations. We ran aligned and unaligned SMC inference 300 times (and with 3 warmup runs) for each experiment for 10^4, 10^5, and 10^6 executions (also known as *particles* in SMC literature).

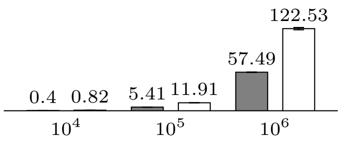
We compare aligned lightweight MCMC to lightweight MCMC.† We implement both versions as compilers from Miking CorePPL to Miking Core, which in turn compiles to OCaml (version 4.12). The lightweight MCMC databases are functional-style maps from the OCaml `Map` library. We set the global step probability to 0.1 for both aligned lightweight MCMC and lightweight MCMC. We ran aligned lightweight and lightweight MCMC inference 300 times for each experiment. We burned 10% of samples in all MCMC runs.

For all experiments, we used an Intel Xeon 656 Gold 6136 CPU (12 cores) and 64 GB of memory running Ubuntu 18.04.5.

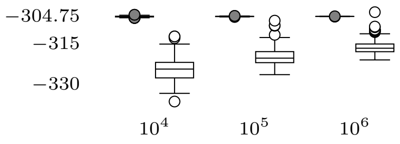
7.1 SMC: Constant Rate Birth-Death (CRBD)

This experiment considers the CRBD diversification model from [39] applied to the Alcedinidae phylogeny (Kingfisher birds, 54 extant species) [19]. We use fixed diversification rates to simplify the model, as unaligned SMC inference accuracy is too poor for the full model with priors over diversification rates. Aligned SMC is accurate for both the full and simplified models. The source code consists of 130 lines of code.† The total experiment execution time was 16 hours.

Fig. 6 presents the experiment results. Aligned SMC is roughly twice as fast and produces superior estimates of the normalizing constant. Unaligned SMC has not yet converged to the correct value -304.75 (available for this particular model due to the fixing the diversification rates) for 10^6 particles, while aligned SMC produces precise estimates already at 10^4 particles. Excess resampling is a significant factor in the increase in execution time for unaligned SMC, as each execution encounters far more resampling checkpoints than in aligned SMC.

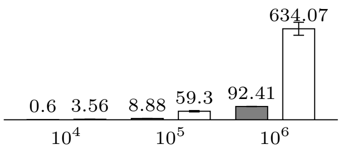


(a) Execution times.

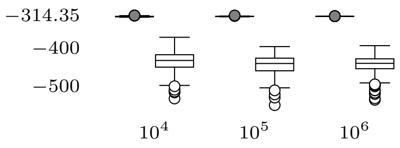


(b) Log normalizing constant estimates.

Fig. 6: SMC experiment results for CRBD. The x-axes give the number of particles. Fig. (a) shows execution times (in seconds) for aligned (gray) and unaligned (white) SMC. Error bars show one standard deviation. Fig. (b) shows box plot log normalizing constant estimates for aligned (gray) and unaligned (white) SMC. The analytically computed log normalizing constant is −304.75.



(a) Execution times.



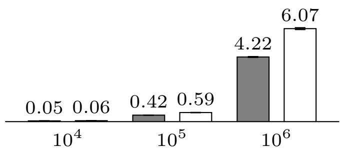
(b) Log normalizing constant estimates.

Fig. 7: SMC experiment results for ClaDS. The x-axes give the number of particles. Fig. (a) shows execution times (in seconds) for aligned (gray) and unaligned (white) SMC. Error bars show one standard deviation. Fig. (b) shows box plot log normalizing constant estimates for aligned (gray) and unaligned (white) SMC. The average estimate for aligned SMC with 10^6 particles is −314.35.

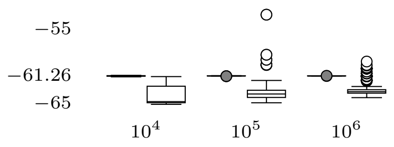
7.2 SMC: Cladogenetic Diversification Rate Shift (ClaDS)

A limitation of CRBD is that the diversification rates are constant. ClaDS [28,39] is a set of diversification models that allow *shifting* rates over phylogenies. We evaluate the ClaDS2 model for the Alcedinidae phylogeny. As in CRBD, we use fixed (initial) diversification rates to simplify the model on account of unaligned SMC. The source code consists of 147 lines of code.† Automatic alignment simplifies the ClaDS2 model significantly, as manual alignment requires collecting and passing weights around in unaligned parts of the program, which are later consumed by aligned `weight`s. The total experiment execution time was 67 hours.

Fig. 7 presents the experiment results. 12 unaligned runs for 10^6 particles and nine runs for 10^5 particles ran out of the preallocated stack memory for each particle (10 kB). We omit these runs from Fig. 7. The consequence of not aligning SMC is more severe than for CRBD. Aligned SMC is now almost seven times faster than unaligned SMC and the unaligned SMC normalizing constant estimates are significantly worse compared to the aligned SMC estimates. The unaligned SMC estimates do not even improve when moving from 10^4 to 10^6 particles (we need even more particles to see improvements). Again, aligned SMC produces precise estimates already at 10^4 particles.



(a) Execution times.



(b) Log normalizing constant estimates.

Fig. 8: SMC experiment results for the state-space aircraft localization model. The x-axes give the number of particles. Fig. (a) shows execution times (in seconds) for aligned (gray) and unaligned (white) SMC. Error bars show one standard deviation. Fig. (b) shows box plot log normalizing constant estimates on the y-axis for aligned (gray) and unaligned (white) SMC. The average estimate for aligned SMC with 10^6 particles is -61.26.

7.3 SMC: State-Space Aircraft Localization

This experiment considers an artificial but non-trivial state-space model for aircraft localization. The source code consists of 62 lines of code.† The total experiment execution time was 1 hour.

Fig. 8 presents the experiment results. The execution time difference is not as significant as for CRBD and ClaDS. However, the unaligned SMC normalizing constant estimates are again much less precise. Aligned SMC is accurate (centered at approximately -61.26) already at 10^4 particles. The model's straightforward control flow explains the less dramatic difference in execution time—there are at most ten unaligned likelihood updates in the aircraft model, while the number is, in theory, unbounded for CRBD and ClaDS. Therefore, the cost of extra resampling compared to aligned SMC is not as significant.

7.4 MCMC: Latent Dirichlet Allocation (LDA)

This experiment considers latent Dirichlet allocation (LDA), a topic model used in the evaluations by Wingate et al. [47] and Ritchie et al. [38]. We use a synthetic data set, comparable in size to the data set used by Ritchie et al. [38], with a vocabulary of 100 words, 10 topics, and 25 documents each containing 30 words. Note that we are not using methods based on collapsed Gibbs sampling [17], and the inference task is therefore computationally challenging even with a rather small number of words and documents. The source code consists of 31 lines of code.† The total experiment execution time was 41 hours.

The LDA model consists of only aligned random draws. As a consequence, aligned lightweight and lightweight MCMC reduces to the same inference algorithm, and we can compare the algorithms by just considering the execution times. The experiment also justifies the correctness of both algorithms.†

Fig. 9 presents the experiment results. Aligned lightweight MCMC is almost three times faster than lightweight MCMC. To justify the execution times with our implementations, we also implemented and ran the experiment with

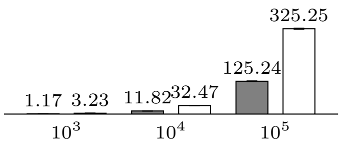


Fig. 9: MCMC experiment results for LDA showing execution time (in seconds) for aligned lightweight MCMC (gray) and lightweight MCMC (white). Error bars show one standard deviation and the x-axis the number of MCMC iterations.

lightweight MCMC in WebPPL [14] for 10^5 iterations, repeated 50 times (and with 3 warmup runs). The mean execution time was 383 s with standard deviation 5 s. We used WebPPL version 0.9.15 and Node version 16.18.0.

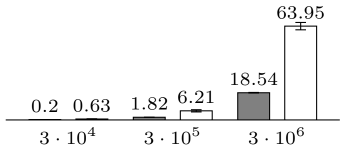
7.5 MCMC: Constant Rate Birth-Death (CRBD)

This experiment again considers CRBD. MCMC is not as suitable for CRBD as SMC, and therefore we use a simple synthetic phylogeny with six leaves and an age span of 5 age units (Alcedinidae used for the SMC experiment has 54 leaves and an age span of 35 age units). The source code for the complete model is the same as in Section 7.1, but we now allow the use of proper prior distributions for the diversification rates. The total experiment execution time was 7 hours.

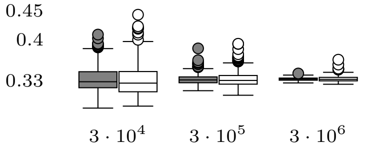
Unlike LDA, the CRBD model contains both unaligned and aligned random draws. Because of this, aligned lightweight MCMC and standard lightweight MCMC do *not* reduce to the same algorithm. To judge the difference in inference accuracy, we consider the mean estimates of the birth diversification rate produced by the two algorithms, in addition to execution times. The experiment results shows that the posterior distribution over the birth rate is unimodal[†], which motivates using the posterior mean as a measure of accuracy.

Fig. 10 presents the experiment results. Aligned lightweight MCMC is approximately 3.5 times faster than lightweight MCMC. There is no obvious difference in accuracy. To justify the execution times and correctness of our implementations, we also implemented and ran the experiment with lightweight MCMC in WebPPL [14] for $3 \cdot 10^6$ iterations, repeated 50 times (and with 3 warmup runs). The mean estimates agreed with Fig. 10. The mean execution time was 37.1 s with standard deviation 0.8 s. The speedup compared to standard lightweight MCMC in Miking CorePPL is likely explained by the use of early termination in WebPPL, which benefits CRBD. Early termination easily combines with alignment but relies on execution suspension, which we do not currently use in our implementations. Note that aligned lightweight MCMC is faster than WebPPL even without early termination.

In conclusion, the experiments clearly demonstrate the need for alignment.



(a) Execution times.



(b) Birth rate mean estimates.

Fig. 10: MCMC experiment results for CRBD. The x-axes give the number of iterations. Fig. (a) shows execution times (in seconds) for aligned lightweight MCMC (gray) and lightweight MCMC (white). Error bars show one standard deviation. Fig. (b) shows box plot posterior mean estimates of the birth rate for aligned lightweight MCMC (gray) and lightweight MCMC (white). The average estimate for aligned lightweight MCMC with $3 \cdot 10^6$ iterations is 0.33.

8 Related Work

The approach by Wingate et al. [47] is closely related to ours. A key similarity with alignment is that executions reaching the same aligned checkpoint also have matching stack traces according to Wingate et al.'s addressing transform. However, Wingate et al. do not consider the separation between unaligned and aligned parts of the program, their approach is not static, and they do not generalize to other inference algorithms such as SMC.

Ronquist et al. [39], Turing [12], Anglican [48], Paige and Wood [36], and van de Meent et al. [46] consider the alignment problem. Manual alignment is critical for the models in Ronquist et al. [39] to make SMC inference tractable, which strongly motivates the automatic alignment approach. The documentation of Turing states that: "The `observe` statements [i.e., likelihood updates] should be arranged so that every possible run traverses all of them in exactly the same order. This is equivalent to demanding that they are not placed inside stochastic control flow" [1]. Turing does not include any automatic checks for this property. Anglican [48] checks, at runtime (resulting in overhead), that all SMC executions encounter the same number of likelihood updates, and thus resamples the same number of times. If not, Anglican reports an error: "some `observe` directives [i.e., likelihood updates] are not global". This error refers to the alignment problem, but the documentation does not explain it further. Probabilistic C, introduced by Paige and Wood [36], similarly assumes that the number of likelihood updates is the same in all executions. Van de Meent et al. [46] state, in reference to SMC: "Each breakpoint [i.e., checkpoint] needs to occur at an expression that is evaluated in every execution of a program". Again, they do not provide any formal definition of alignment nor an automatic solution to enforce it.

Lundén et al. [24] briefly mention the general problem of selecting optimal resampling locations in PPLs for SMC but do not consider the alignment problem in particular. They also acknowledge the overhead resulting from not all SMC executions resampling the same number of times, which alignment avoids.

The PPLs Birch [31], Pyro [3], and WebPPL [14] support SMC inference. Birch and Pyro enforce alignment for SMC as part of model construction. Note that this is only true for SMC in Pyro—other Pyro inference algorithms use other modeling approaches. The approaches in Birch and Pyro are sound but demand more of their users compared to the alignment approach. WebPPL does not consider alignment and resamples at all likelihood updates for SMC.

Ritchie et al. [38] and Nori et al. [35] present MCMC algorithms for probabilistic programs. Ritchie et al. [38] optimize lightweight MCMC by Wingate et al. [47] through execution suspensions and callsite caching. The optimizations are independent of and potentially combines well with aligned lightweight MCMC. Another MCMC optimization which potentially combines well with alignment is due to Nori et al. [35]. They use static analysis to propagate observations backwards in programs to improve inference.

Information flow analyses [40] may determine if particular parts of a program execute as a result of different program inputs. Specifically, if program input is random, such approaches have clear similarities to the alignment analysis.

Many other PPLs exist, such as Gen [10], Venture [29], Edward [44], Stan [8], and AugurV2 [18]. Gen, Venture, and Edward focus on simplifying the joint specification of a model and its inference to give users low-level control, and do not consider automatic alignment specifically. However, the incremental inference approach [9] in Gen does use the addressing approach by Wingate et al. [47]. Stan and AugurV2 have less expressive modeling languages to allow more powerful inference. Alignment is by construction due to the reduced expressiveness.

Borgström et al. [6], Staton et al. [43], Ścibior et al. [41], and Vákár et al. [45] treat semantics and correctness for PPLs, but do not consider alignment.

9 Conclusion

This paper gives, for the first time, a formal definition of alignment in PPLs. Furthermore, we introduce a static analysis technique and use it to align checkpoints in PPLs and apply it to SMC and MCMC inference. We formalize the alignment analysis, prove its correctness, and implement it in Miking CorePPL. We also implement aligned SMC and aligned lightweight MCMC, and evaluate the implementations on non-trivial CRBD and ClaDS models from phylogenetics, the LDA topic model, and a state-space model, demonstrating significant improvements compared to standard SMC and lightweight MCMC.

Acknowledgments We thank Lawrence Murray, Johannes Borgström, and Jan Kudlicka for early discussions on the alignment idea, and Viktor Senderov for implementing ClaDS in Miking CorePPL. We also thank the anonymous reviewers at ESOP for their valuable comments.

References

1. Turing.jl. https://turing.ml/dev/ (2022), accessed: 2022-02-24
2. Miking DPPL. https://github.com/miking-lang/miking-dppl (2023), accessed: 2023-01-02
3. 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)
4. Bishop, C.M.: Pattern Recognition and Machine Learning (Information Science and Statistics). Springer-Verlag (2006)
5. Blei, D.M., Ng, A.Y., Jordan, M.I.: Latent Dirichlet allocation. Journal of Machine Learning Research **3**, 993–1022 (2003)
6. Borgström, J., Dal Lago, U., Gordon, A.D., Szymczak, M.: A lambda-calculus foundation for universal probabilistic programming. In: Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming. pp. 33–46. Association for Computing Machinery (2016)
7. 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. pp. 55–60. Association for Computing Machinery (2019)
8. 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)
9. Cusumano-Towner, M., Bichsel, B., Gehr, T., Vechev, M., Mansinghka, V.K.: Incremental inference for probabilistic programs. In: Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 571–585. Association for Computing Machinery, New York, NY, USA (2018)
10. 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. Association for Computing Machinery (2019)
11. 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. pp. 237–247. Association for Computing Machinery, New York, NY, USA (1993)
12. 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)
13. 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)
14. Goodman, N.D., Stuhlmüller, A.: The design and implementation of probabilistic programming languages. http://dippl.org (2014), accessed: 2022-02-24
15. Goodman, N.D., Tenenbaum, J.B., Contributors, T.P.: Probabilistic Models of Cognition. http://probmods.org/v2 (2016), accessed: 2022-06-10

16. Gothoskar, N., Cusumano-Towner, M., Zinberg, B., Ghavamizadeh, M., Pollok, F., Garrett, A., Tenenbaum, J., Gutfreund, D., Mansinghka, V.: 3DP3: 3D scene perception via probabilistic programming. In: Advances in Neural Information Processing Systems. vol. 34, pp. 9600–9612. Curran Associates, Inc. (2021)
17. Griffiths, T.L., Steyvers, M.: Finding scientific topics. Proceedings of the National academy of Sciences **101**(suppl_1), 5228–5235 (2004)
18. 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. Association for Computing Machinery, New York, NY, USA (2017)
19. 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 (2012)
20. Kahn, G.: Natural semantics. In: Proceedings of the 4th Annual Symposium on Theoretical Aspects of Computer Science. pp. 22–39. Springer-Verlag, Berlin, Heidelberg (1987)
21. Kiselyov, O.: Problems of the lightweight implementation of probabilistic programming. In: Proceedings of Workshop on Probabilistic Programming Semantics (2016)
22. Kozen, D.: Semantics of probabilistic programs. Journal of Computer and System Sciences **22**(3), 328–350 (1981)
23. Lew, A., Agrawal, M., Sontag, D., Mansinghka, V.: PClean: Bayesian data cleaning at scale with domain-specific probabilistic programming. In: Proceedings of The 24th International Conference on Artificial Intelligence and Statistics. vol. 130, pp. 1927–1935. PMLR (2021)
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., Öhman, J., Kudlicka, J., Senderov, V., Ronquist, F., Broman, D.: Compiling universal probabilistic programming languages with efficient parallel sequential monte carlo inference. In: Programming Languages and Systems. pp. 29–56. Springer International Publishing, Cham (2022)
26. Lundén, D., Caylak, G., Ronquist, F., Broman, D.: Artifact: Automatic alignment in higher-order probabilistic programming languages (Jan 2023). https://doi.org/10.5281/zenodo.7572555
27. Lundén, D., Caylak, G., Ronquist, F., Broman, D.: Automatic alignment in higher-order probabilistic programming languages. arXiv e-prints p. arXiv:2301.11664 (2023)
28. Maliet, O., Hartig, F., Morlon, H.: A model with many small shifts for estimating species-specific diversification rates. Nature Ecology & Evolution **3**(7), 1086–1092 (2019)
29. Mansinghka, V.K., Schaechtle, U., Handa, S., Radul, A., Chen, Y., Rinard, M.: Probabilistic programming with programmable inference. In: Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation. p. 603–616. Association for Computing Machinery, New York, NY, USA (2018)
30. Midtgaard, J.: Control-flow analysis of functional programs. ACM Computing Surveys **44**(3) (2012)
31. Murray, L.M., Schön, T.B.: Automated learning with a probabilistic programming language: Birch. Annual Reviews in Control **46**, 29–43 (2018)
32. Naesseth, C., Lindsten, F., Schön, T.: Elements of Sequential Monte Carlo. Foundations and Trends in Machine Learning Series, Now Publishers (2019)

33. Nee, S.: Birth-death models in macroevolution. Annual Review of Ecology, Evolution, and Systematics **37**(1), 1–17 (2006)
34. Nielson, F., Nielson, H.R., Hankin, C.: Principles of Program Analysis. Springer-Verlag (1999)
35. Nori, A., Hur, C.K., Rajamani, S., Samuel, S.: R2: An efficient MCMC sampler for probabilistic programs. Proceedings of the AAAI Conference on Artificial Intelligence **28**(1) (2014)
36. Paige, B., Wood, F.: A compilation target for probabilistic programming languages. In: Xing, E.P., Jebara, T. (eds.) Proceedings of the 31st International Conference on Machine Learning. vol. 32, pp. 1935–1943. PMLR, Bejing, China (22–24 Jun 2014)
37. Pierce, B.C.: Types and programming languages. MIT press (2002)
38. Ritchie, D., Stuhlmüller, A., Goodman, N.: C3: Lightweight incrementalized MCMC for probabilistic programs using continuations and callsite caching. In: Proceedings of the 19th International Conference on Artificial Intelligence and Statistics. vol. 51, pp. 28–37. PMLR, Cadiz, Spain (2016)
39. 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 (2021)
40. Sabelfeld, A., Myers, A.: Language-based information-flow security. IEEE Journal on Selected Areas in Communications **21**(1), 5–19 (2003)
41. Ścibior, A., Kammar, O., Vákár, M., Staton, S., Yang, H., Cai, Y., Ostermann, K., Moss, S.K., Heunen, C., Ghahramani, Z.: Denotational validation of higher-order Bayesian inference. Proceedings of the ACM on Programming Languages **2**(POPL) (2017)
42. Shivers, O.G.: Control-flow analysis of higher-order languages or taming lambda. Carnegie Mellon University (1991)
43. Staton, S., Yang, H., Wood, F., Heunen, C., Kammar, O.: Semantics for probabilistic programming: Higher-order functions, continuous distributions, and soft constraints. In: Proceedings of the 31st Annual ACM/IEEE Symposium on Logic in Computer Science. pp. 525–534. Association for Computing Machinery (2016)
44. Tran, D., Hoffman, M.D., Saurous, R.A., Brevdo, E., Murphy, K., Blei, D.M.: Deep probabilistic programming. In: International Conference on Learning Representations (2017)
45. Vákár, M., Kammar, O., Staton, S.: A domain theory for statistical probabilistic programming. Proceedings of the ACM on Programming Languages **3**(POPL) (2019)
46. van de Meent, J.W., Paige, B., Yang, H., Wood, F.: An introduction to probabilistic programming. arXiv e-prints p. arXiv:1809.10756 (2018)
47. Wingate, D., Stuhlmueller, A., Goodman, N.: Lightweight implementations of probabilistic programming languages via transformational compilation. In: Proceedings of the 14th International Conference on Artificial Intelligence and Statistics. vol. 15, pp. 770–778. PMLR (2011)
48. Wood, F., Meent, J.W., Mansinghka, V.: A new approach to probabilistic programming inference. In: Proceedings of the 17th International Conference on Artificial Intelligence and Statistics. vol. 33, pp. 1024–1032. PMLR (2014)