



# Suspension Analysis and Selective Continuation-Passing Style for Universal Probabilistic Programming Languages

Daniel Lundén[1](✉), Lars Hummelgren[2], Jan Kudlicka[3], Oscar Eriksson[2], and David Broman[2,4]

[1] Oracle, Stockholm, Sweden, daniel.lunden@oracle.com
[2] EECS and Digital Futures, KTH Royal Institute of Technology, Stockholm, Sweden, {larshum,oerikss,dbro}@kth.se
[3] Department of Data Science and Analytics, BI Norwegian Business School, Oslo, Norway, jan.kudlicka@bi.no
[4] Computer Science Department, Stanford University, California, USA broman@stanford.edu

**Abstract.** Universal probabilistic programming languages (PPLs) make it relatively easy to encode and automatically solve statistical inference problems. To solve inference problems, PPL implementations often apply Monte Carlo inference algorithms that rely on execution suspension. State-of-the-art solutions enable execution suspension either through (i) continuation-passing style (CPS) transformations or (ii) efficient, but comparatively complex, low-level solutions that are often not available in high-level languages. CPS transformations introduce overhead due to unnecessary closure allocations—a problem the PPL community has generally overlooked. To reduce overhead, we develop a new efficient selective CPS approach for PPLs. Specifically, we design a novel static suspension analysis technique that determines parts of programs that require suspension, given a particular inference algorithm. The analysis allows selectively CPS transforming the program only where necessary. We formally prove the correctness of the analysis and implement the analysis and transformation in the Miking CorePPL compiler. We evaluate the implementation for a large number of Monte Carlo inference algorithms on real-world models from phylogenetics, epidemiology, and topic modeling. The evaluation results demonstrate significant improvements across all models and inference algorithms.

**Keywords:** Probabilistic programming · Static analysis · Continuation-passing style.

## 1 Introduction

Probabilistic programming languages (PPLs), such as Anglican [50], Birch [36], WebPPL [18], Stan [10], Pyro [6], and Gen [11], make it possible to encode and solve statistical inference problems. Such inference problems are of significant interest in many research fields, including phylogenetics [43], computer vision [25],


S. Weirich (Ed.): ESOP 2024, LNCS 14577, pp. 302–330, 2024.
https://doi.org/10.1007/978-3-031-57267-8_12

topic modeling [7], inverse graphics [20], and cognitive science [19]. A particularly appealing feature of PPLs is the separation between the inference problem specification (the language) and the inference algorithm used to solve the problem (the language implementation). This separation allows PPL users to focus solely on encoding their inference problems while inference algorithm experts deal with the intricacies of inference implementation.

Implementations of PPLs apply many different inference algorithms. Monte Carlo inference algorithms—such as Markov chain Monte Carlo (MCMC) [16] and sequential Monte Carlo (SMC) [12]—are popular due to their asymptotic correctness and relative ease of implementation for *universal*[5] PPLs. The central idea behind all Monte Carlo methods in PPLs is to execute probabilistic programs multiple times to generate *samples* that approximate the target distribution for the encoded inference problem. However, repeated execution is expensive, and PPL implementations must avoid unnecessary overhead.

Monte Carlo algorithms often need to *suspend* executions. For example, MCMC algorithms can suspend at *random draws* in the program to avoid unnecessary re-execution when proposing new executions, and SMC algorithms can suspend at *likelihood updates* to *resample* executions. Languages such as WebPPL [18] and Anglican [50], and the approach described by Ritchie et al. [41], apply *continuation-passing style (CPS)* transformations [3] to enable arbitrary suspension during execution. The main benefit of CPS transformations is that they are relatively easy to implement in functional programming languages. However, one disadvantage with CPS transformations is that high-performance low-level languages, without higher-order functions, do not support them. For this reason, there are also more direct low-level alternatives to CPS, including non-preemptive multitasking (e.g., coroutines [15]) and PPL control-flow graphs [30]. These more direct alternatives can additionally avoid much of the overhead resulting from CPS[6], but are more complex to implement.

We consider how to bridge the performance gap between CPS-based PPLs and lower-level PPLs that rely on, e.g., direct implementation of coroutines. We consider optimizations at the CPS transformation level, and not the translation from CPS-based PPLs to lower-level representations. CPS overhead is a result of closure allocations for continuations. We make the important observation that PPLs do not require the arbitrary suspensions provided by full CPS transformations. Most Monte Carlo inference algorithms require suspension only in very specific parts of programs. Current state-of-the-art CPS-based PPLs do not consider inference-specific suspension requirements to reduce CPS overhead.

We design a new static suspension analysis and a new selective CPS transformation for PPLs that together significantly reduce runtime overhead com-

[5] A term that first appeared in Goodman et al. [17], indicating expressive PPLs where the number and types of random variables are not always known statically.

[6] Note that CPS only results in overhead if programs reify the continuations at runtime to, e.g., suspend computations. Traditional CPS-based compilers often only use CPS as an intermediate form during compilation, which does not result in runtime overhead.

pared to a traditional full CPS transformation. Current state-of-the-art functional PPLs that use CPS for execution suspension can therefore greatly benefit from our new approach. The suspension analysis identifies all parts of programs that may require suspension as a result of applying a particular inference algorithm. We formalize the suspension analysis algorithm using a core PPL calculus equipped with a big-step operational semantics. Specifically, the challenge lies in capturing how suspension requirements propagate through the program in the presence of higher-order functions. Furthermore, we formalize the selective CPS transformation and justify its correctness when guided by the suspension analysis. Prior work on selective CPS for general-purpose programming languages, e.g., by Nielsen [38] and Asai and Uehara [4], focuses on analyses based on type systems and type inference. In contrast, we instead build our suspension analysis using 0-CFA [46] and it operates directly on an untyped calculus.

Overall, we (i) prove that the suspension analysis is correct, (ii) show that the resulting selective CPS transformation gives significant performance gains compared to using a full CPS transformation, and (iii) show that the overall approach is directly applicable to a large set of inference algorithms. Specifically, we evaluate the approach for the following inference algorithms: likelihood weighting, the SMC bootstrap particle filter, the SMC alive particle filter [24], aligned lightweight MCMC [29,49], and particle-independent Metropolis–Hastings [40]. We consider each inference algorithm for four real-world models from phylogenetics, epidemiology, and topic modeling.

We implement the suspension analysis and selective CPS transformation in Miking CorePPL [30,9]. Similarly to WebPPL and Anglican, the implementation supports the co-existence of many inference problems and applications of inference algorithms to these problems within the same program. However, compared to full CPS, such programs are more challenging to handle with selective CPS, as the CPS transformation of an inference problem also depends on the applied inference algorithm—different inference algorithms generally require different suspensions. To complicate things further, different inference problems may share some code, or the PPL user may apply two different inference algorithms to the same inference problem. The compiler must then apply different CPS transformations to different parts of the program, and sometimes even many different CPS transformations to separate copies of the *same* part of the program. To solve this, we develop an approach that, for any given Miking CorePPL program, *extracts* all possible inference problems and corresponding inference algorithm applications. This extraction procedure allows the correct application of selective CPS throughout the program.

In summary, we make the following contributions.

- We design, formalize, and prove the correctness of a suspension analysis for PPLs, where the suspension requirements come from a given inference algorithm (Section 4).
- We design and formalize a new selective CPS transformation for PPLs. Compared to full CPS, selectively CPS transforming PPL programs guided by

the suspension analysis significantly reduces runtime overhead resulting from unnecessary closure allocations (Section 5).
- We implement the suspension analysis and selective CPS transformation in the Miking CorePPL compiler. Unlike full CPS, selective CPS introduces challenges for probabilistic programs containing many inference problems and inference algorithm applications. We implement an approach that correctly applies selective CPS to such programs by extracting individual inference problems (Section 6).

Section 7 presents the evaluation and its results for the implementations in Miking CorePPL, Section 8 discusses related work in more detail, and Section 9 concludes. We first consider a motivating example in Section 2 and introduce the underlying PPL calculus in Section 3.

An extended version of the paper is available at arXiv [31]. 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

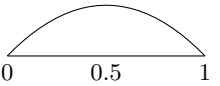
This section introduces the running example in Fig. 1 and uses it to present the basic idea behind PPLs and how inference algorithms such as SMC and MCMC make use of CPS to suspend executions. Most importantly, we illustrate the motivation and key ideas behind selective CPS for PPLs.

Consider the probabilistic program in Fig. 1a, written in a functional-style PPL. The program encodes an inference problem for estimating the probability distribution over the bias of a coin, *conditioned* on the outcome of four experimental coin flips: true, true, false, and true (true = heads and false = tails). At line 1, we use the PPL-specific `assume` construct to define our *prior* belief in the bias $a_1$ of the coin. We set this prior belief to a Beta$(2, 2)$ probability distribution, illustrated in Fig. 1b. In the illustration, 0 indicates a coin that always results in false, 1 a coin that always results in true, and 0.5 a fair coin. We see that our prior belief is quite evenly spread out, but with more probability mass towards a fair coin. To condition this prior distribution on the observed coin flips, we conceptually execute the program in Fig 1a infinitely many times, *sampling* values from the prior Beta distribution at `assume` (line 1) and, as a side effect, *accumulating the product of weights* given as argument to the PPL-specific `weight` construct (line 4). We make the four consecutive calls `weight` ($f_{\text{Bernoulli}}$ $a_1$ true), `weight` ($f_{\text{Bernoulli}}$ $a_1$ true), `weight` ($f_{\text{Bernoulli}}$ $a_1$ false), and `weight` ($f_{\text{Bernoulli}}$ $a_1$ true)[7], using the recursive function *iter*. The function application $f_{\text{Bernoulli}}$ $a_1$ $o$ gives the probability of the outcome $o$ given a bias $a_1$ for the coin. I.e., $f_{\text{Bernoulli}}$ $a_1$ true $= a_1$ and $f_{\text{Bernoulli}}$ $a_1$ false $= 1 - a_1$. So, for example, a sample $a_1 = 0.4$ gets the accumulated weight $0.4 \cdot 0.4 \cdot 0.6 \cdot 0.4$

[7] PPLs also commonly use a similar built-in function `observe` to update the weight. For example, `observe` (Bernoulli $a_1$) true is equivalent to `weight` ($f_{\text{Bernoulli}}$ $a_1$ true).

```
1 let a1 = assume (Beta 2 2) in
2 let rec iter = λobs.
3   if null obs then () else
4     weight (f_Bernoulli a1 (head obs));
5     iter (tail obs)
6 in
7 iter [true,true,false,true];
8 a1
```

(a) Program $\mathbf{t}_{\text{example}}$.




(b) Beta(2,2).


(c) Distribution of $\mathbf{t}_{\text{example}}$.

```
1 Suspension_assume(Beta 2 2, λa1.
2   let rec iter = λobs.
3     if null obs then () else
4       weight (f_Bernoulli(a1)
5                 (head obs));
6       iter (tail obs)
7   in
8   iter [true,true,false,true];
9   a1)
```

(d) Suspension at `assume`.

```
1 let a1 = assume (Beta 2 2) in
2 let rec iter = λk. λobs.
3   if null obs then k ()
4   else
5     Suspension_weight(
6       f_Bernoulli(a1) (head obs),
7       (λ_. iter k (tail obs)))
8 in
9 iter (λ_. a1)
10   [true,true,false,true];
```

(e) Suspension at `weight`.

```
1 let k7 = λt6.
2  let k8 = λt7.
3   Suspension_assume(t7, λa1.
4    let rec iter = λk1. λobs.
5     let k2 = λt1.
6      if t1 then k1 () else
7       let k3 = λt2.
8        let k4 = λt3.
9         let k5 = λt4.
10          Suspension_weight(t4, λ_.
11           let k6 = λt5. iter k1 t5 in
12           tail_CPS k6 obs)
13         in t2 k5 t3
14        in head_CPS k4 obs
15       in f_Bernoulli_CPS k3 a1
16     in null_CPS k2 obs
17    in iter (λ_. a1)
18         [true,true,false,true])
19  in t6 k8 2
20 in Beta_CPS k7 2
```

(f) Full CPS.

Fig. 1: A probabilistic program $\mathbf{t}_{\text{example}}$ modeling the bias of a coin. Fig. (a) gives the program. The function $f_{\text{Bernoulli}}$ is the probability mass function of the Bernoulli distribution. Fig. (b) illustrates the distribution for $a_1$ at line 1 in (a). Fig. (c) shows the set of (weighted) samples resulting from conceptually running $\mathbf{t}_{\text{example}}$ infinitely many times. Fig. (d) and Fig. (e) show the selective CPS transformations required for suspension at `assume` and `weight`, respectively. Fig. (f) gives $\mathbf{t}_{\text{example}}$ in full CPS, with suspensions at `assume` and `weight`. The CPS subscript indicates CPS-versions of intrinsic functions such as *head* and *tail*.

and $a_1 = 0.7$ the accumulated weight $0.7 \cdot 0.7 \cdot 0.3 \cdot 0.7$. The end result is an infinite set of *weighted* samples of $a_1$ (the program returns $a_1$ at line 8) that approximate the *posterior* or *target* distribution of Fig. 1a, illustrated in Fig 1c. Note that, because we observed three true outcomes and only one false, the weights shift the probability mass towards 1 and narrows it slightly as we are now more sure about the bias of the coin. Increasing the number of experimental coin flips would make Fig. 1c more and more narrow.

We can approximate the infinite number of samples by running the program a large (but finite) number of times. This basic inference algorithm is known as *likelihood weighting*. The problem with likelihood weighting is that it is only accurate enough for simple models. For complex models, it is common that only a few likelihood weighting samples (often only one) get much larger weights relative to the other samples, greatly reducing inference accuracy. Real-world models require more powerful inference algorithms based on, e.g., SMC or MCMC. A key requirement in both SMC and MCMC is the ability to *suspend* executions of probabilistic programs at calls to `weight` and/or `assume`. One way to enable suspensions is by writing programs in CPS. We first illustrate a simple use of CPS to suspend at `assume` in Fig. 1d. Here, the program immediately returns an object $\text{Suspension}_{\texttt{assume}}(\text{Beta } 2\ 2, k)$, indicating that execution stopped at an `assume` with the argument Beta 2 2 and a continuation $k$ (i.e., the abstraction binding $a_1$) that executes the remainder of the program. With likelihood weighting, we would simply sample a value $a_1$ from the Beta 2 2 distribution and resume execution by calling $k\ a_1$. This call then runs the program until termination and results in the actual return value of the program, which is $a_1$. Many MCMC inference algorithms reuse samples from previous executions at $\text{Suspension}_{\texttt{assume}}$, and the suspensions are thus useful to avoid unnecessary re-execution [41].

As a second example, we illustrate suspension at `weight` for, e.g., SMC inference in Fig. 1e. Here, we require suspensions in the middle of the recursive call to *iter*, and writing the program in CPS is more challenging. We rewrite the *iter* function to take a continuation $k$ as argument, and call the continuation with the return value () at line 3 instead of directly returning () as in Fig. 1a at line 3. This continuation argument $k$ is precisely what allows us to construct and return $\text{Suspension}_{\texttt{weight}}$ objects at line 5. To illustrate the suspensions, consider executing the program with likelihood weighting. First, the program returns the object $\text{Suspension}_{\texttt{weight}}(f_{\text{Bernoulli}(a_1)}\ \text{true}, k')$, where $k'$ is the continuation that line 7 constructs. Likelihood weighting now updates the weight for the execution with the value $f_{\text{Bernoulli}(a_1)}\ \text{true}$ and resumes execution by calling $k'$ (). Similarly, this next execution returns $\text{Suspension}_{\texttt{weight}}(f_{\text{Bernoulli}(a_1)}\ \text{true}, k'')$ for the second recursive call to *iter*, and we again update the weight and resume by calling $k''$ (). We similarly encounter $\text{Suspension}_{\texttt{weight}}(f_{\text{Bernoulli}(a_1)}\ \text{false}, k''')$ and $\text{Suspension}_{\texttt{weight}}(f_{\text{Bernoulli}(a_1)}\ \text{true}, k'''')$ before the final call $k''''$ () runs the program to termination and produces the actual return value $a_1$. In SMC, we run many executions concurrently and wait until they all have returned a $\text{Suspension}_{\texttt{weight}}$ object. At this point, we resample the executions according to their weights (the first value in $\text{Suspension}_{\texttt{weight}}$), which discards executions

with low weight and replicates executions with high weight. After resampling, we continue to the next suspension and resampling by calling the continuations.

PPL implementations enable suspensions at `assume` and/or `weight` through automatic and full CPS transformations. Fig. 1f illustrates such a transformation for Fig. 1a. We indicate CPS versions of intrinsic functions with the $_{\text{CPS}}$ subscript. Note that the full CPS transformation results in many additional closure allocations compared to Fig. 1d and Fig. 1e. As a result, runtime overhead increases significantly. The contribution in this paper is a static analysis that allows an automatic and selective CPS transformation of programs, as in Fig. 1d and Fig. 1e. With a selective transformation, we avoid many unnecessary closure allocations, and can significantly reduce runtime overhead while still allowing suspensions as required for a given inference algorithm.

## 3 Syntax and Semantics

This section introduces the PPL calculus used to formalize the suspension analysis in Section 4 and selective CPS transformation in Section 5. Section 3.1 gives the abstract syntax and Section 3.2 a big-step operational semantics. Section 3.3 introduces A-normal form—a prerequisite for both the suspension analysis and the selective CPS transformation.

### 3.1 Syntax

We build upon the standard untyped lambda calculus, representative of functional universal PPLs such as Anglican, WebPPL, and Miking CorePPL. We define the abstract syntax below.

**Definition 1 (Terms, values, and environments).** *We define terms* $\mathbf{t} \in T$ *and values* $\mathbf{v} \in V$ *as*

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

*The countable set* $X$ *contains variable names,* $C$ *intrinsic values and operations, and* $D \subset C$ *intrinsic probability distributions. The set* $P$ *contains* evaluation environments*, i.e., maps from variables in* $X$ *to values in* $V$*.*

**Definition 2 (Target language terms).** *As a target language for the selective CPS transformation in Section 5, we additionally extend Definition 1 to target language terms* $\mathbf{t} \in T^+$ *by*

$$\mathbf{t} \mathrel{+}= \text{Suspension}_{\texttt{assume}}(\mathbf{t}, \mathbf{t}) \mid \text{Suspension}_{\texttt{weight}}(\mathbf{t}, \mathbf{t}). \tag{2}$$

Fig. 1a gives an example of a term in $T$, and Fig. 1d and Fig. 1e of terms in $T^+$. However, note that the programs in Fig. 1 also use the list constructor `[...]` (not part of the above definitions) to make the example more interesting.

In addition to the standard variable, abstraction, and application terms in the untyped lambda calculus, we include explicit `let` expressions for convenience. Furthermore, we use the syntactic sugar `let rec` $f$ `=` $\lambda x.\mathbf{t}_1$ `in` $\mathbf{t}_2$ to define recursive functions (translating to an application of a call-by-value fixed-point combinator). We use $\mathbf{t}_1$ `;` $\mathbf{t}_2$ as a shorthand for `(`$\lambda$`_.`$\mathbf{t}_2$`)` $\mathbf{t}_1$, where `_` means that we do not use the argument. That is, we evaluate $\mathbf{t}_1$ for side effects only.

We include a set $C$ of intrinsic operations and constants essential to inference problems encoded in PPLs. The set of intrinsics includes boolean truth values, the unit value, real numbers, and probability distributions. We can also add further operations and constants to $C$. For example, we can let $+ \in C$ to support addition of real numbers. To allow control flow to depend on intrinsic values, we include `if` expressions that use intrinsic booleans as condition.

We saw examples of the `assume` and `weight` constructs in Section 2. The `assume` construct takes distributions $D \subset C$ as argument, and produces random variables distributed according to these distributions. For example, we can let $\mathcal{N} \in C$ be a function that constructs normal distributions. Then, `assume (`$\mathcal{N}$ `0 1)`, where $\mathcal{N}\ 0\ 1 \in D$, defines a random variable with a standard normal distribution. Partially constructed distributions, e.g., $\mathcal{N}\ 0$, are also in $C$, but not in $D$ (they are not yet proper distributions). As we saw in Section 2, the `weight` construct updates the likelihood with the real number given as argument, and allows conditioning on data (e.g., the four coin flips in Fig. 1).

### 3.2 Semantics

We construct a call-by-value big-step operational semantics, based on Lundén et al. [29], describing how to evaluate terms $\mathbf{t} \in T$. Such a semantics is a key component when formally defining the probability distributions corresponding to terms $\mathbf{t} \in T$ (e.g., the distribution in Fig. 1c corresponding to the program in Fig. 1a) and also when proving various properties of PPLs and their inference algorithms (e.g., inference correctness). See, e.g., the work by Borgström et al. [8] and Lundén et al. [28] for full formal treatments.

We use the semantics to formally define suspension, and use this definition to state the soundness of the suspension analysis in Section 4 (Theorem 1). We use a big-step semantics, as we do not require the additional control provided by a small-step semantics. For example, we do not concern ourselves with details of termination, as the soundness of the analysis relates only to terminating executions. Fig. 2 presents the full semantics as a relation $\rho \vdash \mathbf{t}\ {}^{s}\Downarrow_{u}^{w}\ \mathbf{v}$ over tuples $(P, T, S, \{\text{false}, \text{true}\}, \mathbb{R}, V)$. $S$ is a set of *traces* capturing the random draws at `assume` during evaluation. Intuitively, $\rho \vdash \mathbf{t}\ {}^{s}\Downarrow_{u}^{w}\ \mathbf{v}$ holds iff $\mathbf{t}$ evaluates to $\mathbf{v}$ in the environment $\rho$ with the trace $s$ and the total probability density (i.e., the accumulated weight) $w$. We describe the suspension flag $u$ later in this section.

Most of the rules are standard and we focus on explaining key properties related to PPLs and suspension. We first consider the rule (Const-App), which uses the $\delta$-function to evaluate intrinsic operations.

$$\frac{\rho \vdash \mathbf{t}_1 \ {}^{s_1}\Downarrow^{w_1}_{u_1} \ \langle \lambda x.\mathbf{t}, \rho' \rangle \quad \rho \vdash \mathbf{t}_2 \ {}^{s_2}\Downarrow^{w_2}_{u_2} \ \mathbf{v}_2 \quad \rho', x \mapsto \mathbf{v}_2 \vdash \mathbf{t} \ {}^{s_3}\Downarrow^{w_3}_{u_3} \ \mathbf{v}}{\rho \vdash \mathbf{t}_1 \ \mathbf{t}_2 \ {}^{s_1 \| s_2 \| s_3}\Downarrow^{w_1 \cdot w_2 \cdot w_3}_{u_1 \vee u_2 \vee u_3} \ \mathbf{v}} (\textsc{App})$$

$$\frac{}{\rho \vdash x \ {}^{[]}\Downarrow^{1}_{\text{false}} \ \rho(x)} (\textsc{Var}) \qquad \frac{\rho \vdash \mathbf{t}_1 \ {}^{s_1}\Downarrow^{w_1}_{u_1} \ c_1 \quad \rho \vdash \mathbf{t}_2 \ {}^{s_2}\Downarrow^{w_2}_{u_2} \ c_2}{\rho \vdash \mathbf{t}_1 \ \mathbf{t}_2 \ {}^{s_1 \| s_2}\Downarrow^{w_1 \cdot w_2}_{u_1 \vee u_2} \ \delta(c_1, c_2)} (\textsc{Const-App})$$

$$\frac{}{\rho \vdash \lambda x.\mathbf{t} \ {}^{[]}\Downarrow^{1}_{\text{false}} \ \langle \lambda x.\mathbf{t}, \rho \rangle} (\textsc{Lam}) \qquad \frac{\rho \vdash \mathbf{t}_1 \ {}^{s_1}\Downarrow^{w_1}_{u_1} \ \mathbf{v}_1 \quad \rho, x \mapsto \mathbf{v}_1 \vdash \mathbf{t}_2 \ {}^{s_2}\Downarrow^{w_2}_{u_2} \ \mathbf{v}}{\rho \vdash \texttt{let } x = \mathbf{t}_1 \texttt{ in } \mathbf{t}_2 \ {}^{s_1 \| s_2}\Downarrow^{w_1 \cdot w_2}_{u_1 \vee u_2} \ \mathbf{v}} (\textsc{Let})$$

$$\frac{}{\rho \vdash c \ {}^{[]}\Downarrow^{1}_{\text{false}} \ c} (\textsc{Const}) \qquad \frac{\rho \vdash \mathbf{t}_1 \ {}^{s_1}\Downarrow^{w_1}_{u_1} \ \text{true} \quad \rho \vdash \mathbf{t}_2 \ {}^{s_2}\Downarrow^{w_2}_{u_2} \ \mathbf{v}_2}{\rho \vdash \texttt{if } \mathbf{t}_1 \texttt{ then } \mathbf{t}_2 \texttt{ else } \mathbf{t}_3 \ {}^{s_1 \| s_2}\Downarrow^{w_1 \cdot w_2}_{u_1 \vee u_2} \ \mathbf{v}_2} (\textsc{If-True})$$

$$\frac{\rho \vdash \mathbf{t} \ {}^{s}\Downarrow^{w}_{u} \ d \quad w' = f_d(c)}{\rho \vdash \texttt{assume } \mathbf{t} \ {}^{s \| [c]}\Downarrow^{w \cdot w'}_{suspend_{\texttt{assume}} \vee u} \ c} (\textsc{Assume}) \qquad \frac{\rho \vdash \mathbf{t} \ {}^{s}\Downarrow^{w}_{u} \ w'}{\rho \vdash \texttt{weight } \mathbf{t} \ {}^{s}\Downarrow^{w \cdot w'}_{suspend_{\texttt{weight}} \vee u} \ ()} (\textsc{Weight})$$

Fig. 2: A big-step operational semantics for $\mathbf{t} \in T$. We omit the rule (If-False) for brevity; it is analogous to (If-True). The environment $\rho, x \mapsto \mathbf{v}$ denotes $\rho$ extended with a binding $\mathbf{v}$ for $x$. For each $d \in D$, the function $f_d$ is its probability density or probability mass function. E.g., $f_{\mathcal{N}(0,1)}(x) = e^{x^2/2}/\sqrt{2\pi}$, the density function of the standard normal distribution. We use the following notation: $\|$ for sequence concatenation, $\cdot$ for multiplication, and $\vee$ for logical disjunction.

**Definition 3 (Intrinsic arities and the $\delta$-function).** *For each $c \in C$, we let $|c| \in \mathbb{N}$ denote its* arity*. We also assume the existence of a partial function $\delta : C \times C \to C$ such that if $\delta(c, c_1) = c_2$, then $|c| > 0$ and $|c_2| = |c| - 1$.*

For example, $\delta((\delta(+, 1)), 2) = 3$. We use the arity property of intrinsics to formally define traces.

**Definition 4 (Traces).** *For all $s \in S$, $s$ is a sequence of intrinsics with arity 0, called a* trace*. We write $s = [c_1, c_2, \ldots, c_n]$ to denote a trace $s$ with $n$ elements.*

The rule (Assume) formalizes random draws and consumes elements of the trace. Specifically, (Assume) updates the evaluation's total probability density $w \in \mathbb{R}$ with the density $w'$ of the first trace element with respect to the distribution given as argument to `assume`. The rule (Weight) furthermore directly modifies the total probability density according to the `weight` argument.

We now consider the special suspension flag $u$ in the derivation $\rho \vdash \mathbf{t} \ {}^{s}\Downarrow^{w}_{u} \ \mathbf{v}$.

**Definition 5 (Suspension requirement).** *A derivation $\rho \vdash \mathbf{t} \ {}^{s}\Downarrow^{w}_{u} \ \mathbf{v}$* requires suspension *if the suspension flag $u$ is true.*

For example, the rule (App) requires suspension if $u_1 \vee u_2 \vee u_3$—i.e., if any subderivation requires suspension. To reflect the particular suspension requirements in SMC and MCMC inference, we limit the source of suspension requirements to `assume` and `weight`. We turn the individual sources on and off through the

```
let t1 = 2 in
let t2 = 2 in
let t3 = Beta in
let t4 = t3 t1 in
let t5 = t4 t2 in
let a1 = assume t5 in
let rec iter = λobs.
  let t6 = null in
  let t7 = t6 obs in
  let t8 =
    if t7 then
      let t9 = () in
      t9
    else
      let t10 = f_Bernoulli in
      let t11 = t10 a1 in
      let t12 = head in
      let t13 = t12 obs in
      let t14 = t11 t13 in
      let w1 = weight t14 in
      let t15 = tail in
      let t16 = t15 obs in
      let t17 = iter t16 in
      t17
  in
  t8
in
let t18 = true in
let t19 = false in
let t20 = true in
let t21 = true in
let t22 = [t21,t20,t19,t18] in
let t23 = iter t22 in
a1
```

Fig. 3: The running example $\mathbf{t}_{\text{example}}$ from Fig. 1a transformed to ANF.

boolean variables $\mathit{suspend}_{\texttt{assume}}$ and $\mathit{suspend}_{\texttt{weight}}$ in Fig. 2. For the examples in the remainder of this paper, we let $\mathit{suspend}_{\texttt{weight}} = \text{true}$ and $\mathit{suspend}_{\texttt{assume}} = \text{false}$ (i.e., only `weight` requires suspension, as in SMC inference).

To illustrate the semantics, consider $\mathbf{t}_{\text{example}}$ of Fig. 1a again. Because $\mathbf{t}_{\text{example}}$ evaluates precisely one `assume`, the only valid traces for $\mathbf{t}_{\text{example}}$ are singleton traces $[a_1]$, where $a_1 \in \mathbb{R}_{[0,1]}$ due to the Beta prior for $a_1$. By initially setting $\rho$ to the empty environment $\varnothing$ and following the rules of Fig. 2, we derive $\varnothing \vdash \mathbf{t}_{\text{example}} \overset{[a_1]}{\Downarrow}_{\text{true}}^{f_{\text{Beta}(2,2)}(a_1)\cdot a_1^3(1-a_1)} a_1$. Note that every evaluation of $\mathbf{t}_{\text{example}}$ has $u = \text{true}$, as there are always four calls to `weight` during evaluation. That is, the derivation requires suspension. However, many subderivations of $\mathbf{t}_{\text{example}}$ do *not* require suspension. For example, the subderivations `assume (Beta 2 2)` and *null obs* do not (i.e., have $u = \text{false}$). Section 4 presents a suspension analysis that conservatively approximates which subderivations require suspension. The analysis enables, e.g., the selective CPS transformation in Fig. 1e.

### 3.3 A-Normal Form

We simplify the suspension analysis in Section 4 and the selective CPS transformation in Section 5 by requiring that terms are in *A-normal form* (ANF) [13].

**Definition 6 (A-normal form).** *We define the A-normal form terms* $\mathbf{t}_{ANF} \in T_{ANF}$ *as follows.*

$$
\begin{aligned}
\mathbf{t}_{\text{ANF}} &::= x \mid \texttt{let}\ x = \mathbf{t}'_{\text{ANF}}\ \texttt{in}\ \mathbf{t}_{\text{ANF}} \\
\mathbf{t}'_{\text{ANF}} &::= x \mid c \mid \lambda x.\ \mathbf{t}_{\text{ANF}} \mid x\ y \\
&\quad \mid \texttt{if}\ x\ \texttt{then}\ \mathbf{t}_{\text{ANF}}\ \texttt{else}\ \mathbf{t}_{\text{ANF}} \mid \texttt{assume}\ x \mid \texttt{weight}\ x
\end{aligned}
\tag{3}
$$

It holds that $T_{\text{ANF}} \subset T$. Furthermore, there exist standard transformations to convert terms in $T$ to $T_{\text{ANF}}$. Fig. 3 illustrates Fig. 1a transformed to ANF. We will use Fig. 1a as a running example in Section 4 and Section 5.

Restricting programs to ANF significantly simplifies the suspension analysis and selective CPS transformation. From now on we require that all variable bindings in programs are unique, and together with ANF, the result is that every expression in a program $\mathbf{t} \in T_{\text{ANF}}$ is *uniquely labeled* by a variable name from a `let` expression. This property is essential for the treatment in Section 4.

## 4 Suspension Analysis

This section presents the main technical contribution: the suspension analysis. The analysis goal is to identify program expressions that may require suspension in the sense of Definition 5. Identifying such expressions leads to the selective CPS transformation in Section 5, enabling transformations such as in Fig 1e.

The suspension analysis builds upon the 0-CFA algorithm [46,39], and we formalize our algorithms based on Lundén et al. [29]. The main challenge we solve is how to model the propagation of suspension in the presence of higher-order functions. The 0 in 0-CFA stands for *context insensitivity*—the analysis considers every part of the program in one global context. Context insensitivity makes the analysis more conservative compared to context-sensitive approaches such as $k$-CFA, where $k \in \mathbb{N}$ indicates the level of context sensitivity [33]. We use 0-CFA for two reasons: (i) the worst-case time complexity for the analysis is polynomial, while it is exponential for $k$-CFA already at $k = 1$, and (ii) the limitations of 0-CFA rarely matter in practical PPL applications. For example, $k$-CFA provides no benefits over 0-CFA for the programs in Section 7.

We assume $\langle \lambda x.\ \mathbf{t}, \rho \rangle \notin C$ (recall that $C$ is the set of intrinsics). That is, we assume that closures are not part of the intrinsics. In particular, this disallows intrinsic operations (including the use of `assume` $d$, $d \in D \subset C$) to produce closures, which would needlessly complicate the analysis without any benefit.

Consider the program in Fig. 3, and assume that `weight` requires suspension. Clearly, the expression labeled by $w_1$ at line 20 then requires suspension. Furthermore, $w_1$ evaluates as part of the larger expression labeled by $t_8$ at line 10. Consequently, the evaluation of $t_8$ also requires suspension. Also, $t_8$ evaluates as part of an application of the abstraction binding *obs* at line 7. In particular, the abstraction binding *obs* binds to *iter*, and we apply *iter* at lines 23 and 33. Thus, the expressions named by $t_{17}$ and $t_{22}$ require suspension. In summary, we have that $w_1$, $t_8$, $t_{17}$, and $t_{22}$ require suspension, and we also note that all applications of the abstraction binding *obs* require suspension.

We proceed to the formalization and first introduce standard *abstract values*.

**Definition 7 (Abstract values).** *We define the abstract values* $\mathbf{a} \in A$ *as* $\mathbf{a} ::= \lambda x.y \mid \texttt{const}_x\, n$ *for* $x, y \in X$ *and* $n \in \mathbb{N}$.

The abstract value $\lambda x.y$ represents all closures originating at, e.g., a term $\lambda x.$ `let` $y$ `=` $1$ `in` $y$ in a program at runtime (recall that we assume that the variables $x$ and $y$ are unique). Note that the $y$ indicates the name returned by the body (formalized by the function NAME in Algorithm 1). The abstract value

**Algorithm 1** Constraint generation for the suspension analysis. We write the functional-style pseudocode for the algorithm itself in sans serif font to distinguish it from terms in $T$.

**function** GENERATECONSTRAINTS($\mathbf{t}$): $T_{\text{ANF}} \to \mathcal{P}(R)$ =

```
1  match t with
2  | x → ∅
3  | let x = t1 in t2 →
4    generateConstraints(t2) ∪
5      match t1 with
6      | y → {S_y ⊆ S_x}
7      | c → if |c| > 0 then {const_x |c| ∈ S_x}
8            else ∅
9      | λy. t_b → generateConstraints(t_b)
10       ∪ {λy. name t_b ∈ S_x}
11       ∪ {suspend_n ⇒ suspend_y
12            | n ∈ suspendNames(t_b)}
13     | lhs rhs → {
14         ∀z∀y λz.y ∈ S_lhs
15           ⇒ (S_rhs ⊆ S_z) ∧ (S_y ⊆ S_x),
16         ∀y∀n const_y n ∈ S_lhs ∧ n > 1
17           ⇒ const_y n − 1 ∈ S_x,
18         ∀y λy._ ∈ S_lhs
19           ⇒ (suspend_y ⇒ suspend_x),
20         ∀y const_y _ ∈ S_lhs
21           ⇒ (suspend_y ⇒ suspend_x),
22         suspend_x ⇒
23           (∀y λy._ ∈ S_lhs ⇒ suspend_y)
24           ∧ (∀y const_y _ ∈ S_lhs ⇒ suspend_y)
25       }
26     | assume _ →
27       if suspend_assume then {suspend_x} else ∅
28
29     | weight _ →
30       if suspend_weight then {suspend_x} else ∅
31     | if y then t_t else t_e →
32       generateConstraints(t_t)
33         ∪ generateConstraints(t_e)
34         ∪ {S_name t_t ⊆ S_x, S_name t_e ⊆ S_x}
35         ∪ {suspend_n ⇒ suspend_x
36              | n ∈ suspendNames(t_t)
37                    ∪ suspendNames(t_e)}
38
39 function name(t): T_ANF → X =
40   match t with
41   | x → x
42   | let x = t1 in t2 → name(t2)
43
44 function suspendNames(t): T_ANF → P(X) =
45   match t with
46   | x → ∅
47   | let x = t1 in t2 →
48     suspendNames(t2) ∪
49       match t1 with
50       | lhs rhs → {x}
51       | if y then t_t else t_e → {x}
52       | assume _ →
53         if suspend_assume then {x} else ∅
54       | weight _ →
55         if suspend_weight then {x} else ∅
56       | _ → ∅
```

$\texttt{const}_x\ n$ represents all intrinsic functions of arity $n$ originating at $x$. For example, $\texttt{const}_x$ 2 originates at, e.g., a term `let x = + in t`.

The central objects in the analysis are sets $S_x \in \mathcal{P}(A)$ and boolean values $\mathit{suspend}_x$ for all $x \in X$. The set $S_x$ contains all abstract values that may flow to the expression labeled by $x$, and $\mathit{suspend}_x$ indicates whether or not the expression requires suspension. A trivial but useless solution is $S_x = A$ and $\mathit{suspend}_x = \text{true}$ for all variables $x$ in the program. To get more precise information regarding suspension, we wish to find smaller solutions to the $S_x$ and $\mathit{suspend}_x$.

To formalize the set of sound solutions for $S_x$ and $\mathit{suspend}_x$, we generate *constraints* $\mathbf{c} \in R$ for programs.[†] Algorithm 1 formalizes the necessary constraints for programs $\mathbf{t} \in T_{\text{ANF}}$ with a function GENERATECONSTRAINTS that recursively traverses the program $\mathbf{t}$ to generate a set of constraints. Due to ANF, there are only two cases in the top match (line 1). Variables generate no constraints, and the important case is for `let` expressions at lines 3–30. The algorithm makes use of an auxiliary function NAME (line 39) that determines the name of an ANF expression, and a function SUSPENDNAMES (line 44) that determines the names of all top-level expressions within an expression that may suspend (namely, applications, `if` expressions, and `assume` and/or `weight`).

We next illustrate and motivate the generated constraints by considering the set of constraints GENERATECONSTRAINTS($\mathbf{t}_{\text{example}}$), where $\mathbf{t}_{\text{example}}$ is the program in Fig. 3. Many constraints are standard, and we therefore focus on the new suspension constraints introduced as part of this paper. In particular, the challenge is to correctly capture the flow of suspension requirements across function applications and higher-order functions. First, we see that defining aliases (line 6) generates constraints of the form $S_y \subseteq S_x$, that constants introduce `const` abstract values (e.g., $\texttt{const}_{t_6} 1 \in S_{t_6}$), and that `assume` and `weight` introduce suspension requirements, e.g., $suspend_{w_1}$ (shorthand for $suspend_{w_1} = \text{true}$).

First, we consider the constraints generated for $\lambda obs.$ (line 7 in Fig. 3) through the case at lines 9-12 in Algorithm 1. To keep the example simple, we treat the unexpanded `let rec` as an ordinary `let` in the analysis (for this particular example, the analysis result is unaffected). Omitting the recursively generated constraints for the abstraction body, the generated constraints are

$$\{\lambda obs.\, t_8 \in S_{iter}\} \cup \{suspend_n \Rightarrow suspend_{obs} \mid n \in \{t_7, t_8\}\}. \tag{4}$$

The first constraint is standard and states that the abstract value $\lambda obs.\, t_8$ flows to $S_{iter}$ as the variable naming the $\lambda obs$ expression is $t_8$ at line 26 in Fig. 3 (difficult to notice due to the column breaks). The remaining constraints are new and sets up the flow of suspension requirements. Specifically, the abstraction *obs* itself requires suspension if any expression bound by a top-level `let` in its body requires suspension. For efficiency, we only set up dependencies for expressions that may suspend (formalized by SUSPENDNAMES in Algorithm 1). Note here that we do not add the constraint $suspend_{w_1} \Rightarrow suspend_{obs}$, as $w_1$ is not at top-level in the body of *obs*. Instead, we later add the constraint $suspend_{w_1} \Rightarrow suspend_{t_8}$, and $suspend_{w_1} \Rightarrow suspend_{obs}$ follows by transitivity.

The constraints generated for the `if` bound to $t_8$ at line 10 through the case at lines 31-37 in Algorithm 1 are (omitting recursively generated constraints)

$$\begin{aligned}&\{S_{t_9} \subseteq S_{t_8}, S_{t_{17}} \subseteq S_{t_8}\}\\ &\qquad \cup \{suspend_n \Rightarrow suspend_{t_8} \mid n \in \{t_{11}, t_{13}, t_{14}, w_1, t_{16}, t_{17}\}\}.\end{aligned} \tag{5}$$

The first two constraints are standard, and state that abstract values in the results of both branches flow to the result $S_{t_8}$. The last set of constraints is new and similar to the abstraction suspension constraints. The constraints capture that all expressions at top-level in both branches that require suspension also cause $t_8$ to require suspension.

Consider the application at line 23 in Fig. 3. The generated constraints through the case at lines 13-25 in Algorithm 1 are

$$\begin{aligned}\{\ &\forall z \forall y\ \ \lambda z.y \in S_{iter} \Rightarrow (S_{t_{16}} \subseteq S_z) \wedge (S_y \subseteq S_{t_{17}}),\\ &\forall y \forall n\ \ \texttt{const}_y\, n \in S_{iter} \wedge n > 1 \Rightarrow \texttt{const}_y\, n - 1 \in S_{t_{17}},\\ &\forall y\ \ \lambda y.\_ \in S_{iter} \Rightarrow (suspend_y \Rightarrow suspend_{t_{17}}),\\ &\forall y\ \ \texttt{const}_y\ \ \_ \in S_{iter} \Rightarrow (suspend_y \Rightarrow suspend_{t_{17}}),\\ &suspend_{t_{17}} \Rightarrow (\forall y\ \ \lambda y.\_ \in S_{iter} \Rightarrow suspend_y)\\ &\qquad\qquad \wedge (\forall y\ \ \texttt{const}_y\ \ \_ \in S_{iter} \Rightarrow suspend_y)\ \ \}.\end{aligned} \tag{6}$$

The first two constraints are standard and state how abstract values flow as a result of applications. The last three constraints are new and relate to suspension. The third and fourth constraints state that if an abstraction or intrinsic requiring suspension flows to *iter*, the result $t_{17}$ of the application also requires suspension. The fifth constraint states that if the result $t_{17}$ requires suspension, then *all* abstractions and constants flowing to *iter* require suspension. This last constraint is not strictly required to later prove the soundness of the analysis in Theorem 1, but, as we will see in Section 5, it is required for the selective CPS transformation.

We find a solution to the constraints through a fairly standard algorithm that propagates abstract values according to the constraints until fixpoint.[†] However, we extend the algorithm to support the new suspension constraints. The algorithm is a function ANALYZESUSPEND: $T_{\text{ANF}} \to ((X \to \mathcal{P}(A)) \times \mathcal{P}(X))$. The function returns a map $\mathsf{data} : X \to \mathcal{P}(A)$ that assigns sets of abstract values to all $S_x$ and a set $\mathsf{suspend} : \mathcal{P}(X)$ that assigns $suspend_x = \text{true}$ iff $x \in \mathsf{suspend}$. Importantly, the assignments to $S_x$ and $\mathsf{suspend}_x$ satisfy all generated constraints. To illustrate the algorithm, here are the analysis results ANALYZESUSPEND($\mathbf{t}_{\text{example}}$):

$$\begin{gathered}
S_{iter} = \{\lambda obs.t_8\} \quad S_{t_6} = \{\texttt{const}_{t_6}1\} \quad S_{t_{10}} = \{\texttt{const}_{t_{10}}2\} \\
S_{t_{11}} = \{\texttt{const}_{t_{10}}1\} \quad S_{t_{12}} = \{\texttt{const}_{t_{12}}1\} \quad S_{t_{15}} = \{\texttt{const}_{t_{15}}1\} \\
S_n = \varnothing \mid \text{all other } n \in X \\
suspend_n = \text{true} \mid n \in \{obs, w_1, t_8, t_{17}, t_{22}\} \\
suspend_n = \text{false} \mid \text{all other } n \in X.
\end{gathered} \tag{7}$$

The above results confirm our earlier reasoning: the expressions labeled by *obs*, $w_1$, $t_8$, $t_{17}$, and $t_{22}$ may require suspension.

We now consider the soundness of the analysis. First, the soundness of 0-CFA is well established (see, e.g., Nielson et al. [39]) and extends to our new constraints, and we take the following lemma to hold without proof.

**Lemma 1 (0-CFA soundness).** *For every* $\mathbf{t} \in T_{ANF}$*, the solution given by* ANALYZESUSPEND($\mathbf{t}$) *for* $S_x$ *and* $suspend_x$*,* $x \in X$*, satisfies the constraints* GENERATECONSTRAINTS($\mathbf{t}$).

Next, we must show that the constraints themselves are sound. Consider the evaluation of an arbitrary term $\mathbf{t} \in T_{\text{ANF}}$. For each subderivation of $\mathbf{t}$, labeled by a name $x$ (due to ANF), it must hold that $suspend_x = \text{true}$ if the subderivation requires suspension. Otherwise, the analysis is unsound. Theorem 1 formally captures the soundness. Note that the analysis is conservative (i.e., incomplete), because it may find $suspend_x = true$ even if the subderivation for $x$ does not require suspension.

**Theorem 1 (Suspension analysis soundness).** *Let* $\mathbf{t} \in T_{ANF}$*,* $s \in S$*,* $u \in \{false, true\}$*,* $w \in \mathbb{R}$*, and* $\mathbf{v} \in V$ *such that* $\varnothing \vdash \mathbf{t} \; {}^{s}\!\Downarrow_{u}^{w} \mathbf{v}$*. Now, let* $S_x$ *and* $suspend_x$ *for* $x \in X$ *according to* ANALYZESUSPEND($\mathbf{t}$)*. For every subderivation* $(\rho \vdash \texttt{let}\ x = \mathbf{t}_1\ \texttt{in}\ \mathbf{t}_2 \; {}^{s_1 \| s_2}\!\Downarrow_{u_1 \vee u_2}^{w_1 \cdot w_2} \mathbf{v}')$ *of* $(\varnothing \vdash \mathbf{t} \; {}^{s}\!\Downarrow_{u}^{w} \mathbf{v})$*,* $u_1 = true$ *implies* $suspend_x = true$*.*

**Algorithm 2** Selective continuation-passing style transformation. We define $\mathbf{t}_{\text{id}} = \lambda x.x$. The term $c_{\text{CPS}}$ is the CPS version of $c$. We write the functional-style pseudocode for the algorithm itself in sans serif font to distinguish it from terms in $T$.

**function** CPS(vars, **t**): $\mathcal{P}(X) \times T_{\text{ANF}} \to T^+$ =

```
1  return CPS'(t_id, t)
2
3  function CPS'(cont,t): T × T_ANF → T⁺ =
4    match t with
5    | x → if cont = t_id then t else cont t
6    | let x = t1 in t2 →
7      let t2' = CPS'(cont, t2) in
8      match t1 with
9      | y → let x = t1 in t2'
10     | c → let x =
11             (if x ∈ vars then c_CPS else c) in t2'
12     | λy. t_b →
13       let t1' = if y ∈ vars
14         then λk.λy. CPS'(k, t_b)
15         else λy. CPS'(t_id, t_b)
16       in
17       let x = t1' in t2'
18     | lhs rhs →
19       if x ∈ vars then
20         if TAILCALL(t)
21         then lhs cont rhs
22         else lhs (λx.t2') rhs
23       else let x = t1 in t2'
24
25
26
27
28     | if y then t_t else t_e →
29       if x ∈ vars then
30         if TAILCALL(t) then
31           if y then CPS'(cont, t_t)
32           else CPS'(cont, t_e)
33         else
34           let k = λx.t2' in
35           if y then CPS'(k, t_t) else CPS'(k, t_e)
36       else let x = if y then CPS'(t_id, t_t)
37                    else CPS'(t_id, t_e) in t2'
38     | assume y → let x = t1 in t2'
39       if x ∈ vars then
40         if TAILCALL(t)
41         then Suspension_assume(y, cont)
42         else Suspension_assume(y,λx.CPS'(cont, t2))
43       else let x = t1 in t2'
44     | weight y → let x = t1 in t2'
45       if x ∈ vars then
46         if TAILCALL(t)
47         then Suspension_weight(y, cont)
48         else Suspension_weight(y,λx.CPS'(cont, t2))
49       else let x = t1 in t2'
50
51 function TAILCALL(t): T_ANF → {false, true} =
52   match t with
53   | let x = _ in x → true
54   | _ → false
```

The proof uses Lemma 1 and structural induction over the derivation $\varnothing \vdash \mathbf{t} \overset{s}{\Downarrow}{}^{w}_{u} \mathbf{v}$.†

Next, we use the suspension analysis to selectively CPS transform programs.

## 5 Selective CPS Transformation

This section presents the second technical contribution: the selective CPS transformation. The transformations themselves are standard, and the challenge is to correctly use the suspension analysis results for a selective transformation.

Algorithm 2 is the full algorithm. Using terms in ANF as input significantly helps reduce the algorithm's complexity. The main function CPS takes as input a set vars : $\mathcal{P}(X)$, indicating which expressions to CPS transform, and a program $\mathbf{t} \in T_{\text{ANF}}$ to transform. It is the new vars argument that separates the transformation from a standard CPS transformation. For the purposes of this paper, we always use vars = $\{x \mid \mathit{suspend}_x = \text{true}\}$, where the $\mathit{suspend}_x$ come from ANALYZESUSPEND(**t**). One could also use vars = $X$ for a standard full CPS transformation (e.g., Fig 1f), or some other set vars for other application domains. The value returned from the CPS function is a (non-ANF) term of the

```
 1 let t1 = 2 in
 2 let t2 = 2 in
 3 let t3 = Beta in
 4 let t4 = t3 t1 in
 5 let t5 = t4 t2 in
 6 let a1 = assume t5 in
 7 let rec iter = λk. λobs.
 8   let t6 = null in
 9   let t7 = t6 obs in
10   if t7 then
11     let t9 = () in
12     t9
13   else
14     let t10 = f_Bernoulli in
15     let t11 = t10 a1 in
16     let t12 = head in
17     let t13 = t12 obs in
18     let t14 = t11 t13 in
19     Suspension_weight(t14,
20       λ_.
21         let t15 = tail in
22         let t16 = t15 obs in
23         iter k t16)
24 in
25 let t18 = true in
26 let t19 = false in
27 let t20 = true in
28 let t21 = true in
29 let t22 = [t21,t20,t19,t18] in
30 let k' = λ_. a1 in
31 iter k' t22
```

Fig. 4: The running example from Fig. 3 after selective CPS transformation. The program is semantically equivalent to Fig. 1e.

type $T^+$. The helper function CPS′, initially called at line 1, takes as input a continuation term cont, indicating the continuation to apply in tail position. Initially, this continuation term is $\mathbf{t}_{\text{id}}$, which indicates no continuation. Similarly to Algorithm 1, the top-level match at line 4 has two cases: a simple case for variables (line 5) and a complex case for `let` expressions (lines 6–49). To enable optimization of tail calls, the auxiliary function TAILCALL indicates whether or not an ANF expression is a tail call (i.e., of the form `let` $x$ `=` $\mathbf{t}'$ `in` $x$).

We now illustrate Algorithm 2 by computing CPS($\mathsf{vars}_{\text{example}}, \mathbf{t}_{\text{example}}$), where $\mathsf{vars}_{\text{example}} = \{obs, w_1, t_8, t_{17}, t_{22}\}$ is from (7), and $\mathbf{t}_{\text{example}}$ is from Fig. 3. Fig. 4 presents the final result. First, we note that the transformation does not change expressions not labeled by a name in $\mathsf{vars}_{\text{example}}$, as they do not require suspension. In the following, we therefore focus only on the transformed expressions. First, consider the abstraction *obs* defined at line 7 in Fig. 3, handled by the case at line 12 in Algorithm 2. As $obs \in \mathsf{vars}_{\text{example}}$, we apply the standard CPS transformation for abstractions: add a continuation parameter to the abstraction and recursively transform the body with this continuation. Next, consider the transformation of the `weight` expression $w_1$ at line 20 in Fig. 3, handled by the case at line 44 in Algorithm 2. The expression is not at tail position, so we build a new continuation containing the subsequent `let` expressions, recursively transform the body of the continuation, and then wrap the end result in a Suspension object. The `if` expression $t_8$ at line 10 in Fig. 3, handled by the case at line 28 in Algorithm 2, is in tail position (it is directly followed by returning $t_8$). Consequently, we transform both branches recursively. Finally, we have the applications $t_{17}$ and $t_{22}$ at lines 23 and 33 in Fig. 3, handled by the case at line 18 in Algorithm 2. The application $t_{17}$ is at tail position, and we transform it by adding the current continuation as an argument. The application at $t_{22}$ is not at tail position, so we construct a continuation $k'$ that returns the final value $a_1$ (line 34 in Fig. 3), and then add it as an argument to the application.

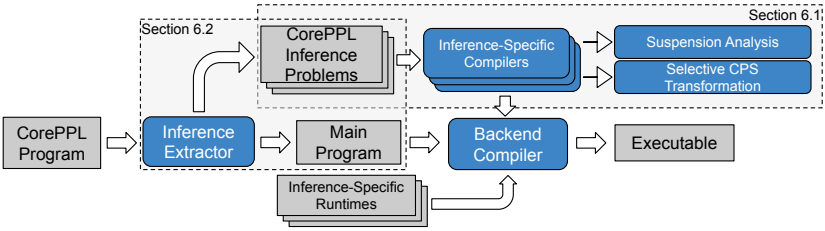



Fig. 5: Overview of the Miking CorePPL compiler implementation. We divide the overall compiler into two parts, (i) *suspension analysis and selective CPS* (Section 6.1) and (ii) *inference problem extraction* (Section 6.2). The figure depicts artifacts as gray rectangular boxes and transformation units and libraries as blue rounded boxes. Note how the *inference extractors* transformation separates the program into two different paths that are combined again after the inference-specific compilation. The white inheritance arrows (pointing to *suspension analysis* and *selective CPS transformations*) mean that these libraries are used within the inference-specific compiler transformation.

It is not guaranteed that Algorithm 2 produces a correct result. Specifically, for all applications *lhs rhs*, we must ensure that (i) if we CPS transform the application, we must also CPS transform *all* possible abstractions that can occur at *lhs*, and (ii) if we do *not* CPS transform the application, we must *not* CPS transform any abstraction that can occur at *lhs*. We control this through the argument vars. In particular, assigning vars according to the suspension analysis produces a correct result. To see this, consider the application constraints at lines 13–25 in Algorithm 1 again, and note that if any abstraction or intrinsic operation that requires suspension occur at *lhs*, $suspend_x = \text{true}$. Furthermore, the last application constraint ensures that if $suspend_x = \text{true}$, then *all* abstractions and intrinsic operations that occur at *lhs* require suspension. Consequently, for all $\lambda y.\ \_$ and $const_y\ \_$, either all $suspend_y = \text{true}$ or all $suspend_y = \text{false}$.

## 6 Implementation

We implement the suspension analysis and selective CPS transformation in Miking CorePPL [30], a core PPL implemented in the domain-specific language construction framework Miking [9]. We choose Miking CorePPL for the implementation over other CPS-based PPLs, as the language implementation contains an existing 0-CFA base implementation which simplifies the suspension analysis implementation. Fig. 5 presents the organization of the CorePPL compiler. The input is a CorePPL program that may contain many inference problems and applications of inference algorithms, similar to WebPPL and Anglican. The output is an executable produced by one of the Miking backend compilers. Section 6.1 gives the details of the suspension analysis and selective CPS implementations, and in particular the differences compared to the core calculus in Section 3. Sec-

tion 6.2 presents the inference extractor and its operation combined with selective CPS. The suspension analysis, selective CPS transformation, and inference extraction implementations consist of roughly 1500 lines of code (a contribution in this paper). The code is available on GitHub [2].

### 6.1 Suspension Analysis and Selective CPS

Miking CorePPL extends the abstract syntax in Definition 1 with standard functional data structures and features such as algebraic data types (records, tuples, and variants), lists, and pattern matching. The suspension analysis and selective CPS implementations in Miking CorePPL extend Algorithm 1 and Algorithm 2 to support these language features. Furthermore, compared to $suspend_{\texttt{weight}}$ and $suspend_{\texttt{assume}}$ in Fig. 2, the implementation allows arbitrary configuration of suspension sources. In particular, the implementation uses this arbitrary configuration together with the alignment analysis by Lundén et al. [29]. This combination allows selectively CPS transforming to suspend at a subset of `assume`s or `weight`s for aligned versions of SMC and MCMC inference algorithms.

Miking CorePPL also includes a framework for inference algorithm implementation. Specifically, to implement new inference algorithms, users implement an *inference-specific compiler* and *inference-specific runtime*. Fig. 5 illustrates the different compilers and runtimes. Each inference-specific compiler applies the suspension analysis and selective CPS transformation to suit the inference algorithm's particular suspension requirements.

Next, we show how Miking CorePPL handles programs containing many inference problems solved with different inference algorithms.

### 6.2 Inference Problem Extraction

Fig. 5 includes the inference extraction compiler procedure. First, the compiler applies an inference extractor to the input program. The result is a set of inference problems and a main program containing remaining glue code. Second, the compiler applies inference-specific compilers to each inference problem. Finally, the compiler combines the main program and the compiled inference problems with inference-specific runtimes and supplies the result to a backend compiler.

Consider the example in Fig. 6a. We define a function `m` that constructs a minimal inference problem on lines 7–10, using a single call to `assume` and a single call to `observe` (modifying the execution weight similar to `weight`). The function takes an initial probability distribution `d` and a data point `y` as input. We apply aligned lightweight MCMC inference for the inference problem through the `infer` construct on lines 12–16. The first argument to `infer` gives the inference algorithm configuration, and the second argument the inference problem. Inference problems are thunks (i.e., functions with a dummy unit argument). We construct the inference problem thunk by an application of `m` with a uniform initial distribution and data point 1.0. The inference result `d0` is another probability distribution, and we use it as the first initial distribution in the recursive

```
1 mexpr
2 let data = [
3   24.0, 42.2, 96.7, 9.2, 85.8,
4   34.2, 41.7, 53.4, 85.6, 45.4
5 ] in
6
7 let m = lam d. lam y. lam.
8  let x = assume d in
9  observe y (Gaussian x 0.1);
10 x in
11
12 let d0 =
13  infer (LightweightMCMC
14   { iterations = 100,
15     aligned = true })
16   (m (Uniform 0.0 4.0) 1.0) in
17
18 recursive let repeat =
19  lam data. lam d.
20   match data with [y] ++ data then
21    let posterior =
22     infer (BPF {particles = 100})
23           (m d y) in
24    repeat data posterior
25   else d
26 let d1 = repeat data d0 in
27 match distEmpiricalSamples d1
28 with (samples, weights) in
29 iter
30  (lam s.
31   print
32    (concat (float2string s) "\n"))
33  samples
```

(a) Miking CorePPL program.

```
1 let m = lam d. lam y. lam.
2   let x = assume d in
3   observe y (Gaussian x 0.1);
4   x in
5 m (Uniform 0.0 4.0) 1.0 ()
```

(b) Extracted inference problem from line 13 in (a).

```
1 let m = lam d. lam y. lam.
2   let x = assume d in
3   observe y (Gaussian x 0.1);
4   x in
5 m d y ()
```

(c) Extracted inference problem from line 22 in (a).

Fig. 6: Example Miking CorePPL program in (a) with two non-trivial uses of `infer`. Figures (b) and (c) show the extracted and selectively CPS-transformed inference problems at lines 13 and 22 in (a), respectively. The compiler handles the free variables `d` and `y` in (c) in a later stage.

`repeat` function (lines 19–24). This function repeatedly performs inference using the SMC bootstrap particle filter (lines 21–23), again using the function `m` to construct the sequence of inference problems. Each `infer` application uses the result distribution from the previous iteration as the initial distribution and consumes data points from the `data` sequence. We extract and print the samples from the final result distribution `d1` at lines 29–33. A limitation with the current extraction approach is that we do not yet support nested `infer`s.

A key challenge in the compiler design is how to handle different inference algorithms within one probabilistic program. In particular, inference algorithms require different selective CPS transformations, applied to different parts of the code. To allow the separate handling of inference algorithms, we apply the extraction approach by Hummelgren et al. [22] on the `infer` applications, producing separate inference problems for each occurrence of `infer`. Although the compiler design mostly concerns rather comprehensive engineering work, special care must be taken to handle the non-trivial problem of name bindings when transforming and combining different code entities together. For instance, the compiler must selectively CPS transform Fig. 6b to suspend at `assume` (required by MCMC) and selectively CPS transform Fig. 6c to suspend at `observe` (re-

quired by SMC). We design a robust and modular solution, where it is possible to easily add new inference algorithms without worrying about name conflicts.

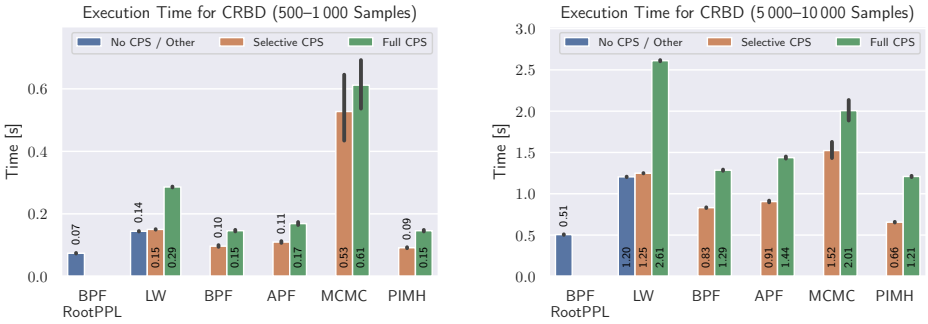
## 7 Evaluation

This section presents the evaluation of the suspension analysis and selective CPS implementations. Our main claims are that (i) the approach of selective CPS significantly improves performance compared to traditional full CPS, and (ii) that this holds for a significant set of inference algorithms, evaluated on realistic inference problems. We use four PPL models and corresponding data sets from the Miking benchmarks repository, available on GitHub [1]. The models are: constant rate birth-death (CRBD) in Section 7.1, cladogenetic diversification rate shift (ClaDS) in Section 7.2, latent Dirichlet allocation (LDA) in Section 7.3, and vector-borne disease (VBD) in Section 7.4. All models are significant and actively used in different research areas: CRBD and ClaDS in evolutionary biology and phylogenetics [37,43,32], LDA in topic modeling [7], and VBD in epidemiology [14,34]. In addition to the Miking CorePPL models from the Miking benchmarks, we also implement CRBD in WebPPL and Anglican.

We add a number of popular inference algorithms in Miking CorePPL with support for selective CPS. The first is standard likelihood weighting (LW), as introduced in Section 2. LW does not strictly require CPS, but we implement it with suspensions at `weight` to highlight the difference between no CPS, selective CPS, and full CPS. LW gives a good direct measure of CPS overhead as the algorithm simply executes programs many times. Suspending at `weight` can also be useful in LW to stop executions with weight 0 (i.e., useless samples) early. However, we do not use early stopping to isolate the effect CPS has on execution time. Next, we add the bootstrap particle filter (BPF) and alive particle filter (APF). Both are SMC algorithms that suspend at `weight` to resample executions. BPF is a standard algorithm often used in PPLs, and APF is a related algorithm introduced in a PPL context by Kudlicka et al. [24]. The final two inference algorithms we add are aligned lightweight MCMC (just MCMC for short) and particle-independent Metropolis–Hastings. Aligned lightweight MCMC [29] is an extension to the standard PPL Metropolis–Hastings approach introduced by Wingate et al. [49], and suspends at a subset of calls to `assume`. Particle-independent Metropolis–Hastings (PIMH) is an MCMC algorithm that repeatedly uses the BPF (suspending at `weight`) within a Metropolis–Hastings MCMC algorithm [40]. We limit the scope to single-core CPU inference.

In addition to the inference algorithms in Miking CorePPL, we also use three other state-of-the-art PPLs for CRBD: Anglican, WebPPL, and the special high-performance RootPPL compiler for Miking CorePPL [30]. For Anglican, we apply LW, BPF, and PIMH inference. For WebPPL, we use BPF and (non-aligned) lightweight MCMC. For the RootPPL version of Miking CorePPL, we use BPF inference (the only supported inference algorithm).

We consider two configurations for each model: 1 000 and 10 000 samples. An exception is for CRBD and ClaDS, where we adjust APF to use 500 and 5 000




| | Anglican LW | Anglican BPF | WebPPL BPF | WebPPL MCMC |
|---|---|---|---|---|
| 1 000 samples | 11.6 ± 0.36 s | 5.65 ± 2.71 s | 2.42 ± 0.20 s | 1.42 ± 0.07 s |
| 10 000 samples | 90.4 ± 2.12 s | 29.1 ± 2.35 s | 53.9 ± 4.03 s | 3.10 ± 0.77 s |

Fig. 7: Mean execution times for the CRBD model. The error bars show 95% confidence intervals (using the option `('ci', 95)` in Seaborn's `barplot`). The table shows standard deviations.

samples to make the inference accuracy comparable to the related BPF. We run each experiment 300 times (with one warmup run) and measure execution time (excluding compile time). To justify the efficiency of the suspension analysis and selective CPS transformation that are part of the compiler, we note here that they, combined, run in only 1–5 ms for all models.

The experiments do *not* compare the performance of different inference algorithms. To do this, one would also need to consider how accurate the inference results are for a given amount of execution time. Accuracy varies dramatically between different combinations of inference algorithms and models. We evaluate the execution time of selective and full CPS in isolation for individual inference algorithms. Selective CPS is solely an execution time optimization—the algorithms themselves and their accuracy remain unchanged.†

For Miking CorePPL, we used OCaml 4.12.0 as backend compiler for the implementation in Section 6 and GCC 7.5.0 for the separate RootPPL compiler. We used Anglican 1.1.0 (OpenJDK 11.0.19) and WebPPL 0.9.15 (Node.js 16.18.0). We ran the experiments on an Intel Xeon Gold 6148 CPU with 64 GB of memory using Ubuntu 18.04.6.

### 7.1 Constant Rate Birth-Death

CRBD is a diversification model, used by evolutionary biologists to infer distributions over birth and death rates for observed evolutionary trees of groups of species, called *phylogenies*. For the CRBD experiment, we use the Alcedinidae phylogeny (Kingfisher birds, 54 extant species) [43,23]. We compare CRBD in Miking CorePPL (55 lines of code)†, Anglican (129 lines of code)†, and WebPPL (66 lines of code)†. The total experiment execution time was 9 hours.

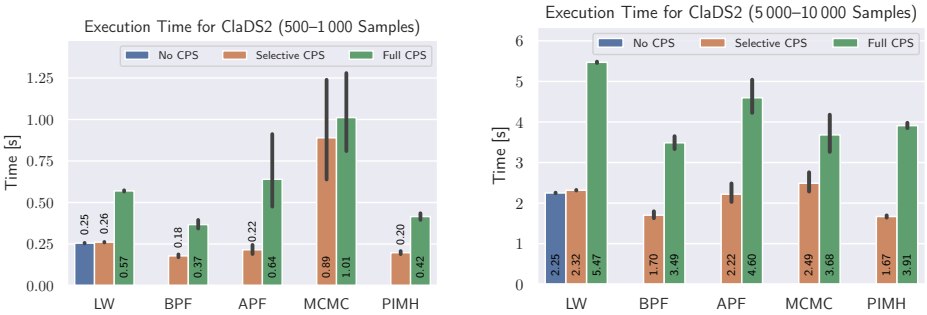



Fig. 8: Mean execution times for the ClaDS model. The error bars show 95% confidence intervals (using the option `('ci', 95)` in Seaborn's `barplot`).

Fig. 7 presents the results. We note that selective CPS is faster than full CPS in all cases. Unlike full CPS, the overhead of selective CPS compared to no CPS is marginal for LW. The execution time for early MCMC samples is sensitive to initial conditions, and we therefore see more variance for MCMC compared to the other algorithms. When we increase the number of samples to 10 000, the variance reduces. With the exception of MCMC in WebPPL, the execution times for Anglican and WebPPL are one order of magnitude slower than the equivalent algorithms in Miking CorePPL. However, note that the comparison is only for reference and not entirely fair, as Anglican and WebPPL use different execution environments compared to Miking CorePPL. Lastly, we note that the Miking CorePPL BPF implementation with selective CPS is not much slower than when compiling Miking CorePPL to RootPPL BPF—a compiler designed specifically for efficiency (but with other limitations, such as the lack of garbage collection). RootPPL does not use CPS, and instead enables suspension through a low-level transformation using the concept of PPL control-flow graphs [30].

### 7.2 Cladogenetic Diversification Rate Shift

ClaDS is another diversification model used in evolutionary biology [32,43]. Unlike CRBD, it allows birth and death rates to change over time. We again use the Alcedinidae phylogeny. The source code consists of 72 lines of code.† The total experiment execution time was 3 hours. Fig. 8 presents the results. We note that selective CPS is faster than full CPS in all cases.

### 7.3 Latent Dirichlet Allocation

LDA [7] is a model from natural language processing used to categorize documents into *topics*. We use a synthetic data set with size comparable to the data set in Ritchie et al. [41]: a vocabulary of 100 words, 10 topics, and 25 observed documents (30 words in each). We do not apply any optimization techniques such as collapsed Gibbs sampling [21]. Solving the inference problem using a PPL is

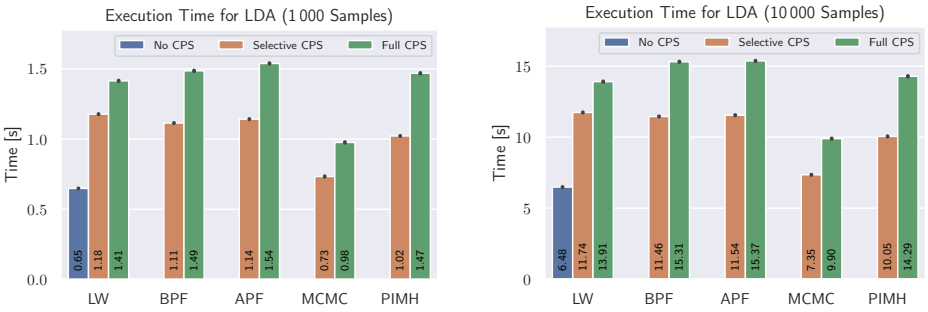



Fig. 9: Mean execution times for the LDA model. The error bars show 95% confidence intervals (using the option `('ci', 95)` in Seaborn's `barplot`).

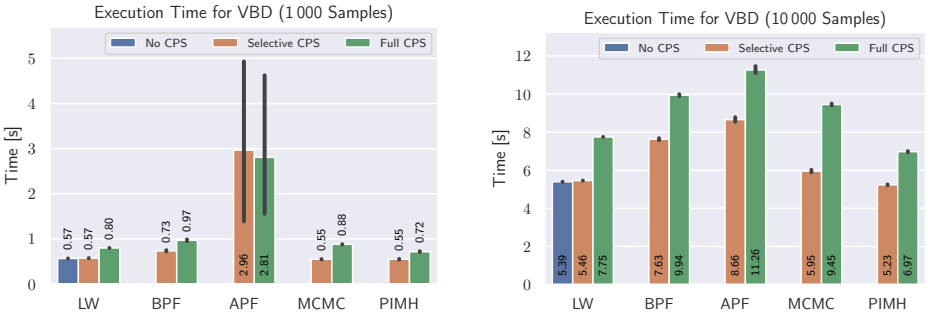



Fig. 10: Mean execution times for the VBD model. The error bars show 95% confidence intervals (using the option `('ci', 95)` in Seaborn's `barplot`).

therefore challenging already for small data sets. The source code consists of 26 lines of code.† The total experiment execution time was 12 hours.

Fig. 9 presents the results. We note that selective CPS is faster than full CPS in all cases. Interestingly, the reduction in overhead compared to full CPS for LW is not as significant. The reason is that suspension at `weight` for the model requires that we CPS transform the most computationally expensive recursion.

### 7.4 Vector-Borne Disease

We use the VBD model from Funk et al. [14] and later Murray et al. [34]. The background is a dengue outbreak in Micronesia and the spread of disease between mosquitos and humans. The inference problem is to find the true numbers of susceptible, exposed, infectious, and recovered (SEIR) individuals each day, given daily reported numbers of new cases at health centers. The source code consists of 140 lines of code.† The total execution time was 8 hours.

Fig. 10 presents the results. Again, we note that selective CPS is faster than full CPS in all cases, except seemingly for APF and 1 000 samples. This is very likely a statistical anomaly, as the variance for APF is quite severe for the case with 1 000 samples. Compared to the BPF, APF uses a resampling approach for

which the execution time varies a lot if the number of samples is too low [24]. The plots clearly show this as, compared to 1 000 samples, the variance is reduced to BPF-comparable levels for 10 000 samples. In summary, the evaluation demonstrates the clear benefits of selective CPS over full CPS for universal PPLs.

## 8 Related Work

There are a number of universal PPLs that require non-trivial suspension. One such language is Anglican [50], which solves the suspension problem using CPS. Anglican performs a full CPS transformation with one exception—certain statically known functions named *primitive procedures*, that include a subset of the regular Clojure (the host language of Anglican) functions, are guaranteed to not execute PPL code, and Anglican does not CPS transform them [47]. However, higher-order functions in Clojure libraries cannot be primitive procedures, and Anglican must manually reimplement such functions (e.g., `map` and `fold`). Anglican does not consider a selective CPS transformation of PPL code, and always fully CPS transforms the PPL part of Anglican programs.

WebPPL [18] and the approach by Ritchie et al. [41] also make use of CPS transformations to implement PPL inference. They do not, however, consider selective CPS transformations. Ścibior et al. [44] present an architectural design for a probabilistic functional programming library based on monads and monad transformers (and corresponding theory in Ścibior et al. [45]). In particular, they use a coroutine monad transformer to suspend SMC inference. This approach is similar to ours in that it makes use of high-level functional language features to enable suspension. They do not, however, consider a selective transformation.

The PPLs Pyro [6], Stan [10,5], Gen [11,27], and Edward [48] either implement inference algorithms that do not require suspension (e.g., Hamiltonian Monte Carlo), or restrict the language in such a way that suspension is explicit and trivially handled by the language implementation. For example, SMC in Pyro[8] and newer versions of Birch require that users explicitly write programs as a `step` function that the SMC implementation calls iteratively. Resampling only occurs in between calls to `step`, and suspension is therefore trivial.

Work on general-purpose selective CPS transformations include Nielsen [38], Asai and Uehara [4], Rompf et al. [42], and Leijen [26]. They consider typed languages, unlike the untyped language in this paper. The early work by Nielsen [38] considers the efficient implementation of `call/cc` through a selective CPS transformation. The transformation requires manual user annotations, unlike the fully automatic approach in this paper. A more recent approach is due to Asai and Uehara [4], who consider an efficient implementation of delimited continuations using `shift` and `reset` through a selective CPS transformation. Similar to us, they automatically determine where to selectively CPS transform programs. They use an approach based on type inference, while our approach builds upon 0-CFA. Rompf et al. [42] follow a similar approach to Asai and Uehara [4], but for

[8] Note that the main inference algorithm in Pyro is stochastic variational inference, which does not require suspension.

Scala, and additionally require user annotations. Leijen [26] uses a type-directed selective CPS transformation to compile algebraic effect handlers.

There are low-level alternatives to CPS for suspension in PPLs. In particular, there are various languages and approaches that directly implement support for non-preemptive multitasking (e.g., coroutines). Turing [15] and older versions of Birch [36,35] implement coroutines to enable arbitrary suspension, but do not discuss the implementations in detail. Lundén et al. [30] introduces and uses the concept of PPL control-flow graphs to compile Miking CorePPL to the low-level C++ framework RootPPL. The compiler explicitly introduces code that maintains special execution call stacks, distinct from the implicit C++ call stacks. The implementation results in excellent performance, but supports neither garbage collection nor higher-order functions. Another low-level approach is due to Paige and Wood [40], who exploits mutual exclusion locks and the `fork` system call to suspend and resample SMC executions. In theory, many of the above low-level alternatives to CPS can, if implemented efficiently, result in the least possible overhead due to more fine-grained low-level control. The approaches do, however, require significantly more implementation effort compared to a CPS transformation. Comparatively, the selective CPS transformation is a surprisingly simple, high-level, and easy-to-implement alternative that brings the overhead of CPS closer to that of more low-level approaches.

## 9 Conclusion

This paper introduces a selective CPS transformation for the purpose of execution suspension in PPLs. To enable the transformation, we develop a static suspension analysis that determines parts of programs that require a CPS transformation as a consequence of inference algorithm suspension requirements. We implement the suspension analysis, selective CPS transformation, and an inference problem extraction procedure (required as a result of the selective CPS transformation) in Miking CorePPL. Furthermore, we evaluate the implementation on real-world models from phylogenetics, topic-modeling, and epidemiology. The results demonstrate significant speedups compared to the standard full CPS suspension approach for a large number of Monte Carlo inference algorithms.

**Acknowledgments.** This project was financially supported by the Swedish Foundation for Strategic Research (FFL15-0032 and RIT15-0012), partially supported by the Swedish Research Council (Grant No. 2018-04329), and by Digital Futures (the DLL project). 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. We thank Gizem Çaylak for her LDA implementation and Viktor Senderov for his ClaDS implementation.

**Data-Availability Statement.** The paper has an accompanying artifact that supports the evaluation: `https://zenodo.org/doi/10.5281/zenodo.10454311`.

## References


1. The Miking benchmark suite. `https://github.com/miking-lang/miking-benchmarks` (2023), accessed: 2023-01-02
2. Miking DPPL. `https://github.com/miking-lang/miking-dppl` (2023), accessed: 2023-01-02
3. Appel, A.W.: Compiling with Continuations. Cambridge University Press (1991)
4. Asai, K., Uehara, C.: Selective cps transformation for shift and reset. In: Proceedings of the ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation. pp. 40–52. Association for Computing Machinery (2017)
5. Baudart, G., Burroni, J., Hirzel, M., Mandel, L., Shinnar, A.: Compiling stan to generative probabilistic languages and extension to deep probabilistic programming. In: Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation. pp. 497–510. Association for Computing Machinery (2021)
6. 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)
7. Blei, D.M., Ng, A.Y., Jordan, M.I.: Latent Dirichlet allocation. Journal of Machine Learning Research **3**, 993–1022 (2003)
8. 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)
9. 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)
10. 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)
11. 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)
12. Doucet, A., de Freitas, N., Gordon, N.: Sequential Monte Carlo Methods in Practice. Information Science and Statistics, Springer New York (2001)
13. 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 (1993)
14. Funk, S., Kucharski, A.J., Camacho, A., Eggo, R.M., Yakob, L., Murray, L.M., Edmunds, W.J.: Comparative analysis of dengue and zika outbreaks reveals differences by setting and virus. PLOS Neglected Tropical Diseases **10**(12), 1–16 (2016)
15. Ge, H., Xu, K., Ghahramani, Z.: Turing: A language for flexible probabilistic inference. In: Proceedings of the Twenty-First International Conference on Artificial Intelligence and Statistics. vol. 84, pp. 1682–1690. Proceedings of Machine Learning Research (2018)

16. Gilks, W., Richardson, S., Spiegelhalter, D.: Markov Chain Monte Carlo in Practice. Chapman & Hall/CRC Interdisciplinary Statistics, Taylor & Francis (1995)
17. 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)
18. Goodman, N.D., Stuhlmüller, A.: The design and implementation of probabilistic programming languages. `http://dippl.org` (2014), accessed: 2022-10-31
19. Goodman, N.D., Tenenbaum, J.B., Contributors, T.P.: Probabilistic Models of Cognition. `http://probmods.org/v2` (2016), accessed: 2022-06-10
20. 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)
21. Griffiths, T.L., Steyvers, M.: Finding scientific topics. Proceedings of the National academy of Sciences **101**(suppl_1), 5228–5235 (2004)
22. Hummelgren, L., Wikman, J., Eriksson, O., Haller, P., Broman, D.: Expression acceleration: Seamless parallelization of typed high-level languages. arXiv e-prints p. arXiv:2211.00621 (2022)
23. Jetz, W., Thomas, G.H., Joy, J.B., Hartmann, K., Mooers, A.O.: The global diversity of birds in space and time. Nature **491**(7424), 444–448 (2012)
24. Kudlicka, J., Murray, L.M., Ronquist, F., Schön, T.B.: Probabilistic programming for birth-death models of evolution using an alive particle filter with delayed sampling. In: Conference on Uncertainty in Artificial Intelligence (2019)
25. Kulkarni, T.D., Kohli, P., Tenenbaum, J.B., Mansinghka, V.: Picture: A probabilistic programming language for scene perception. In: 2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR). pp. 4390–4399 (2015)
26. Leijen, D.: Type directed compilation of row-typed algebraic effects. In: Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages. pp. 486–499. Association for Computing Machinery (2017)
27. Lew, A.K., Matheos, G., Zhi-Xuan, T., Ghavamizadeh, M., Gothoskar, N., Russell, S., Mansinghka, V.K.: Smcp3: Sequential monte carlo with probabilistic program proposals. In: Proceedings of The 26th International Conference on Artificial Intelligence and Statistics. pp. 7061–7088. Proceedings of Machine Learning Research (2023)
28. 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 (2021)
29. Lundén, D., Çaylak, G., Ronquist, F., Broman, D.: Automatic alignment in higher-order probabilistic programming languages. In: Programming Languages and Systems. pp. 535–563 (2023)
30. 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 (2022)
31. Lundén, D., Hummelgren, L., Kudlicka, J., Eriksson, O., Broman, D.: Suspension analysis and selective continuation-passing style for higher-order probabilistic programming languages. arXiv e-prints p. arXiv:2302.13051 (2024)
32. 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)

33. Midtgaard, J.: Control-flow analysis of functional programs. ACM Computing Surveys **44**(3) (2012)
34. Murray, L., Lundén, D., Kudlicka, J., Broman, D., Schön, T.: Delayed sampling and automatic Rao-Blackwellization of probabilistic programs. In: Proceedings of the Twenty-First International Conference on Artificial Intelligence and Statistics. vol. 84, pp. 1037–1046. Proceedings of Machine Learning Research (2018)
35. Murray, L.M.: Lazy object copy as a platform for population-based probabilistic programming. arXiv e-prints p. arXiv:2001.05293 (2020)
36. Murray, L.M., Schön, T.B.: Automated learning with a probabilistic programming language: Birch. Annual Reviews in Control **46**, 29–43 (2018)
37. Nee, S.: Birth-death models in macroevolution. Annual Review of Ecology, Evolution, and Systematics **37**(1), 1–17 (2006)
38. Nielsen, L.R.: A selective cps transformation. Electronic Notes in Theoretical Computer Science **45**, 311–331 (2001)
39. Nielson, F., Nielson, H.R., Hankin, C.: Principles of Program Analysis. Springer-Verlag (1999)
40. Paige, B., Wood, F.: A compilation target for probabilistic programming languages. In: Proceedings of the 31st International Conference on Machine Learning. vol. 32, pp. 1935–1943. Proceedings of Machine Learning Research (2014)
41. 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. Proceedings of Machine Learning Research (2016)
42. Rompf, T., Maier, I., Odersky, M.: Implementing first-class polymorphic delimited continuations by a type-directed selective cps-transform. In: Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming. pp. 317–328. Association for Computing Machinery (2009)
43. 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)
44. Ścibior, A., Kammar, O., Ghahramani, Z.: Functional programming for modular Bayesian inference. Proceedings of the ACM on Programming Languages **2**(ICFP) (2018)
45. 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)
46. Shivers, O.G.: Control-flow analysis of higher-order languages or taming lambda. Carnegie Mellon University (1991)
47. Tolpin, D., van de Meent, J.W., Yang, H., Wood, F.: Design and implementation of probabilistic programming language anglican. In: Proceedings of the 28th Symposium on the Implementation and Application of Functional Programming Languages. Association for Computing Machinery (2016)
48. Tran, D., Kucukelbir, A., Dieng, A.B., Rudolph, M., Liang, D., Blei, D.M.: Edward: A library for probabilistic modeling, inference, and criticism. arXiv e-prints p. arXiv:1610.09787 (2016)
49. Wingate, D., Stuhlmueller, A., Goodman, N.: Lightweight implementations of probabilistic programming languages via transformational compilation. In: Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics. vol. 15, pp. 770–778. Proceedings of Machine Learning Research (2011)

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