TreePPL: A Universal Probabilistic Programming Language for Phylogenetics

Viktor Senderov
Institut de Biologie de l'École normale supérieure
vsenderov@gmail.com

Jan Kudlicka
Department of Data Science and Analytics, BI Norwegian Business School
jan.kudlicka@bi.no

Daniel Lundén
Oracle
daniel.lunden@oracle.com

Viktor Palmkvist
EECS and Digital Futures, KTH Royal Institute of Technology
viktor.palmkvist@kth.se

Mariana P. Braga
Department of Ecology, Swedish University of Agricultural Sciences
mariana.pires.braga@slu.se

Emma Granqvist
Department of Bioinformatics and Genetics, Swedish Museum of Natural History
emma.granqvist@nrm.se

David Broman[*]
EECS and Digital Futures, KTH Royal Institute of Technology and
Computer Science Department, Stanford University
dbro@kth.se and broman@stanford.edu

Fredrik Ronquist[*]
Department of Bioinformatics and Genetics, Swedish Museum of Natural History
fredrik.ronquist@nrm.se

October 10, 2023

Abstract

We present TreePPL, a language for probabilistic modeling and inference in statistical phylogenetics. Specifically, TreePPL is a domain-specific universal probabilistic programming language (PPL), particularly designed for describing phylogenetic models. The core idea is to express the model as a computer program, which estimates the posterior probability distribution of interest when executed

[*]Equal contribution.

sufficiently many times. The program uses two special probabilistic constructs: `assume` statements, which describe latent random variables in the model, and `observe` statements, which condition random variables in the model on observed data. The `assume` and `observe` statements make it possible for generic inference algorithms, such as sequential Monte Carlo and Markov chain Monte Carlo algorithms, to identify checkpoints that enable them to generate and manipulate simulations from the posterior probability distribution. This means that a user can focus on describing the model, and leave the estimation of the posterior probability distribution to TreePPL's inference machinery. The TreePPL modeling language is inspired by R, Python, and the functional programming language OCaml. The model script can be conveniently run from a Python environment (an R environment is work in progress), which can be used for pre-processing, feeding the model with the observed data, controlling and running the inference, and receiving and post-processing the output data. The inference machinery is generated by a compiler framework developed specifically for supporting domain-specific modeling and inference, the Miking CorePPL framework. It currently supports a range of inference strategies, including several recent innovations that are important for efficient inference on phylogenetic models. It also supports the implementation of novel inference strategies for models described using TreePPL or other domain-specific modeling languages. We briefly describe the TreePPL modeling language and the Python environment, and give some examples of modeling and inference with TreePPL. The examples illustrate how TreePPL can be used to address a range of common problem types considered in statistical phylogenetics, from diversification and co-speciation analysis to tree inference. Although much progress has been made in recent years, developing efficient algorithms for automatic PPL-based inference is still a very active field. A few major challenges remain to be addressed before the entire phylogenetic model space is adequately covered by efficient automatic inference techniques, but several of them are being addressed in ongoing work on TreePPL. We end the paper by discussing how probabilistic programming can support the use of machine learning in designing and fine-tuning inference strategies and in extending incomplete model descriptions in phylogenetics.

1 Introduction

In software for statistical phylogenetics, it has been common practice to implement the entire problem, from model specification to inference machinery, in a general-purpose programming language like C, C++, or Java. If a larger set of models is supported, then such programs typically allow the user to select the desired model within the predefined space using command line settings—e.g. PhyloBayes [1], RaXML [2] and IQ-Tree [3]). Alternatively, they may allow the user to provide an elaborate configuration file—e.g. MrBayes [4, 5].

In the last decade, advances in statistical phylogenetics software have paved the way for more flexibility and generality. Key developments include the adoption of XML-based model specifications in BEAST [6], and the introduction of a model description language grounded in probabilistic graphical models in RevBayes [7, 8]. Very recently, we have seen additional phylogenetic modeling languages introduced [9, 10]. There has also been a growing interest in leveraging generic statistical modeling and inference frameworks for the analysis of phylogenetic models [11, 12].

Probabilistic programming languages. Meanwhile, at the intersection of computer science and statistics, a class of tools called *probabilistic programming languages* (PPLs) has matured [13, 14, 15]. The purpose of a program written in a PPL is not its execution per se, but to implicitly specify a probability distribution, i.e., a statistical model, on the output of the program. In Bayesian inference (also called probabilistic inference), we would like the program to specify the posterior probability distribution, that is, the probability distribution of a model conditioned on some observed data. A PPL typically includes two special probabilistic constructs: `assume` statements, which describe latent random variables in the model, and `observe` statements, which condition random variables in the model on observed data. These statements make it possible for generic inference algorithms, such as sequential Monte Carlo and Markov chain Monte Carlo algorithms, to identify checkpoints that enable them to generate and manipulate simulations from the posterior probability distribution. This means that a user can ideally focus on describing the model, and leave the estimation of the posterior probability distribution to general-purpose inference algorithms developed by statisticians and computer scientists. Given a sufficiently powerful programming language, PPLs can express essentially any probabilistic model regardless of the application domain. The modeling flexibility and the potential for automatic inference has inspired an enormous amount of PPL work across a wide range of scientific disciplines in recent years. To cite a few applications, probabilistic programming languages have been used in affective computing (the study of human emotion with computers) [16], in data cleaning [17], and in phylogenetics [18],

PPLs separate modeling from inference. In the conventional approach, the initial step involves the derivation of an inference algorithm for a specific model, followed by its implementation using a classical programming language. In the PPL approach, in contrast, the model specification is separated from the implementation of the statistical inference machinery, which is automatically provided by the PPL compiler [19]. This opens up the possibility, for example, for the user to try several different inference strategies for the problem at hand, and select the best one without modifying the problem description [20, 21, 22]. Another important benefit of not having to re-implement the inference machinery for every new model is that it reduces the risk of implementation errors. That is, by reusing an existing and well-tested inference algorithm implementation, we can have higher confidence in the inference result being correct. This would minimize the risk of prolonged arguments over the correctness of inference for complicated models, as exemplified in statistical phylogenetics by the BAMM debate [23, 24, 25, 26].

While the PPL approach comes with multiple benefits for domain experts, it poses considerable challenges for the developers of the inference machinery. Indeed, it was not until efficient automatic inference techniques were applied to the most powerful PPLs about a decade ago that the interest in using the PPL approach started to pick up significantly [27]. While the last years have seen an explosion in the development of these techniques, and all techniques (if implemented correctly) result in correct inference for all models, it is still far from guaranteed that the inference machinery is efficient for a particular model, especially if the model is complex. Nonetheless, we are now at a stage of development where automatic inference strategies are efficient enough for PPLs to be used productively in addressing many scientific problems, particularly if the model description is structured in a way that is favorable and the inference compiler implements the tool set required to address the specific challenges posed by the model. The coming years will undoubtedly see rapid progress in addressing many of the current constraints, eventually supporting the PPL vision of automatic inference for all models, regardless of how they are described.

Using PPLs in phylogenetics. Phylogenetic models are complex, and covering the entire model space requires a powerful modeling language. In particular, expressing many common phylogenetic models, such that the entire model becomes available to automatic inference algorithms, requires a language that is universal in the sense that it can express the structure of a probabilistic graph as a random variable. This requires stochastic control flow statements, such as stochastic branching, stochastic recursion or unbounded loops; we will see several concrete examples of this later in the paper. For instance, in a diversification analysis, we are often interested in inferring speciation and extinction rates conditioned on a so-called reconstructed tree, that is, an observed tree of extant (now living) lineages. This requires that the automatic inference algorithms can integrate out any unobserved side lineages that died out before the present. A universal PPL would express such a problem as a simulation over possible extinct side lineages, using stochastic constructs to represent speciation and extinction events in those side lineages, so that the automatic inference machinery can integrate them out [18]. To the best of our knowledge, existing modeling languages designed specifically for phylogenetics are not universal in this sense (see unreleased work on BALI-Phy `https://www.bali-phy.org/models.php` for a possible exception). Whether or not one wishes to call such languages PPLs is a matter of personal preference; in the following, we will restrict the term PPL to universal languages.

It has become increasingly clear in recent years that automatic and efficient inference for phylogenetic models, especially strategies that scale up to realistic problem sizes, requires a set of special techniques, which are not widely available in existing PPLs. For instance, integrating out random variables, where possible, is often important for efficient inference. This can be done using an approach called delayed sampling, which automatically identifies conjugacy relations in the model [28]. Diversification models and similar phylogenetic problems require so-called alignment of checkpoints for inference to be efficient [18]. Strategies for automating checkpoint alignment and exploiting it in SMC and MCMC inference have only been described very recently, and are only available in the Miking PPL framework currently, as far as we are aware [29]. Another automatic inference technique, which is not yet widely available but can lead to considerable efficiency improvements for some models is the alive particle filter (APF) [30]. APF ensures that automatic SMC algorithms sample more efficiently over models where some outcomes are impossible. Diversification models conditioned on a reconstructed tree represent a good example, as they condition the simulation on side lineages going extinct. Thus, a side simulation surviving to the present has to be discarded, decreasing the efficiency of SMC inference unless the APF is used. Finally, many phylogenetic problems are now conditioned on very large data sets, making it important to leverage parallel computing power where possible [19]. Until now, no PPL has provided a complete set of these improved inference techniques.

The Miking CorePPL framework. Bayesian statistical phylogenetics typically relies on Monte Carlo algorithms: Markov chain Monte Carlo (MCMC, e.g. [31, 5, 1]) or, more recently, sequential Monte Carlo (SMC, e.g. [30, 32]). In the last few years, optimization or hybrid techniques have also been explored, such as variational combinatorial SMC [33] and variational Bayesian phylogenetic inference (VBPI) [34, 35]. Monte Carlo methods, in particular, rely on clever and efficient ways of sampling or simulating from the target distribution. In order to implement an inference algorithm in a PPL compiler, facilities must be provided by the compiler for the execution of the probabilistic program to be suspended, likelihood updates to be made, and execution traces (different program runs with random variables and stochastic choices instantiated) to be resampled. To accomplish this, existing PPLs such as WebPPL [21] rely on a technique called full continuation-passing style (CPS) transformation [36, 21, 37], which comes with a performance penalty. In order to overcome this drawback, a partial CPS transformation was investigated in [20]. The results show that selective CPS offers up to twice the performance gain compared to full CPS for MCMC and SMC inference algorithms on both phylogenetic models, such as the cladogenetic diversification shifts (ClaDS [38]) and constant rate birth-death (CRBD [39, 40]) models, and non-phylogenetic models, such as latent Dirichlet allocation (LDA, [41]). Since the CPS transformation is a fundamental feature of a compiler, a novel compiler and intermediate language (Miking CorePPL) needed to be developed with selective CPS at its core. Miking CorePPL [19] is in turn implemented within the general compiler framework called Miking [42]. See `https://miking.org/` for details.

A further distinguishing feature of Miking CorePPL is that it has a modular way of expressing Monte Carlo inference algorithms, with most of the compiler being agnostic as to the particular inference engine. Thus, it is easily extensible with new optimizations and inference strategies. It already supports importance sampling, the bootstrap particle filter (BPF) and the alive particle filter (APF) [30], aligned lightweight MCMC [29, 43], trace MCMC, naive MCMC, and particle MCMC–particle independent Metropolis-Hastings (PMCMC-PIMH, [44]). However, as an intermediate language, Miking CorePPL is not designed to be used by end-users unfamiliar with functional programming and without a deep interest in compiler optimizations. Instead, the intention is that *domain-specific languages* should be designed and built on top of Miking CorePPL, to enable user-friendly environments for specific domain users. TreePPL is one example of such a domain-specific language, a probabilistic programming language designed specifically for phylogenetics problems and with evolutionary biologists in mind as end users.

TreePPL. TreePPL is an ambitious project bringing the methods discussed above (universal probabilistic programming and powerful, flexible, and extensible Monte Carlo inference) to a broader audience in the field of statistical phylogenetics. It aims to provide a front-end: a syntactic layer, as well as library tools and documentation, which will enable practitioners to write probabilistic programs using the advanced Miking CorePPL inference in the back-end. It focuses on providing maximum expressive power (universality), automatic inference suitable for phylogenetic problems, and a simple syntax that would be familiar to computational biologists.

With the publication of this paper, we are inviting the community to try TreePPL. The TreePPL project, at its core, is a project in human-computer interaction. It aims to understand the best practices that would lead to the adoption of universal probabilistic programming in phylogenetics, and to test whether universal probabilistic programming will result in an accelerated pace of scientific discovery in the model-rich field of statistical phylogenetics. In Section 2, we introduce the TreePPL programming environment. In Section 3, we use three examples to illustrate that TreePPL can already push the envelope in statistical phylogenetics. Finally, in Section 4, we wrap up our presentation with a look towards the future.

2 TreePPL

In this section, we describe the TreePPL language and environment. We begin by an overview of the involved components and how they inter-relate (Section 2.1). We continue with an example (Section 2.2) that we will use throughout the rest of the chapter to illustrate the workflow. We then illustrate the necessary pre-processing and post-processing steps as they pertain to the example (Section 2.3). As the central point of the exposition, we dive into the core of probabilistic programming with TreePPL (Section 2.4) and a discussion of optimizations and advanced inference ideas (Section 2.5). In the end, we highlight some specific language features and point users to locations for further learning (Section 2.6)

2.1 Workflow and architecture

To set up the TreePPL language and environment, please refer to our official website at `https://treeppl.org`. TreePPL consists of the following components: the TreePPL compiler and standard library, a Python package, an R package (in development), and a syntax highlighting extension for Visual Studio Code (in development). Although TreePPL can be used as a stand-alone tool, it is designed and intended to be used together with Python or R. In Fig. 1 we illustrate the development workflow and the software architecture of TreePPL.

A TreePPL analysis typically requires the creation of two files by the user: a model file in the TreePPL language (indicated by the "TreePPL program" grey box in Fig. 1) and a script in Python or R (indicated by the grey box "Python or R script" in Fig. 1). The model program only contains the specification of the problem and not a statistical inference procedure. The Python or R script file is responsible for:

1. *Preparing data:* utilize Python or R to pre-process and format biological data, such as sequences in the FASTA or the Nexus format.
2. *Running the model and processing its output:* the Python or R environments, using the provided packages, manage the execution of the TreePPL model, handling tasks like data visualization and output processing. Communication is done using the JSON format.

In the left side of Fig. 1, we illustrate the stages through which a TreePPL program goes until an executable is produced. The TreePPL compiler `tpplc` operates in stages: during the initial compilation, the `tpplc` front-end translates the high-level (phylogenetic) model into the intermediate representation, a Miking CorePPL program. In turn, the CorePPL compiler, `cppl`, continues compiling the program by applying optimizations [29, 20]; some optimizations are applied automatically, while others can be controlled by the user or the Python or R environments. The ones that can be set are:

1. Alignment analysis and placement of resampling points (see checkpoint discussions in Sections 2.5, 3.1), three possibilities:
   (a) Where applicable, resample immediately after each checkpoint;
   (b) Resample after aligned likelihood updates;
   (c) Manual resampling by supplying inference hints;
2. CPS transformation, variants: none, partial, or full (see Section 1);
3. Static delayed sampling (in development, see Section 2.5).

CorePPL then adds code for the selected inference strategy: one of importance sampling, bootstrap particle filter, alive particle filter [30], aligned lightweight MCMC [29, 43], trace MCMC, naive MCMC, or particle MCMC–particle independent Metropolis-Hastings (PMCMC-PIMH, [44]). The code obtained in this way is converted either into C++ (upcoming feature) or pure Miking code, which is subsequently processed by `nvcc` or `mi` to obtain respectively a CUDA-enabled executable (upcoming feature) or a regular executable.

2.2 Example: diversification models

We will now briefly introduce diversification models, and use these as an example in the rest of this section. Diversification models are an important class of problems in evolutionary biology, and they describe the speciation and extinction processes that generate an evolutionary tree. In these problems, the observed data usually come in the form of a rooted binary tree. The leaves represent current biological taxa, while each node signifies the common ancestor of its descendants. The root node is known as the most recent common ancestor (MRCA). The objective is to estimate the unknown parameters of the diversification process (the speciation-extinction process) that most likely generated the tree. In a very simple case, the constant rate birth-death model (CRBD), we want to estimate the unknown (latent) parameters λ (speciation rate) and μ (extinction rate). Introducing the second parameter μ already presents significant challenges for the solution of the model, as it implies that the tree that we are observing in the present ("reconstructed

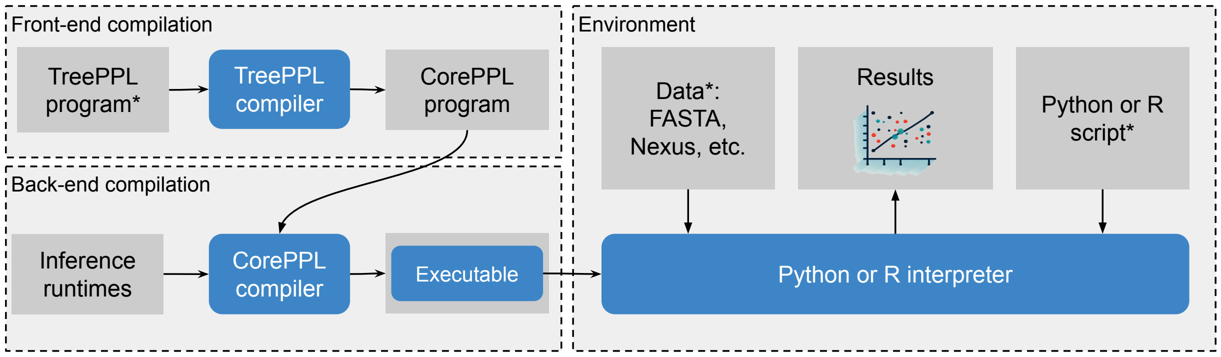


Figure 1: Illustration of the TreePPL programming environment and compiler. The grey boxes represent source code files or data, acting as inputs to or outputs from various workflow stages; the blue boxes represent the compilers or the interpreters involved acting on the source code and data. With * we have indicated the components that the user supplies, all the others are generated by the programs or supplied with the compilers. The executable is a special case, as it is both the output of the `cppl` compiler and *its output* is processed by the Python or R interpreter to generate plots and do other analysis tasks.

Listing 1: CRBD model implementation in TreePPL/Python (Python part)

```
1  import treeppl
2  import numpy as np
3  import pandas as pd
4  import matplotlib.pyplot as plt
5  import seaborn as sns
6
7  alcedinidae = treeppl.Tree.load("trees/Alcedinidae.phyjson", format="phyjson")
8  samples = None
9  with treeppl.Model(filename="crbd.tppl", samples=10_000) as crbd:
10     for _ in range(1_000):
11         res = crbd(tree=alcedinidae)
12         samples = pd.concat([
13             samples,
14             pd.DataFrame({
15                 "lambda": res.subsample(10),
16                 "lweight": res.norm_const
17             })
18         ])
19         plt.clf()
20         sns.kdeplot(
21             data=samples,
22             x="lambda",
23             weights=np.exp(samples.lweight - samples.lweight.max()),
24         )
25         plt.pause(0.05)
```

tree") is only part of the full tree, with some unobserved sub-trees having gone extinct in the past. Even though in this particular simple case, an analytical solution for the likelihood of the reconstructed tree, conditioned on the time of the MRCA and integrating out hidden branches, is known (see e.g. the supplementary information in [18]), for most problems of the class (e.g ClaDS [38], BAMM [24], TDBD [45], LSBDS [46]; also see Section 3.1) we need to rely on simulating the latent evolutionary lineages and conditioning on them going extinct before the present. Thus, the CRBD example used here works as a template that can be extended to implement a wide range of complex diversification models [18]. Before describing in detail how to implement CRBD in a TreePPL program, let us consider how such a program would be used.

2.3 Pre- and post-processing with Python or R

TreePPL offers a Python package[2] that serves as a versatile interface for creating and executing inference on TreePPL programs. This approach offers users the advantage of working with input data and processing inference results using a familiar language that comes with a wide range of pre- and post-processing capabilities. These include popular libraries, such as `Biopython`, `Numpy`, `Pandas`, `Matplotlib` and `Seaborn`. An R package with similar functionality is under development and will be released in an upcoming software update.

Listing 1 exemplifies the use of the Python interface for leveraging a CRBD model implemented in TreePPL to estimate the posterior distribution of the speciation rate.

The Python program first reads in the observed tree data from a PhyJSON file (line 7; other common file formats, such as Nexus and Newick, are also supported through `Biopython`).

Next, a `treeppl.Model` instance is created (line 9). This step involves specifying the name of the TreePPL model file (`filename`) and the number of samples to be returned during the inference (`samples`). The instantiation process encompasses the reading and compilation of the TreePPL model. Note that various other parameters are supported, including specifying the source code as an inline string (`source`), alternative inference methods (`method`; `smc-bpf` by default) and additional arguments for the `tpplc` compiler.

The model instance can be called directly (line 11) to perform inference on the TreePPL model, passing the input data as named arguments aligned with the TreePPL model function's parameters. This process involves encoding of the provided input data into the JSON format, which is then passed to the compiled model executable. Subsequently, the

[2]https://github.com/treeppl/treeppl-python

output of the model executable, also in JSON format, is parsed. The resulting output includes samples, weights, and the marginal likelihood, which are returned as a `treeppl.InferenceResult` instance.

In this example program, the SMC-based inference is repeated $1\,000$ times (line 10). In each iteration, 10 samples of the $10,000$ SMC particles (line 15) are added to a Pandas data frame (lines 12–18) to reduce the total memory footprint. The collected subsamples are then used to update the estimate of the posterior distribution shown to the user (lines 20–24). By repeating the inference many times, as in this example, we can estimate the accuracy of the inference result.

As shown in this example, users can use Python visualization libraries to plot the resulting posterior distributions. In Fig. 2 we have repeated the analysis twice with two different inference methods—the alive particle filter [47] and the bootstrap particle filter [48] (achieved via changing the arguments on line 9).

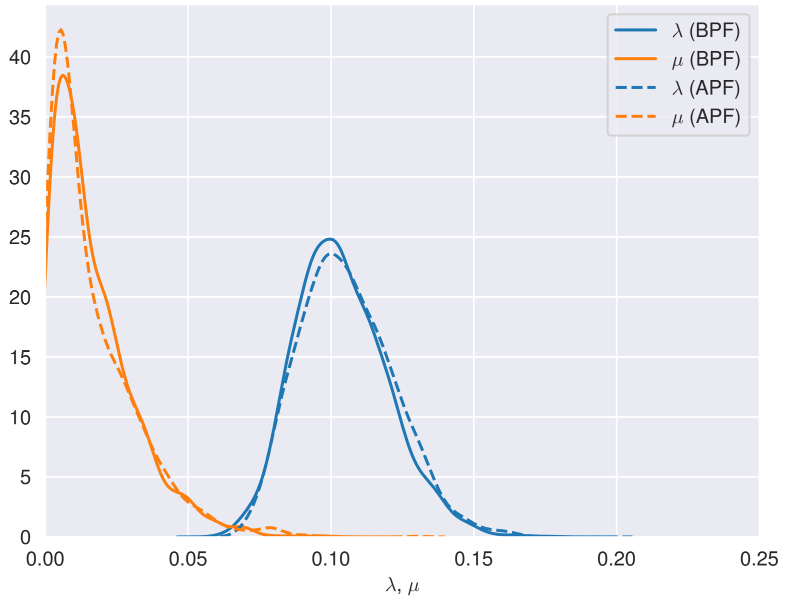


Figure 2: Posterior distributions of the speciation rate λ and extinction rate μ under the CRBD model inferred using two different inference methods (`smc-bpf` and `smc-apf`).

2.4 Writing a TreePPL program

In this subsection, we explain the main features of a TreePPL program using the CRBD example. Our aim is to provide the reader with the fundamentals of writing a probabilistic program in TreePPL. For a more extensive discussion of probabilistic programming techniques, we refer to [21, 15], and for a comprehensive TreePPL language reference to our official website.

A useful way to think about the meaning of a probabilistic program is that it describes a Bayesian data distribution in a *generative way*. In Monte Carlo-oriented PPLs, such as TreePPL, running the program many times generates a valid sample from the posterior distribution. This repetitive execution is done automatically by the inference engine supplied by the compiler.

Let us use θ to refer to the random variable of the latent model parameters and use y to refer to the data. In both cases they may be multi-dimensional. You write a probabilistic program as a recipe, which generates samples from (simulates from) the prior distribution of the latent parameters $p(\theta)$ (using `assume`-type statements, details in Section 2), and then constructs in a programmatic way the data distribution $p(y|\theta)$ (using `observe`-type statements, details in Section 2); the posterior distribution of the latent parameters $p(\theta|y)$ (specified with the `return` statement) and, optionally for some algorithms, the marginal likelihood $p(y)$ emerge as a result of the inference (Eq. 1).

$$\overbrace{p(\theta|y)}^{\text{Posterior distribution}} = \frac{\overbrace{p(y|\theta)}^{\substack{\text{Data distribution /}\\ \text{likelihood}}} \; \overbrace{p(\theta)}^{\text{Prior distribution}}}{\underbrace{p(y)}_{\text{Marginal likelihood}}} \tag{1}$$

Listing 2: CRBD model implementation in TreePPL/Python (TreePPL part)

```
 1 function simulateSubtree(time: Real, lambda: Real, mu: Real): () {
 2   assume waitingTime ~ Exponential(lambda + mu);
 3   if waitingTime > time {
 4     weight 0.0;
 5     resample;
 6   } else {
 7     assume isSpeciation ~ Bernoulli(lambda / (lambda + mu));
 8     if isSpeciation {
 9       simulateSubtree(time - waitingTime, lambda, mu);
10       simulateSubtree(time - waitingTime, lambda, mu);
11     }
12   }
13 }
14
15 function walk(node: Tree, time:Real, lambda: Real, mu: Real): () {
16   assume waitingTime ~ Exponential(lambda);
17   if time - waitingTime > node.age {
18     simulateSubtree(time - waitingTime, lambda, mu);
19     weight 2.0;
20     observe 0 ~ Poisson(mu * waitingTime);
21     walk(node, time - waitingTime, lambda, mu);
22   } else {
23     observe 0 ~ Poisson(mu * (time - node.age));
24     if node is Node {
25       observe 0.0 ~ Exponential(lambda);
26     }
27     resample;
28     if node is Node {
29       walk(node.left, node.age, lambda, mu);
30       walk(node.right, node.age, lambda, mu);
31     }
32   }
33 }
34
35 model function crbd(tree: Tree): Real {
36   assume lambda ~ Gamma(1.0, 1.0);
37   assume mu ~ Gamma(1.0, 0.5);
38   walk(tree.left, tree.age, lambda, mu);
39   walk(tree.right, tree.age, lambda, mu);
40   return lambda;
41 }
```

When statistical inference is invoked, the return object (in JSON format in the case of TreePPL) contains samples from the posterior distribution and may, under certain inference scenarios (e.g. SMC), also contain the normalizing constant or the marginal likelihood of the data integrated over the priors.

Let us turn this into practice. Consider now Listning 2 that shows a CRBD model expressed as a TreePPL program. In such a program, we can have many functions, where one function must be the main entry point, the *model function*. Specifically, this program has three functions: `crbd`—this is the model function, which takes care of input and output and of calling the helper functions; `simulateSubtree`—simulation of the hidden "data-augmented" evolutionary lineages; and `walk`—the function conditioning the execution on the actually observed input data tree.

Throughout the program we made use of some of the following constructs specific to probabilistic programming:

- Constructs associating a *random variable* appearing in the program with a specific distribution: `assume` (used on lines 2, 7, 16, 36, and 37), and `iid`, a construct indicating a sequence of independent and identically distributed random variables. This latter construct is not shown in the example. Please refer to the language documentation for guidance on usage.
- Constructs for updating the weight of the computation based on *observed data*: `observe` (lines 20, 23, 25), `weight` (lines 4 and 19), and `logWeight` (not shown in the example).

These program points marked with `assume`, `observe`, etc., are called *checkpoints*. They are essential for compiling different inference algorithms efficiently [19].

Now, let us look at the individual functions.

Model function. The prior for the speciation rate λ and the extinction rate μ are specified on lines 36 and 37, whose posteriors are shown in Fig. 2. The observed data is the phylogenetic tree, which is input through the identifier `tree` (refer to previous section). The `walk` function is invoked to condition the simulation on the input tree. Finally, the `return` statement (line 40) specifies which latent variables are interesting to us, and will be reported by the program.

Function `walk`. This function traverses the observed tree (lines 21, 29 and 30) and simulates unobserved speciation events (in which only one descendant survives to the present day). For each unobserved speciation event the program calls the `simulateSubtree` function to simulate the evolution of the descendant destined for extinction (line 18; more details provided in the next paragraph). The observe statements on lines 20 and 23 are associated with the absence of extinction events along the observed tree, while the statement on line 25 pertains to the observed speciation events (in which both descendants survive until the present day).

Function `simulateSubtree`. In our CRBD example we have refrained from using the analytical expressions for the likelihood of the data-augmented tree and instead implemented the hidden tree simulation in order to illustrate universality. The purpose of the function `simulateSubtree` on lines 1–13 in Listing 2 is to simulate a birth-death process starting from a hidden speciation event in the past, thus augmenting the reconstructed tree with a simulated lineage; the simulation *must* result in the side-tree going extinct in the past: otherwise we would have observed its surviving species in the data.

The function `simulateSubtree` is a recursive function designed to model random waiting times until the next event (line 2). In the event of a speciation, it calls itself on lines 9 and 10, simulating evolutionary processes for both descendant lineages. If the next event is projected to occur in the future, the simulation is deemed incompatible with the observed data (by setting the weight to 0 on line 4). Note that the recursion itself is controlled by a random process—namely, whether the next event is a speciation or not is modeled using a Bernoulli distribution with probability $\lambda/(\lambda + \mu)$ (line 7). Thus, the recursion depth cannot be known in advance of actually running the simulation. The fact that we are allowed to introduce new random variables in a recursive function—where the number of invocations is unknown a priori—is the essence of *universality*.

2.5 Optimizations and inference

In order to make the program more efficient, the compiler will attempt to optimize the code based on the settings provided by the user (see Section 2.1). However, the model program itself can be structured in a way that may result in efficiency gains.

Factorizing the likelihood. Since `observe`-type statements indicate a potential likelihood update (a checkpoint—see [29] for an in-depth discussion), another way of thinking about TreePPL (and PPLs in general) is that it provides a way of specifying the likelihood function $\mathcal{L}(\theta; y)$ in a programmatic way, instead of using a closed-form expression. In fact, we do not have to explicitly construct the data distribution and then observe from it—we can instead directly factor in the likelihood via `weight` or `logWeight` as we have shown on lines 4 (ensuring that the side-branch dies) and line 19 (correcting for the rotation factor of the tree).

Regardless of whether we use `observe` or directly request a likelihood update, more than one such "conditioning" checkpoint is allowed to exist in a probabilistic program. This technique has been used throughout the Listing 2 on all lines that have `observe` or `weight`. Factorizing the likelihood in several parts is encouraged as it allows for more sophisticated inference algorithms to be utilized.

If SMC is used as the inference algorithm, one key optimization that can be performed by the compiler is to automatically decide where to do resampling in the program. This alignment analysis will be performed by the TreePPL compiler (see [29, 20] for details, also Section 3.1).

Delayed sampling and belief propagation. In the PPL literature, `assume` statements are also called `sample` statements because they often involve direct sampling of the random variable. However, in many cases it is advantageous not to sample the value of the random variable directly, justifying the syntax adopted by TreePPL. Instead, one can either represent the possible values as a probability vector over outcomes (for numerical integration or summation), or defer sampling until it is possible to determine whether any conjugacy relations allow the variable to be integrated out analytically. The former is closely related to belief propagation, an example of which is Felsenstein's pruning algorithm

[49], frequently used in statistical phylogenetics. The latter is a more recent technique in PPL inference algorithms, referred to as delayed sampling [28], although the exploration of conjugacy relations is as old as Bayesian inference itself. Both methods are known to dramatically accelerate the performance of inference algorithms in many cases. Conjugacy-utilizing optimizations are ideally done by the compiler, and the end-user does not have to worry whether the generative model described with `assume` actually samples the random variables, or instead uses a different technique to achieve the desired outcome. Automatic application of delayed sampling and Felsenstein's pruning algorithm are both work in progress in TreePPL.

Extending the inference. Inference can be controlled by selecting an inference strategy and changing algorithm settings using the Python or R environment, as illustrated above. If the user wants to implement a novel inference strategy, this can be done by writing inference code at the Miking CorePPL level. In order to do this, the user must implement both a runtime library (cf. Fig. 1) and a compiler routine for the inference strategy [20]. There are a number of reusable Miking CorePPL compiler and runtime components which the user may take advantage of in their inference strategy implementation. Some examples of such components are continuation-passing style transformations and automatic alignment. The user may also use existing inference strategy implementations as templates. Please see `https://miking.org/` for details.

2.6 Other TreePPL language features

Many TreePPL language features should be familiar to users with some previous programming experience. For instance, the syntax follows a brace-bracket style in the tradition of C, C++, etc. We can also comment in the usual C-style with `//` or `/* ... */`. We refer the reader to the TreePPL web site for a detailed language reference, and a set of examples of both general modeling and phylogenetic modeling in TreePPL. Here, we just highlight a couple of powerful language features, which may be unfamiliar to potential users but that facilitate the description of phylogenetic models and the application of automatic inference.

Functional programming and immutability. The TreePPL language is essentially a functional programming language. From a practical coding perspective, this means that functions cannot have side effects except for printing and changing the weight (probability) of a simulation. Also, the code binds variable names to values, and these bindings cannot change when running the program (immutability). Both features greatly facilitate the application of automatic inference algorithms. Most computational biologists should be somewhat familiar with the functional programming style, as it is supported in both R and Python, even though the enforcement of immutability may require some adjustment of coding practices.

Algebraic data types. TreePPL uses a flexible system for data types, allowing users to define their own composite types. For instance, consider the representation of phylogenetic trees, a type provided in the standard library:

Listing 3: Definition of the `Tree` type in the standard library

```
type Tree =
  | Node {left: Tree, right: Tree, age: Real}
  | Leaf {age: Real}
```

The `Tree` type is a so called *algebraic data type (ADT)* and is defined using two constructors: `Node` and `Leaf`. The `Node` constructor encapsulates a structure with two sub-trees (`left` and `right`) and an associated age. Conversely, the `Leaf` constructor represents terminal nodes with only an age attribute.

The specific type of a particular instance of an ADT can be checked using the `if ... is` construct (seen line 24 in the CRBD example). This is a form of *pattern-matching* and checks if the instance is matched by a particular constructor. Pattern-matching allows different components of a phylogenetic tree to be handled in a natural way in TreePPL programs.

The basic Tree type can easily be extended by the user to cover more complex tree data structures with different model variables associated with its nodes and branches. New complex ADTs can be created from scratch by the user. The incorporation of ADTs in TreePPL provides a versatile framework for modeling intricate biological data structures, ensuring that the language remains both semantically rich and syntactically concise.

Type annotation. TreePPL is an explicitly typed language and requires the user to specify types in function signatures using type annotation. Specifically, the type is added after the parameter or the function signature, separated using a colon, as demonstrated on line 1 in the CRBD example. Static typing using type annotation enables good error messages, readability, and certain compiler optimizations.

3 Scientific Modeling in TreePPL

In this section, we give three examples of modeling and inference in TreePPL. The examples represent very different types of phylogenetic models, and span a large portion of phylogenetic model space. They demonstrate a range of TreePPL modeling and inference features, and also highlight some challenges that remain for future work. The examples are by no means exhaustive. Users who want to describe their own models and use automated inference for them are encouraged to clone the TreePPL GitHub repository and explore the full set of examples provided there.

3.1 Diversification models

The CRBD model script used earlier in this paper forms a useful basis for exploring more complex diversification models, for which it is challenging to develop correct and efficient inference machinery. The basic techniques are the same as those recently used to analyze a range of diversification models accommodating different types of variation across lineages in diversification rates in the WebPPL and Birch PPLs [18]. An example of a further model, more complex than CRBD, implemented in the native TreePPL syntax is ClaDS [38]. For brevity's sake we refer the reader to the `models/phylo/clads` directory in the source code repository for the code. The difference between ClaDS and CRBD is that at each speciation (both observed and hidden), we propose a shift in the speciation and extinction rates, but the general template on implementing it in TreePPL is the same.

An interesting challenge in applying PPLs to diversification models is the alignment problem [18, 29]. In an SMC setting, a resampling step is a step where some "unpromising" execution traces are pruned. Intuitively, it seems reasonable to compare execution traces when they have reached the same point in the simulation, that is, the same checkpoint in the code. Indeed, empirical results show that this is often considerably more efficient than comparing execution traces that are not aligned [29]. However, finding the checkpoints that are always present and comparable across executions is not straightforward. For example, in Listing 2, the checkpoint on line 19 may occur or it may not occur depending on the outcome of the random event on line 16. Similar for lines 20, 23, and 25. It makes sense to wait with resampling until we reach line 27 because we can ensure that all executions pass through this point in the code. Unlike WebPPL and Birch, where alignment points have to be manually encoded in the PPL program, the TreePPL compiler automatically finds these checkpoints. Thus, the user does not have to worry about alignment. Due to automatic alignment, TreePPL provides efficient inference strategies for many diversification models, which would otherwise have been intractable for automatic PPL inference algorithms.

The application of PPLs to diversification models made SMC inference algorithms available for these models for the first time [18]. Because SMC generates an estimate of the model likelihood as a byproduct, this opened up the possibility of resolving existing debates concerning the performance of different models in explaining empirical data using Bayesian model comparison. The PPL framework also opens up the possibility of exploring a number of interesting variants of current models such as the birth-death shift model [50], for which a PPL implementation exists and is in preparation.

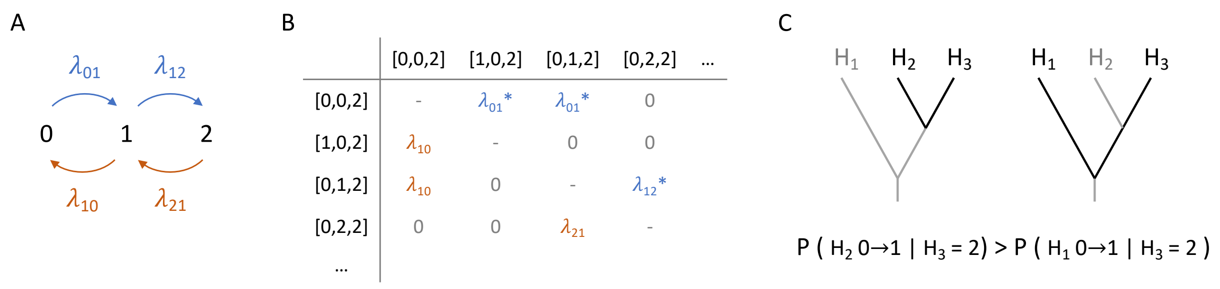
3.2 Host repertoire model

The host repertoire model is arguably one of the most realistic models considered to date for the evolution of host-symbiont associations [51]. Given a set of host taxa, the host repertoire of a symbiont taxon is represented by an integer vector, where each element describes the interaction of the symbiont with each host taxon. For instance, one may consider three levels of interaction—non-host, potential host and actual host—as originally proposed when the model was introduced [51]. One can also consider a simpler model with just two levels of interaction—non-host and host—or more complex scenarios with more than three levels of interaction. Given a time-calibrated symbiont tree and a matrix of observed symbiont-host interactions as data, the model describes how the host repertoire evolves along the symbiont tree through changes in the level of interaction with each of the host taxa included in the interaction matrix.

Because ancestral host repertoires can include multiple hosts, this model can capture a wide range of complex scenarios of host-symbiont co-evolution, from strict specialization in one-to-one interactions to complex pulses between generalist and specialist phases in the evolution of host ranges. Finally, with this model, one can also test if the probability of gaining a new host depends on the phylogenetic distance between the new host and the host(s) used at the time of the event. Originally proposed to study the evolution of butterfly-plant associations, this model is now being used for a variety of host-symbiont systems, for instance in analyses of beetle microbiomes [52] and parasitic mussels in fishes [53].

The host repertoire model poses several interesting modeling and inference challenges. First, the number of possible host repertoires is huge. For a model with n hosts and three different levels of host interaction, there are $3^n - 1$ possible host repertoires. The rate of change from one repertoire to another is computed from the loss and gain rates for the different levels of host interaction, potentially taking into account the phylogenetic dissimilarity between hosts (Fig. 3).

The size of the rate matrix means that we cannot integrate out ancestral host repertoires and changes between them for realistic-size problems, like we would do, say, for a standard substitution model describing the evolution of nucleotides in a DNA sequence. Instead, we need to sample over ancestral repertoires and histories of change between them.



	[0,0,2]	[1,0,2]	[0,1,2]	[0,2,2]	...
[0,0,2]	-	λ_{01}*	λ_{01}*	0	
[1,0,2]	λ_{10}	-	0	0	
[0,1,2]	λ_{10}	0	-	λ_{12}*	
[0,2,2]	0	0	λ_{21}	-	
...					

Figure 3: A. In the host repertoire model, different loss and gain rates are used to describe the rates of moving between levels of host interaction. B. A section of the rate matrix for a repertoire with three hosts, where all possible events have one of the four rates shown in A. Asterisks mark the gain rates, which are modified by a function of the phylogenetic distance between hosts. C. Phylogenetic distances between hosts. Since H2 is closer to H3 (the only host being used), the probability of gaining H2 is higher than gaining H1.

Second, it is challenging to sample ancestral host repertoires and change histories directly from the full model. It is more natural to sample from a simpler model, the independence model, in which host interactions evolve independently from each other. This is the same as sampling from n independently evolving discrete characters, each representing a different host. Each character would have three possible states corresponding to the different levels of host interaction. We can sample from this model, conditioned on the tip states, using well-known algorithms [54]. This is done by drawing from the appropriate probability distributions using *assume* statements (Fig. 4). We then weight the sampled history according to its probability under the full model, factoring out the probability of the proposed state under the independence model. This is accomplished using *weight* statements, as illustrated in line 21 of Fig. 4A.

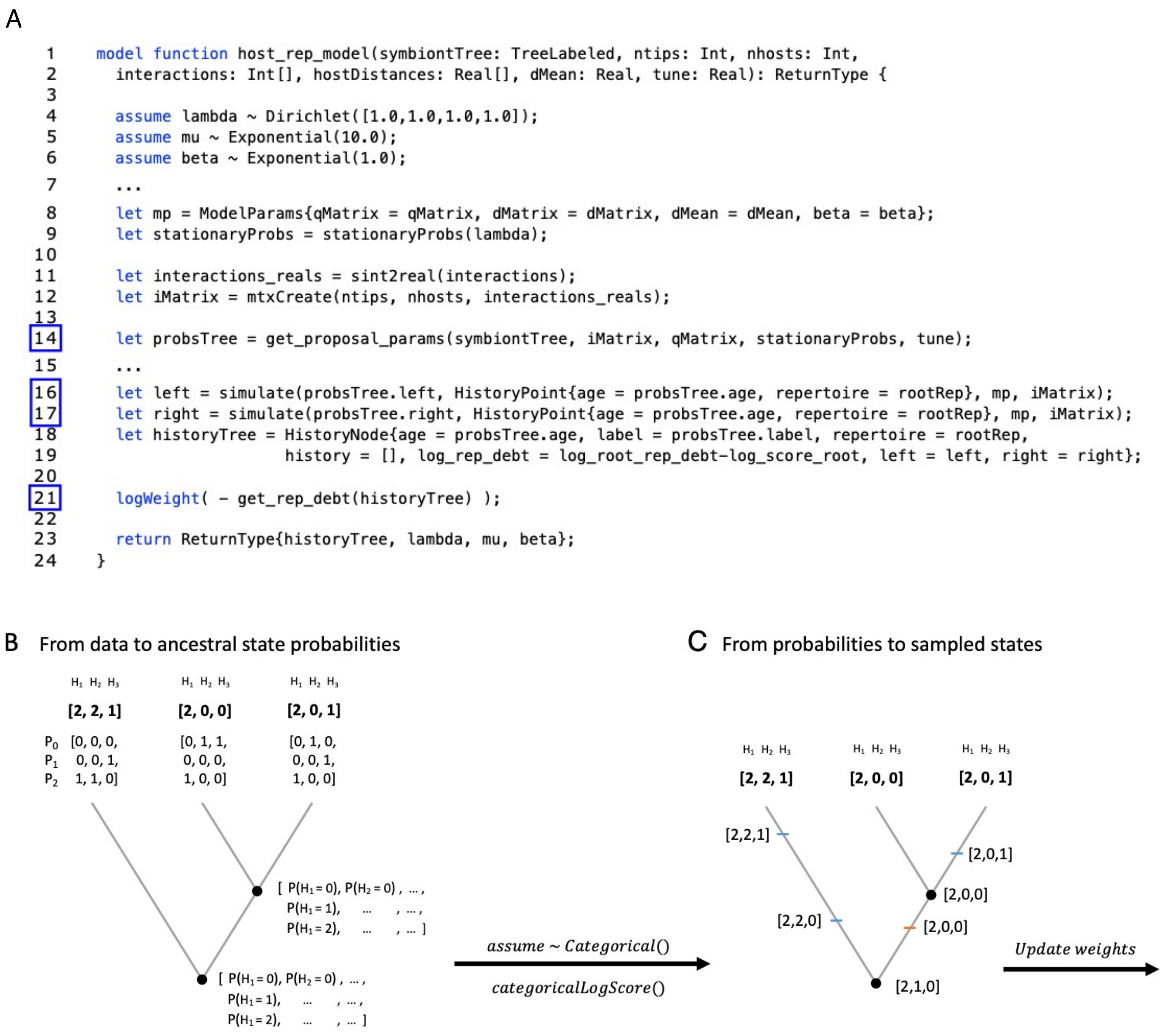
Because of the complexity of the host repertoire model, it is quite challenging to find efficient inference strategies. The only current implementation uses dedicated Markov chain Monte Carlo (MCMC) algorithms coded in C++ in the back end of the RevBayes platform [51]. Unfortunately, the host repertoire model is hard for the MCMC algorithm because of the difficulties of moving from one plausible history of repertoire change to another. Once the model is described in TreePPL, however, we get access to the full range of TreePPL inference strategies. They include sequential Monte Carlo (SMC) algorithms, which do not depend on moving between points in model space. TreePPL also supports mixed strategies involving both SMC and MCMC steps, and opens up for future inference strategies developed for all TreePPL models.

By modifying the TreePPL description of the model, we can easily extend the original model. For instance, one can associate an observation of the host repertoire of a leaf in the symbiont tree with some uncertainty about the true level of interaction with each host, make different assumptions about the root state (for example, assuming stationarity or not), or change the number of host interaction levels in the model.

3.3 Tree inference

To describe the problem of tree inference in a PPL, such that the code lends itself to automated inference using generic methods, we use stochastic recursion. The core idea is to control a recursive function using a random variable, such that successive iterations generate a valid draw from the prior probability distribution over tree space.

The PPL code accomplishing this can be structured as a forward simulation, in which the tree is generated from the root and forward in time until it is complete with all the observed leaves. It can also be structured as a backward simulation, in which one starts with all the leaves and simulates coalescent events backward in time until the root is reached. The latter, the coalescent process, is a more convenient description for our purposes, because it can be used to condition a simulation on observed data earlier than would otherwise be possible. For instance, in an SMC algorithm, the backward simulation makes it possible to weight a particle describing a subtree according to the probability of the substitution process generating the tip sequences on that subtree. In an MCMC algorithm, conditioning on the observed sequences early on in the simulation increases the probability of generating a complete tree that represents a good starting point for the Markov chain. Crucially for the end user, the model remains the same regardless of which inference strategy is used.



```
1   model function host_rep_model(symbiontTree: TreeLabeled, ntips: Int, nhosts: Int,
2     interactions: Int[], hostDistances: Real[], dMean: Real, tune: Real): ReturnType {
3
4     assume lambda ~ Dirichlet([1.0,1.0,1.0,1.0]);
5     assume mu ~ Exponential(10.0);
6     assume beta ~ Exponential(1.0);
7     ...
8     let mp = ModelParams{qMatrix = qMatrix, dMatrix = dMatrix, dMean = dMean, beta = beta};
9     let stationaryProbs = stationaryProbs(lambda);
10
11    let interactions_reals = sint2real(interactions);
12    let iMatrix = mtxCreate(ntips, nhosts, interactions_reals);
13
14    let probsTree = get_proposal_params(symbiontTree, iMatrix, qMatrix, stationaryProbs, tune);
15    ...
16    let left = simulate(probsTree.left, HistoryPoint{age = probsTree.age, repertoire = rootRep}, mp, iMatrix);
17    let right = simulate(probsTree.right, HistoryPoint{age = probsTree.age, repertoire = rootRep}, mp, iMatrix);
18    let historyTree = HistoryNode{age = probsTree.age, label = probsTree.label, repertoire = rootRep,
19                      history = [], log_rep_debt = log_root_rep_debt-log_score_root, left = left, right = right};
20
21    logWeight( - get_rep_debt(historyTree) );
22
23    return ReturnType{historyTree, lambda, mu, beta};
24  }
```

Figure 4: A. Simplified TreePPL program for inference of host repertoire evolution. In order to sample ancestral host repertoires, first (B) ancestral state probabilities are calculated based on the observed data and the independence model (line 14 in A). Then (C) a sample is drawn using *assume* statements and the probability weight of the sample is recorded (lines 16-17 in A). Lastly, probability weights are updated by factoring in the probability under the full model (also in lines 16-17 in A) and factoring out the recorded probability under the independence model (line 21 in A).

The disadvantage of simulating the process backwards is that we need to say something about ancestral DNA sequences in the tree before we can fully express the probability distribution they are drawn from. We solve this problem by using a proposal distribution, similar to the mechanism used to simulate from the host repertoire model using an initial draw from the independence model.

Assume, for simplicity, that we are modeling the evolution of DNA sequences using the Jukes-Cantor (JC) model. The coalescent process would start with a set of leaves, each leaf being associated with a DNA sequence. The leaves together form a set of trees, also known as a forest. This is expressed in TreePPL as a sequence of `Tree` variables, each `Tree` initially being an instance of a `Leaf` variable. A random pair is then drawn from this tree set, see line 6 of the code in Figure 5, and a coalescent time is drawn from the prior, see line 10.

Now we propose an ancestral DNA sequence from the stationary state distribution of the JC model, that is, we draw each nucleotide from a uniform distribution over the state set $\{A, C, G, T\}$, see line 11. Even though this is not the correct distribution, we correct the weight of the simulation later, when we know the correct probability distribution,

```
 1 function cluster(q: Tensor[Real], trees: SeqTree[], maxAge: Real, seqLen: Int): SeqTree[] {
 2     let n = length(trees);
 3     if (eqi(n, 1)) {
 4         return trees;
 5     }
 6     let pairs = pickpair(n);
 7     let leftChild = trees[pairs[1]];
 8     let rightChild = trees[pairs[2]];
 9     assume t ~ Exponential(10.0);
10     let age = maxAge + t;
11     let seq = iid(categorical, CategoricalParam{probs=[0.25,0.25,0.25,0.25]}, seqLen);
12     for i in 1 to seqLen {
13         let p1 = ctmc(seq[i], q, age-leftChild.age);
14         observe leftChild.seq[i] ~ Categorical(p1);
15         if (leftChild is Node) {
16             logWeight -(categoricalLogScore(leftChild.seq[i], CategoricalParam{probs=[0.25,0.25,0.25,0.25]}));
17         }
18         let p2 = ctmc(seq[i], q, age-rightChild.age);
19         observe rightChild.seq[i] ~ Categorical(p2);
20         if (rightChild is Node) {
21             logWeight -(categoricalLogScore(rightChild.seq[i], CategoricalParam{probs=[0.25,0.25,0.25,0.25]}));
22         }
23     }
24     let parent = Node{age=age, seq=seq, left=leftChild, right=rightChild};
25     let min = mini(pairs[1], pairs[2]);
26     let max = maxi(pairs[1], pairs[2]);
27     let newTrees = join([slice(trees, 1, min), slice(trees, addi(min, 1), max), slice(trees, addi(max, 1), addi(n, 1)), [parent]]);
28     return cluster(q, newTrees, age, seqLen);
29 }
```

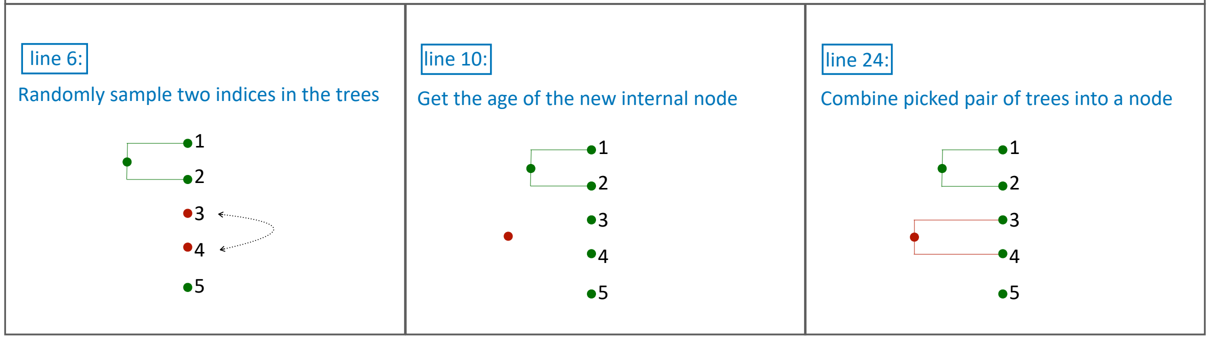

line 6:
Randomly sample two indices in the trees
1
2
3
4
5
line 10:
Get the age of the new internal node
1
2
3
4
5
line 24:
Combine picked pair of trees into a node
1
2
3
4
5

Figure 5: Illustration of the cluster function, used in phylogenetic tree inference.

just as in the host repertoire model. In principle, we could use any distribution for this proposal, as long as the weight of the simulation is later adjusted accordingly.

Once the ancestral DNA sequence has been specified, we can describe the probability of the two descendant sequences using the appropriate distribution from the continuous time Markov chain defined by the JC model over the appropriate time interval (lines 12 - 23 in Figure 5). If the descendant is a leaf, this is done using an observe statement. If it is an internal node, however, its DNA sequence has been proposed in an earlier pass through the function as a draw from the stationary distribution of the JC model. Therefore, we correct the weight of the simulation by factoring out the original draw and factoring in the probability of drawing the sequence from the correct continuous time Markov chain distribution (see lines 16 and 21).

The program described above defines the correct probability distribution but current automated inference algorithms do not scale very well on it because they will sample over all possible ancestral DNA sequences. This is inefficient, as the sampled parameter space will be huge even for DNA sequences of moderate length. The standard solution is to compute tree probabilities by summing over all possible ancestral sequences using Felsenstein's pruning algorithm [55]. In TreePPL, the pruning algorithm can be expressed explicitly, in which case the current version of TreePPL will support efficient SMC inference of phylogeny as described by [32]. By implementing appropriate proposal distributions for the selection of lineages to coalesce, along the lines of previous work on MCMC sampling over tree space [56, 57], it would seem possible to compete successfully with state of the art inference strategies. In ongoing work, we are exploring the possibility of automatically implementing the pruning algorithm as part of the inference machinery, so that the user does not have to worry about expressing it in the model script.

The current version of TreePPL also supports MCMC inference of tree topology from the same script. Again, with suitable proposal distributions, the inference would be quite efficient. However, the current MCMC algorithm does not support the stochastic tree topology moves that are currently used in most Bayesian MCMC phylogenetic software, such as stochastic nearest neighbor interchange or subtree pruning and regrafting [58]. Extending the automatic inference to support these techniques is left for future work.

Finally, we note that the basic script described above can be modified to support a range of related inference problems, such as inference of substitution model parameters, phylogenetic placement of unknown sequences in a known phylogenetic tree, or online inference of phylogeny, in which sequences are added sequentially to a growing phylogenetic tree. We point the interested reader to the TreePPL web site for an up-to-date set of such model scripts.

4 Discussions and Conclusions

Our examples hopefully illustrate that a universal PPL, such as TreePPL, has sufficient expressive power to adequately describe a range of common inference problems in phylogenetics. In principle, if it is possible to write a simulation from the model using a Turing-complete programming language, the model can be expressed in a universal PPL. It is usually the conditioning on observed data, which makes the application of automatic inference techniques difficult. With some understanding of how automatic inference techniques work, it is possible to avoid many problems by structuring the model code appropriately. For instance, if it is possible to condition on observed data early on in the simulation, it is often advantageous for automatic inference. The host repertoire program accomplishes this by using the observed data already in deriving a proposal distribution that is similar to the posterior probability distribution. The tree inference program instead reverses the time flow in the simulation, so that it starts with the observed data and goes backwards in time.

Another potential caveat is the conditioning of the simulation on specific values of continuous random variables. Where possible, this should be done using observe statements weighting the simulation using the analytical probability density function. If one were to sample random variables from continuous probability distributions and then condition on them taking on specific values, one can even specify conditional probability distributions that are not computable [59].

An interesting challenge is provided by models for which it is impossible to simulate directly from the prior distribution, but where it is possible to sample one or more latent variables conditional on the state of all the others. An iconic example is the Ising model, a mathematical model of ferromagnetism, but similar models could be envisioned in phylogenetics. For instance, one might be interested in eco-evolutionary models where the lineages interact so that some lineage-specific parameters are dependent on the state of all other lineages in the community. As pointed out in [12], such problems can be addressed in a PPL setting by proposing an initial state from a simpler model, and then weighting the simulation by the probability of the full model, as we exemplified in the host repertoire and tree inference programs.

Clearly, the PPL approach comes with numerous benefits for the end user. Most importantly, perhaps, the clean separation of modeling from inference allows a domain expert to rapidly explore alternative models that could explain a phenomenon of interest, rather than spending considerable time and resources on implementing inference machinery from scratch for each and every model explored. This aspect is particularly important when scientific progress is tied to the exploration of new models that have not been considered previously, as is often the case in phylogenetics and evolutionary biology. Another important advantage of separating inference from modeling is that it makes it possible to reuse inference algorithms that have been extensively used and tested on other scientific modeling and inference problems, thereby reducing the risk of inference errors.

The potential disadvantage of the PPL approach is that the available automatic inference algorithms may be less efficient than dedicated inference algorithms written for a specific problem. Sometimes the difference is small enough that it makes no difference in practice. The convenience of automatic inference and the speed with which it allows new models to be explored and applied to real data may well outweigh the potential loss in inference efficiency. In other cases, the automatic inference algorithms may be so slow that they do not allow realistic-size problems to be addressed at all. Research into efficient automatic inference algorithms is currently very active, and is successively addressing these problems of efficient inference for challenging types of problems. Importantly, such advances are applicable across scientific domains, making it possible for PPL users to benefit from them as soon as they become available in the PPL they are using. Meanwhile, as demonstrated in this paper and elsewhere [18], it is often possible to structure a PPL description so that it lends itself to efficient inference with current automatic inference techniques, before the appropriate solutions that would allow a more natural model description become available in PPL compilers. This is not an ideal solution, but it vastly expands the utility of current PPLs, and helps to point out the direction for future work on automatic inference techniques.

Given the current state of affairs, we argue that the evolutionary biologist interested in exploring PPLs should choose a PPL that is easy to use when describing phylogenetic models, and that has an inference toolbox that includes the techniques found to be essential in providing efficient inference for such models. The TreePPL language is designed to address both concerns. The language is inspired by R and Python, which are commonly used by computational evolutionary biologists. Therefore, many TreePPL language features should be familiar to users in this community. Other features may appear more exotic, such as algebraic data types and the adoption of an immutable functional programming style. These features are used to simplify automatic inference, but they also introduce abstractions and programming practices that are powerful in expressing phylogenetic models, and therefore worth the learning effort. The advanced features are typically inspired by OCaml, and provide a natural bridge to the CorePPL language used to express automatic inference algorithms in the Miking CorePPL framework. Importantly, the TreePPL language comes with a set of built-in types and functions, which facilitate the specification of phylogenetic models. For instance, the algebraic data type provided in the standard library for phylogenetic trees is easy to use and extend for models using more complex tree structures with many latent branch and node variables. The functions include many operations that are commonly used in phylogenetics, such as counting the number of leaves on a tree, computing total tree length, or rotating subtrees. Finally, the TreePPL web site provides a number of example programs describing common phylogenetic simulation and inference problems beyond the ones discussed in this paper.

The TreePPL Python package offers powerful utilities for pre-processing and post-processing, and for controlling and running TreePPL programs. This opens up the possibility of using a wide range of Python tools developed specifically for data processing and visualization of data and results relevant for the phylogenetics community. The R package for TreePPL, which is work in progress, will provide similar functionality for R users.

With respect to automatic inference algorithms, the current version of TreePPL implements a number of techniques, which have been found to be important for phylogenetic problems but that are not yet widely available in PPLs. These include selective CPS transformation [19], automatic alignment [29] and suspension analysis [20], and the alive particle filter [30]. Further examples of important techniques for efficient inference on phylogenetic problems include automatic application of delayed sampling, Felsenstein's pruning algorithm, and effective MCMC samplers of tree topology. These techniques are currently all work in progress in TreePPL. In addition to these specialized techniques, TreePPL includes a range of more commonly supported automatic inference techniques for PPLs, some of which have rarely been used for phylogenetic problems previously. Importantly, TreePPL also provides scalable inference algorithms that make effective use of parallel hardware architectures.

There is currently a large set of generic PPLs, including Stan [60], Gen [61, 62], Edward [63], Turing [64], Anglican [22], WebPPL [21], Birch [65] and Pyro [66], to name a few. Arguably, none of them supports phylogenetic model specification as well as TreePPL, and—to our knowledge—none of them provides such a complete range of tools for efficient automatic inference on phylogenetic models. Nevertheless, several of these generic PPLs have large and active user communities, and their capabilities expand very rapidly. Several of them have been shown to be useful for phylogenetic problems already with their current set of features [18, 11]. Therefore, we strongly recommend phylogeneticists and evolutionary biologists interested in the PPL approach to also keep an eye on these more generic PPLs. Eventually, we will probably see the different PPL subcommunities converge on a smaller set of languages and inference compilers, but there is likely to be several years of intense parallel development of PPL techniques before the field matures in this way.

We end by pointing out that the PPL approach represents an important step towards the adoption of machine learning techniques to accelerate progress in statistical phylogenetics. There are several ways in which machine learning could to extend current PPL capabilities. For instance, many automatic inference algorithms come with tuning parameters. Adjusting those tuning parameters so that inference becomes efficient for a particular model script is a problem that is well suited to machine learning. Finding an optimal strategy by selecting among a predefined set of inference techniques or possible combinations of techniques, including the setting of the tuning parameters involved, is a similar but more challenging problem, where machine learning could be useful. For early work in this direction for maximum likelihood optimization algorithms, see [67].

Even more exciting, perhaps, is the prospect of using generative machine learning in extending PPL models to accommodate phenomena that are not completely understood by the modeler. For instance, many phylogenomic data sets come with poorly characterized systematic errors that originated during the sequencing and genome assembly steps. These errors can cause serious problems in phylogenomic inference, as they may give rise to strongly misleading signal when probabilistic inference under evolutionary models is applied to big data sets [68, 69]. Generative machine learning techniques that start from a PPL script describing known evolutionary processes and extend the model by looking for and accommodating unexplained patterns in big data sets could significantly improve the accuracy of phylogenomic inference and help biologists resolve the most challenging branches in the tree of life. Similar approaches could potentially also be used to characterize poorly understood genomic processes against a background of well-known evolutionary phenomena. Undoubtedly, we will see several developments in these directions over the coming years.

Acknowledgements

This project was in part financially supported by the Swedish Foundation for Strategic Research (FFL15–0032 and RIT15–0012), by Digital Futures and by the Swedish Research Council (grant 2018–04329 awarded to DB, grants 2018-04620 and 2021–04830 awarded to FR, and International Postdoc Grant 2020–06422 awarded to MPB), and by the European Union's Horizon 2020 research and innovation program under the Marie Skłodowska–Curie grant agreement PhyPPL No. 898120 to V.S. The research has also partially been carried out as part of the Vinnova Competence Center for Trustworthy Edge Computing Systems and Applications at KTH Royal Institute of Technology.

TreePPL Technical Details

Item	**Details**
Version Number	TreePPL 2023 release candidate 1
Webpage	`https://treeppl.org`
Download Site – Language	`https://github.com/treeppl/treeppl`
Download Site – Python environment	`https://github.com/treeppl/treeppl-python`

References

[1] Nicolas Lartillot, Thomas Lepage, and Samuel Blanquart. Phylobayes 3: a bayesian software package for phylogenetic reconstruction and molecular dating. *Bioinformatics*, 25(17):2286–2288, 2009.

[2] Alexandros Stamatakis. Raxml version 8: a tool for phylogenetic analysis and post-analysis of large phylogenies. *Bioinformatics*, 30(9):1312–1313, 2014.

[3] Lam-Tung Nguyen, Heiko A Schmidt, Arndt Von Haeseler, and Bui Quang Minh. Iq-tree: a fast and effective stochastic algorithm for estimating maximum-likelihood phylogenies. *Molecular biology and evolution*, 32(1):268–274, 2015.

[4] John P Huelsenbeck and Fredrik Ronquist. Mrbayes: Bayesian inference of phylogenetic trees. *Bioinformatics*, 17(8):754–755, 2001.

[5] Fredrik Ronquist, Maxim Teslenko, Paul Van Der Mark, Daniel L Ayres, Aaron Darling, Sebastian Höhna, Bret Larget, Liang Liu, Marc A Suchard, and John P Huelsenbeck. Mrbayes 3.2: efficient bayesian phylogenetic inference and model choice across a large model space. *Systematic biology*, 61(3):539–542, 2012.

[6] Remco Bouckaert, Joseph Heled, Denise Kühnert, Tim Vaughan, Chieh-Hsi Wu, Dong Xie, Marc A Suchard, Andrew Rambaut, and Alexei J Drummond. Beast 2: a software platform for bayesian evolutionary analysis. *PLoS computational biology*, 10(4):e1003537, 2014.

[7] Sebastian Höhna, Tracy A Heath, Bastien Boussau, Michael J Landis, Fredrik Ronquist, and John P Huelsenbeck. Probabilistic graphical model representation in phylogenetics. *Systematic Biology*, 63(5):753–771, 2014.

[8] Sebastian Höhna, Michael J Landis, Tracy A Heath, Bastien Boussau, Nicolas Lartillot, Brian R Moore, John P Huelsenbeck, and Fredrik Ronquist. Revbayes: Bayesian phylogenetic inference using graphical models and an interactive model-specification language. *Systematic biology*, 65(4):726–736, 2016.

[9] Alexei J Drummond, Kylie Chen, Fábio K Mendes, and Dong Xie. Linguaphylo: a probabilistic model specification language for reproducible phylogenetic analyses. *PLOS Computational Biology*, 19(7):e1011226, 2023.

[10] B. Bredelings. BAli-Phy [version 4.0-beta6]. https://github.com/bredelings/BAli-Phy/releases/tag/4.0-beta6, 2024. GitHub repository.

[11] Mathieu Fourment, Christiaan J Swanepoel, Jared G Galloway, Xiang Ji, Karthik Gangavarapu, Marc A Suchard, and Frederick A Matsen IV. Automatic differentiation is no panacea for phylogenetic gradient computation. *Genome Biology and Evolution*, 15(6):evad099, 2023.

[12] Alexandre Bouchard-Côté, Kevin Chern, Davor Cubranic, Sahand Hosseini, Justin Hume, Matteo Lepur, Zihui Ouyang, and Giorgio Sgarbi. Blang: Bayesian declarative modelling of general data structures and inference via algorithms based on distribution continua. *arXiv preprint arXiv:1912.10396*, 2019.

[13] Andrew D Gordon, Thomas A Henzinger, Aditya V Nori, and Sriram K Rajamani. Probabilistic programming. In *Future of Software Engineering Proceedings*, pages 167–181. 2014.

[14] Zoubin Ghahramani. Probabilistic machine learning and artificial intelligence. *Nature*, 521(7553):452–459, 2015.

[15] Jan-Willem van de Meent, Brooks Paige, Hongseok Yang, and Frank Wood. An introduction to probabilistic programming. *arXiv preprint arXiv:1809.10756*, 2018.

[16] Desmond C Ong, Harold Soh, Jamil Zaki, and Noah D Goodman. Applying probabilistic programming to affective computing. *IEEE Transactions on Affective Computing*, 12(2):306–317, 2019.

[17] Alexander Lew, Monica Agrawal, David Sontag, and Vikash Mansinghka. Pclean: Bayesian data cleaning at scale with domain-specific probabilistic programming. In *International Conference on Artificial Intelligence and Statistics*, pages 1927–1935. PMLR, 2021.

[18] Fredrik Ronquist, Jan Kudlicka, Viktor Senderov, Johannes Borgström, Nicolas Lartillot, Daniel Lundén, Lawrence Murray, Thomas B Schön, and David Broman. Universal probabilistic programming offers a powerful approach to statistical phylogenetics. *Communications biology*, 4(1):244, 2021.

[19] Daniel Lundén, Joey Öhman, Jan Kudlicka, Viktor Senderov, Fredrik Ronquist, and David Broman. Compiling universal probabilistic programming languages with efficient parallel sequential monte carlo inference. In *ESOP*, pages 29–56, 2022.

[20] Daniel Lundén, Lars Hummelgren, Jan Kudlicka, Oscar Eriksson, and David Broman. Suspension analysis and selective continuation-passing style for higher-order probabilistic programming languages. *arXiv preprint arXiv:2302.13051*, 2023.

[21] Noah D Goodman and Andreas Stuhlmüller. The design and implementation of probabilistic programming languageshttp. *dippl. org*, 2014.

[22] David Tolpin, Jan-Willem van de Meent, Hongseok Yang, and Frank Wood. Design and implementation of probabilistic programming language anglican. In *Proceedings of the 28th Symposium on the Implementation and Application of Functional Programming Languages*, IFL 2016, pages 6:1–6:12, New York, NY, USA, 2016. ACM.

[23] Daniel L Rabosky, Michael Grundler, Carlos Anderson, Pascal Title, Jeff J Shi, Joseph W Brown, Huateng Huang, and Joanna G Larson. Bamm tools: an r package for the analysis of evolutionary dynamics on phylogenetic trees. *Methods in Ecology and Evolution*, 5(7):701–707, 2014.

[24] Daniel L Rabosky. Automatic detection of key innovations, rate shifts, and diversity-dependence on phylogenetic trees. *PloS one*, 9(2):e89543, 2014.

[25] Brian R Moore, Sebastian Höhna, Michael R May, Bruce Rannala, and John P Huelsenbeck. Critically evaluating the theory and performance of bayesian analysis of macroevolutionary mixtures. *Proceedings of the National Academy of Sciences*, 113(34):9569–9574, 2016.

[26] Daniel L Rabosky, Jonathan S Mitchell, and Jonathan Chang. Is bamm flawed? theoretical and practical concerns in the analysis of multi-rate diversification models. *Systematic biology*, 66(4):477–498, 2017.

[27] David Tolpin, Jan-Willem van de Meent, and Frank Wood. Probabilistic programming in anglican. In *Machine Learning and Knowledge Discovery in Databases: European Conference, ECML PKDD 2015, Porto, Portugal, September 7-11, 2015, Proceedings, Part III 15*, pages 308–311. Springer, 2015.

[28] Lawrence Murray, Daniel Lundén, Jan Kudlicka, David Broman, and Thomas Schön. Delayed sampling and automatic rao-blackwellization of probabilistic programs. In *International Conference on Artificial Intelligence and Statistics*, pages 1037–1046. PMLR, 2018.

[29] Daniel Lundén, Gizem Çaylak, Fredrik Ronquist, and David Broman. Automatic alignment in higher-order probabilistic programming languages. *Programming Languages and Systems LNCS 13990*, page 535, 2023.

[30] Jan Kudlicka, Lawrence M Murray, Fredrik Ronquist, and Thomas B Schön. Probabilistic programming for birth-death models of evolution using an alive particle filter with delayed sampling. In *Uncertainty in Artificial Intelligence*, pages 679–689. PMLR, 2020.

[31] Alexei J Drummond, Marc A Suchard, Dong Xie, and Andrew Rambaut. Bayesian phylogenetics with beauti and the beast 1.7. *Molecular biology and evolution*, 29(8):1969–1973, 2012.

[32] Alexandre Bouchard-Côté, Sriram Sankararaman, and Michael I. Jordan. Phylogenetic Inference via Sequential Monte Carlo. *Systematic Biology*, 61(4):579–593, 2012.

[33] Antonio Moretti, Liyi Zhang, and Itsik Pe'er. Variational combinatorial sequential monte carlo for bayesian phylogenetic inference. *Machine Learning in Computational Biology*, 2020.

[34] Cheng Zhang. Improved variational bayesian phylogenetic inference with normalizing flows. *Advances in neural information processing systems*, 33:18760–18771, 2020.

[35] Hazal Koptagel, Oskar Kviman, Harald Melin, Negar Safinianaini, and Jens Lagergren. Vaiphy: a variational inference based algorithm for phylogeny. *Advances in Neural Information Processing Systems*, 35:14758–14770, 2022.

[36] Frank Wood, Jan Willem Meent, and Vikash Mansinghka. A new approach to probabilistic programming inference. In *Artificial intelligence and statistics*, pages 1024–1032. PMLR, 2014.

[37] Daniel Ritchie, Andreas Stuhlmüller, and Noah Goodman. C3: Lightweight incrementalized mcmc for probabilistic programs using continuations and callsite caching. In *Artificial Intelligence and Statistics*, pages 28–37. PMLR, 2016.

[38] Odile Maliet and Hélène Morlon. Fast and accurate estimation of species-specific diversification rates using data augmentation. *Systematic biology*, 71(2):353–366, 2022.

[39] George Udny Yule. Ii.—a mathematical theory of evolution, based on the conclusions of dr. jc willis, fr s. *Philosophical transactions of the Royal Society of London. Series B, containing papers of a biological character*, 213(402-410):21–87, 1925.

[40] Sean Nee. Birth-death models in macroevolution. *Annu. Rev. Ecol. Evol. Syst.*, 37:1–17, 2006.

[41] David M Blei, Andrew Y Ng, and Michael I Jordan. Latent dirichlet allocation. *Journal of machine Learning research*, 3(Jan):993–1022, 2003.

[42] David Broman. 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*, SLE '19, pages 55–60. ACM, 2019.

[43] David Wingate, Andreas Stuhlmüller, and Noah Goodman. Lightweight implementations of probabilistic programming languages via transformational compilation. In *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics*, pages 770–778. JMLR Workshop and Conference Proceedings, 2011.

[44] Brooks Paige and Frank Wood. A compilation target for probabilistic programming languages. In *International Conference on Machine Learning*, pages 1935–1943. PMLR, 2014.

[45] David G Kendall. On the generalized" birth-and-death" process. *The annals of mathematical statistics*, 19(1):1–15, 1948.

[46] Sebastian Höhna, William A Freyman, Zachary Nolen, John P Huelsenbeck, Michael R May, and Brian R Moore. A bayesian approach for estimating branch-specific speciation and extinction rates. *BioRxiv*, page 555805, 2019.

[47] Pierre Del Moral, Ajay Jasra, Anthony Lee, Christopher Yau, and Xiaole Zhang. The alive particle filter and its use in particle markov chain monte carlo. *Stochastic Analysis and Applications*, 33(6):943–974, 2015.

[48] Arnaud Doucet, Nando De Freitas, Neil James Gordon, et al. *Sequential Monte Carlo methods in practice*, volume 1. Springer, 2001.

[49] Joseph Felsenstein. Evolutionary trees from gene frequencies and quantitative characters: finding maximum likelihood estimates. *Evolution*, pages 1229–1242, 1981.

[50] Ignacio Quintero, Nicolas Lartillot, and Hélène Morlon. The birth-death diffusion leading to present-day mammal diversity. *bioRxiv*, pages 2022–08, 2022.

[51] Mariana P Braga, Michael J Landis, Sören Nylin, Niklas Janz, and Fredrik Ronquist. Bayesian Inference of Ancestral Host-parasite Interactions under a Phylogenetic Model of Host Repertoire Evolution. *Systematic Biology*, 69(6):1149–1162, 2020.

[52] Yueqing An, Mariana P. Braga, Sarahi L. Garcia, Magdalena Grudzinska-Sterno, and Peter A. Hambäck. Host Phylogeny Structures the Gut Bacterial Community Within Galerucella Leaf Beetles. *Microbial Ecology*, pages 1–11, 2023.

[53] Sakina Neemuchwala, Nathan A. Johnson, John M. Pfeiffer, Manuel Lopes-Lima, André Gomes-dos Santos, Elsa Froufe, David M. Hillis, and Chase H. Smith. Coevolution With Host Fishes Shapes Parasitic Life Histories in a Group of Freshwater Mussels (Unionidae: Quadrulini). *Bulletin of the Society of Systematic Biologists*, 2(1):1–25, 2023.

[54] Rasmus Nielsen. Mapping mutations on phylogenies. *Systematic biology*, 51(5):729–739, 2002.

[55] Joseph Felsenstein. Maximum likelihood and minimum-steps methods for estimating evolutionary trees from data on discrete characters. *Systematic Zoology*, (3):240–249, 1973.

[56] Sebastian Höhna and Alexei J Drummond. Guided tree topology proposals for bayesian phylogenetic inference. *Systematic Biology*, 69(5):1016–1032, 2020.

[57] Chi Zhang, John P Huelsenbeck, and Fredrik Ronquist. Using parsimony-guided tree proposals to accelerate convergence in bayesian phylogenetic inference. *Systematic Biology*, 69(5):1016–1032, 2020.

[58] Clemens Lakner, Paul van der Mark, John P Huelsenbeck, Bret Larget, and Fredrik Ronquist. Efficiency of markov chain monte carlo tree proposals in bayesian phylogenetics. *Systematic Biology*, 57(1):86–103, 2008.

[59] Nathanael L Ackerman, Cameron E Freer, and Daniel M Roy. On the computability of conditional probability. *arXiv preprint arXiv:1005.3014v4*, 2019.

[60] Bob Carpenter, Andrew Gelman, Matthew D Hoffman, Daniel Lee, Ben Goodrich, Michael Betancourt, Marcus A Brubaker, Jiqiang Guo, Peter Li, and Allen Riddell. Stan: A probabilistic programming language. *Journal of statistical software*, 76, 2017.

[61] Marco F Cusumano-Towner, Feras A Saad, Alexander K Lew, and Vikash K Mansinghka. Gen: a general-purpose probabilistic programming system with programmable inference. In *Proceedings of the 40th acm sigplan conference on programming language design and implementation*, pages 221–236, 2019.

[62] Alexander K Lew, George Matheos, Tan Zhi-Xuan, Matin Ghavamizadeh, Nishad Gothoskar, Stuart Russell, and Vikash K Mansinghka. Smcp3: Sequential monte carlo with probabilistic program proposals. In *International Conference on Artificial Intelligence and Statistics*, pages 7061–7088. PMLR, 2023.

[63] Dustin Tran, Alp Kucukelbir, Adji B Dieng, Maja Rudolph, Dawen Liang, and David M Blei. Edward: A library for probabilistic modeling, inference, and criticism. *arXiv preprint arXiv:1610.09787*, 2016.

[64] Hong Ge, Kai Xu, and Zoubin Ghahramani. Turing: a language for flexible probabilistic inference. In *International conference on artificial intelligence and statistics*, pages 1682–1690. PMLR, 2018.

[65] Lawrence M Murray and Thomas B Schön. Automated learning with a probabilistic programming language: Birch. *Annual Reviews in Control*, 46:29–43, 2018.

[66] Eli Bingham, Jonathan P Chen, Martin Jankowiak, Fritz Obermeyer, Neeraj Pradhan, Theofanis Karaletsos, Rohit Singh, Paul Szerlip, Paul Horsfall, and Noah D Goodman. Pyro: Deep universal probabilistic programming. *The Journal of Machine Learning Research*, 20(1):973–978, 2019.

[67] Dana Azouri, Shiran Abadi, Yishay Mansour, Itay Mayrose, and Tal Pupko. Harnessing the power of machine learning to guide phylogenetic-tree search algorithms. *Nature Communications*, 12:1983, 2021.

[68] Paschalia Kapli and Maximilian J Telford. Topology-dependent asymmetry in systematic errors affects phylogenetic placement of ctenophora and xenacoelomorpha. *Science Advances*, 6:eabc5162, 2020.

[69] Erik Tihelka, Chenyang Cai, Mattia Giacomelli, Jesus Lozano-Fernandez, Omar Rota-Stabelli, Diying Huang, Michael S Engel, Philip C J Donoghue, and Davide Pisani. The evolution of insect biodiversity. *Science Advances*, 31:R1299–R1311, 2021.