# Venture: a higher-order probabilistic programming platform with programmable inference

**Vikash Mansinghka** VKM@MIT.EDU
**Daniel Selsam** DSELSAM@MIT.EDU
**Yura Perov** PEROV@MIT.EDU



## Abstract

We describe Venture, an interactive virtual machine for probabilistic programming that aims to be sufficiently expressive, extensible, and efficient for general-purpose use. Like Church, probabilistic models and inference problems in Venture are specified via a Turing-complete, higher-order probabilistic language descended from Lisp. Unlike Church, Venture also provides a compositional language for custom inference strategies, assembled from scalable implementations of several exact and approximate techniques. Venture is thus applicable to problems involving widely varying model families, dataset sizes and runtime/accuracy constraints. We also describe four key aspects of Venture's implementation that build on ideas from probabilistic graphical models. First, we describe the *stochastic procedure interface* (SPI) that specifies and encapsulates primitive random variables, analogously to conditional probability tables in a Bayesian network. The SPI supports custom control flow, higher-order probabilistic procedures, partially exchangeable sequences and "likelihood-free" stochastic simulators, all with custom proposals. It also supports the integration of external models that dynamically create, destroy and perform inference over latent variables hidden from Venture. Second, we describe *probabilistic execution traces* (PETs), which represent execution histories of Venture programs. Like Bayesian networks, PETs capture conditional dependencies, but PETs also represent existential dependencies and exchangeable coupling. Third, we describe partitions of execution histories called *scaffolds* that can be efficiently constructed from PETs and that factor global inference problems into coherent sub-problems. Finally, we describe a family of *stochastic regeneration algorithms* for efficiently modifying PET fragments contained within scaffolds without visiting conditionally independent random choices. Stochastic regeneration insulates inference algorithms from the complexities introduced by changes in execution structure, with runtime that scales linearly in cases where previous approaches often scaled quadratically and were therefore impractical. We show how to use stochastic regeneration and the SPI to implement general-purpose inference strategies such as Metropolis-Hastings, Gibbs sampling, and blocked proposals based on hybrids with both particle Markov chain Monte Carlo and mean-field variational inference techniques.

**Keywords:** Probabilistic programming, Bayesian inference, Bayesian networks, Markov chain Monte Carlo, sequential Monte Carlo, particle Markov chain Monte Carlo, variational inference

**Acknowledgements:** The authors thank Vlad Firoiu and Alexey Radul for contributions to multiple Venture implementations, and Daniel Roy, Cameron Freer and Alexey Radul for helpful discussions, suggestions, and comments on early drafts. This work was supported by the DARPA PPAML program, grants from the ONR and ARO, and Google's "Rethinking AI" project. Any opinions, findings, and conclusions or recommendations expressed in this work are those of the authors and do not necessarily reflect the views of any of the above sponsors.

## Contents

## 1. Introduction

Probabilistic modeling and approximate Bayesian inference have proven to be powerful tools in multiple fields, from machine learning (Bishop, 2006) and statistics (Green et al., 2003; Gelman et al., 1995) to robotics (Thrun et al., 2005), artificial intelligence (Russell and Norvig, 2002), and cognitive science (Tenenbaum et al., 2011). Unfortunately, even relatively simple probabilistic models and their associated inference schemes can be difficult and time-consuming to design, specify, analyze, implement, and debug. Applications in different fields, such as robotics and statistics, involve differing modeling idioms, inference techniques and dataset sizes. Different fields also often impose varying speed and accuracy requirements that interact with modeling and algorithmic choices. Small changes in the modeling assumptions, data, performance/accuracy requirements or compute budget frequently necessitate end-to-end redesign of the probabilistic model and inference strategy, in turn necessitating reimplementation of the underlying software.

These difficulties impose a high cost on practitioners, making state-of-the-art modeling and inference approaches impractical for many problems. Minor variations on standard templates can be out of reach for non-specialists. The cost is also high for experts: the development time and failure rate make it difficult to innovate on methodology except in simple settings. This limits the richness of probabilistic models of cognition and artificial intelligence systems, as these kinds of models push the boundaries of what is possible with current knowledge representation and inference techniques.

Probabilistic programming languages could potentially mitigate these problems. They provide a formal representation for models — often via executable code that makes a random choice for every latent variable — and attempt to encapsulate and automate inference. Several languages and systems have been built along these lines over the last decade (Lunn et al., 2000; Stan Development Team, 2013; Milch et al., 2007; Pfeffer, 2001; McCallum et al., 2009). Each of these systems is promising in its own domain; some of the strengths of each are described below. However, none of the probabilistic programming languages and systems that have been developed thus far is suitable for general purpose use. Examples of drawbacks include inadequate and unpredictable runtime performance, limited expressiveness, batch-only operation, lack of extensibility, and overly restrictive and/or opaque inference schemes. In this paper, we describe Venture, a new probabilistic language and inference engine that attempts to address these limitations.

Several probabilistic programming tools have sought efficiency by restricting expressiveness. For example, Microsoft's Infer.NET system (Minka et al., 2010) leverages fast message passing

techniques originally developed for graphical models, but as a result restricts the use of stochastic choice in the language so that it cannot influence control flow. Such random choices would yield models over sets of random variables with varying or even unbounded size, and therefore preclude compilation to a graphical model. BUGS, arguably the first (and still most widely used) probabilistic programming language, has essentially the same restrictions (Lunn et al., 2000). Random compound data types, procedures and stochastic control flow constructs that could lead to a priori unbounded executions are all out of scope. STAN, a BUGS-like language being developed in the Bayesian statistics community, has limited support for discrete random variables, as these are incompatible with the hybrid (gradient-based) Monte Carlo strategy it uses to overcome convergence issues with Gibbs sampling (Stan Development Team, 2013). Other probabilistic programming tools that have seen real-world use include FACTORIE (McCallum et al., 2009) and Markov Logic (Richardson and Domingos, 2006); applications of both have emphasized problems in information extraction. The probabilistic models that can be defined using FACTORIE and Markov Logic are finite and undirected, specified imperatively (for FACTORIE) or declaratively (for Markov Logic). Both systems make use of specialized, efficient approximation algorithms for inference and parameter estimation. Infer.NET, STAN, BUGS, FACTORIE and Markov Logic each capture important modeling and approximate inference idioms, but there are also interesting models that each cannot express. Additionally, a number of probabilistic extensions of classical logic programming languages have also been developed (Poole, 1997; Sato and Kameya, 1997; De Raedt and Kersting, 2008), motivated by problems in statistical relational learning. As with FACTORIE and Markov Logic, these languages have interesting and useful properties, but have thus far not yielded compact descriptions of many useful classes of probabilistic generative models from domains such as statistics and robotics.

In contrast, probabilistic programming languages such as BLOG (Milch et al., 2007), IBAL (Pfeffer, 2001), Figaro (Pfeffer, 2009) and Church (Goodman*, Mansinghka*, Roy, Bonowitz, and Tenenbaum, 2008; Mansinghka, 2009) emphasize expressiveness. Each language was designed around the needs of models whose fine-grained structure cannot be represented using directed or undirected graphical models, and where standard inference algorithms for graphical models such as belief propagation do not directly apply. Examples include probabilistic grammars (Jelinek et al., 1992), nonparametric Bayesian models (Rasmussen, 1999; Johnson et al., 2007; Rasmussen and Williams, 2006; Griffiths and Ghahramani, 2005), probabilistic models over worlds with a priori unknown numbers of objects, models for learning the structure of graphical models (Heckerman, 1998; Friedman and Koller, 2003; Mansinghka et al., 2006), models for inductive learning of symbolic expressions (Grosse et al., 2012; Duvenaud et al., 2013) and models defined in terms of complex stochastic simulation software that lack tractable likelihoods (Marjoram et al., 2003).

Each of these model classes is the basis of real-world applications, where inference over richly structured models can address limitations of classic statistical modeling and pattern recognition techniques. Example domains include natural language processing (Manning and Schütze, 1999), speech recognition (Baker, 1979), information extraction (Pasula et al., 2002), multitarget tracking and sensor fusion (Oh et al., 2009; Arora et al., 2010), ecology (Csilléry et al., 2010) and computational biology (Friedman et al., 2000; Yu et al., 2004; Toni et al., 2009; Dowell and Eddy, 2004). However, the performance engineering needed to turn specialized inference algorithms for these models into viable implementations is challenging. Direct deployment of probabilistic program implementations in real-world applications is often infeasible. The elaborations on these models that expressive probabilistic languages enable can thus seem completely impractical.

Church makes the most extreme tradeoffs with respect to expressiveness and efficiency. It can represent models from all the classes listed above, partly through its support for higher-order probabilistic procedures, and it can also represent generative models defined in terms of algorithms for simulation and inference in arbitrary Church programs. This flexibility makes Church especially suitable for nonparametric Bayesian modeling (Roy et al., 2008), as well as artificial intelligence and cognitive science problems that involve reasoning about reasoning, such as sequential decision making, planning and theory of mind (Goodman and Tenenbaum, 2013; Mansinghka, 2009). Additionally, probabilistic formulations of learning Church programs from data — including both program structure and parameters — can be formulated in terms of inference in an ordinary Church program (Mansinghka, 2009). But although various Church implementations provide automatic Metropolis-Hastings inference mechanisms that in principle apply to all these problems, these mechanisms have exhibited limitations in practice. It has not been clear how to make general-purpose sampling-based inference in Church sufficiently scalable for typical machine learning applications, including problems for which standard techniques based on graphical models have been applied successfully. It also is not easy for Church programmers to override built in inference mechanisms or add new higher-order stochastic primitives.

In this paper we describe Venture, an interactive, Turing-complete, higher order probabilistic programming platform that aims to be sufficiently expressive, extensible and efficient for general-purpose use. Venture includes a virtual machine, a language for specifying probabilistic models, and a language for specifying inference problems along with custom inference strategies for solving those problems. Venture's implementation of standard MCMC schemes scales linearly with dataset size on problems where many previous inference architectures scale quadratically and are therefore impractical. Venture also supports a larger class of primitives — including "likelihood-free" primitives arising from complex stochastic simulators — and enables programmers to incrementally migrate performance-critical portions of their probabilistic program to optimized external inference code. Venture thus improves over Church in terms of expressiveness, extensibility, and scalability. Although it remains to be seen if these improvements are sufficient for general-purpose use, unoptimized Venture prototypes have begun to be successfully applied in real-world system building, Bayesian data analysis, cognitive modeling and machine intelligence research.

### 1.1 Contributions

This paper makes two main contributions. First, it describes key aspects of Venture's design, including support for interactive modeling and programmable inference. Due to these and other innovations, Venture provides broad coverage in terms of models, approximation strategies for those models and overall applicability to inference problems with varying model/data complexities and time/accuracy requirements. Second, this paper describes key aspects of Venture's implementation: the *stochastic procedure interface* (SPI) for encapsulating primitives, the *probabilistic execution trace* (PET) data structure for efficient representation and updating of execution histories, and a suite of *stochastic regeneration algorithms* for scalable inference within trace fragments called *scaffolds*. Other important aspects of Venture, including the VentureScript front-end syntax, formal language definitions, software architecture, standard library, and performance measurements of optimized implementations, are all beyond the scope of this paper.

It is helpful to consider the relationships between PETs and the SPI. The SPI generalizes the notion of elementary random procedure associated with many previous probabilistic programming

languages. The SPI encapsulates Venture primitives and enables interoperation with external modeling components, analogously to a foreign function interface in a traditional programming language. External modeling components can represent sets of latent variables hidden from Venture and that use specialized inference code. The SPI also supports custom control flow, higher-order probabilistic procedures, exchangeable sequences, and the "likelihood-free" stochastic primitives that can arise from complex simulations. Probabilistic execution traces are used to represent generative models written in Venture, along with particular realizations of these models and the data values they must explain. PETs generalize Bayesian networks to handle the patterns of conditional dependence, existential dependence and exchangeable coupling amongst invocations of stochastic procedures conforming to the SPI. PETs thus must handle a priori unbounded sets of random variables that can themselves be arbitrary probabilistic generative processes written in Venture and that may lack tractable probability densities.

Using these tools, we show how to define coherent local inference steps over arbitrary sets of random choices within a PET, conditioned on the surrounding trace. The core idea is the notion of a *scaffold*. A scaffold is a subset of a PET that contains those random variables that must exist regardless of the values chosen for a set of variables of interest, along with a set of random variables whose values will be conditioned on. We show how to construct scaffolds efficiently. Inference given a scaffold proceeds via *stochastic regeneration algorithms* that efficiently consume and either restore or resample PET fragments without visiting conditionally independent random choices. The proposal probabilities, local priors, local likelihoods and gradients needed for several approximate inference strategies can all be obtained via small variations on stochastic regeneration.

We use stochastic regeneration to implement both single-site and composite Metropolis-Hastings, Gibbs sampling, and also blocked proposals based on hybrids with conditional sequential Monte Carlo and variational techniques. The uniform implementation of these approaches to incremental inference, along with analytical tools for converting randomly chosen local transition operators into ergodic global transition operators on PETs, constitutes another contribution.

## 2. The Venture Language

Consider the following example Venture program for determining if a coin is fair or tricky:

```
[ASSUME is_tricky_coin (bernoulli 0.1)]
[ASSUME coin_weight (if is_tricky_coin (uniform 0.0 1.0) 0.5)]
[OBSERVE (bernoulli coin_weight) True]
[OBSERVE (bernoulli coin_weight) True]
[INFER (mh default one 10)]
[PREDICT (bernoulli coin_weight)]
```

We will informally discuss this program before defining the Venture language more precisely.

The `ASSUME` instructions induce the hypothesis space for the probabilistic model, including a random variable for whether or not the coin is tricky, and either a deterministic coin weight or a potential random variable corresponding to the unknown weight. The model selection problem is expressed via an `if` with a stochastic predicate, with the alternative models on the consequent and alternate branches. After executing the `ASSUME` instructions, particular values for `is_tricky_coin` and `coin_weight` will have been sampled, though the meaning of the program so far corresponds to a probability distribution over possible executions.

The OBSERVE instructions describe a data generator that produces two flips of a coin with the generated weight, along with data that is assumed to be generated by the given generator. In this program, the OBSERVEs encode constraints that both of these coin flips landed heads up.

The INFER instruction causes Venture to find a hypothesis (execution trace) that is probable given the data using 10 iterations of its default Markov chain for inference[1]. INFER evolves the probability distribution over execution traces inside Venture from whatever distribution is present before the instruction — in this case, the prior — closer to its conditional given any observations that have been added prior to that INFER, using a user-specified inference technique. In this example program, the resulting marginal distribution on whether or not the coin is tricky shifts from the prior to an approximation of the posterior given the two observed flips, increasing the probability of the coin being tricky ever so slightly. Increasing 10 to 100 shifts the distribution closer to the true posterior; other inference strategies, including exact sampling techniques, will be covered later. The execution trace inside Venture after the instruction is sampled from this new distribution.

Once inference has finished, the PREDICT instruction causes Venture to report a sampled prediction for the outcome of another flip of the coin. The weight used to generate this sample comes from the current execution trace.

### 2.1 Modeling and Inference Instructions

Venture programs consist of sequences of modeling instructions and inference instructions, each given a positive integer index by a global instruction counter[2]. Interactive Venture sessions have the same structure. The modeling instructions are used to specify the probabilistic model of interest, any conditions on the model that the inference engine needs to enforce, and any requests for prediction of values against the conditioned distribution.

The core modeling instructions in Venture are:

1. [ASSUME <name> <expr>]: binds the result of simulating the model expression <expr> to the symbol <name> in the global environment. This is used to build up the model that will be used to interpret data. Returns the value taken on by <name>, along with the index of the instruction.

2. [OBSERVE <expr> <literal-value>]: adds the constraint that the model expression <expr> must yield <literal-value> in every execution. Note that this constraint is not enforced until inference is performed.

3. [PREDICT <expr>]: samples a value for the model expression <expr> from the current distribution on executions in the engine and returns the value. As the amount of inference done since the last OBSERVE approaches infinity, this distribution converges to the conditioned distribution that reconciles the OBSERVEs.

1. This default Markov chain is a variant of the algorithm from Church (Goodman*, Mansinghka*, Roy, Bonowitz, and Tenenbaum, 2008; Mansinghka, 2009). This is a simple random scan single-site Metropolis-Hastings algorithm that chooses random choices uniformly at random from the current execution, resimulates them conditioned on the rest of the trace, and accepts or rejects the result.
2. Venture implementations also support labels for the instruction language. However, the recurrence of line numbers is reflective of the ways the current instruction language is primitive as compared to the modeling language. For example, it currently lacks control flow constructs, procedural abstraction, and recursion, let alone runtime generation and execution of statements in the instruction language. Current research is focused on addressing these limitations.

4. `[FORGET <instruction-index-or-label>]`: This instruction causes the engine to undo and then forget the given instruction, which must be either an `OBSERVE` or `PREDICT`. Forgetting an observation removes the constraint it represents from the inference problem. Note that the effect may not be visible until an `INFER` is performed.

Venture supports additional instructions for inference and read-out, including:

1. `[INFER <inference-expr>]`: This instruction first incorporates any observations that have occurred after the last `INFER`, then evolves the probability distribution on executions according to the inference strategy described by `<inference-expr>`. Two inference expressions, corresponding to general-purpose exact and approximate sampling schemes, are useful to consider here:

   (a) `(rejection default all)` corresponds to the use of rejection sampling to generate an exact sample from the conditioned distribution on traces. The runtime requirements may be substantial, and exact sampling applies to a smaller class of programs than approximate sampling. However, rejection is crucial for understanding the meaning of a probabilistic model and for debugging models without simultaneously debugging inference strategies.

   (b) `(mh default one 1)` corresponds to one transition of the standard uniform mixture of single-site Metropolis-Hastings transition operators used as a general-purpose "automatic" inference scheme in many probabilistic programming systems. As the number of transitions is increased from 1 towards $\infty$, the semantics of the instruction approach an exact implementation of conditioning via rejection.

2. `[SAMPLE <expr>]`: This instruction simulates the model expression `<expr>` against the current trace, returns the value, and then forgets the trace associated with the simulation. It is equivalent to `[PREDICT <expr>]` followed by `[FORGET <index-of-predict>]`, but is provided as a single instruction for convenience.

3. `[FORCE <expr> <literal-value>]`: Modify the current trace so that the simulation of `<expr>` takes on the value `<literal-value>`. Its implementation can be roughly thought of as an `OBSERVE` immediately followed by an `INFER` and then a `FORGET`. This instruction can be used for controlling initialization and for debugging.

### 2.2 Modeling Expressions

Venture modeling expressions describe stochastic generative processes. The space of all possible executions of all the modeling expressions in a Venture program constitute the hypothesis space that the program represents. Each Venture program thus represents a probabilistic model by defining a stochastic generative process that samples from it.

At the expression level, Venture is similar to Scheme and to Church, though there are several differences[3]. For example, branching and procedure construction can both be desugared into applications of stochastic procedures — that is, ordinary combinations — and do not need to be treated as

3. We sometimes refer to the s-expression syntax (including syntactic sugar) as Venchurch, and the desugared language (represented as JSON objects corresponding to parse trees) as Venture.

special forms. Additionally, Venture supports a dynamic scoping construct called `scope_include` for tagging portions of an execution history such that they can be referred to by inference instructions; to the best of our knowledge, analogous constructs have not yet been introduced in other probabilistic programming languages.

Venture modeling expressions can be broken down into a few simple cases:

1. **Self-evaluating or "literal" values:** These describe constant values in the language, and are discussed below.

2. **Combinations:** `(<operator-expr> <operand0-expr> ... <operandk-expr>)` first evaluates all of its expressions in arbitrary order, then applies the value of `<operator-expr>` (which must be a stochastic procedure) to the values of all the `<operand-expr>`s. It returns the value of the application as its own result.

3. **Quoted expressions:** `(quote <expr>)` returns the expression value `<expr>`. As compared to combinations, `quote` suppresses evaluation.

4. **Lambda expressions:** `(lambda <args> <body-expr>)` returns a stochastic procedure with the given formal parameters and procedure body. `<args>` is a specific list of argument names `(<arg0> ... <argk>)`.

5. **Conditionals:** `(if <predicate-expr> <consequent-expr> <alternate-expr>)` evaluates the `<predicate-expr>`, and then if the resulting value is `true`, evaluates and returns value of the `<consequent-expr>`, and if not, evaluates and returns the value of the `<alternate-expr>`.

6. **Inference scope annotations:** `(scope_include <scope-expr> <block-expr> <expr>)` provides a mechanism for naming random choices in a probabilistic model, so that they can be referred to during inference programming. The `scope_include` form simulates `<scope-expr>` and `<block-expr>` to obtain a scope value and a block value, and then simulates `<expr>`, tagging all the random choices in that process with the given scope and block. More details on inference scopes are given below.

### 2.3 Inference Scopes

Venture programs may attach metadata to fragments of execution traces via a dynamic scoping construct called *inference scopes*. Scopes are defined in modeling expressions, via the special form `(scope_include <scope-expr> <block-expr> <expr>)`, that assigns all random choices required to simulate `<expr>` to a scope that is named by the value resulting from simulating `<scope-expr>` and a *block* within that scope that is named by the value that results from simulating `<block-expr>`. A single random choice can be in multiple inference scopes, but can only be in one block within each scope. Also, a random choice gets annotated with a scope each time the choice is simulated within the context of a `scope_include` form, not just the first time it is simulated.

Inference scopes can be referred to in inference expressions, thus providing a mechanism for associating custom inference strategies with different model fragments. For example, in a parameter estimation problem for hidden Markov models, it might be natural to have one scope for the hidden states, another for the hyperparameters, and a third for the parameters, where the blocks for the hidden state scope correspond to indexes into the hidden state sequence. We will see later how to

write cycle hybrid kernels that use Metropolis-Hastings to make proposals for the hyperparameters and either single-site or particle Gibbs over the hidden states. Inference scopes also provide a means of controlling the allocation of computational effort across different aspects of a hypothesis, e.g. by only performing inference over scopes whose random choices are conditionally dependent on the choices made by a given `PREDICT` instruction of interest.

Random choices are currently tagged with `(scope, block)` pairs. Blocks can be thought of as subdivisions of scopes into meaningful (and potentially ordered) subsets. We will see later how inference expressions can make use of block structure to provide fine-grained control over inference and enable novel inference strategies. For example, the order in which a set of random choices is traversed by conditional sequential Monte Carlo can be controlled via blocks, regardless of the order in which they were constructed during initial simulation.

Scopes and blocks can be produced by random choices; `<scope-expr>`s and `<block-expr>`s are ordinary Venture modeling expressions[4]. This enables the use of random choices in one scope to control the scope or block allocation of random choices in another scope. The random choices used to construct scopes and blocks may be auxiliary variables independent of the rest of the model, or latent variables whose distributions depend on the interaction of modeling assumptions, data and inference. At present, the only restriction is that inference on the random choices in a set of blocks cannot add or remove random choices from that set of blocks, though the probability of membership can be affected. Applications of randomized, inference-driven scope and block assignments include variants of cluster sampling techniques, beyond the spin glass (Swendsen and Wang, 1986) and regression (Nott and Green, 2004) settings where they have typically been deployed.

The implementation details needed to handle random scopes and blocks are beyond the scope of this paper. However, we will later see analytical machinery that is sufficient for justifying the correctness of complex transition operators involving randomly chosen scopes and blocks.

Venture provides two built-in scopes[5]:

1. `default` — This scope contains every random choice. Previously proposed inference schemes for Church as well as concurrently developed generic inference schemes for variants of Venture correspond to single-line inference instructions acting on this default, global scope.

2. `latents` — This scope contains all the latent random choices made by stochastic procedures but hidden from Venture. Using this scope, programmers can control the frequency with which any external inference systems are invoked, and interleave inference over external variables with inference over the latent variables managed by Venture.

### 2.4 Inference Expressions

*Inference expressions* specify transition operators that evolve the probability distribution on traces inside the Venture virtual machine. This is in contrast to *instructions*, which extend a model, add data, or initiate inference using a valid transition operator.

Venture provides several primitive forms for constructing transition operators that leave the conditioned distribution invariant, each of which is a valid inference expression. In each of these

4. Our current implementations restrict the values of scope and block names to symbols and integers for simplicity, but this restriction is not intrinsic to the Venture specification.
5. Some implementations so far have merged the default and latent scopes, or triggered inference over latents automatically after every transition over the default scope.

forms, `scope` must be a literal scope, and `block` must either be a literal block within that scope, or the keyword `one` or `all`. The "selected" set of random choices on which each inference expression acts is given by the specified scope and block. If the block specification is `all`, then the union of all blocks within the scope is taken. If the block specification is `one`, then one block is chosen uniformly at random from the set of all blocks within the given scope.

The core set of inference expressions in Venture are as follows:

1. `(mh <scope> <block> <#-transitions>)` — Propose new values for the selected choices either by resimulating them or by invoking a custom local proposal kernel if one has been provided. Accept or reject the results via the Metropolis-Hastings rule, accounting for changes to the mapping between random choices and scopes/blocks using the machinery provided later in this paper. Repeat the whole process `#-transitions` times.

2. `(rejection <scope> <block> <#-transitions>)` — Use rejection sampling to generate an exact sample from the conditioned distribution on all the selected random choices. Repeat the whole process `#-transitions` times, potentially improving convergence if the selected set is randomly chosen, i.e. `block` is `one`. This transition operator is often computationally intractable, but is optimal, in the sense that it makes the most progress per completed transition towards the conditioned distribution on traces. All the other transition operators exposed by the Venture inference language can be viewed as asymptotically convergent approximations to it.

3. `(pgibbs <scope> <block> <#-particles> <#-transitions>)` — Use conditional sequential Monte Carlo to propose from an approximation to the conditioned distribution over the selected set of random choices. If `block` is `ordered`, all the blocks in the scope are sorted, and each distribution in the sequence of distributions includes all the random choices from the next block. Otherwise, each distribution in the sequence includes a single random choice drawn from the selected set, and the ordering is arbitrary.

4. `(meanfield <scope> <block> <iters> <#-transitions>)` — Use `iters` steps of stochastic gradient to optimize the parameters of a partial mean-field approximation to the conditioned distribution over the random choices in the given `scope` and `block` (with block interpreted as with `mh`). Make a single Metropolis-Hastings proposal using this approximation. Repeat the process `#-transitions` times.

5. `(enumerative_gibbs <scope> <block> <#-transitions>)` — Use exhaustive enumeration to perform a transition over all the selected random choices from a proposal corresponding to the optimal conditional proposal (conditioned on the values of any newly created random variables). Random choices whose domains cannot be enumerated are resimulated from their prior unless they have been equipped with custom simulation kernels. If all selected random choices are discrete and no new random choices are created, this is equivalent to the `rejection` transition operator, and corresponds to a discrete, enumerative implementation of Gibbs sampling, hence the name. The computational cost scales exponentially with the number of random choices, as opposed to the KL divergence between the prior and the conditional (Freer et al., 2010).

Both `mh` and `pgibbs` are implemented by in-place mutation. However, versions of each that use simultaneous particles to represent alternative possibilities are given by `func-mh` and `func-pgibbs`.

These are prepended with `func` to signal an aspect of their implementation: these simultaneously accessible sets of particles are implemented using persistent data structure techniques typically associated with pure functional programming. `func-pgibbs` can yield improvements in order of growth of runtime as compared to `pgibbs`, but it imposes restrictions on the selected random choices[6].

There are also currently two composition rules for transition operators, enabling the creation of cycle and mixture hybrids:

1. `(cycle (<inference-expr-1> <inference-expr-2>) ... <#-transitions>)` — This produces a cycle hybrid of the transition operators represented by the given inference expression: each transition operator is run in sequence, and the whole sequence is repeated `#-transitions` times.

2. `(mixture ((<w1> <inference-expr-1>) (<w2> <inference-expr-2>) ...) <#-transitions>)` — This produces a mixture hybrid of the given transition operators, using the given mixing weights, that is invoked `#-transitions` times.

This language is flexible enough to express a broad class of standard approximate inference strategies as well as novel combinations of standard inference algorithm templates such as conditional sequential Monte Carlo, Metropolis-Hastings, Gibbs sampling and mean-field variational inference. Additionally, the ability to use random variables to map random choices to inference strategies and to perform inference over these variables may enable new cluster sampling techniques. That said, from an aesthetic standpoint, the current inference language also has many limitations, some of which seem straightforward to relax. For example, it seems natural to expand inference expressions to support arbitrary modeling expressions, and thereby also support arbitrary computation to produce inference schemes. The machinery needed to support these and other natural extensions is discussed later in this paper.

### 2.5 Values

Venture values include the usual scalar and symbolic data types from Scheme, along with extended support for collections and additional datatypes corresponding to primitive objects from probability theory and statistics. Venture also supports the *stochastic procedure* datatype, used for built-in and user-added primitive procedures as well as compound procedures returned by `lambda`. A full treatment of the value hierarchy is out of scope, but we provide a brief list of the most important values here:

1. **Numbers**: roughly analogous to floating point numbers, e.g. 1, 2.4, -23, and so on.

2. **Atoms**: discrete items with no internal structure or ordering. These are generated by categorical draws, but also Dirichlet and Pitman-Yor processes.

3. **Symbols**: symbol values, such as the name of a formal argument being passed to lambda, the name associated with an `ASSUME` instruction, or the result of evaluating a `quote` special form.

6. To support multiple simultaneous particles, all stochastic procedures within the given `scope` and `block` must support a clone operation for their auxiliary state storage (or have the ability to emulate it). This is feasible for standard exponential family models, but may be not be feasible for external inference systems hosted on distributed hardware.

4. **Collections:** vectors, which map numbers to values and support O(1) random access, and maps, which map values to values and support O(1) amortized random access (via a hash table that relies on the built-in hash function associated with each kind of value).

5. **Stochastic procedures:** these include the components of the standard library, and can also be created by `lambda` and other stochastic procedures.

### 2.6 Automatic inference versus inference programming

Although Venture programs can incorporate custom inference strategies, it does not *require* them. Interfaces that are as automatic as existing probabilistic programming systems are straightforward to implement. Single-site Metropolis-Hastings and Gibbs sampling algorithms — the sole automatic inference option in many probabilistic programming systems — can be invoked with a single instruction. We have also seen that global sequential Monte Carlo and mean field algorithms are similarly straightforward to describe. Support for programmable inference does not necessarily increase the education burden on would-be probabilistic programmers, although it does provide a way to avoid limiting probabilistic programmers to a potentially inadequate set of inference strategies.

The idea that inference strategies can be formalized as structured, compositionally specified inference programs operating on model programs is, to the best of our knowledge, new to Venture. Under this view, standard inference algorithms actually correspond to primitive inference programming operations or program templates, some of which depend on specific features of the model program being acted upon. This perspective suggests that far more complex inference strategies should be possible if the right primitives, means of combination and means of abstraction can be identified. Considerations of modularity, analyzability, soundness, completeness, and reuse will become central. and will be complicated by the interaction between inference programs and model programs. For example, inference programmers will need to be able to predict the asymptotic scaling of inference instructions, factoring out the contribution of the computational complexity of the model expressions to which a given inference instruction is being applied. Another example comes from considering abstraction and reuse. It should be possible to write compound inference procedures that can be reused across different models, and perhaps even use inference to learn these procedures via an appropriate hierarchical Bayesian formulation.

Another view, arguably closer to the mainstream view in machine learning, is that inference algorithms are better thought of by analogy to mathematical programming and operations research, with each algorithm corresponding to a "solver" for a well-defined class of problems with certain structure. This perspective suggests that there is likely to be a small set of monolithic, opaque mechanisms that are sufficient for most important problems. In this setting, one might hope that inference mechanisms can be matched to models and problems via simple heuristics, and that the problem of automatically generating high-quality inference strategies will prove easier than query planning for databases, and will be vastly easier than automatic programming.

It remains to be seen whether the traditional view is sufficient in practice or if it underestimates the richness of inference and its interaction with modeling and problem specification.

### 2.7 Procedural and Declarative Interpretations

We briefly consider the relationship between procedural and declarative interpretations of Venture programs.

Venture code has a direct procedural reading: it defines a probabilistic generative process that samples hypotheses, checks constraints, and invokes inference instructions that trigger specific algorithms for reconciling the hypotheses with the constraints. Re-orderings of the instructions can significantly impact runtime and change the distribution on outputs. The divergence between the true conditioned distribution on execution traces and the distribution encoded by the program may depend strongly on what inference instructions are chosen and how they are interleaved with the incorporation of data.

Venture code also has declarative readings that are unaffected by some of these procedural details. One way to formalize the of meaning of a Venture program is as a probability distribution over execution traces. A second approach is to ignore the details of execution and restrict attention to the joint probability distribution of the values of all `PREDICT`s so far. A third approach, consistent with Venture's interactive interface, is to equate the meaning of a program with the probability distribution of the values of all the `PREDICT`s in all possible sequences of instructions that could be executed in the future. Under the second and third readings, many programs are equivalent, in that they induce the same distribution albeit with different scaling behavior.

As the amount of inference performed at each `INFER` instruction increases, these interpretations coalesce, recovering a simple semantics based on sequential Bayesian reasoning. Consider replacing all inference instructions are replaced with exact sampling — `[INFER (rejection default one 1)]` — or a sufficiently large number of transitions of a generic inference operator, such as `[INFER (mh default one 1000000)]`. In this case, each `INFER` implements a single step of sequential Bayesian reasoning, conditioning the distribution on traces with all the `OBSERVE`s since the last `INFER`. The distribution after each `INFER` becomes equivalent to the distribution represented by all programs with the same `ASSUME`s, all the `OBSERVE`s before the infer (in any order), and a single `INFER`. The computational complexity varies based on the ordering and interleaving of `INFER`s and `OBSERVE`s, but the declarative meaning is unchanged. Although correspondence with these declarative, fully Bayesian semantics may require an unrealistic amount of computation in real-world applications, close approximations can be useful tools for debugging, and the presence of the limit may prove useful for probabilistic program analysis and transformation.

Venture programs represent distributions by combining modeling operations that sample values for expressions, constraint specification operations that build up a conditioner, and inference operations that evolve the distribution closer to the conditional distribution induced by a conditioner. Later in this paper we will see how to evaluate the partial probability densities of probabilistic execution traces under these distributions, as well as the ratios and gradients of these partial densities that are needed for a wide range of inference schemes.

### 2.8 Markov chain and sequential Monte Carlo architectures

The current Venture implementation maintains a single probabilistic execution trace per virtual machine instance. This trace is initialized by simulating the code from the `ASSUME` and `OBSERVE` instructions, and stochastically modified during inference via transition operators that leave the current conditioned distribution on traces invariant. The prior and posterior probability distributions on traces are implicit, but can be probed by repeatedly re-executing the program and forming Monte Carlo estimates.

This Markov chain architecture has been chosen for simplicity, but sequential Monte Carlo architectures based on weighted collections of traces are also possible and indeed straightforward.

The number of initial traces could be specified via an INFER instruction at the beginning of the program. Forward simulation would be nearly unchanged. OBSERVE instructions would attach weights to traces based on the "likelihood" probability density corresponding to the constrained random choice in each observation, and PREDICT instructions would read out their values from a single, arbitrarily chosen "active" trace. An [INFER (resample <k>)] instruction would then implement multinomial resampling, and also change the active trace to ensure that PREDICTs are always mutually consistent. Venture's other inference programming instructions could be treated as rejuvenation kernels (Del Moral et al., 2006), and would not need to modify the weights[7]. This kind of sequential Monte Carlo implementation would have the advantage that the weights could be used to estimate marginal probability densities of given OBSERVEs, and that another source of non-embarrassing parallelism would be exposed. Integrating sophisticated coupling strategies from the αSMC framework (Whiteley et al., 2013) into the inference language could also prove fruitful.

Running separate Venture virtual machines is guaranteed to produce independent samples from the distribution represented by the Venture program. This distribution will typically only approximate some desired conditional. If there is no need to quantify uncertainty precisely, then Venture programmers can append repetitions of a sequence of INFER and PREDICT instructions to their program. Unless the INFER instructions use rejection sampling, this choice yields PREDICT outputs that are dependent under both Markov chain and sequential Monte Carlo architectures. It only approximates the behavior of independent runs of the program. Application constraints will determine what approximation strategies are most appropriate for each problem.

## 2.9 Examples

Here we give simple illustrations of the Venture language, including some standard modeling idioms as well as the use of custom inference instructions. Venture has been also used to implement applications of several advanced modeling and inference techniques; examples include generative probabilistic graphics programming (Mansinghka*, Kulkarni*, Perov, and Tenenbaum, 2013) and topic modeling (Blei et al., 2003). A description of these and other applications is beyond the scope of this paper.

### 2.9.1 Hidden Markov Models

To represent a Hidden Markov model in Venture, one can use a stochastic recursion to capture the hidden state sequence, and index into it by a stochastic observation procedure. Here we give a variant with continuous observations, a binary latent state, and an a priori unbounded number of observation sequences to model:

```
[ASSUME observation_noise (scope_include 'hypers 'unique (gamma 1.0 1.0))]

[ASSUME get_state
  (mem (lambda (seq t)
      (scope_include 'state t
         (if (= t 0)
             (bernoulli 0.3)
             (transition_fn (get_state seq (- t 1))))))))]

[ASSUME get_observation
```

7. The only subtlety is that transition operators must not change which random choice is being constrained, as this would require changing the weight of the trace.

```
  (mem (lambda (seq t)
         (observation_fn (get_state seq t))))]

[ASSUME transition_fn
  (lambda (state)
      (scope_include 'state t (bernoulli (if state 0.7 0.3))))]

[ASSUME observation_fn
  (lambda (state)
    (normal (if state 3 -3) observation_noise))]

[OBSERVE (get_observation 1 1) 3.6]
[INFER (mh default one 10)]
[OBSERVE (get_observation 1 2) -2.8]
[INFER (mh default one 10)]
<...>
```

This is a sequentialized variant of the "default" resimulation-based Metropolis-Hastings inference scheme. If all but the last `INFER` statement were removed, the program would yield the same stochastic transitions as several Church implementations, but with linear (rather than quadratic) scaling in the length of the sequence. Interleaving inference with the addition of observations improves over bulk incorporation of observations by mitigating some of the strong conditional dependencies in the posterior.

Another inference strategy is particle Markov chain Monte Carlo. For example, one could combine Metropolis-Hastings moves on the hyperparameters, given the latent states, with a conditional sequential Monte Carlo approximation to Gibbs over the hidden states given the hyper parameters and observations. Here is one implementation of this scheme, where 10 transitions of Metropolis-Hastings are done on the hyper parameters for every 5 transitions of approximate Gibbs based on 30 particles, all repeated 50 times:

```
[INFER (cycle ((mh hypers one 10) (pgibbs state ordered 30 5)) 50)]
```

The global particle Gibbs algorithm from (Wood et al., 2014) with 30 particles would be expressed as follows:

```
[INFER (pgibbs default ordered 30 100)]
```

Note that Metropolis-Hastings transitions can be more effective than pure conditional sequential Monte Carlo for handling global parameters (Andrieu et al., 2010). This is because MH moves allow hyperparameter inference to be constrained by all latent states.

In real-time applications, hyperparameter inference is sometimes skipped. Here is one representation of a close relative[8] of a standard 30-particle particle filter that uses randomly chosen hyper parameters and yields a single latent trajectory:

```
[INFER (pgibbs state ordered 30 1)]
```

### 2.9.2 Hierarchical Nonparametric Bayesian Models

Here we show how to implement one version of a multidimensional Dirichlet process mixture of Gaussians (Rasmussen, 1999):

8. The only difference is that a particle filter exposes all its weighted particles for forming Monte Carlo expectations or for rapidly obtaining a set of approximate samples. Only straightforward modifications are needed for Venture to expose a set of weighted traces instead of a single trace and literally recover particle filtering.

```
[ASSUME alpha (scope_include 'hypers 0 (gamma 1.0 1.0))]
[ASSUME scale (scope_include 'hypers 1 (gamma 1.0 1.0))]

[ASSUME crp (make_crp alpha)]

[ASSUME get_cluster (mem (lambda (id)
  (scope_include 'clustering id (crp))))]

[ASSUME get_mean (mem (lambda (cluster dim)
  (scope_include 'parameters cluster (normal 0 10)))))]

[ASSUME get_variance (mem (lambda (cluster dim)
  (scope_include 'parameters cluster (gamma 1 scale))))]

[ASSUME get_component_model (lambda (cluster dim)
  (lambda () (normal (get_mean cluster dim) (get_variance cluster dim)))))]

[ASSUME get_datapoint (mem (lambda (id dim)
  ((get_component_model (get_cluster id dim))))))]

[OBSERVE (get_datapoint 0 0) 0.2]
<...>

; default resimulation-based Metropolis-Hastings scheme
[INFER (mh default one 100)]
```

The parameters are explicitly represented, i.e. "uncollapsed", rather than integrated out as they often are in practice. While the default resimulation-based Metropolis-Hastings scheme can be effective on this problem, it is also straightforward to balance the computational effort differently:

```
[INFER (cycle ((mh hypers one 1)
                       (mh parameters one 5)
                       (mh clustering one 5))
                       1000)]
```

On each execution of this `cycle`, one hyperparameter transition, five parameter transitions (each to both parameters of a randomly chosen cluster), and five cluster reassignments are made. Which hyperparameter, parameters and cluster assignments are chosen is random. As the number of data points grows, the ratio of computational effort devoted to inference over the hyperparameters and cluster parameters to the cluster assignments is higher than it is for the default scheme. Note that the complexity of this inference instruction, as well as the computational effort ratio, depend on the number of clusters in the current trace.

It is also straightforward to use a structured particle Markov chain Monte Carlo scheme over the cluster assignments:

```
[INFER (mixture ((0.2 (mh hypers one 10))
                           (0.5 (mh parameters one 5))
                           (0.3 (pgibbs clustering ordered 2 2)))
                           100)]
```

Due to the choice of only 2 particles for the `pgibbs` inference strategy, this scheme closely resembles an approximation to blocked Gibbs over the indicators based on a sequential initialization of the complete model. Also note that despite the low mixing weight on the `clustering` scope in the `mixture`, this inference program allocates asymptotically greater computational effort to inference over the cluster assignments than the previous strategy. This is because the `pgibbs` transition operator is guaranteed to reconsider every single cluster assignment.

### 2.9.3 INVERSE INTERPRETATION

We now describe *inverse interpretation*, a modeling idiom that is only possible in Turing-complete languages. Recall that Venture modeling expressions are easy to represent as Venture data objects, and Venture models can invoke the evaluation and application of arbitrary Venture stochastic procedures. These Scheme-like features make it straightforward to write an evaluator — perhaps better termed a simulator — for a Turing-complete, higher-order probabilistic programming language.

This application highlights Turing-completeness and also embodies a new potentially appealing path for solving problems of probabilistic program synthesis. In less expressive languages, learning programs (or structure) requires custom machinery that goes beyond what is provided by the language itself. In Venture, the same inference machinery used for state estimation or causal inference can be brought to bear on problems of probabilistic program synthesis. The dependency tracking and inference programming machinery that is general to Venture can be brought to bear on the problem of approximately Bayesian learning of probabilistic programs in a Venture-like language[9].

We first define some utility procedures for manipulating references, symbols and environments:

```
[ASSUME make_ref (lambda (x) (lambda () x))]
[ASSUME deref (lambda (x) (x))]

[ASSUME incremental_initial_environment
  (lambda ()
    (list
      (dict
        (list (quote bernoulli)
              (quote normal)
              (quote plus)
              (quote times)
              (quote branch))
        (list (make_ref bernoulli)
              (make_ref normal)
              (make_ref plus)
              (make_ref times)
              (make_ref branch))))))]

[ASSUME extend_env
  (lambda (outer_env syms vals)
    (pair (dict syms vals) outer_env))]

[ASSUME find_symbol
  (lambda (sym env)
    (if (contains (first env) sym)
        (lookup (first env) sym)
        (find_symbol sym (rest env)))))]
```

The most interesting of these are `make_ref` and `deref`. These use closures to pass references around the trace, using an idiom that avoids unnecessary growth of scaffolds. Consider an execution trace in which a value that is the argument to `make_ref` becomes the principal node of a transition. Only those uses of the reference to the value that have been passed to `deref` will become resampling nodes. The value of the reference is unchanged, though the value the reference refers to is not. This permits dependence tracking through the construction of complex data structures.

9. Although we are still far from a study of the expressiveness of probabilistic languages via definitional interpretation (Reynolds, 1972; Abelson and Sussman, 1983), it seems likely that probabilistic programming formulations of probabilistic program synthesis — inference over a space of probabilistic programs, possibly including inference instructions, and an interpreter for those programs — will be revealing.

Given this machinery, it is straightforward to write an evaluator for a simple function language that has access to arbitrary Venture primitives:

```
[ASSUME incremental_venture_apply
  (lambda (op args) (eval (pair op (map_list deref args)) (get_empty_environment)))]

[ASSUME incremental_apply
  (lambda (operator operands)
    (incremental_eval (deref (lookup operator 2))
                      (extend_env (deref (lookup operator 0))
                                  (deref (lookup operator 1))
                                  operands))))]

[ASSUME incremental_eval
  (lambda (expr env)
    (if (is_symbol expr)
        (deref (find_symbol expr env))
            (if (not (is_pair expr))
                expr
            (if (= (deref (lookup expr 0)) (quote lambda))
                          (pair (make_ref env) (rest expr))
                          ((lambda (operator operands)
                             (if (is_pair operator)
                               (incremental_apply operator operands)
                                 (incremental_venture_apply operator operands)))
                           (incremental_eval (deref (lookup expr 0)) env)
                           (map_list (lambda (x)
                                       (make_ref (incremental_eval (deref x) env)))
                                     (rest expr)))))))]
```

It is also possible to generate the input `expr`s using another Venture program, and use general-purpose, Turing-complete inference mechanisms to explore a hypothesis space of expressions given constraints on the values that result. We call this the *inverse interpretation* approach to probabilistic program synthesis. As in Church — and contrary to (Liang et al., 2010) — inverse interpretation algorithms are not limited to rejection sampling. But while Church was limited to a single-site Metropolis-Hastings scheme, Venture programmers have more options. In Venture it is possible to associate portions of the program source (and portions of the induced program's executions) with custom inference strategies.

Here is an example expression grammar that can be used with the incremental evaluator:

```
[ASSUME genBinaryOp (lambda () (if (flip) (quote plus) (quote times)))]
[ASSUME genLeaf (lambda () (normal 4 3))]
[ASSUME genVar (lambda (x) x)]

[ASSUME genExpr
  (lambda (x)
    (if (flip 0.4)
      (genLeaf)
      (if (flip 0.8)
         (genVar x)
         (list (make_ref (genBinaryOp)) (make_ref (genExpr x)) (make_ref (genExpr x)))))))]

[ASSUME noise (gamma 5 .2)]
[ASSUME expr (genExpr (quote x))]

[ASSUME f
  (mem
    (lambda (y)
      (incremental_eval expr
                        (extend_env (incremental_initial_environment)
```

```
                                          (list (quote x))
                                          (list (make_ref y)))))))]

[ASSUME g (lambda (z) (normal (f z) noise))]

[OBSERVE (g 1) 10] ;f(x) = 5x + 5
[OBSERVE (g 3) 20]
<...>
```

Indirection via references substantially improves the asymptotic scaling of programs like these. When a given production rule in the grammar is resimulated, only those portions of the execution of the program that depend on the changed source code are resimulated. A naive implementation of an evaluator would not have this property.

Scaling up this approach to larger symbolic expressions and small programs will require multiple advances. Overall system efficiency improvements will be necessary. Inverse interpretation also may benefit from additional inference operators, such as Hamiltonian Monte Carlo for the continuous parameters. Expression priors with inference-friendly structures would also help. For example, a prior where resimulation recovers some of the search moves from (Duvenaud et al., 2013) may be expressible by separately generating symbolic expressions and the contents of the environment into which they are evaluated. Longer term, it may be fruitful to explore formalizations of some of the knowledge taught to programmers using probabilistic programming.

## 3. Stochastic Procedures

Random choices in Venture programs arise due to the invocation of *stochastic procedures* (SPs). These stochastic procedures accept input arguments that are values in Venture and sample output values given those inputs. Venture includes a built-in stochastic procedure library, which includes SPs that construct other SPs, such as the SP `make_csp` which the special form `lambda` gets desugared to. Stochastic procedures can also be added as extensions to Venture, and provide a mechanism for incremental optimization of Venture programs. Model fragments for which Venture delivers inadequate performance can be migrated to native inference code that interoperates with the enclosing Venture program.

### 3.1 Expressiveness and extensibility

Many typical random variables, such as draws from a Bernoulli or Gaussian distribution, are straightforward to represent computationally. One common approach has been to use a pair of functions: a *simulator* that maps from an input space of values $\mathcal{X}$ and a stream of random bits $\{0,1\}^*$ to an output space of values $\mathcal{Y}$, and a *marginal density* that maps from $(x,y)$ pairs to $[0,\infty)$. This representation corresponds to the "elementary random procedures" supported by early Church implementations. Repeated invocations of such procedures correspond to IID sequences of random variables whose densities are known. While simple and intuitive, this simple interface does not naturally handle many useful classes of random objects. In fact, many objects that are easy to express as compound procedures in Church and in Venture cannot be made to fit in this form.

Stochastic procedures in Venture support a broader class of random objects:

1. **Higher-order stochastic procedures, such as `mem` (including stochastic memorization), `apply` and `map`.** Higher-order procedures may accept procedures as arguments, apply these

procedures internally, and produce procedures as return values. Stochastic procedures in Venture are equipped with a simple mechanism for handling these cases. In fact, it turns out that all structural changes to execution traces — including those arising from the execution of constructs that affect control flow, such as `IF` — can be handled by this mechanism. This simplifies the development of inference algorithms, and permits users to extend Venture by adding new primitives that affect the flow of control.

2. **Stochastic procedures whose applications are exchangeably coupled.** Examples include collapsed representations of conjugate models from Bayesian statistics, combinatorial objects from Bayesian nonparametrics such as the Chinese Restaurant and Pitman-Yor processes, and probabilistic sequence models (such as HMMs) whose hidden state sequences can be efficiently marginalized out. Support for these primitives whose applications are coupled is important for recovering the efficiency of manually optimized samplers, which frequently make use of collapsed representations. Whereas exchangeable primitives in Church are thunks, which prohibits collapsing many important models such as HMMs, Venture supports primitives whose applications are row-wise partially exchangeable across different sets of arguments. The formal requirement is that the cumulative log probability density of any sequence of input-output pairs is invariant under permutation.

3. **Likelihood-free stochastic procedures that lack tractable marginal densities.** Complex stochastic simulations can be incorporated into Venture even if the marginal probability density of the outputs of these simulations given the inputs cannot be efficiently calculated. Models from the literature on Approximate Bayesian Computation (ABC), where priors are defined over the outcome of forward simulation code, can thus be naturally supported in Venture. Additionally, a range of "doubly intractable" inference problems, including applications of Venture to reasoning about the behavior of approximately Bayesian reasoning agents, can be included using this mechanism.

4. **Stochastic procedures with external latent variables that are invisible to Venture.** There will always be models that admit specialized inference strategies whose efficiency cannot be recovered by performing generic inference on execution traces. One of the principal design decisions in Venture is to allow these strategies to be exploited whenever possible by supporting a broad class of stochastic procedures with custom inference over internal latent variables, hidden from the rest of Venture. The stochastic procedure interface thus serves as a flexible bridge between Venture and foreign inference code, analogous to the role that foreign function interfaces (FFIs) play in traditional programming languages.

### 3.2 Primitive stochastic procedures

Informally, a primitive stochastic prodecure (PSP) is an object that can simulate from a family of distributions indexed by some arguments. In addition to simulating, PSPs may be able to report the logdensity of an output given an input, and may incorporate and unincorporate information about the samples drawn from it using mutation, e.g. in the case of a conjugate prior. This mutation cannot be aribitrary: the draws from the PSP must remain row-wise partially exchangeable as discussed above. A PSP may also have custom proposal kernels, in which case it must be able to return its contribution to the Metropolis-Hastings acceptance rate. For example, the PSP that simulates

Gaussian random variables may provide a drift kernel that proposes small steps around its previous location, rather than resampling from the prior distribution.

Primitive stochastic procedures are parameterized by the following properties and behaviors:

1. `isStochastic()` — does this PSP consume randomness when it is invoked?

2. `canAbsorbArgumentChanges()` — can this PSP absorb changes to its input arguments? If `true`, then this PSP must correctly implement `logdensity()` as described below.

3. `childrenCanAbsorbAtApplications()` — does this PSP return an SP that implements the short-cut "absorbing at applications" optimization, needed to integrate optimized expressions for the log marginal probability of sufficient statistics in standard conjugate models.

4. `value = simulate(args)` — samples a `value` for an application of the PSP, given the arguments `args`

5. `logp = logdensity(value, args)` — an optional procedure that evaluates the log probability density[10] of an output given the input arguments $p_{psp}(\texttt{simulate}(\texttt{args}) = \texttt{value} | \texttt{args})$.

6. `incorporate(aux, value, args)` — incorporate the value stored in the variable named `value` into the auxiliary storage `aux` associated with the SP that contains the PSP. `incorporate()` is used to implement SPs whose applications are exchangeably coupled. While it is always sufficient to store and update the full set of `values` returned for each observed `args`, often only the counts (or some other sufficient statistics) are necessary.

7. `unincorporate(aux, value, args)` — remove `value` from the auxiliary storage, restoring it to a state consistent with all other values that have been `incorporated` but not `unincorporated`; this is done when an application is unevaluated.

### 3.3 The Stochastic Procedure Interface

The stochastic procedure interface specifies the contract that Venture primitives must satisfy to preserve the coherence of Venture's inference mechanisms. It also serves as the vehicle by which external inference systems can be integrated into Venture. This interface preserves the ability of primitives to dynamically create and destroy internal latent variables hidden from Venture and to perform custom inference over these latent variables.

#### 3.3.1 Definition

**Definition 3.1** (Stochastic procedure)**.** A stochastic prodecure is a pair (**request**, **output**) of PSPs, along with a latent variable simulator, where

1. **request** returns:

10. This density is implicitly defined with respect to a PSP-specific (but argument independent) choice of dominating measure. For PSPs that are guaranteed to produce discrete outputs, the measure is assumed to be the counting measure, so `logdensity` is equivalent to the log probability mass function. A careful measure-theoretic treatment of Venture is left for future work.

(a) A list of tuples $(\mathbf{addr}, \mathbf{expr}, \mathbf{env})$ that represent expressions whose values must be available to **output** before it can start its simulation. We refer to requests of this form as *exposed simulation requests* (ESRs),

(b) A list of opaque tokens that can be interpreted by the SP as the latent variables that **output** will need in order to simulate its output, along with the values of the exposed simulation requests. We refer to requests of this form as *latent simulation requests* (LSRs).

2. The latent variable simulator responds to LSRs by simulating any latent variables requested.

3. **output** returns the final output of the procedure, conditioned on the inputs, the results of any of the exposed simulation requests, and the results of any latent simulation requests.

### 3.3.2 Exposed Simulation Requests

We want our procedures to be able to pass $(\mathbf{expr}, \mathbf{env})$ pairs to Venture for evaluation, and make use of the results in some way. A procedure may also have multiple applications all make use of a shared evaluation, e.g. **mem**, and in these cases the procedure must take care to request the same **addr** each time, and the $(\mathbf{expr}, \mathbf{env})$ will only be evaluated the first time and then reused thereafter.

Specifically, an ESR request of the $(\mathbf{addr}, \mathbf{expr}, \mathbf{env})$ is handled by Venture as follows. First Venture checks the requesting SP's namespace to see if it already has an entry with address **addr**. If it does not, then Venture evaluates **expr** in **env**, and adds the mapping $\mathbf{addr} \rightarrow \mathbf{root}$ to the SP's namespace, where **root** is the root of the evaluation tree. If the SP's namespace does contain **addr**, then Venture can look up **root**. Either way, Venture wires in **root** as an extra argument to the output node.

### 3.3.3 Latent Simulation Requests and the Foreign Inference Interface

Some procedures may want to simulate and perform inference over latent variables that are hidden from Venture. Consider an optimized implementation of a hidden Markov model integrated into the following Venture program:

```
[ASSUME my_hmm (make_hmm 10 0.1 5 0.2)]
[OBSERVE (my_hmm 0 0) 0]
[OBSERVE (my_hmm 0 1) 1]
[OBSERVE (my_hmm 0 2) 0]
[INFER (mh default one 20)]
[PREDICT (my_hmm 0 3)]
[PREDICT (my_hmm 1 0)]
```

Here we have a constructor SP `(make_hmm <num-states> <transition-hyper> <num-output-symbols> <observation-hyper)` that generates an (uncollapsed) hidden Markov model by generating the rows of the transition and observation matrices at random. The assumption is that `make_hmm` is a primitive, although it would be straightforward to implement `make_hmm` as a compound procedure in Venture, using variations of the examples presented earlier. The constructor returns a procedure bound to the symbol `my_hmm` that permits observations from this process to be queried via `(my_hmm <sequence-id> <index>)`. The program adds a sequence fragment of length three then requests predictions for the next observation in the sequence, as well as the initial observation from an entirely new sequence.

This probabilistic program captures a common pattern: integrating Venture with an foreign probabilistic model fragment that can be dynamically queried and contains latent variables hidden from Venture. It is useful to partition the random choices in this program as follows. The transition and observation matrices of the HMM could be viewed as part of the value of the `my_hmm` SP and therefore returned by the output PSP of the `make_hmm` SP. The observations are managed by Venture as the applications of the `my_hmm` SP. The hidden states, however, are fully latent from the standpoint of Venture, yet need to be created, updated and destroyed as invocations of `my_hmm` are created or destroyed and as their arguments (or the arguments to `make_hmm`) change.

Venture makes it possible for procedures to instantiate latent variables only as necessary to simulate a given program. The mechanism is similar to that for exposed simulation requests, except in this case the requests–which we call latent simulation requests (LSRs)–are opaque to Venture. Venture only calls appropriate methods on the SP at appropriate times to ensure that all the bookkeeping is handled correctly.

This framework is straightforward to apply to `make_hmm` and the stochastic procedure(s) that it returns. If `my_hmm` is queried for an observation at time $t$, the requestPSP can return the time $t$ as an LSR. When Venture tells the HMM to simulate that LSR, the HMM will either do nothing if $x_t$ already exists in its internal store of simulations, or else continue simulating from its current position up until $x_t$. The outputPSP then samples $o_t$ conditioned on the latent $x_t$. If the application at time $t$ is ever unevaluated, Venture will tell the HMM to detach the LSR $t$, which will cause the HMM to place the latents that are no longer necessary to simulate the program into a "latentDB", which it returns to Venture. Later on, Venture may tell the HMM to restore latents from a latentDB, for example if a proposal is rejected and the starting trace is being restored.

The main reason to encapsulate latent variables in this way, as opposed to requesting them as ESRs, is so that the SP can use optimized implementations of inference over their values, potentially utilizing special-purpose inference methods. For example, the uncollapsed HMM can implement forwards-filtering backwards-sampling to efficiently sample the latent variables conditioned on all observations. Such procedures are integrated into Venture by defining an "Arbitrary Ergodic Kernel" (AEKernel) which Venture may call during inference, and which is simply a black-box to Venture. Note that this same mechanism may be used by SPs that do not make latent simulation requests at all, but which have latent variables instantiated upon creation, such as a finite-time HMM or an uncollapsed Dirichlet-Multinomial.

To implement this functionality, stochastic procedures must implement three procedures in addition to the procedures needed for their ESR requestor, LSR requestor and output PSPs:

1. `simulateLatents(aux, LSR, shouldRestore, latentDB)` — simulate the latents corresponding to `LSR`, using the tokens in `latentDB` (indexed by `LSR`) to find a previous value if `shouldRestore` is true.

2. `detachLatents(aux, LSR, latentDB)` — signal that the latents corresponding to the request `LSR` are no longer needed, and store enough information in `latentDB` so that the value can be recovered later.

3. `AEInfer(aux)` — Trigger the external implementation to perform inference over all latent variables using the contents of `aux`. It is often convenient for *simulateLatents* to store latent variables in the `aux` and for `incorporate` to store the return values of applications in tt aux, along with the arguments that produced them.

Examples of this use of the stochastic procedure interface can be found in current releases of the Venture system.

### 3.3.4 Optimizations for higher-order SPs

Venture provides a special mechanism that allows certain SPs to exploit the ability to quickly compute the logdensity of its applications. Consider the following program:

```
[ASSUME alpha (gamma 1 1)]
[ASSUME collapsed_coin (make_beta_bernoulli alpha alpha)]
[OBSERVE (collapsed_coin) False]
[OBSERVE (collapsed_coin) True]
<repeat 10^9 times>
[OBSERVE (collapsed_coin) False]
[INFER]
```

A hand-written inference scheme would only keep track of the counts of the observations, and could perform rapid Metropolis-Hastings proposals on alpha by exploiting conjugacy. On the other hand, a naive generic inference scheme might visit all one billion observation nodes to compute the acceptance ratio for each proposal. We can achieve this efficient inference scheme by letting `make_beta_bernoulli` be responsible for tracking the sufficient statistics from the applications of `collapsed_coin`, and for evaluating the log density of all those applications as a block. Stochastic procedures that return other stochastic procedures and implement this optimization are said to be *absorbing at applications*, often abbreviated AAA. We will discuss techniques for implementing this mechanism in a later section.

### 3.3.5 Auxiliary State

An SP is itself stateless, but may have an associated auxiliary store, called *SPAux*, that carries any mutable information. SPAuxs have several uses:

1. If an SP makes exposed simulation requests, Venture uses the SPAux to store mappings from the addresses of the ESRs to the node that stores the result of that simulation.

2. If a PSP keeps track of its sample counts or other sufficient statistics, such as the collapsed-beta-bernoulli which stores the number of **true**s and **false**s, it will store this information in the SPAux.

3. If an SP makes latent simulation requests, then all latent variables it simulates to respond to those requests are stored in the SPAux.

4. Some SPs may optionally store part of the value of the SP directly in the SPAux. This is necessary if an SP cannot easily store its value, its latents, and its outputs separately.

## 4. Probabilistic Execution Traces

Bayesian networks decompose probabilistic models into nodes for random variables, equipped with conditional probability tables, and edges for conditional dependencies. They can be interpreted as

probabilistic programs via the ancestral simulation algorithm (Frey, 1997). The network is traversed in an order consistent with the topology, and each random variable is generated given its immediate parents. Bayesian networks can also be viewed as expressing a function for evaluating the joint probability density of all the nodes in terms of a factorization given by the graph structure.

Here we describe *probabilistic execution traces* (PETs), which serve analogous functions for Venture programs and address the additional complications that arise from Turing-completeness and the presence of higher order probabilistic procedures. We also describe recursive procedures for constructing and destroying probabilistic execution traces as Venture modeling language expressions are evaluated and unevaluated.

### 4.1 Definition of a probabilistic execution trace

Probabilistic execution traces consist of a directed graphical model representing the dependencies in the current execution, along with the stateful auxiliary data for each stochastic procedure, the Venture program itself, and metadata for existential dependencies and exchangeable coupling. We will typically identify executions and PETs, and denote them via the symbols $\rho$ or $\xi$.

PETs contain the following nodes:

1. One constant node for the global environment.
2. One constant node for every value bound in the global environment, which includes all built-in SPs.
3. One constant node for every call to **eval** on a expression that is either self-evaluating or quoted.
4. One lookup node for every call to **eval** that triggers a symbol lookup.
5. One request node and one output node for every call to **eval** that triggers an SP application. We refer to the operator nodes and operand nodes of request nodes and output nodes, but note that these are not special node types.

PETs also contain the following edges:

1. One lookup edge to each lookup node from the node it is looking up.
2. One operator edge to every request node from its operator node.
3. One operator edge to every output node from its operator node.
4. One operand edge to every request node from each of its operand nodes.
5. One operand edge to every output node from each of its operand nodes.
6. One requester edge to every output node from its corresponding request node.
7. One ESR edge from the root node of every SP family to each SP application that requests it.

Every node represents a random variable and has a value that cannot change during an execution. The PET also includes the SPAux for every SP that needs one. Unlike the values in the nodes, the SPAuxs may be mutated during an execution, for example to increment the number of *true*s for a beta-bernoulli.

### 4.2 Families

We divide our traces into families: one Venture family for every *assume*, *predict*, *observe* directive, and one SP family for every unique ESR requested during forward simulation. Because of our uniform treatment of conditional simulation, executions of programs satisfy the following property: the structure of every family is a function of the expression only, and does not depend on the random choices made while evaluating that expression. The only part of the topology of the graph that can change is which ESRs are requested.

### 4.3 Exchangeable coupling

Given our exchangeability assumptions for SPs, we can cite (generalized) de Finetti (Orbanz and Roy) to conclude that there is, in addition to the observed random variables explicitly represented in the PET, one latent random variable $\theta_f$ for every SP $f$ in the PET corresponding to the unobserved de Finetti measure, and one latent variable $\theta_{f,\mathbf{args}}$ for every set of arguments **args** that $f$ is called on, with an edge from $f$'s maker-node to $\theta_f$, edges from $\theta_f$ to every $\theta_{f,\mathbf{args}}$, and an edge from $\theta_{f,\mathbf{args}}$ to every node corresponding to an application of the form $f(\mathbf{args})$. However, each $\theta_f$ and $\theta_{f,\mathbf{args}}$ is marginalized out by each $f$ by way of mutation on its SPAux, effectively introducing a hyperedge that indirectly couples the application nodes of all applications of $f$.

We have chosen not to represent these dependencies in the graphical structure of the PET for the following reasons. First, once we integrate out $\theta_f$ and $\theta_{f,\mathbf{args}}$, we can only encode these dependencies at all with directed edges once we fix a specific ordering for the applications. Second, for the orderings we will be interested in, the graph that combines both types of directed edges would be cyclic. This would complicate future efforts to develop a causal semantics for PETs and Venture programs.

### 4.4 Existential dependence and contingent evaluation

An SP family is existentially dependent on the nodes that request it as part of an ESR, in the sense that if at any point it is not requested by *any* nodes, then the family would not have been computed while simulating the PET. Existential dependence is then handled with garbage collection semantics, whereby an SP family can no longer be part of the PET if it is not selected by any $W$ nodes, and thus should be unevaluated.

### 4.5 Examples

Here we briefly give example PETs for simple Venture programs.

#### 4.5.1 TRICK COIN

This is a variant of our running example. It defines a model that can be used for inferences about whether or not a coin is tricky, where a trick coin is allowed to have any weight between 0 and 1. The version we use includes one observation that a single flip of the coin came up heads.

To keep the PET as simple as possible, we give both the program and the PET in the form where `IF` has already been desugared into an SP application: `(IF <predicate> <consequent> <alternate>)` has been replaced with `(branch <predicate> (quote <consequent>) (quote <alternate>))`, where `branch` is an ordinary stochastic procedure.

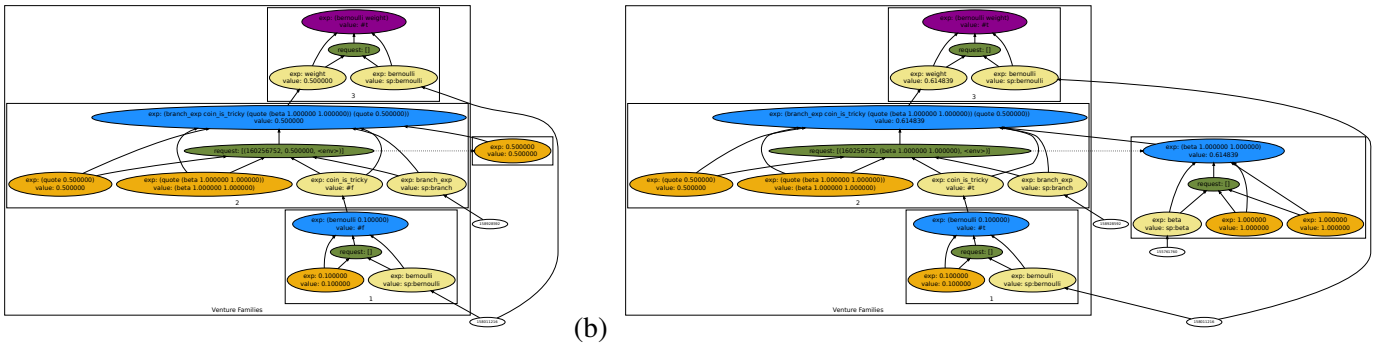



Figure 1: **The two different PET structures corresponding to the trick coin program.** (a) An execution trace where the coin is fair. (b) An execution trace where the coin is tricky, containing additional nodes that depend existentially on the coin flip.

```
[ASSUME coin_is_tricky (bernoulli 0.1)]
[ASSUME weight (branch coin_is_tricky (quote (beta 1.0 1.0)) (quote 0.5))]
[OBSERVE (bernoulli weight) true]
```

Figure 1 shows the two PET structures that can arise from simulating this program, along with arbitrarily chosen values.

### 4.5.2 A SIMPLE BAYESIAN NETWORK

Figure 2 shows a PET for the following program, implementing a simple Bayesian network:

```
[ASSUME rain (bernoulli 0.2)]
[ASSUME sprinkler (bernoulli (branch rain 0.01 0.4))]
[ASSUME grassWet
    (bernoulli (branch rain
                       (branch sprinkler 0.99 0.8)
                       (branch sprinkler 0.9 0.00001)))]
[OBSERVE grassWet True]
```

Note that each of the Venture families in the PET corresponds to a node in the Bayesian network. A coarsened version of the PET contains the same conditional dependence and independence information as the Bayesian network would.

### 4.5.3 STOCHASTIC MEMOIZATION

We now illustrate a program that exhibits stochastic memoization. This program constructs a stochastically memoized procedure and then applies it three times. Unlike deterministic memoization, a stochastically memoized procedure has a stochastic **request** PSP which sometimes returns a previously sampled value, and sometimes samples a fresh one. These random choices follow a Pitman-Yor process. Figure 3 shows a PET corresponding to a typical simulation; note the overlapping requests.

```
[ASSUME f (pymem bernoulli 1.0 0.1)]
```

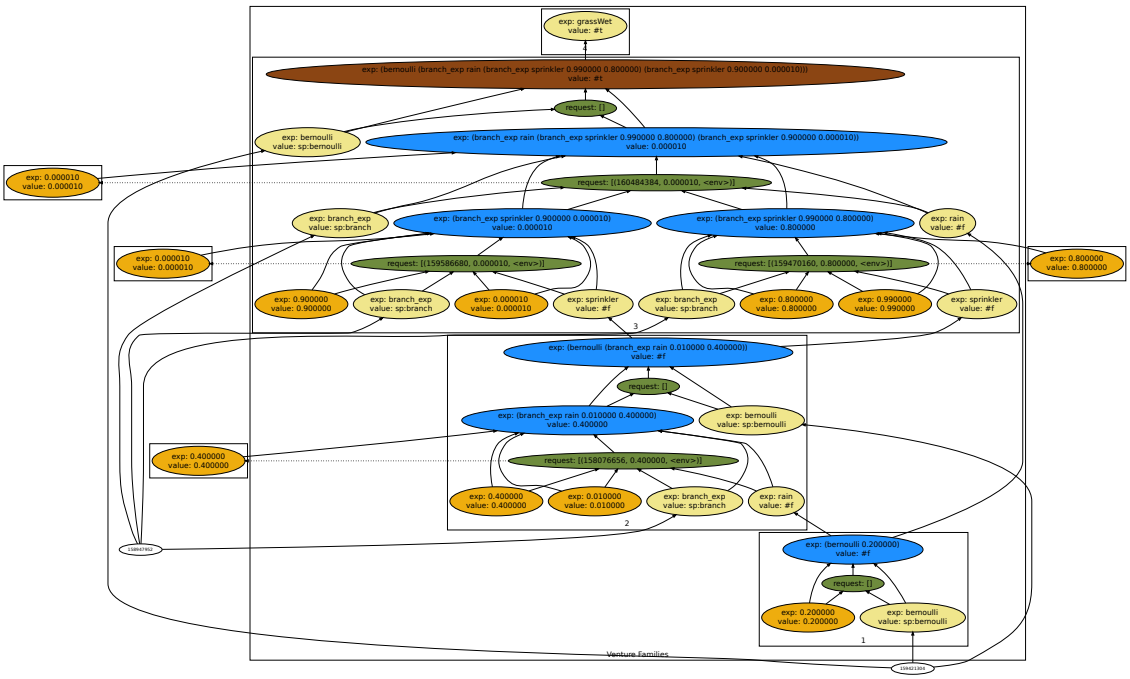



Figure 2: **The PET corresponding to a three-node Bayesian network.** Each of the three numbered families correspond to a node in the Bayesian network, capturing the execution history needed to simulate the node given its parents.

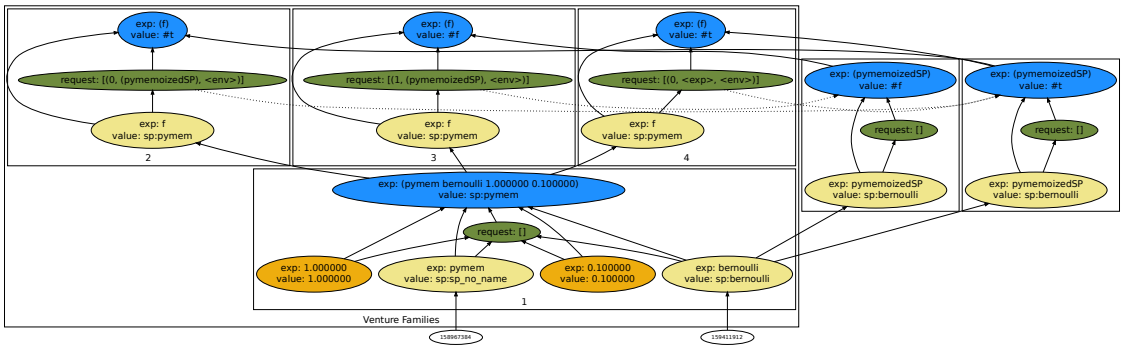



Figure 3: **A PET corresponding to an execution of a program with stochastic memoization.** The procedure being stochastically memoized, `bernoulli`, is applied twice, based on the value of the requests arising in invocations of `(f)`.

```
[PREDICT (f)]
[PREDICT (f)]
[PREDICT (f)]
```

### 4.6 Constructing PETs via forward simulation

Let **P** be a constrained program. Venture's primary inference strategies require an execution of **P** with positive probability before they can even begin. Therefore the first thing we do is simply evaluate the program by interpreting it in a fairly standard way, except with all conditional evaluation handled uniformly through the ESR machinery presented above. For simplicity, we elide details related to inference scoping.

#### 4.6.1 Pseudocode for EVAL, APPLY and EVAL-REQUESTS

Evaluation is generally similar to a pure Scheme, but there are a few noteworthy differences. First, evaluation creates nodes for every recursive call, and connects them together to form the directed graph structure of the probabilistic execution trace. Second, there are no distinctions between primitive procedures and compound procedures. We call the top-level evaluation procedure EVALFAMILY to emphasize the family block structure in PETs. Third, a *scaffold* and a database of random choices *db* are threaded through the recursions, to support their use in inference, including restoring trace fragments when a transition is rejected by reusing random choices from the *db*. Note that an environment model evaluator (Abelson and Sussman, 1983) is used, even though the underlying language is pure, with the PET storing both the environment structure and the recursive invocations of `eval`.

```
EVALFAMILY(trace, exp, env, scaffold, db)
 1  if isSelfEvaluating?(exp)
 2      return (0, trace.createConstantNode(exp))
 3  elseif isQuoted?(exp)
 4      return (0, trace.createConstantNode(TextOfQuotation(exp)))
 5  elseif isVariable?(exp)
 6      sourceNode = env.findSymbol(exp)
 7      REGENERATE(trace, exp, scaffold, FALSE, db)
 8      return (0, trace.createLookupNode(sourceNode)
 9  else
10      weight, operatorNode = EVALFAMILY(trace, exp.operator, env, scaffold, db)
11      operandNodes = []
12      for operand in exp.operands
13          (w, operandNode) = EVALFAMILY(trace, operand, env, scaffold, db)
14          weight = weight + w
15          operandNodes.append(operandNode)
16
17      (requestNode, outputNode) = trace.createApplicationNodes(operatorNode, operandNodes, env)
18      weight = weight + APPLYSP(trace, requestNode, outputNode, scaffold, FALSE, db)
19      return (weight, outputNode)
```

The call to REGENERATE ensures that if EVALFAMILY is called in the context of a pre-existing PET, it is traversed in an order that is compatible with the dependence structure of the program. We will sometimes refer to such orders as "evaluation-consistent". From the standpoint of forward simulation, however, REGENERATE can be safely ignored.

```
APPLYSP(trace, requestNode, outputNode, scaffold, restore?, db)
1  weight = APPLYPSP(trace, requestNode, scaffold, restore?, db)
2  weight = weight + EVALREQUESTS(trace, requestNode, scaffold, restore?, db)
3  weight = weight + REGENERATEESRPARENTS(trace, outputNode, scaffold, restore?, db)
4  weight = weight + APPLYPSP(trace, outputNode, scaffold, restore?, db)
5  return weight
```

Stochastic procedures are allowed to request evaluations of expressions[11]; this enables higher-order procedures as well as the encapsulation of custom control flow constructs. For example, compound procedures do this to evaluate their body in an environment extended with their formal parameters and argument values.

11. To prevent this flexibility from introducing arbitrary dependencies, the expressions and environments are restricted to those constructible from the arguments to the procedure or the procedure that constructed it.

```
EVALREQUESTS(trace, node, scaffold, restore?, db)
 1  sp, spaux = trace.getSP(node), trace.getSPAux(node)
 2  weight = 0
 3  request = trace.getValue(node)
 4  for id, exp, env, block, subblock in request.esrs
 5      if not spaux.containsFamily(id)
 6          if restore?
 7              esrParent = db.getESRParent(sp, id)
 8              weight = weight + RESTOREFAMILY(trace, esrParent, scaffold, db)
 9          else
10              (w, esrParent) = EVALFAMILY(trace, exp, env, scaffold, db)
11              weight = weight + w
12          trace.registerFamily(node, id, esrParent)
13      else
14          weight = weight + REGENERATE(trace, spaux.getFamily(id), scaffold, restore?, db)
15      esrParent = spaux.getFamily(id)
16      if not block == NONE
17          trace.registerBlock(block, subblock, esrParent)
18      trace.addESREdge(esrParent, node.outputNode)
19  for lsr in request.lsrs
20      if db.hasLatentDBFor(sp)
21          latentDB = db.getLatentDB(sp)
22      else
23          latentDB = NONE
24      weight = weight + sp.simulateLatents(spaux, lsr, restore?, latentDB)
25  return weight
```

Stochastic procedures are also allowed to perform opaque operations to produce outputs given the value of their arguments and any requested evaluations. Note that this may result in random choices being added to the record maintained by the trace.

```
APPLYPSP(trace, node, scaffold, restore?, db)
 1  psp, args = trace.getPSP(node), trace.getArgs(node)
 2  if db.hasValueFor(node)
 3      oldValue = db.getValue(node)
 4  else
 5      oldValue = NONE
 6  // Determine new value
 7  if shouldRestore
 8      newValue = oldValue
 9  elseif scaffold.hasKernelFor(node)
10      newValue = scaffold.getKernel(node).simulate(trace, oldValue, args)
11  else
12      newValue = psp.simulate(args)
13  // Determine weight
14  if scaffold.hasKernelFor(node)
15      weight = scaffold.getKernel(node).weight(trace, newValue, oldValue, args)
16  else
17      weight = 0
18  trace.setValue(node, newValue)
19  psp.incorporate(newValue, args)
20  if isSP?(newValue)
21      PROCESSMADESP(trace, node, scaffold.isAAA?(node))
22  if psp.isRandom?()
23      trace.registerRandomChoice(node)
24  return weight
```

This functionality is supported by book-keeping to link new stochastic procedures into the trace, and register any customizations they implement with the trace so that inference transitions can make use of them:

```
PROCESSMADESP(trace, node, isAAA?)
1  sp = trace.getValue(node)
2  trace.setMadeSP(node, sp)
3  trace.setValue(node, spref(sp))
4  if not isAAA?
5      trace.setMadeSPAux(node, sp.constructSPAux())
6      if sp.hasAEKernel()
7          trace.registerAEKernel(node)
```

### 4.7 Undoing simulation of PET fragments

Venture also includes requires unevaluation procedures that are dual to the evaluation procedures described earlier. This is because PET fragments need to be removed from the trace in two situations. First, when `FORGET` instructions are triggered, the expression corresponding to the directive is removed. Second, during inference, changes to the values of certain requestPSPs make cause some

SP families to no longer be requested. In the first case, all of the random choices are permanently removed, whereas in the second case, the random choices may need to be restored if the proposal is rejected.

### 4.7.1 PSEUDOCODE FOR UNEVAL, UNAPPLY AND UNEVAL-REQUESTS

When a trace fragment is unevaluated, we must visit all application nodes in the trace fragment so that the PSPs have a chance to unincorporate the (input,output) pairs. The operations needed to do this are essentially inverses of the simulation procedures described above, designed to visit nodes in the reverse order that evaluation does, to ensure compatibility with exchangeable coupling. Here we give pseudocode for these operations, eliding the details of garbage collection[12]:

```
UNEVALFAMILY(trace, node, scaffold, db)
 1  if node.isConstantNode?
 2      weight = 0
 3  elseif node.isLookupNode?
 4      trace.disconnectLookup(node)
 5      weight = EXTRACT(trace, node.sourceNode, scaffold, db)
 6  else
 7      weight = UNAPPLYSP(trace, node, scaffold, FALSE, db)
 8      for operandNode in node.operandNodes
 9          weight = weight + UNEVALFAMILY(trace, operandNode, scaffold, db)
10      weight = weight + UNEVALFAMILY(trace, node.operatorNode, scaffold, db)
11  return weight
```

To unapply a stochastic procedure, Venture must undo its random choices and also unapply any requests it generated:

```
UNAPPLYSP(trace, node, scaffold, db)
1  weight = UNAPPLYPSP(trace, node, scaffold, db)
2  weight = weight + EXTRACTESRPARENTS(trace, outputNode, scaffold, db)
3  weight = weight + UNEVALREQUESTS(trace, node.requestNode, scaffold, db)
4  weight = weight + UNAPPLYPSP(trace, node.requestNode, scaffold, db)
5  return weight
```

To unapply a primitive stochastic procedure, we remove its random choices from the trace, unincorporate them, and update the weight accordingly. Note that we store the value in the *db* so that we can restore it later if necessary.

12. Previous work has anecdotally explored the possibility of preserving unused trace fragments and treating them as auxiliary variables, following the treatment of component model parameters from Algorithm 8 in (Neal, 1998). In theory, this delay of garbage collection, where multiple copies of each trace fragment are maintained for each branch point, could support adaptation to the posterior. A detailed empirical evaluation of these strategies is pending a comprehensive benchmark suite for Venture as well as a high-performance implementation.

```
UNAPPLYPSP(trace, node, scaffold, db)
psp, args = trace.getPSP(node), trace.getArgs(node)
if psp.isRandom()
    trace.unregisterRandomChoice(node)
if isSP(trace.getValue(node))
    TEARDOWNMADESP(trace, node, scaffold.isAAA(node))
oldValue = trace.getValue(node)
psp.unincorporate(oldValue, args)
if scaffold.hasKernelFor(node)
    weight = scaffold.getKernel(node).reverseWeight(trace, oldValue, args)
else
    weight = 0
db.extractValue(node, oldValue)
trace.clearValue(node)
return weight
```

Handling of requests during unevaluation involves two additional subtleties. First, latent random choices must be handled appropriately. Second, requests must be unevaluated only if no other application of the SP refers to them.

```
UNEVALREQUESTS(trace, node, scaffold, db)
sp, spaux = trace.getSP(node), trace.getSPAux(node)
weight = 0
request = trace.getValue(node)
if request.lsrs and not db.hasLatentDB(trace.getSP(node))
    db.registerLatentDB(sp, sp.constructLatentDB())
for lsr in reversed(request.lsrs)
    weight = weight + sp.detachLatents(spaux, lsr, db.getLatentDB(sp))
for id, exp, env in reversed(request.esrs)
    esrParent = trace.popLastESRParent(node.outputNode)
    if trace.getNumberOfRequests(esrParent) == 0
        trace.unregisterFamily(node, id)
        db.registerFamily(sp, id, esrParent)
        weight = weight + UNEVALFAMILY(trace, esrParent, scaffold, db)
return weight
```

When an SP is finally destroyed — i.e. when the output PSP of its maker SP application is unapplied — then the SP can be destroyed and its auxiliary storage can be garbage collected. Because unapplication of the maker must happen after all applications of the made SP have been unapplied — due to the constraint that unevaluation visits nodes in the reverse order of evaluation and regeneration — there is no information in the auxiliary storage that needs to be preserved in the latent DB.

```
TEARDOWNMADESP(trace, node, isAAA?)
1  sp = trace.getMadeSP(node)
2  trace.setValue(node, sp)
3  trace.clearMadeSP(node)
4  if not isAAA?
5      if sp.hasAEKernel()
6          trace.unregisterAEKernel(node)
7      trace.clearMadeSPAux(node)
```

### 4.8 Enforcing constraints via CONSTRAIN and UNCONSTRAIN

Unfortunately, the observations may not have positive probability given the program execution. Venture may not happen to sample the right data, and but if noise is added to each data generating expression to ensure positive probability, it may take a long time for Venture to incorporate the observed data at all. On the other hand, the general problem of finding a program execution for which some output has positive probability is intractable in general – it can encode **SAT** without even making use of contingent evaluation or special primitives – and Venture is not designed to attempt to invert such programs.

Venture implements a middle ground. It is designed with sufficiently stochastic probabilistic programs in mind, and in particular is not a **SAT**-solver. However, in order to support common cases from Bayesian statistics, we do support inverting a simple kind of determinism. We introduce a method `constrain(<directive>,<value>)` that recursively walks backwards along *id*-**on-requests** edges to find the outermost application of a different kind of PSP. If the constrained value has positive probability at this PSP application, we constrain the value and we are done. Otherwise, we unevaluate the entire program and evaluate it again, in hopes that this execution will be constrainable.

Simple versions of constrain and unconstrain can be implemented as follows:

```
CONSTRAIN(trace, node, value)
 1  if node.isLookupNode?
 2      return CONSTRAIN(trace, node.sourceNode, value)
 3  elseif trace.getPSP(node).isESRReference?
 4      return CONSTRAIN(trace, trace.getESRParents(node)[0], value)
 5  else
 6      psp, args = trace.getPSP(node), trace.getArgs(node)
 7      psp.unincorporate(trace.getValue(node), args)
 8      weight = psp.logDensity(value, args)
 9      trace.setValue(node, value)
10      psp.incorporate(value, args)
11      if psp.isRandom?
12          trace.registerConstrainedChoice(node)
13      return weight
```

UNCONSTRAIN(*trace*, *node*)

```
1  if node.isLookupNode?
2      return UNCONSTRAIN(trace, node.sourceNode)
3  elseif trace.getPSP(node).isESRReference?
4      return UNCONSTRAIN(trace, trace.getESRParents(node)[0])
5  else
6      oldValue = trace.getValue(node)
7      psp, args = trace.getPSP(node), trace.getArgs(node)
8      if psp.isRandom?
9          trace.unregisterConstrainedChoice(node)
10     psp.unincorporate(oldValue, args)
11     weight = psp.logDensity(oldValue, args)
12     psp.incorporate(oldValue, args)
13     return weight
```

It is instructive to consider the following program:

```
[ASSUME x_1 (bernoulli 0.5)]
[ASSUME x_2 (bernoulli 0.5)]
...
[ASSUME x_N (bernoulli 0.5)]
[OBSERVE (xor x_1 ... x_N) True]
```

If `xor()` happens to be true by chance — i.e. `constrain()` did not have to adjust any random choices — then the PET that was sampled is already drawn from the true conditioned distribution. However, the probability of this decreases rapidly in `N` and because `xor()` is deterministic, repeated rejection may appear to be the only option. To avoid this problem, we frequently restrict ourselves to `OBSERVE`s whose expressions have the form `(<some-stochastic-SP> ...)`, i.e. they are applications of some fixed stochastic procedure. Then we can always successfully initialize and begin inference. Venture was designed with such "sufficiently stochastic" probabilistic programs in mind; see (Freer et al., 2010) for some rationale. We also note that to ensure ergodicity, periodic whole-trace proposals that attempt to restart from the prior (via the `INFER` directive) are sufficient, although this kind of independence sampler is unlikely to be efficient.

We note that without additional restrictions on observations, the presence of `constrain()` could introduce significant complexity. Constraining a node $X$ that is used in multiple places would not necessarily give us an execution with positive probability, since the probability of all children of $X$ will now have changed.

To resolve this and to simplify our algorithms, we make the following strong assumption on Venture programs:

*For a Venture program to be valid, it must guarantee that if a node gets constrained during inference (as opposed to from a call to observe), every node along every outgoing directed path must be either a deterministic output PSP, a null-request PSP, or a lookup node. Moreover, any observed nodes on this path must agree on the observed value.*

In addition to the code given above, our implementation of constrain also propagates the constrained value along these outgoing deterministic paths, but the details of the implementation are beyond the scope of this paper.

## 5. Partitioning Traces into Scaffolds for Scalable, Incremental Inference

Most scalable inference algorithms for Bayesian networks take advantage of the network's representation of conditional dependence. We have seen that probabilistic execution traces make analogous scalability gains possible for probabilistic programs written in Turing-complete, higher-order languages, by representing not just conditional dependences but also existential dependencies and exchangeable coupling.

However, PETs and the SPI also introduce complexities that can be avoided in the setting of Bayesian networks. For example, in a Bayesian network, the random variables that could possibly directly depend on a given random variable are easy to identify: they are the immediate children of the node according to the direct graph structure of the network. In a PET, the value of a random choice may change which children exist. Additionally, in Bayesian networks, all nodes are equipped with conditional probability densities, but SPs that lack conditional density functions are permitted in Venture. Joint densities for chains of random choices in Venture may not be available, let alone the conditional density of a choice given its immediate descendants.

Here we describe *scaffolds*, the mechanism used in Venture to handle these complexities. Scaffolds carve out coherent subproblems of inference in the context of a global inference problem analogously to the inference subproblems in a Bayesian network that arise by conditioning on a subset of the nodes and querying the nodes contained within it.

### 5.1 Motivation and Notation

Let $\rho$ be an execution and let $\mathbf{X} = \{X\}$ denote a set of nodes that we wish to propose new values for as a block. If we only propose new values to these nodes, we may end up with a trace that has probability 0. For example, consider the following program:

```
[ASSUME x (normal 0 1)]
[PREDICT (+ x 1)]
```

If we only propose a change to the value of $x$ from 1 to 2, and do not propagate the change through the application of $+$, then $+$ will report probability 0 of sampling its output 2 given its inputs 1 and 2.

Similarly, in the program

```
[ASSUME x (normal 0 1)]
[PREDICT (if (< x 0) (normal -10 1) (normal 10 1))]
```

if we only propose a new value for $x$ and its sign flips, then the requester for IF will report probability 0 of its old request given its new inputs.

SPs that cannot report their likelihoods introduce a similar problem. In the program

```
[ASSUME x (normal 0 1)]
[PREDICT (likelihood-free-sp x)]
```

if we only propose a new value for $x$, then we cannot compute the Metropolis-Hastings acceptance ratio since we have no way to account for the probability of the likelihood-free-sp's old value given its new arguments.

In all three cases, the problem can be solved by proposing new values to other nodes downstream of the nodes in $\mathbf{X}$. However, we do not want to resimulate the rest of the program just to make a single proposal. We want to find a middleground between rejecting at $+$ and running the entire program. As a default, we propose new values to downstream nodes until we reach applications of stochastic PSPs for which we can definitely compute the logdensity of its original output given any new values for its inputs. We say that these nodes "absorb the flow of change" and so we call them *absorbing nodes*.

Note that in the case of the IF statement that may switch from one branch to another, the nodes in the old branch may no longer exist in the proposed trace, and other nodes may come into existence.

To simplify notation, we will not explicitly condition on the parents of a random variable in our derivations. For example, in the Bayesian network $A \to B$, we will write $P(A)$ for $P(A)$ and $P(B)$ for $P(B|A)$. Although this would be unacceptable in the case of Bayesian networks where we marginalize and condition arbitrarily, in our discussion of probabilistic programming we will only ever compute probabilities of nodes given their parents, so that this notation will not be problematic.

### 5.2 Partitioning traces and defining scaffolds

As before, let $\rho$ be an execution and let $\mathbf{X} = \{X\}$ denote a set of nodes that we wish to propose new values for as a block. We call $\mathbf{X}$ the *principal nodes* of the proposal. Let $\xi$ denote the proposed trace. We introduce the following definitions:

1. $\mathbf{D} = \mathbf{D}(\rho, \mathbf{X})$: the nodes that will definitely still exist in $\xi$, whose values might change. This includes $\mathbf{X}$, as well as any nodes at which we do not want to absorb at (e.g. deterministic nodes) or are unable to absorb at (e.g. applications of likelihood-free SPs). Note that we have $\mathbf{D}(\rho) = \mathbf{D}(\xi) = \mathbf{D}$ by symmetry.

2. $\mathbf{brush}(\rho) = \mathbf{brush}(\rho, \mathbf{X})$: the nodes that may no longer be requested if $\mathbf{D}$ is resampled. This includes any nodes in branches that may be abandoned, or any requests whose operator may change.

3. $\mathbf{R}(\rho) = \mathbf{R}(\rho, \mathbf{X}) = \mathbf{D}(\rho, \mathbf{X}) \cup \mathbf{brush}(\rho, \mathbf{X})$: the regenerated subset of $\rho$. These are all the nodes that may either change value or no longer exist in $\xi$. Conversely $\mathbf{R}(\xi)$ consists of precisely the set of new values we are proposing for $\xi$.

4. $\mathbf{A} = \mathbf{A}(\rho, \mathbf{X})$: the children of $\mathbf{D}$, or equivalently, the "absorbing" nodes. For each absorbing, the differences between the logdensity of the output given the new value and the logdensity of the output of the old value appears in the Metropolis-Hastings acceptance ratio.

5. $\mathbf{torus}(\rho, \mathbf{X}) = \rho \setminus \mathbf{R}(\rho)$: the nodes in $\rho$ that are guaranteed to still exist in $\xi$, and whose values are guaranteed not to change. By construction we have $\mathbf{torus}(\rho) = \mathbf{torus}(\xi) = \mathbf{torus}$.

6. $\mathbf{P}(\rho) = \mathbf{P}(\rho, \mathbf{X})$: all parents of all nodes in $\mathbf{R}(\rho) \cup \mathbf{A}$, excluding any nodes in those sets. These are the nodes whose values would have been required to simulate $\mathbf{R}(\rho)$ from $\mathbf{torus}$ and to calculate the new log densities at $\mathbf{A}$, but which definitely exist and whose values cannot change.

7. $\mathbf{I}(\rho)$: the set of all nodes that would never be referenced at all while regenerating $\mathbf{R}(\rho)$.

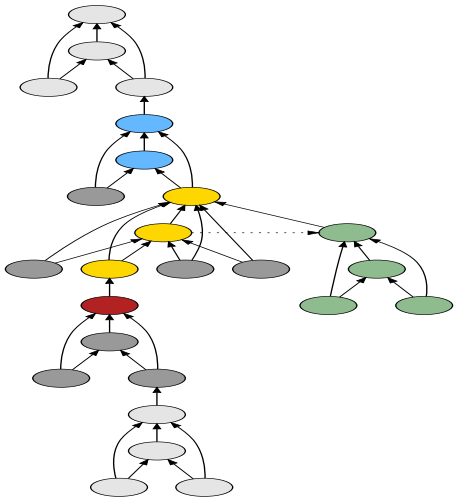


Figure 4: The scaffold partitions the trace into five groups: the gold nodes that will definitely still exist in the proposal trace but whose values may change (drg), the blue nodes at which we will definitely compute likelihoods (absorbing), the green nodes that may no longer exist (the brush), the dark grey nodes that are parents of nodes in these three groups (parents), and all other light grey nodes that need never be visited at all (ignored). Note that future graphs will not distinguish between these last two groups.

8. $\mathbf{O} = \mathbf{P}(\rho) \cup \mathbf{I}(\rho) = \mathbf{P}(\xi) \cup \mathbf{I}(\xi)$. Note that we may not have $\mathbf{P}(\rho) = \mathbf{P}(\xi)$, since $\mathbf{R}(\rho)$ and $\mathbf{R}(\xi)$ may lookup different symbols in different environments, and may also request different simulations.

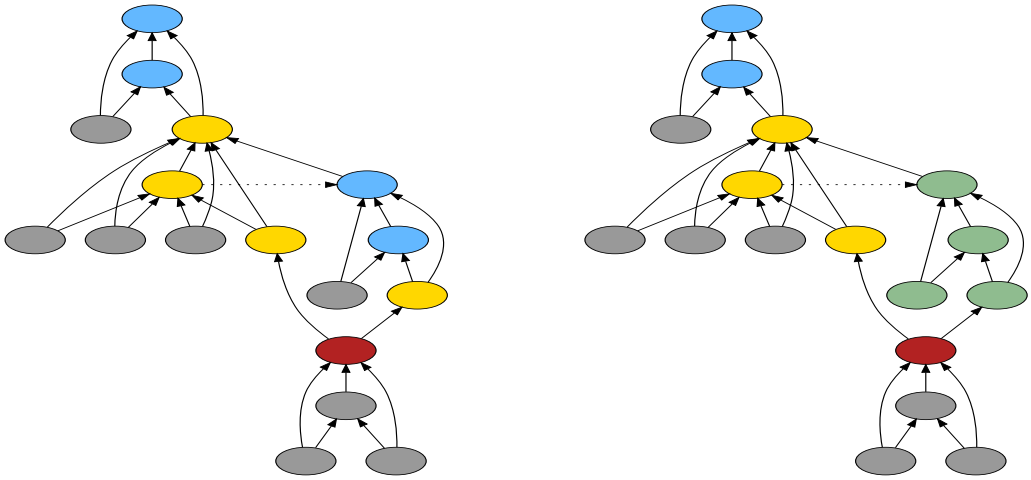
These definitions give us the following partitions:

$$\rho = \mathbf{D} \cup \mathbf{R}(\rho) \cup \mathbf{A} \cup \mathbf{O} \tag{1}$$
$$\xi = \mathbf{D} \cup \mathbf{R}(\xi) \cup \mathbf{A} \cup \mathbf{O} \tag{2}$$

We refer to the pair $(\mathbf{D}, \mathbf{A})$ as the **scaffold** of the set $\mathbf{X}$ of principal nodes. As we will see, the scaffold is the fundamental entity in the inference methods we will consider. It induces a set $\Xi = \Xi[\mathbf{O}, \textbf{scaffold}]$ of executions that can be reached by resimulating along the scaffold, consulting parents in $\mathbf{O}$ as needed. Moreover, for any $\xi \in \Xi$, the scaffold **scaffold** will yield the same set $\mathbf{O}$ in the factorization $\xi = \textbf{scaffold} \cup \textbf{brush}(\xi) \cup \mathbf{O}$.

### 5.3 Constructing the scaffold

Given a trace $\rho$ and a set of principal nodes $\mathbf{X}$, we construct the scaffold $\mathbf{D}(\rho, \mathbf{X})$ as follows. First, we walk downstream from each of the principal nodes depth-first until every path either terminates or reaches the application of a stochastic PSP for which we are certain we can compute a logdensity. As we go along, we mark nodes as $\mathbf{D}$ or $\mathbf{A}$ as appropriate. However, some nodes that we marked during this process may actually be in the brush. Therefore our second step is to recursively "disable" all requests that may change value or that may no longer exist, and to mark as brush every node that is



(a) (b)

Figure 5: **Scaffold construction.** The principal node is shown in red, resampling nodes are shown in gold, absorbing nodes are shown in blue, and the brush is shown in green. (a) To construct a scaffold, Venture first walks downstream from the (red) principal node, identifying all the nodes whose values could possibly change, stopping at nodes that are guaranteed to be able to absorb the change (albeit perhaps with a probability density of 0). (b) Venture then identifies the brush — those nodes that may no longer exist depending on the values chosen for the nodes being resampled — using a separate recursion. Once the brush has been identified, the definite regeneration graph (gold) and the border (blue) are straightforward to identify. See the main text for additional details.

only requested by these disabled requests. We then remove from **D** and **A** every node in the brush. Figure 5 gives an overview of this process, while Figure 6 illustrates the need for removing the brush of the source trace ρ before making a proposal.

### 5.4 Pseudocode for constructing scaffolds

The overall construction procedure for scaffolds involves several passes, all linear in the size of the scaffold:

CONSTRUCTSCAFFOLD(*trace*, *principalNodes*, *kernelInfo*)
1 *cDRG*, *cAbsorbing*, *cAAA* = FINDCANDIDATESCAFFOLD(*trace*, *principalNodes*)
2 *brush* = FINDBRUSH(*trace*, *cDRG*, *cAbsorbing*, *cAAA*)
3 *drg*, *absorbing*, *aaa* = RemoveBrush(*cDRG*, *cAbsorbing*, *cAAA*, *brush*)
4 *border* = FINDBORDER(*drg*, *absorbing*, *aaa*)
5 *regenCounts* = COMPUTEREGENCOUNTS(*trace*, *drg*, *absorbing*, *aaa*, *border*, *brush*)
6 *lkernels* = loadKernels(*trace*, *drg*, *kernelInfo*)
7 **return** Scaffold(*drg*, *absorbing*, *aaa*, *border*, *regenCounts*, *lkernels*)

Finding a candidate scaffold given a set of principal nodes involves walking downstream from each principal node until it is possible to absorb. To find the brush, Venture must count the number of times a given requested family is reachable from the DRG. If this equals the number of times

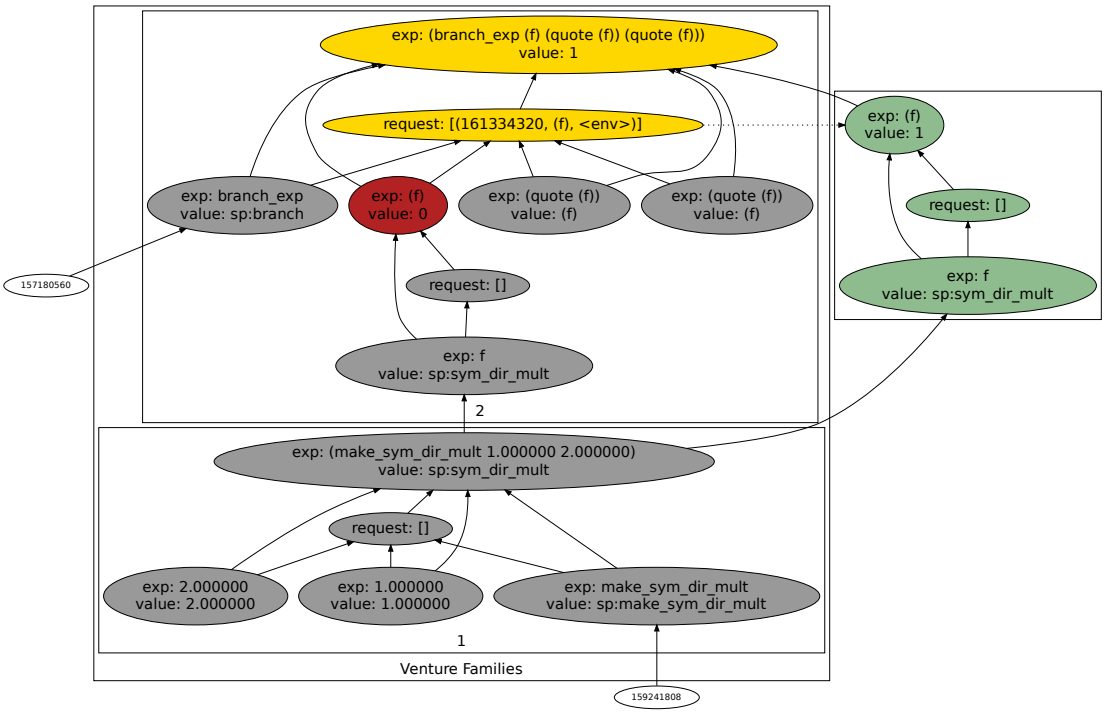



Figure 6: **The need for detaching the brush before making a proposal.** This trace includes the invocation of `make_symmetric_dirichlet_discrete`, an SP that produces SPs whose applications are exchangeably coupled. In this trace, the result of one application determines whether or not another application is made. Thus the second application is in the brush of the scaffold generated by the first application. If we do not detach the brush first, we may propose to the first application conditioned on an application that will not exist in the new trace.

the family is requested, then all references to the family originate in the DRG, and it is therefore possible for the family's existence to depend on the values of the DRG nodes.

FINDBRUSH$(trace, cDRG, cAbsorbing, cAAA)$

1 $disableCounts = \text{map}(0)$
2 $disabledRequestNodes = \text{set}()$
3 $brush = \text{set}()$
4 **for** *node* **in** *cDRG*
5 **if** $node.isRequestNode$?
6 DISABLEREQUESTS$(trace, node, disableCounts, disabledRequestNodes, brush)$
7 **return** *brush*

```
DISABLEREQUESTS(trace, node, disableCounts, disabledRequestNodes, brush)
1  if node in disabledRequests
2      return
3  for esrParent in trace.getESRParents(node.outputNode)
4      disableCounts[esrParent] = disableCounts[esrParent] + 1
5      if disableCounts[esrParent] == trace.getNumberOfRequests(esrParent)
6          DISABLEFAMILY(trace, esrParent, disableCounts, disabledRequestNodes, brush)
```

```
DISABLEFAMILY(trace, node, disableCounts, disabledRequestNodes, brush)
1  brush.insert(node)
2  if node.isOutputNode?
3      brush.insert(node.requestNode)
4      DISABLEREQUESTS(trace, node.requestNode, disableCounts, disabledRequestNodes, brush)
5      DISABLEFAMILY(trace, node.operatorNode, disableCounts, disabledRequestNodes, brush)
6      for operatorNode in node.operatorNodes
7          DISABLEFAMILY(trace, operatorNode, disableCounts, disabledRequestNodes, brush)
```

The border consists of nodes where resampling stops. This includes absorbing nodes, AAA nodes, and also resampling nodes with no children, e.g. from top-level `PREDICT` directives:

```
FINDBORDER(trace, drg, absorbing, aaa)
1  border = union(absorbing, aaa)
2  for node in drg
3      if not hasChildInAorD(trace, node, drg, absorbing)
4          border.insert(node)
5  return border
```

For stochastic regeneration to proceed correctly, the nodes in the scaffold must be annotated with counts that allow determination of when a given node is no longer referenced and can therefore be removed.

```
COMPUTEREGENCOUNTS(trace, drg, absorbing, aaa, border, brush)
 1  regenCounts = map(0)
 2  for node in drg
 3      if node in aaa
 4          regenCounts[node] = 1 // will be added to shortly
 5      elseif node in border
 6          regenCounts[node] = trace.getChildren(node).size + 1
 7      else
 8          regenCounts[node] = trace.getChildren(node).size
 9  // Now determine the number of times each reference to an AAA node will be regenerated
10  for node in union(drg, absorbing)
11      for parent in trace.getParents(node)
12          MAYBEINCREMENTAAAREGENCOUNT(parent)
13  for node in brush
14      if node.isLookupNode?
15          MAYBEINCREMENTAAAREGENCOUNT(node.sourceNode)
16      elseif node.isOutputNode?
17          MAYBEINCREMENTAAAREGENCOUNT(trace.getESRParents(node)[0])
18  return regenCounts

MAYBEINCREMENTAAAREGENCOUNT(trace, node, regenCounts)
1  value = trace.getValue(node)
2  if value.isSPRef? and not value.makerNode == node and aaa.contains?(value.makerNode)
3      regenCounts[value.makerNode] = regenCounts[value.makerNode] + 1
```

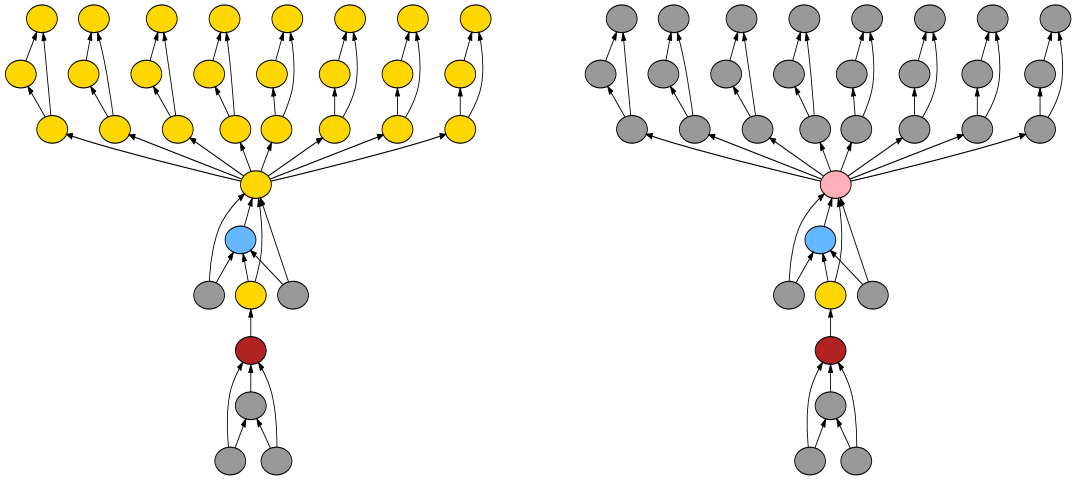
#### 5.4.1 ABSORBING AT APPLICATIONS

As we discussed earlier, there are cases in which we do not need to visit all the absorbing nodes explicitly to account for their logdensities because the SP can keep track of its outputs and report the joint logdensity of all of its applications. If this is the case, we say that the application of the SP is absorbing at applications (AAA), and our preliminary walk to the find the scaffold can stop upon reaching an AAA node. Figure 7 gives a graphical illustration. Note that the AAA node itself is in **D**–not **A**–since its value may change.

Handling AAA introduces many subtle bookkeeping challenges, and the code for computing the regenCounts for AAA nodes will seem mysterious until inspecting the special cases for AAA nodes in regen and extract in the next section.

### 5.5 Breaking down a global inference problem into collections of local inference problems

If the principal nodes are chosen to be all the random choices in the current trace, then the border will correspond to the random choices constrained by the `OBSERVE`s, and the terminal resampling nodes will be any `ASSUME`s that are not subsequently referred to, as well as all the `PREDICT`s. The definite regeneration graph will only contain random choices that are guaranteed to exist in every execution of the probabilistic program. Conditionally simulating the entire program can be reduced to conditionally simulating a completion of a **torus** given the absorbing nodes **A**.



(a) (b)

Figure 7: **Scaffolds with and without the "absorbing-at-applications" optimization.** Repeated applications of stochastic procedures are common in probabilistic programs for machine learning and statistics. (a) shows a PET fragment typical of this repetition. The principal node is in red, resampling nodes are in yellow, and the constructor SP is in blue. Note the size of the scaffold, which grows linearly with the number of applications. (b) shows the same PET fragment but where the "constructor" SP (whose resulting SP is repeatedly applied) can absorb at applications. The SP is now pink, and it because it absorbs at applications (implementing the weight calculations needed for inference via information in its auxiliary storage) the scaffold size no longer scales linearly with the number of applications.

This equivalence is an important feature of Venture's design. Any inference strategy that works for entire programs can be run on a program fragment, conditioned on the remainder, and vice versa. Venture can break down the problem of sampling from the conditioned distribution over the entire PET, which might be extremely high dimensional, into collections of overlapping lower-dimensional inference subproblems. Any Venture strategy applicable to the overall problem becomes applicable to each subproblem. Current research is exploring the use of Venture programs to construct custom proposals, where each `PREDICT` directive is mapped to a principal node or a resampling node in the border, and each `OBSERVE` statement is matched to an absorbing node.

## 5.6 Local kernels and scaffolds

Later sections describe techniques for building up transition operators that efficiently modify contents of scaffolds, sampling its definite regeneration graph and brush from a distribution that leaves its conditional invariant. These transition operators are built out of stochastic proposals for individual random choices that we call *local kernels*. These kernels must be able to sample new values for individual random choices as well as calculate the ratio of forward and reverse transition probabilities necessary for using the sampled value as a proposal for a Metropolis-Hastings scheme. Venture supports local kernels of two types: simulation and delta kernels.

### 5.6.1 Local simulation kernels: resimulation & bottom-up proposals

A local simulation kernel for a random choice $x$ cannot look at previous values of $x$ when making a proposal. The simplest simulation kernel is the one that samples $x$ according to the `simulate()` provided by its associated stochastic procedure. We will sometimes refer to this as a "resimulation" kernel.

1. Let $\pi_{\rightarrow}$ denote the order in which the local kernels are called during sample. Then **sample** returns a new value $x$ along with the following weight:
$$\frac{P_{\pi_{\rightarrow}}(x)}{\mathbf{K}_{\pi_{\rightarrow}}(x)}$$
2. Let $\pi_{\leftarrow}$ denote the reverse of the order in which the local kernels are called during unsample. Then **unsample** returns the following weight:
$$\frac{\mathbf{K}_{\pi_{\leftarrow}}(x)}{P_{\pi_{\leftarrow}}(x)}$$

For the resimulation kernel, the weights from this local kernel are identically 1. Simulation kernels can also be conditioned on anything in **torus** without breaking asymptotic convergence. This is because the values of random choices in the torus are all available at proposal time and also unaffected by the transition. Simulation kernels can thus be used to make bottom-up proposals, using other values in the source trace as input. This provides one mechanism for augmenting the top-down processing that is typical in probabilistic programming with custom bottom-up information.

### 5.6.2 Local delta kernels and reuse of random choices

Venture also supports delta kernels. When these kernels transition the trace from $x' \rightarrow x$, they have access to the previous state $x'$ in addition to the contents of the torus. A Gaussian drift kernel, where $x \sim \mathcal{N}(x', \sigma^2)$, is a typical example. They satisfy the following contract:

1. **sample** returns a state $x'$ with weight:

$$\frac{P(x)\mathbf{K}(x \to x')}{P(x')\mathbf{K}(x' \to x)}$$

2. **unsample** returns 0.

These kernels cannot be applied to stochastic procedures that exhibit exchangeable coupling between applications, nor can they be used in particle methods. Since they receive the source state as an input, they also provide a mechanism for re-using the old value of a random choice if its parent choices have not changed. Delta kernels can thus reduce unnecessary or undesirable resampling.

## 6. Stochastic Regeneration Algorithms for Scaffolds

We now show how to coherently modify a trace, given a partition and a valid scaffold, using an algorithm we introduce called *stochastic regeneration*. Variations on stochastic regeneration, some of which can be expressed as simple parameterizations, can be used to implement a wide range of stochastic inference strategies.

The simplest use of stochastic regeneration is to implement a Metropolis-Hastings transition operator for PETs as follows. First, the border of the scaffold is "detached" from the trace in an arbitrary order, and the random choices within the trace fragment that has been detached are stored in an "extract". We will sometimes abbreviate this process as **detach**. Next, the border of the scaffold is "regenerated and attached" in reverse order, yielding a new trace. We will sometimes abbreviate this process as **regen**. This new trace can be accepted or rejected. If it is accepted, the algorithm has finished. If it is rejected, it is detached again, and the original extract is fed to the regeneration algorithm. This restores the trace to its original state.

Assuming simulation and density evaluation of the constituent PSPs is constant, the runtime of this process scales with the size of the scaffold and the brush, not the size of the entire PET. Nodes that cannot influence or be influenced by the transition are never visited. PET nodes are regenerated in the opposite order that they were visited during detach; see Figure 8 for a trace that illustrates the need for this symmetry.

### 6.1 Detaching along a scaffold with DETACH-AND-EXTRACT

Detach-and-Extract takes a scaffold and a trace, and converts the trace into a torus and a full omegaDB. At that point, the torus is ready for a new proposal, while the omegaDB then contains all the information needed to restore the original trace. The first step is to disconnect (detach) the trace fragment from the border of the scaffold. The second step is to extract all the random choices from the trace fragment and store them in the omegaDB.

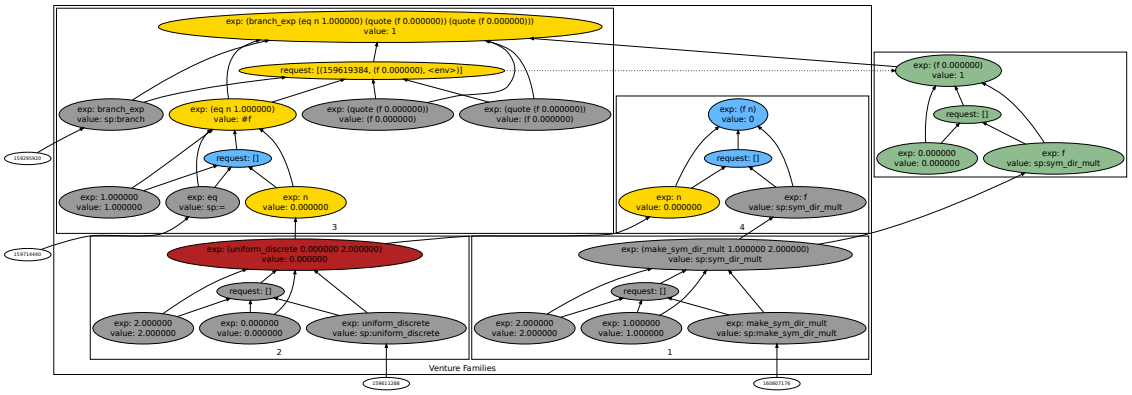



Figure 8: **Due to partially exchangeable SPs, detach and regen must visit nodes in opposite orders.** A procedure with partial-exchangeable coupling is applied in both the absorbing border and the brush. The probability of the absorbing nodes depends on the order in which regeneration takes place. As Venture is detaching, it needs to compute the probability of the absorbing nodes that it would have calculated if it had proposed this trace by regenerating along the scaffold. In particular, if it would have visited the absorbing application first during **regen**, then it must visit the absorbing application last during **detach**. Venture solves this problem by having **detach** always detach all nodes in the opposite order from how **regen** would have generated them.

DETACHANDEXTRACT($trace, border, scaffold$)

```
1  weight = 0
2  db = DB()
3  for node in reversed(border)
4      if scaffold.isAbsorbing(node)
5          weight = weight + DETACH(trace, node, scaffold, db)
6      else
7          if node.isObservation
8              weight = weight + UNCONSTRAIN(trace, node)
9          weight = weight + EXTRACT(trace, node, scaffold, db)
10 return weight, db
```

To detach a node from the border, Venture must unincorpoate the value from the PSP that was previously responsible for generating it, adjust the weights, and then extract its parents:

DETACH($trace, node, scaffold, db$)

```
1  psp, args = trace.getPSP(node), trace.getArgs(node)
2  groundValue = trace.getGroundValue(node)
3  psp.unincorporate(groundValue, args)
4  weight = psp.logDensity(groundValue, args)
5  weight = EXTRACTPARENTS(trace, node, scaffold, db)
6  return weight
```

To extract the parents of a node, Venture loops over the parents in the correct order:

```
EXTRACTPARENTS(trace, node, scaffold, db)
1  weight = EXTRACTESRPARENTS(trace, node, scaffold, db)
2  for parent in reversed(trace.getDefiniteParents(node))
3      weight = weight + EXTRACT(trace, parent, scaffold, db)
4  return weight
```

```
EXTRACTESRPARENTS(trace, node, scaffold, db)
1  weight = 0
2  for parent in reversed(trace.getESRParents(node))
3      weight = weight + EXTRACT(trace, parent, scaffold, db)
4  return weight
```

The key idea in this process is that when a node is no longer referenced by any others — that is, its `regenCount` is 0 — it can be unevaluated. Also, if a node is a request node, before unevaluating it all requests it generates should be unevaluated, which may trigger the trace fragments corresponding to the request to be unevaluated if they are no longer referenced.

```
EXTRACT(trace, node, scaffold, db)
 1  weight = 0
 2  value = trace.getValue(node)
 3  if value.isSPRef? and not value.makerNode == node and scaffold.isAAA(value.makerNode)
 4      weight = weight + EXTRACT(trace, value.makerNode, scaffold, db)
 5  if scaffold.isResampling(node)
 6      scaffold.decrementRegenCount(node)
 7      if scaffold.getRegenCount(node) == 0
 8          if node.isLookup?
 9              trace.clearValue(node)
10          else
11              if node.isRequestNode?
12                  weight = weight + UNEVALREQUESTS(trace, node, scaffold, db)
13              weight = weight + UNAPPLYPSP(trace, node, scaffold, db)
14          weight = weight + EXTRACTPARENTS(trace, node, scaffold, db)
15  return weight
```

### 6.2 Regenerating a new trace along a scaffold with REGENERATE-AND-ATTACH

Regenerate-and-Attach, often abbreviated as regen, takes a scaffold, a torus, and an omegaDB, as well as a control parameter that indicates whether or not it is restoring (i.e. replaying random choices from the omegaDB)[13]. The key idea in regen is that it ensures the PET is constructed via sequence of self-supporting PETs, that is PETs where all parents of a node — including all choices on which a given random choice directly depends — have been regenerated before the node itself is.

13. It also takes a map from nodes to the gradient of their log densities, for use by variational kernels; we will discuss this later in the advanced inference section.

```
REGENERATEANDATTACH(trace, border, scaffold, restore?, db)
1  weight = 0
2  for node in border
3      if scaffold.isAbsorbing?(node)
4          weight = weight + ATTACH(trace, node, scaffold, restore?, db)
5      else
6          weight = weight + REGENERATE(trace, node, scaffold, restore?, db)
7          if node.isObservation?
8              weight = weight + CONSTRAIN(trace, node, node.observedValue)
9  return weight
```

To attach a node from the border, one first regenerates its parents, then "attaches" the trace fragment to the current trace, making it responsible for generating the current node in the border. This involves updating the weight and also incorporating the value into the stochastic procedure that is newly responsible for generating it.

```
ATTACH(trace, node, scaffold, restore?, db)
1  psp, args = trace.getPSP(node), trace.getArgs(node)
2  groundValue = trace.getGroundValue(node)
3  weight = REGENERATEPARENTS(trace, node, scaffold, restore?, db)
4  weight = weight + psp.logDensity(groundValue, args)
5  psp.incorporate(groundValue, args)
6  return weight
```

```
REGENERATEPARENTS(trace, node, scaffold, restore?, db)
1  weight = 0
2  for parent in trace.getDefiniteParents(node)
3      weight = weight + REGENERATE(trace, parent, scaffold, restore?, db)
4  weight = weight + REGENERATEESRPARENTS(trace, node, scaffold, restore?, db)
5  return weight
```

```
REGENERATEESRPARENTS(trace, node, scaffold, restore?, db)
1  weight = 0
2  for parent in trace.getESRParents(node)
3      weight = weight + REGENERATE(trace, parent, scaffold, restore?, db)
4  return weight
```

Regenerating a node involves updating its regeneration counts, to ensure that after regeneration, detach can be correctly called on the new trace:

```
REGENERATE(trace, node, scaffold, restore?, db)
 1  weight = 0
 2  if scaffold.isResampling?(node)
 3      if scaffold.getRegenCount(node) == 0
 4          weight = weight + REGENERATEPARENTS(trace, node, scaffold, restore?, db)
 5          if node.isLookup?
 6              trace.setValue(node, trace.getValue(sourceNode))
 7          else
 8              weight = weight + APPLYPSP(trace, node, scaffold, restore?, db)
 9              if node.isRequestNode?
10                  weight = weight + EVALREQUESTS(trace, node, scaffold, restore?, db)
11      scaffold.incrementRegenCount(node)
12  value = trace.getValue(node)
13  if value.isSPRef? and not value.makerNode == node and scaffold.isAAA?(value.makerNode)
14      weight = weight + REGENERATE(trace, value.makerNode, scaffold, restore?, db)
15  return weight
```

### 6.3 Building invariant transition operators using stochastic regeneration

Stochastic regeneration can be used to implement a variety of inference schemes over probabilistic execution traces. In addition to modifying the trace in an undoable fashion as described above, stochastic regeneration provides numerical weights that are needed for inference schemes such as Metropolis-Hastings.

#### 6.3.1 WEIGHTS ASSUMING SIMULATION KERNELS

Assume that the local kernels used for each stochastic procedure application is a simulation kernel — that is, the value it proposes is independent of the value of the random choice in $\rho$. Let $\pi$ be the (after the fact) **regen** order for $\xi$, which is the order in which the local kernels are called during **regen**, and which is also by construction the reverse of the order in which the local kernels would be called during **detach**. Then

1. **regen** returns
$$\frac{P_\pi(\mathbf{R}(\xi), \mathbf{A})}{\mathbf{K}_{\mathbf{regen}}(\mathbf{R}(\xi))}$$
2. **detach** returns:
$$\frac{\mathbf{K}_{\mathbf{regen}}(\mathbf{R}(\xi))}{P_\pi(\mathbf{R}(\xi), \mathbf{A})}$$

where $\mathbf{K}_{\mathbf{regen}}$ is the kernel that proposes to $\mathbf{R}(\cdot)$ by calling local kernels in the **regen** order, and **detach** computes the reverse probability correctly since each local kernel is unapplied in the reverse of the order in which it would have been applied during **regen**. These weights can be used to implement Metropolis-Hastings transitions between $\rho$ and $\xi$.

Note that if all local kernels are resimulation kernels, only the "likelihood ratio" induced by the absorbing nodes is involved in the Metropolis-Hastings acceptance ratio.

#### 6.3.2 WEIGHTS ASSUMING DELTA KERNELS

Now assume that at least one of the local kernels is a delta kernel — that is, the value it proposes depends on the value of the random choice being modified from $\rho$. In this case, **regen** and **detach** do not return useful quantities individually, but their product (**detachAndRegen**) is the same as in the case of simulation kernels, and is suitable for implementing Metropolis-Hastings transitions:

$$\frac{P_{\pi_\xi}(\mathbf{R}(\xi),\mathbf{A})\mathbf{K}_{\mathbf{regen}}(\mathbf{R}(\xi)\rightarrow\mathbf{R}(\rho))}{P_{\pi_\rho}(\mathbf{R}(\rho),\mathbf{A})\mathbf{K}_{\mathbf{regen}}(\mathbf{R}(\rho)\rightarrow\mathbf{R}(\xi))} \tag{3}$$

### 6.4 Context-independent versus context-specific inference schemes

The contents of the scaffold and brush do not depend on the values of the principal nodes for a given transition. For example, every random choice whose existence might be affected by a value chosen for a principal node (or a descendant of a principal node that must be resampled) must be included in the brush, and regenerated during a transition, even if the particular value(s) sampled for the principal node were consistent with the old execution trace. It could be asymptotically more efficient in some circumstances to exploit context-specific independencies, both for the values of random choices and for their existence.

Unfortunately, context-specificity comes at the cost of significant additional complexity. Other Turing-complete probabilistic programming systems — for example, the original implementation in (Mansinghka, 2009), as well as MIT-Church, and an earlier version of Venture — were context sensitive. Monte was context-independent but significantly more limited than Venture. Of these, we believe only the earlier version of Venture achieved the correct order of growth for all scale parameters in Latent Dirichlet Allocation, and none supported reprogrammable inference.

It seems likely that scaffold construction and stochastic regeneration can be interleaved to retain the flexibility of our current architecture while achieving the efficiency of an update scheme sensitive to context-specific independencies. We leave this for future work.

## 7. General-purpose Inference Strategies via Stochastic Regeneration

Venture's inference programming language provides a number of primitive inference mechanisms . The effect of each strategy is the evolution of a PET according to a transition operator that leaves the posterior distribution on PETs invariant, with modifications constrained to lie within the scaffold. Each of these strategies can be applied to any scaffold, including the global scaffold. Each is implemented using stochastic regeneration, with **regen** and **detach** performing mutation of the trace in place and returning weights that can be used for any rejection steps necessary to maintain invariance.

Before describing specific inference strategies, we establish some preliminary results that simplify reasoning about transition operators on PETs.

### 7.1 Factorization of the acceptance ratio

Let $\rho$ be a PET representing an execution and $\pi$ be an ordering of its nodes. Suppose we visit each node in this order, for each SP application (i.e. random choice) computing the probability density of its output given its input, and then passing that input-output pair to the incorporate method asso-

ciated with the SP, to handle exchangeable coupling. Define $P_\pi(\rho)$ to be the total probability density of the sequence of random choices in $\rho$, i.e. the product of the probability densities calculated above. Recall that every SP satisfies the property that the probability of any sequence of input-output pairs is invariant under permutation. Thus we have that $P_\pi(\rho) = P(\rho)$.

For an arbitrary subset $Z$ of the nodes in $\rho$, we do not necessarily have $P_{\pi_1}(Z) = P_{\pi_2}(Z)$. The equivalence only holds over the probability of all of $\rho$, not over arbitrary fragments of it. For example, consider the following program:

```
(assume f (make-ccoin 1 1))
(predict (f))
(predict (f))
```

Now let $\rho$ be an execution with (say) both predicts **true**. Then the probability of the first predict depends on whether or not it is weighted before or after the second predict, but the total probability will be the same no matter which order we proceed in.

However, the probabilities will be equal if $Z$ is a prefix in both orderings, because we can view $Z$ as the complete trace associated with some modified probabilistic program $\rho'$.

**Proposition 7.1.** *Let* $\rho, \xi \in \Xi[\mathbf{torus}, \mathbf{D}]$. *Let* $\pi_\rho$ *be a permutation on* $\rho$ *such that* $\pi_\rho(\mathbf{P} \cup \mathbf{I}) < \pi_\rho(\mathbf{R}(\rho) \cup \mathbf{A})$*, and likewise for* $\pi_\xi$*. Then*

$$\frac{P(\rho)}{P(\xi)} = \frac{P_{\pi_\rho}(\mathbf{R}(\rho), \mathbf{A})}{P_{\pi_\xi}(\mathbf{R}(\xi), \mathbf{A})} \tag{4}$$

*Proof.* We can factor $\rho$ and $\xi$ into disjoint subsets:

$$\rho = (\mathbf{P} \cup \mathbf{I}) \dot{\cup} (\mathbf{R}(\rho) \cup \mathbf{A}) \tag{5}$$

$$\xi = (\mathbf{P} \cup \mathbf{I}) \dot{\cup} (\mathbf{R}(\xi) \cup \mathbf{A}) \tag{6}$$

We have $P_{\pi_\rho}(\mathbf{P} \cup \mathbf{I}) = P_{\pi_\xi}(\mathbf{P} \cup \mathbf{I})$ since the set is a prefix in both orderings. The result follows immediately. □

### 7.2 Auxiliary variables for state-dependent stochastic selection of kernels

We would like to be able to use Metropolis-Hastings to construct valid transition operators that leave a conditional distribution $P()$ on PETs invariant. In some cases, we also would like these transition operators to be individually ergodic. For any random choice guaranteed to be in all executions — that is, the SP application is in the DRG of the global scaffold — this is straightforward. Standard cycle or mixture hybrid kernels (see e.g. (Bonawitz, 2008; Andrieu et al., 2003; Mansinghka, 2009)) can be used to sequence Metropolis-Hastings proposals on individual variables. Alternately, if we are only interested in inference strategies that choose random choices at uniformly at random, we can make a compound Metropolis-Hastings proposal that first selects a random variable to change (using it to seed a scaffold) and then makes a within that scaffold. Because the choice of random choices involves only a simple state dependence, and it is (nearly) trivial to integrate out the number of paths from some trace $\rho$ to another trace $\xi$ under this sequence of choices, it is straightforward to accommodate it in a single Metropolis-Hastings step.

Venture supports a broader range of transition operators: Metropolis-Hastings proposals where the proposal kernel is randomly chosen from a state-dependent distribution. Let $I$ be a (possibly

infinite) set, $f$ a stochastic function from traces to $I$ with induced density $P_f()$. For every $i \in I$, let $Q_i$ be a proposal kernel on PETs, and let $\rho$ be the current PET. Then the following cycle kernel on the extended state space $\Omega \times I$ preserves $\pi$:

1. Sample $i|\rho \sim f(\rho)$.
2. (a) Propose $\xi \sim Q_i(\rho \to \xi)$.
   (b) Accept if
$$U \sim \textbf{Uniform}(0,1) < \frac{P_f(f(\xi) = i)}{P_f(f(\rho) = i)} \cdot \frac{P(\xi) Q_i(\xi \to \rho)}{P(\rho) Q_i(\rho \to \xi)}$$

This scheme treats the choice of kernel as a transient auxiliary variable $I$ that is created solely for the duration of the ordinary Metropolis-Hastings proposal that depends on it. Invariance is ensured by accounting for the effect of $Q_i$ on $P_f(f(\xi) = i)$. This construction can be used to straightforwardly justify sophisticated custom proposal strategies such as Algorithm 8 from (Neal, 1998) without need for any approximation.

Stochastic index selection makes it straightforward to compositionally analyze richer transition operators on PETs. For example, one can select scaffolds by choosing a single principal node uniformly at random, as in typical Church implementations. Alternately, one could estimate the entropy of the conditional distributions for each random choice given its containing scaffold, and choose scaffolds with probability proportional to this estimate. One could also select principal nodes by starting with some query variable of interest — e.g. a random choice representing an important prediction or action variable — and walk the PET, favoring random choices that are "close" to the query variable.

We anticipate that the evolution of the Venture inference programming language will depend on the expansion and refinement of a library of techniques for converting proposals into invariant transition operators.

### 7.3 Metropolis-Hastings via Stochastic Regeneration

Here we describe the basic Metropolis-Hastings algorithm for incremental inference over a PET. With all of the tools we have developed so far, a generic version of this transition operator is straightforward to specify:

1. Sample a set of principal nodes **principals** $\sim f(\rho)$, and record the probability $P(f(\rho) =$ **principals**$)$ of sampling it.
2. Construct **D** with **principals** as the set of principal nodes.
3. Set $\alpha =$ **detachAndRegen**$(\mathbf{D})$
4. Calculate $P(f(\xi) =$ **principals**$)$.
5. Accept if
$$U < \frac{P(f(\xi) = \textbf{principals})}{P(f(\rho) = \textbf{principals})} \cdot \alpha$$

Correctness follows immediately from 7.1 and (3), and applies to a broad class of customized kernels. Because stochastic regeneration does not visit portions of the PET that are conditionally independent of the random choices affected by the transition, it is more scalable than approaches based on transformational compilation. In particular, for typical machine learning problems on datasets of size $N$, the number of random choices scales linearly with $N$ but the connections in the PET are sparse. Thus an inference sweep of $N$ single-site Metropolis-Hastings transitions — enough to consider each random choice roughly once — requires $O(N)$ time, as opposed to $O(N^2)$ time for transformational compilers.

### 7.4 Approximating optimal proposals via Stochastic Variational Inference

We can use the same auxiliary variable technique to lift any local proposal into one that preserves global stationarity. Here we show how to learn a local variational approximation to the posterior distribution on $\mathbf{R}(\xi)$ given $\mathbf{A}$ using **regen** and **detach**, and then wrap it in Metropolis-Hastings as discussed above.

#### 7.4.1 Posing the optimization problem

Let $\rho$ be a trace and $\mathbf{D}$ a definite regeneration graph inducing **torus**. Let $Q$ be some family of distributions on $\Xi[\textbf{torus}, \mathbf{D}]$ where every $Q_\theta \in Q$ is defined by its parameters $\theta \in \Theta$, which control the local kernels along $\mathbf{D}$, and where each $Q_\theta$ reverts to the resimulation kernel on the brush. Our goal is to I-Project $P(\mathbf{R})$ onto the space $Q$, which involves solving the following optimization problem:

$$\begin{aligned} &\text{minimize} \quad f(\theta) = \mathbb{E}_{\xi \sim Q_\theta}\left[\log \frac{Q_\theta(\mathbf{R}(\xi))}{P(\mathbf{R}(\xi), \mathbf{A})}\right] \\ &\text{subject to} \quad \theta \in \Theta \end{aligned}$$

#### 7.4.2 Stochastic gradient descent

One generic approach is stochastic gradient descent. Under assumptions of differentiability that may not hold in arbitrary programs, the gradient of the objective function has the following nice form (as originally shown in (Wingate and Weber, 2013) for entire traces):

$$\begin{aligned} \nabla_\theta f(\theta) &= \nabla_\theta \mathbb{E}_{\xi \sim Q_\theta}\left[\log \frac{Q_\theta(\mathbf{R}(\xi))}{P(\mathbf{R}(\xi), \mathbf{A})}\right] && (7) \\ &= \nabla_\theta \int_{\xi \in \Xi} Q_\theta(\mathbf{R}(\xi)) \log \frac{Q_\theta(\mathbf{R}(\xi))}{P(\mathbf{R}(\xi), \mathbf{A})} d\xi && (8) \\ &= \int_{\xi \in \Xi} \nabla_\theta \left\{ Q_\theta(\mathbf{R}(\xi)) \log \frac{Q_\theta(\mathbf{R}(\xi))}{P(\mathbf{R}(\xi), \mathbf{A})} \right\} d\xi && (9) \\ &= \int_{\xi \in \Xi} \nabla_\theta Q_\theta(\mathbf{R}(\xi)) \log \frac{Q_\theta(\mathbf{R}(\xi))}{P(\mathbf{R}(\xi), \mathbf{A})} d\xi + \int_{\xi \in \Xi} Q_\theta(\mathbf{R}(\xi)) \left( \nabla_\theta \log \frac{Q_\theta(\mathbf{R}(\xi))}{P(\mathbf{R}(\xi), \mathbf{A})} \right) d\xi && (10) \\ & && (11) \end{aligned}$$

$$= \int_{\xi \in \Xi} \nabla_\theta Q_\theta(\mathbf{R}(\xi)) \log \frac{Q_\theta(\mathbf{R}(\xi))}{P(\mathbf{R}(\xi), \mathbf{A})} d\xi \tag{12}$$

$$= \int_{\xi \in \Xi} Q_\theta(\mathbf{R}(\xi)) \nabla_\theta \log Q_\theta(\mathbf{R}(\xi)) \log \frac{Q_\theta(\mathbf{R}(\xi))}{P(\mathbf{R}(\xi), \mathbf{A})} d\mathbf{R}(\xi) \tag{13}$$

$$= \mathbb{E}_{\xi \sim Q_\theta} \left[ \nabla_\theta \log Q_\theta(\mathbf{R}(\xi)) \log \frac{Q_\theta(\mathbf{R}(\xi))}{P(\mathbf{R}(\xi), \mathbf{A})} \right] \tag{14}$$

where we used that

$$\int_{\xi \in \Xi} Q_\theta(\mathbf{R}(\xi)) \left( \nabla_\theta \log \frac{Q_\theta(\mathbf{R}(\xi))}{P(\mathbf{R}(\xi), \mathbf{A})} \right) d\xi = \int_{\xi \in \Xi} Q_\theta(\mathbf{R}(\xi)) \left( \nabla_\theta \log Q_\theta(\mathbf{R}(\xi)) \right) d\xi \tag{15}$$

$$= \int_{\xi \in \Xi} \nabla_\theta Q_\theta(\mathbf{R}(\xi)) d\xi \tag{16}$$

$$= \nabla_\theta \int_{\xi \in \Xi} Q_\theta(\mathbf{R}(\xi)) d\xi \tag{17}$$

$$= 0 \tag{18}$$

Thus for a given setting of $\theta^{(t)}$, we can approximate the gradient $\nabla_\theta f(\theta)|_{\theta^{(t)}}$ using Monte Carlo, by forward-sampling trajectories from $Q_{\theta^{(t)}}$. This is the scheme proposed in (Wingate and Weber, 2013):

1. Sample a trajectory from $Q_{\theta^{(t)}}$.
2. Calculate the noisy gradient estimate $\delta^{(t)}$.
3. Take a noisy gradient descent step $\theta^{(t+1)} := \theta^{(t)} - \alpha^{(t)} \delta^{(t)}$.

### 7.4.3 As a proposal for Metropolis-Hastings

We can solve this optimization problem as above to construct a proposal distribution $Q_{\theta^\star}$, and then sample $\xi$ from it and accept or reject it as a single M-H proposal. Note that while solving the optimization problem, we did not inspect any nodes that were in $\rho$ but that might not be in $\xi$; therefore we can use the same $Q_{\theta^\star}$ to compute the probability of the reverse transition.

### 7.4.4 Using regen and detach

The stochastic regeneration recursions can be easily extended to carry additional metadata that enables them to evaluate local gradients for each random choice, and store this information as a local kernel that modifies additional state capturing variational parameters. Once this is done, it turns out that both the stochastic gradient optimization of the variational approximation and the use of it as a proposal can be expressed using stochastic regeneration:

1. Construct $\mathbf{D}$.
2. Detach $\rho$.
3. For every application node in $\mathbf{D}$ whose operator cannot change and that can compute gradients, initialize variational parameters to the current arguments.

4. Loop until satisfied with the quality of the variational approximation:

   (a) Attach variational kernels to $\mathbf{D}$ to define $Q_{\theta^{(t)}}$.

   (b) Call **regen** to sample $\xi \sim Q_{\theta^{(t)}}$, and to compute

   $$\log\left(\frac{P(\xi,\mathbf{A})}{Q_{\theta^{(t)}}(\mathbf{R}(\xi))}\right)$$

   as well as the local gradients $\{\nabla_\theta \log Q_{\theta^{(t)}}(\mathbf{R}(\xi))\}$.

   (c) Take a gradient step.

   (d) Detach.

5. **restore** $\rho$ to compute $\alpha_\rho := \frac{P(\mathbf{R}(\rho),\mathbf{A})}{Q_{\theta^\star}(\mathbf{R}(\rho))}$.

6. **detach**

7. **regen** to sample $\xi$ and compute $\alpha_\xi := \frac{P(\mathbf{R}(\xi),\mathbf{A})}{Q_{\theta^\star}(\mathbf{R}(\xi))}$.

8. Accept if

$$U \sim \textbf{Uniform}(0,1) < \frac{P(f(\xi)=i)}{P(f(\rho)=i)} \cdot \frac{\alpha_\xi}{\alpha_\rho}$$

## 8. Particle-based Inference: Enumerative Gibbs and Particle Markov chain Monte Carlo

Many inference strategies involve weighing multiple alternative states against one another as a way of approximating optimal Gibbs-type transition operators over portions of the hypothesis space. Gibbs sampling on a discrete variable in a Bayesian network can be implemented by enumerating all possible values for the variable, substituting each into the network, evaluating the joint probability of each network configuration, then renormalizing and sampling. Normalized importance sampling for approximating Gibbs involves proposing multiple values for a variable from its prior, weighting by its likelihood, and then normalizing over and sampling from the sampled set. Each of these techniques can be applied iteratively to larger subsets of variables. For example, discrete Gibbs can be iterated over a sequence of variables to generate a proposal from a single-particle particle filter, or extended via enumeration (and perhaps dynamic programming) to larger subsets of random variables. Normalized importance sampling can be also iterated to yield sequential importance sampling with resampling, and embedded within a Markov chain to yield particle Markov Chain Monte Carlo techniques.

It can be useful to think of these inference strategies as instantiations of the idea of weighted speculative execution technique within inference programming. An ensemble of executions is represented, evolved and stochastically weighted and selected via the absorbing nodes, although ultimately only one of these executions will be used.

We refer to each alternative state as a particle, and implement Gibbs sampling, particle Markov chain Monte Carlo, and other techniques using general machinery for particle-based inference methods. This allows us to develop common techniques for handling the dependent random choices in the brush — both in the proposal and in the analysis — that apply to all these inference strategies.

### 8.1 In-place mutation versus simultaneous particles in memory

Particle methods also provide new opportunities for time-space tradeoffs in inference. Although the semantics of particle methods involve multiple alternative traces, it may not always be desirable, or even possible, to represent alternate traces in memory simultaneously. Consider scaffolds whose definite regeneration graph or brush invokes a complex external simulator. If we cannot modify the code of this simulator, or it relies on an expensive external compute resource that cannot be multiplexed, it will be necessary to implement particle methods via mutation of a single PET in place. Otherwise, it is likely to be more efficient to represent all particles in memory simultaneously, sharing a common PET and scaffold as their source, and cloning the auxiliary states of stochastic procedures that are referred to from multiple particles. This choice also enables the parallel evolution of collections of particles via multiple independent threads, synchronizing whenever resampling occurs.

Venture supports both in-place mutation and simultaneous representation of particles. This is necessary to recover asymptotic scaling that is competitive with custom sequential Monte Carlo techniques. In this paper, we focus on the transition operators associated with particle methods, and give pseudocode for implementing them in terms of a generic API for simultaneous particles. The data structures needed to implement this API efficiently, and the algorithm analysis for efficient implementations, involve novel applications of techniques from functional programming and persistent data structures and are beyond the scope of this paper.

### 8.2 The Mix-MH operator for constructing particle-based transition operators

Although the auxiliary variable technique introduced previously is sufficient for lifting a local proposal into a global Metropolis-Hastings proposal, it will be helpful to introduce a variant of the technique that is closed under application — i.e. the kernel that results from applying the technique is a valid proposal kernel that can be parameterized, randomly selected, and used as a proposal in a subsequent application. This will give us significant additional flexibility in explaining and justifying our particle-based inference schemes.

#### 8.2.1 METROPOLIS-HASTINGS

Suppose we have some distribution $\pi$ of interest on $\Omega$.

Let $\mathbf{K}$ be a proposal kernel on $\Omega$. We can create a new kernel $\mathbf{MH}(\mathbf{K},\pi)$ based on it that satisfies detailed balance with respect to $\pi$ by the Metropolis-Hastings rule. Instead of representing it as a black-box kernel, we will keep around its constituent elements, so we have

$$\mathbf{MH}(\mathbf{K},\pi) = (\mathbf{K},\lambda) \tag{19}$$

where

$$\lambda(\rho \to \xi) = \pi(\xi)\mathbf{K}(\xi \to \rho)$$

and where applying $(\mathbf{K},\lambda)$ involves:

1. Sample $\xi$ from $\mathbf{K}(\rho,\xi)$.
2. Let $U \sim [0,1]$. Accept the new trace $\xi$ iff

$$U < \frac{\lambda(\rho \to \xi)}{\lambda(\xi \to \rho)}$$

It is straightforward to show that this transition operator preserves detailed balance with respect to $\pi$, and therefore leaves $\pi$ invariant. Ergodic convergence can be proved under fairly mild assumptions (Andrieu et al., 2003).

### 8.2.2 State-dependent Mixtures of Kernels

We would like to be able to construct compound kernels that are themselves stochastically chosen, and certify detailed balance of the compound kernel given easily verified conditions on the component kernel(s) and the stochastic kernel selection rule.

Let $P()$ be a density of interest, such as the conditioned density on resampling nodes given the absorbing nodes in a scaffold. Let $I$ be a (possibly infinite) set, and $f(\rho) = i$ be a stochastic function from traces to $I$, with $P_f(i)$ the density of $i$ induced by $f$. We will use $f(\rho)$ to select a kernel $\mathbf{Q}_i = (\mathbf{K}_i, \alpha_i)$ at random given the trace $\rho$ (and also evaluate $P_f(f(\xi) = i)$ for reversing the transition). Let $\mathbf{K}_i(\rho \to \xi)$ be a transition operator on traces that can be simulated from, and let $\alpha_i(\rho \to \xi)$ be a procedure that evaluates the factors out of which the Metropolis-Hastings acceptance ratio is constructed:

$$\alpha_i(\rho \to \xi) = P(\xi)\mathbf{K}_i(\xi \to \rho)$$

Consider the following kernel, parameterized by scalars $\lambda_i(\rho \to \xi)$ to be determined:

1. Sample $i \sim f(\rho)$. This will be used to choose a kernel $\mathbf{K}_i$.
2. Sample $\xi \sim \mathbf{K}_i(\rho \to \xi)$.
3. Let $U \sim [0,1]$. Accept $\xi$ iff
$$U < \frac{\lambda_i(\rho \to \xi)}{\lambda_i(\xi \to \rho)}$$

Our goal is to define $\lambda_i(\rho \to \xi)$ such that this kernel preserves detailed balance, analogously to the Metropolis-Hastings kernels described above. We assume that the stochastic function $f$ satisfies the following symmetry condition:

$$P_f(f(\rho) = i)\mathbf{K}_i(\rho \to \xi) > 0 \iff P_f(f(\xi) = i)\mathbf{K}_i(\xi \to \rho) > 0$$

This expresses the constraint that if you can reach $\xi$ from $\rho$ by selecting $\mathbf{Q}_i$ according to $f(\rho)$ then applying it, then with nonzero probability, reaching $\rho$ from $\xi$ is possible by selecting the same $\mathbf{Q}_i$ according to $f(\xi)$ then applying it. Let $I(\rho) = \{i \in I : P_f(f(\rho) = i) > 0\}$ be the set of indices (kernels) reachable from $\rho$, and $I(\rho, \xi) = I(\rho) \cap I(\xi)$ be the set of indices (kernels) reachable from both $\rho$ and $\xi$. To establish detailed balance for this kernel, it suffices to show that

$$P(\rho) \sum_{i \in I(\rho)} P_f(f(\rho) = i)\mathbf{K}_i(\rho \to \xi)\lambda_i(\rho \to \xi) = P(\xi) \sum_{i \in I(\xi)} P_f(f(\xi) = i)\mathbf{K}_i(\xi \to \rho)\lambda_i(\xi \to \rho) \quad (20)$$

Due to our symmetry condition, $i \in I(\rho)$ but $i \notin I(\xi)$ implies that $\mathbf{K}_i(\rho \to \xi) = 0$. Therefore detailed balance is satisfied if and only if the following holds:

$$P(\rho) \sum_{i \in I(\rho,\xi)} P_f(f(\rho) = i)\mathbf{K}_i(\rho \to \xi)\lambda_i(\rho \to \xi) = P(\xi) \sum_{i \in I(\rho,\xi)} P_f(f(\xi) = i)\mathbf{K}_i(\xi \to \rho)\lambda_i(\xi \to \rho) \quad (21)$$

By hypothesis, we have that each $\mathbf{Q}_i = (\mathbf{K}_i, \alpha_i)$ satisfies detailed balance with respect to $P$:

$$P(\rho)\mathbf{K}_i(\rho \to \xi)\alpha_i(\rho \to \xi) = P(\xi)\mathbf{K}_i(\xi \to \rho)\alpha_i(\xi \to \rho) \qquad \text{for all } i \tag{22}$$

Let $\lambda_i(\rho \to \xi) = \alpha_i(\rho \to \xi)P_f(f(\xi) = i)$. If we multiply both sides of (22) by $P_f(f(\rho) = i)P_f(f(\xi) = i)$ then sum both sides over $i \in I(\rho, \xi)$, we establish (21).

#### 8.2.3 THE MIXMH OPERATOR

We define the **MixMH** operator as follows. **MixMH** takes as input stochastic index sampler $f$ and a set of base kernels $\{K_i, \alpha_i\}_{i \in I}$ parameterized by the index $i$ sampled from the index sampler:

$$\mathbf{MixMH}(f, \{(\mathbf{K}_i, \alpha_i)\}_{i \in I})$$

Now let $\lambda_i(\rho \to \xi) = \alpha_i(\rho \to \xi)P(f(\xi) = i)$. To apply $\mathbf{MixMH}(f, \{(\mathbf{K}_i, \alpha_i)\}_{i \in I})$ we:

1. Sample $i$ from $f(\rho)$.
2. Apply $(\mathbf{K}_i, \lambda_i)$.

### 8.3 Particle-based Kernels

Our goal here is to define and sketch correctness arguments for particle-based transition operators. We first describe particle sets obtained from repeated applications of a single kernel. Our formulation incorporates a boosted acceptance ratio that is designed to recover Metropolis-Hastings in the special case where there is only one new particle generated.[14]

#### 8.3.1 GENERATING PARTICLES BY REPEATEDLY APPLYING A SEED KERNEL

Suppose we have a $(\mathbf{torus}, \mathbf{D})$ pair, and we attach local simulation kernels to the nodes in $\mathbf{D}$ to define a distribution $\mathbf{K}$ on $\Xi[\mathbf{torus}, \mathbf{D}]$. By (1), **regenAndAttach** will return

$$w_\xi = \frac{P(\mathbf{R}(\xi), \mathbf{A})}{\mathbf{K}(\xi)}$$

Let $\Upsilon = \{(\xi_i, w_i) : i = 1, \ldots, n\}$ be a multiset of weighted particles in $\Xi$. We can define the kernel $\mathbf{K}_\Upsilon(\cdot \to \cdot)$ on $\Upsilon$ as follows:

$$\mathbf{K}_\Upsilon(\xi_i \to \xi_j) = \frac{\mathbf{nds}(\xi_j, i)w_j}{w_{-i}}$$

where $\mathbf{nds}(\xi_j, i)$ is the number of duplicates of $\xi_j$ in $\Upsilon \setminus \xi_i$.The $\alpha$-factor for $\mathbf{K}_\Upsilon$ in isolation is:

$$\frac{\alpha(\xi_i \to \xi_j)}{\alpha(\xi_j \to \xi_i)} = \frac{P(\xi_j)\mathbf{K}_\Upsilon(\xi_j \to \xi_i)}{P(\xi_i)\mathbf{K}_\Upsilon(\xi_i \to \xi_j)} \tag{23}$$

$$= \frac{P(\mathbf{R}(\xi_j), \mathbf{A})\frac{\mathbf{nds}(\xi_i, j)w_i}{w_{-j}}}{P(\mathbf{R}(\xi_i), \mathbf{A})\frac{\mathbf{nds}(\xi_j, i)w_j}{w_{-i}}} \tag{24}$$

14. This boosting comes at the cost of subtle ergodicity violations if this is the only transition operator used and the hypothesis space contains the right symmetries. Simpler particle-based analyses — where all particles, including the current trace, are resampled from — are possible and indeed straightforward. These yield Boltzmann-style acceptance rules that self-transition more frequently and therefore explore the space less efficiently, but avoid ergodicity issues.

We can sample the kernel $\Upsilon$ starting from $\xi_i$ by generating $n-1$ additional particles from $\mathbf{K_{regen}}$. The probability of generating $\Upsilon$ is the probability of sampling the $n-1$ other particles, multiplied by the number of distinct orders in which they could have been sampled:

$$P(\Upsilon|\xi_i) = \frac{(n-1)!}{\prod_{\gamma \in \mathbf{unique}(\Upsilon \setminus \xi_i)} \mathbf{nds}(\gamma, i)!} \prod_{k \neq i} \mathbf{K_{regen}}(\xi_k) \tag{25}$$

**Claim 8.1.**

$$\frac{P(\Upsilon|\xi_j)}{P(\Upsilon|\xi_i)} = \frac{\mathbf{nds}(\xi_j, i) \prod_{k \neq j} \mathbf{K_{regen}}(\xi_k)}{\mathbf{nds}(\xi_i, j) \prod_{k \neq i} \mathbf{K_{regen}}(\xi_k)}$$

*Proof.*

$$\frac{P(\Upsilon|\xi_j)}{P(\Upsilon|\xi_i)} = \frac{\frac{(n-1)!}{\prod_{\gamma \in \mathbf{unique}(\Upsilon \setminus \xi_j)} \mathbf{nds}(\gamma, j)!} \prod_{k \neq j} \mathbf{K_{regen}}(\xi_k)}{\frac{(n-1)!}{\prod_{\gamma \in \mathbf{unique}(\Upsilon \setminus \xi_i)} \mathbf{nds}(\gamma, i)!} \prod_{k \neq i} \mathbf{K_{regen}}(\xi_k)} \tag{26}$$

$$= \frac{\prod_{\gamma \in \mathbf{unique}(\Upsilon \setminus \xi_i)} \mathbf{nds}(\gamma, i)! \prod_{k \neq j} \mathbf{K_{regen}}(\xi_k)}{\prod_{\gamma \in \mathbf{unique}(\Upsilon \setminus \xi_j)} \mathbf{nds}(\gamma, j)! \prod_{k \neq i} \mathbf{K_{regen}}(\xi_k)} \tag{27}$$

$$= \frac{\mathbf{nds}(\xi_j, i)! \mathbf{nds}(\xi_i, i)! \prod_{k \neq j} \mathbf{K_{regen}}(\xi_k)}{\mathbf{nds}(\xi_i, j)! \mathbf{nds}(\xi_j, j)! \prod_{k \neq i} \mathbf{K_{regen}}(\xi_k)} \tag{28}$$

$$= \frac{\mathbf{nds}(\xi_j, i)}{\mathbf{nds}(\xi_i, j)} \frac{\prod_{k \neq j} \mathbf{K_{regen}}(\xi_k)}{\prod_{k \neq i} \mathbf{K_{regen}}(\xi_k)} \tag{29}$$

□

Thus once we apply **MixMH**, the $\alpha$-factor becomes:

$$\frac{\alpha(\xi_i \to \xi_j)}{\alpha(\xi_j \to \xi_i)} = \frac{P(\mathbf{R}(\xi_j), \mathbf{A}) \frac{\mathbf{nds}(\xi_i, j) w_i}{w_{-j}} P(\Upsilon|\xi_j)}{P(\mathbf{R}(\xi_i), \mathbf{A}) \frac{\mathbf{nds}(\xi_j, i) w_j}{w_{-i}} P(\Upsilon|\xi_i)} \tag{30}$$

$$= \frac{P(\mathbf{R}(\xi_j), \mathbf{A}) \frac{\mathbf{nds}(\xi_i, j) w_i}{w_{-j}} \mathbf{nds}(\xi_j, i) \prod_{k \neq j} \mathbf{K}(\xi_k)}{P(\mathbf{R}(\xi_i), \mathbf{A}) \frac{\mathbf{nds}(\xi_j, i) w_j}{w_{-i}} \mathbf{nds}(\xi_i, j) \prod_{k \neq i} \mathbf{K}(\xi_k)} \tag{31}$$

$$= \frac{P(\mathbf{R}(\xi_j), \mathbf{A}) \frac{w_i}{w_{-j}} \prod_{k \neq j} \mathbf{K}(\xi_k)}{P(\mathbf{R}(\xi_i), \mathbf{A}) \frac{w_j}{w_{-i}} \prod_{k \neq i} \mathbf{K}(\xi_k)} \tag{32}$$

$$= \frac{P(\mathbf{R}(\xi_j), \mathbf{A}) \frac{\left(\frac{P(\mathbf{R}(\xi_i), \mathbf{A})}{\mathbf{K}(\xi_i)}\right)}{w_{-j}} \prod_{k \neq j} \mathbf{K}(\xi_k)}{P(\mathbf{R}(\xi_i), \mathbf{A}) \frac{\left(\frac{P(\mathbf{R}(\xi_j), \mathbf{A})}{\mathbf{K}(\xi_j)}\right)}{w_{-i}} \prod_{k \neq i} \mathbf{K}(\xi_k)} \tag{33}$$

$$= \frac{w_{-i}}{w_{-j}} \tag{34}$$

### 8.3.2 THE $\text{MH}_n$ OPERATOR

This pattern, where a multiset of weighted particles is generated from a "seed" kernel $\mathbf{K}$ and then sampled from, is a sufficiently common operation that we give it a name:

$$\mathbf{MH}_n(\mathbf{K}) = \mathbf{MixMH}(\Upsilon \sim \mathbf{K}, \mathbf{K}_\Upsilon)$$

We will use variations on it throughout this section. The multiset functions as the index for **MixMH**, and the sampling step where a particle is chosen functions as the base proposal kernel.

### 8.3.3 METROPOLIS-HASTINGS AS SPECIAL CASE OF PARTICLE METHODS

Metropolis-Hastings with a simulation kernel is actually just the special case $\mathbf{MH}_2$, since in this case $w_{-\rho} = w_\xi$. Parallelizable extensions of this kind of "locally independent" Metropolis-Hastings scheme are also natural. For example, one could make multiple proposals and weigh them against against one another. This could be viewed as an importance sampling approximation to an optimal Gibbs proposal over a scaffold, where the proposal can be any simulation kernel, including one that conditions on downstream information. That said, the restriction to simulation kernels is significant. Gaussian drift kernels, for example, are not permitted.

### 8.3.4 ENUMERATIVE GIBBS AS A SPECIAL CASE: USING DIFFERENT KERNELS FOR EACH PARTICLE

There are many ways to generate weighted particle sets $\Upsilon$. For instance, we may want to allow a different kernel $\mathbf{K}_i$ for each particle $\xi_i$. We can represent enumeration over some set of variables this way: each distinct tuple in the Cartesian product of the domains of the variables will correspond to a different kernel.

Suppose there are some variables in $\mathbf{D}$ that we want to enumerate over, yielding $n$ total combinations. Assume that $\rho$ has the first combination. We can sample $\Upsilon$ from $\rho$ as follows: call **regen** $n-1$ times as before to generate $n-1$ new particles, but passing $\mathbf{K}_i$ for $\xi_i$, where $\mathbf{K}_i$ is the same as $\mathbf{K}$ except with deterministic kernels replacing the old kernels on the nodes that we are enumerating.

The analysis is even simpler in this case because the particles are distinct. Once we apply **MixMH**, the $\alpha$-factor becomes:

$$\frac{\alpha(\xi_i \to \xi_j)}{\alpha(\xi_j \to \xi_i)} = \frac{P(\xi_j)\mathbf{K}_\Upsilon(\xi_j \to \xi_i)\prod_{k\neq j}\mathbf{K}_k(\xi_k)}{P(\xi_i)\mathbf{K}_\Upsilon(\xi_i \to \xi_j)\prod_{k\neq i}\mathbf{K}_k(\xi_k)} \tag{35}$$

$$= \frac{P(\mathbf{R}(\xi_j),\mathbf{A})\frac{w_i}{w_{-j}}\prod_{k\neq j}\mathbf{K}_k(\xi_k)}{P(\mathbf{R}(\xi_i),\mathbf{A})\frac{w_j}{w_{-i}}\prod_{k\neq i}\mathbf{K}_k(\xi_k)} \tag{36}$$

$$= \frac{P(\mathbf{R}(\xi_j),\mathbf{A})\frac{\left(\frac{P(\mathbf{R}(\xi_i),\mathbf{A})}{\mathbf{K}(\xi_i)}\right)}{w_{-j}}\prod_{k\neq j}\mathbf{K}_k(\xi_k)}{P(\mathbf{R}(\xi_j),\mathbf{A})\frac{\left(\frac{P(\mathbf{R}(\xi_j),\mathbf{A})}{\mathbf{K}(\xi_j)}\right)}{w_{-i}}\prod_{k\neq i}\mathbf{K}_k(\xi_k)} \tag{37}$$

$$= \frac{w_{-i}}{w_{-j}} \tag{38}$$

as above.

### 8.3.5 Particle Markov chain Monte Carlo: adding iteration and resampling

It is often useful to iterate particle-based methods, and to resample collections of particles based on their weights, stochastically allocating particles to regions of the execution space that appear locally promising. This insight is at the heart of particle filtering, i.e. sequential importance sampling with resampling (Doucet et al., 2001), as well as more sophisticated sequential Monte Carlo techniques. For example, (Andrieu et al., 2010) introduces particle Markov chain Monte Carlo methods, a family of techniques for using sequential Monte Carlo to make sensible proposals over subsets of variables as part of larger MCMC schemes.

We can also use conditional SMC to generate $\Upsilon$. As above, suppose we have $\rho$ and $\mathbf{D}$, with local simulation kernels attached to the nodes in $\mathbf{D}$, which determines a simulation kernel $\mathbf{K}$ that generates samples by propagating along $\mathbf{D}$ via **regen**. Suppose we group the sinks into $T$ groups, which in turn partitions all of $\mathbf{D}$ into $T$ groups $\mathbf{D}_1, \ldots, \mathbf{D}_T$ according to the **regen** recursion. We will refer to $\mathbf{K}$ on $\mathbf{R}_t$ as $\mathbf{K}_t$.

To establish notation, first let us consider a simpler case: basic sequential Monte Carlo, instead of conditional SMC. We can propose along $\mathbf{R}_1$ from $\mathbf{K}_1$, and then (inductively) propagate to time $t$ by sampling independently from a proposal kernel,

$$M_t(a_t^{(m)}, \mathbf{R}_t(\xi_t^{(m)})) = \frac{w_{t-1}^{(m)}}{w_{t-1}} \mathbf{K}_t(\mathbf{R}_t(\xi_t^{(m)}))$$

where

$$w_{t-1}^{(m)} = \frac{P(\mathbf{R}_{t-1}(\xi_{t-1}^{(m)}), \mathbf{A}_{t-1})}{\mathbf{K}_{t-1}(\xi_{t-1}^{m})}$$

is the return value from **regen** on the $t-1$st group of sinks, and where $\mathbf{R}_t(\xi_t^{(m)})$ is assumed to be conditioned on $\xi_t^{(m)}$'s parent particle $\xi_{t-1}^{(a_t^{(m)})}$.

It is instructive to view this sampling procedure as a way of generating a single sample of all auxiliary variables $\Gamma$ from the density

$$\psi(\Gamma) = \prod_{m=1}^{n} \mathbf{K}_1(\mathbf{R}_1(\xi_1^{(m)})) \prod_{t=2}^{T} \prod_{m=1}^{n} M_t(a_t^{(m)}, \xi_t^{(m)}) \tag{39}$$

$$= \prod_{m=1}^{n} \mathbf{K}_1(\mathbf{R}_1(\xi_1^{(m)})) \prod_{t=2}^{T} \prod_{m=1}^{n} \frac{w_{t-1}^{(m)}}{w_{t-1}} \mathbf{K}_t(\mathbf{R}_t(\xi_t^{(m)})) \tag{40}$$

$$= \prod_{m=1}^{n} \mathbf{K}_1(\mathbf{R}_1(\xi_1^{(m)})) \prod_{t=2}^{T} \prod_{m=1}^{n} \frac{P(\mathbf{R}_{t-1}(\xi_{t-1}^{(m)}), \mathbf{A}_{t-1}) \mathbf{K}_t(\mathbf{R}_t(\xi_t^{(m)}))}{\mathbf{K}_{t-1}(\mathbf{R}_{t-1}(\xi_{t-1}^{(m)})) w_{t-1}} \tag{41}$$

$$= \left( \prod_{t=2}^{T} \prod_{m=1}^{n} \frac{1}{w_{t-1}} \right) \prod_{m=1}^{n} \mathbf{K}_1(\mathbf{R}_1(\xi_1^{(m)})) \prod_{t=2}^{T} \prod_{m=1}^{n} \frac{P(\mathbf{R}_{t-1}(\xi_{t-1}^{(m)}), \mathbf{A}_{t-1}) \mathbf{K}_t(\mathbf{R}_t(\xi_t^{(m)}))}{\mathbf{K}_{t-1}(\mathbf{R}_{t-1}(\xi_{t-1}^{(m)}))} \tag{42}$$

Now suppose we perform the conditional SMC sweep to sample $\Gamma$. First we sample an index $b_t \in \{1, \ldots, n\}$ for each time step $t = 1, \ldots, T-1$ for the index of the forced resampling of the source particle $\xi_i$. Then we sample the other particles as in standard CSMC.

The probability of $\Gamma$ starting from $\xi_i$ is precisely:

$$P(\Gamma;\xi_i) = \left(\prod_{t=2}^{T} w_{t-1}\right) \frac{\psi(\Gamma)}{n^{T-1}\mathbf{K}_1(\mathbf{R}_1(\xi_1^{(b_1)}))\prod_{t=2}^{T} w_{t-1}^{(b_t)}\mathbf{K}_t(\mathbf{R}_t(\xi_t^{(b_t)}))} \tag{43}$$

$$= \left(\prod_{t=2}^{T} w_{t-1}\right) \frac{w_T^{(i)}\psi(\Gamma)}{n^{T-1}P(\mathbf{R}(\xi^{(i)}),\mathbf{A})} \tag{44}$$

However, we want to index our kernel with multiset semantics for the complete particles as we did above, and so we index by the equivalence class $\overline{\Gamma}$ where $\Gamma \equiv \Gamma'$ if they agree on everything up until the last time step, and then agree on the same multiset of complete particles. We have

$$P(\overline{\Gamma};\xi_i) = P(\Gamma;\xi_i)\frac{(n-1)!}{\prod_{\gamma\in\mathbf{unique}(\Gamma^{(T)}\setminus\xi_i)}\mathbf{nds}(\gamma,i)!}$$

and thus when we apply **MixMH**, the $\alpha$-factor becomes[15]

$$\frac{\alpha(\xi_i\to\xi_j)}{\alpha(\xi_j\to\xi_i)} = \frac{P(\xi_j)\mathbf{K}_{\overline{\Gamma}}(\xi_j\to\xi_i)P(\overline{\Gamma}|\xi_j)}{P(\xi_i)\mathbf{K}_{\overline{\Gamma}}(\xi_i\to\xi_j)P(\overline{\Gamma}|\xi_i)} \tag{45}$$

$$= \frac{P(\xi_j)\mathbf{K}_{\overline{\Gamma}}(\xi_j\to\xi_i)\mathbf{nds}(\xi_j,i)P(\Gamma|\xi_j)}{P(\xi_i)\mathbf{K}_{\overline{\Gamma}}(\xi_i\to\xi_j)\mathbf{nds}(\xi_i,j)P(\Gamma|\xi_i)} \tag{46}$$

$$= \frac{P(\xi_j)\mathbf{K}_{\overline{\Gamma}}(\xi_j\to\xi_i)\mathbf{nds}(\xi_j,i)P(\mathbf{R}(\xi^{(i)}),\mathbf{A})w_j}{P(\xi_i)\mathbf{K}_{\overline{\Gamma}}(\xi_i\to\xi_j)\mathbf{nds}(\xi_i,j)P(\mathbf{R}(\xi^{(j)}),\mathbf{A})w_i} \tag{47}$$

$$= \frac{\mathbf{K}_{\overline{\Gamma}}(\xi_j\to\xi_i)\mathbf{nds}(\xi_j,i)w_j}{\mathbf{K}_{\overline{\Gamma}}(\xi_i\to\xi_j)\mathbf{nds}(\xi_i,j)w_i} \tag{48}$$

$$= \frac{\frac{\mathbf{nds}(\xi_i,j)w_i}{w_{-j}}\mathbf{nds}(\xi_j,i)w_j}{\frac{\mathbf{nds}(\xi_j,i)w_j}{w_{-i}}\mathbf{nds}(\xi_i,j)w_i} \tag{49}$$

$$= \frac{w_{-i}}{w_{-j}} \tag{50}$$

We can think of this as a strict generalization of $\mathbf{MH}_n$ where the kernel is split into a sequence of kernels.

This scheme is called `PGibbs` in the Venture inference programming language, as it enables the approximation of blocked Gibbs sampling over arbitrary scaffolds. Cycles and mixtures of `PGibbs` with other kernels recovers a wide range of existing particle Markov chain Monte Carlo schemes as well as novel algorithms.

### 8.3.6 Pseudocode for `PGibbs` with simultaneous particles

Here we give pseudocode for implementing the `PGibbs` transition operator using an interface for simultaneous particles. The particle interface permits particles to be constructed from source PETs or from one another, sharing the maximum amount of state possible. Stochastic regeneration is used

15. When it is clear we are referring to the final weight, we will refer to $w_T^{(i)}$ as $w_i$ to be consistent with the previous sections.

to prepare the source trace ρ for conditional SMC, to populate the array of particles, and to calculate weights.

```
PGIBBS(trace, border, scaffold, P)
 1  T = border.length
 2  rhoWeights = [T]
 3  rhoDBs = [T]
 4  for t = T to 1
 5      rhoWeights[t], rhoDBs[t] = DETACHANDEXTRACT(trace, border[t], scaffold)
 6  particles = [P]
 7  particleWeights = [P]
 8  for p = 0 to P
 9      particles[p] = PARTICLE(trace)
10  particleWeights[0] =
        REGENERATEANDATTACH(particles[0], border[1], scaffold, TRUE, rhoDBs[1])
11  for p = 1 to P
12      particleWeights[p] =
            REGENERATEANDATTACH(particles[p], border[1], scaffold, FALSE, NIL)
13  newParticles = [P]
14  newParticleWeights = [P]
15  for t = 2 to T
16      newParticles[0] = PARTICLE(particles[0])
17      newParticleWeights[0] =
            REGENERATEANDATTACH(newParticles[0], border[t], scaffold, TRUE, rhoDBs[t])
18      for p = 1 to P
19          parentIndex = SAMPLECATEGORICAL(MAPEXP(particleWeights))
20          newParticles[p] = PARTICLE(particles[parentIndex])
21          newParticleWeights[p] =
                REGENERATEANDATTACH(newParticles[p], border[t], scaffold, FALSE, NIL)
22      particles = newParticles
23      particleWeights = newParticleWeights
24  finalIndex = SAMPLECATEGORICAL(MAPEXP(particleWeights[1 : P]))
25  weightMinusXi = LOGSUMEXP(particleWeights.REMOVE(finalIndex))
26  weightMinusRho = LOGSUMEXP(particleWeights.REMOVE(0))
27  alpha = weightMinusRho − weightMinusXi
28  return particles[finalIndex], alpha
```

This implementation illustrates the use of versions of **regen** and **detach** that act on particles, and the initialization of particles using parent particles as well as source traces. All but the last four lines are simply constructing the randomly chosen kernel to be applied, corresponding to proposing to replace the current race with the contents of a particle.

## 9. Conditional Independence and Parallelizing Transitions

Here we briefly describe the conditional independence relationships that are easy to extract from probabilistic execution traces. This analysis clarifies the potential long-range dependencies in a

PET introduced by the choice to make probabilistic closures into first-class objects in the language. These independence relationships could be used to support probabilistic program analyses and also to justify the correctness of parallelized kernel composition operators in the inference programming language. They also expose parallelism that is distinct from particle-level parallelism.

### 9.1 Markov Blankets, Envelopes, and Conditional Independence

In Bayesian networks, the Markov Blanket of a set of nodes provides a useful characterization of important conditional independencies. These independencies permit parallel simulation of Markov chain transition operators. In constrast, while the scaffold provides a useful factorization of the logdensity of a trace, it does not permit parallel simulation of transition operators. In this section, we show how to formulate a notion of locality that permits parallel transitions on PETs.

We first review Markov Blankets in Bayesian networks. Let $\mathbf{D}$ and $\mathbf{D}'$ be sets of nodes in a Bayesian network, $\mathbf{A}$ the children of $\mathbf{D}$, and $\mathbf{P}$ the parents of $\mathbf{D} \cup \mathbf{A}$ excluding $\mathbf{D}$ and $\mathbf{A}$. Then if $\mathbf{D}' \cap (\mathbf{D} \cup \mathbf{A} \cup \mathbf{P}) = \emptyset$, proposals can be made on the two regeneration graphs in parallel, although both proposals may *read* the same values. We refer to $\mathbf{A} \cup \mathbf{P}$ as the *Markov blanket* of $\mathbf{D}$, and it can be thought of as the set of all nodes that one must read from (but not write to) when resampling $\mathbf{D}$ and computing the new probabilities of its children $\mathbf{A}$.

The situation becomes more complicated in the case of PETs for two main reasons. First, the parents $\mathbf{P} = \mathbf{P}(\rho) \cup \mathbf{P}(\xi)$ are not known a priori, since we do not know $\mathbf{P}(\xi)$ until we have simulated $\xi$ to see what nodes it reads from. Second, the SP auxiliary states that will be mutated are not known a priori, since the PSPs being applied in $\mathbf{R}(\xi)$ are not known.

### 9.2 After-the-fact envelopes

Let $\mathbf{S}(\rho)$ denote the SP auxiliary states for SPs with PSPs that are applied in $\mathbf{R}(\rho)$. Define

$$\mathbf{MB}(\rho) = \mathbf{P}(\rho) \cup \mathbf{A} \tag{51}$$

to be the set of all nodes that are read from but not written to, as before for Bayesian networks. Now define the scope of a proposal

$$\mathbf{Scope}(\rho) = \mathbf{R}(\rho) \cup \mathbf{S}(\rho) \tag{52}$$

to be the set of all parts of the PET that are written to or created while generating $\rho$ from **torus**, and define the envelope

$$\mathbf{Envelope}(\rho) = \mathbf{MB}(\rho) \cup \mathbf{Scope}(\rho) \tag{53}$$

to be the set of all nodes that must be read from or written to during the proposal. Note that in general there may be some nodes in the scope that are written to but never read from, e.g. terminal resampling nodes and the sufficient statistics for uncollapsed SPs, but we will ignore this distinction for now.

We can make the following claim: suppose the proposal $\rho \to \xi$ temporally overlaps with the proposal $\rho' \to \xi'$, and that by some luck,

$$\big(\mathbf{Envelope}(\rho, \xi) \cap \mathbf{Scope}(\rho', \xi')\big) \cup \big(\mathbf{Envelope}(\rho', \xi') \cap \mathbf{Scope}(\rho, \xi)\big) = \emptyset \tag{54}$$

then the transitions will not have clashed. In words, this says that the two proposals will not clash as long as no node that one proposal reads differs in the two traces of the other proposal. If the transitions have not clashed, then they could have been simulated simultaneously.

### 9.3 The envelope of a scaffold

Define:

$$\mathbf{Envelope}[\mathbf{D}] = \bigcup_{\xi \in \Xi[\mathbf{D}]} \mathbf{Envelope}(\xi) \tag{55}$$

$$\mathbf{Scope}[\mathbf{D}] = \bigcup_{\xi \in \Xi[\mathbf{D}]} \mathbf{Scope}(\xi) \tag{56}$$

That is, $\mathbf{Envelope}[\mathbf{D}]$ and $\mathbf{Scope}[\mathbf{D}]$ correspond to the union of the trace-specific envelope and scope, taken over all possible completions of **torus**. Then two proposals on $\mathbf{D}$ and $\mathbf{D}'$ are guaranteed not to clash if

$$(\mathbf{Envelope}[\mathbf{D}] \cap \mathbf{Scope}[\mathbf{D}']) \cup (\mathbf{Envelope}[\mathbf{D}'] \cap \mathbf{Scope}[\mathbf{D}]) = \emptyset \tag{57}$$

Now let $\mathbf{R}$ and $\mathbf{R}'$ be the random variables corresponding to the results of invocations of **regen** (including the weights) on scaffolds with definite regeneration graphs $D$ and $D'$. One approach to formalizing the relationship between conditional independence and parallel simulation in Venture would be to first try to show that (57) holds if and only if $\mathbf{R} \perp \mathbf{R}'$, and second to try to show that random variables can safely be simulated in parallel if and only if they are conditionally independent.

When probabilistic programmers can identify two scaffolds for which (57) holds, they can schedule transitions on them simultaneously. Also, in some circumstances speculative simultaneous simulation may be beneficial, for example if the clash is on small, cheap procedures or an approximate transition that ignores some dependency can be used as a proposal for a serial transition.

It may also be possible to use this notion of envelope to predict beforehand which transitions we can run simultaneously. To do this, we will need to bound the Markov blanket and the scope. One option would be to statically analyze Venture program fragments, implementing a kind of "escape analysis" for random choices. Another would be to introduce language constructs and/or stochastic procedures that came with hints about Markov blanket composition. Future work will explore the efficacy of these and other techniques in practice.

## 10. Related Work

Venture builds on and contains complementary insights to a number of other probabilistic programming languages, implementations, and inference techniques. For example, Monte (Bonawitz and Mansinghka, 2009) was the first commercially developed prototype interpreter for a Church-like probabilistic programming language. Its architecture is the most direct antecedent of Venture. For example, it included a scaffold-like decomposition of local transitions in terms of "proposal blankets" for single-variable changes. Preliminary prototypes of Venture (Wu, 2013) and of an approximate multi-core scheme for Church-like languages (Perov and Mansinghka, 2012) also introduced early, ad-hoc versions of some of the ideas in this paper, all in an attempt to control the asymptotic cost of each inference transition.

The most salient inefficiency in many Church systems has been that for typical machine learning problems with $N$ datapoints, a single sweep of inference over all random variables requires runtime that scales as $O(N^2)$. This quadratic scaling, combined with the absolute constant factors, has made it impossible to apply these systems beyond hundreds of datapoints. For "lightweight" implementations based on direct transformational compilation or augmented interpretation, such as Bher (Wingate et al., 2011b), this is a basic consequence of the approach: the entire program must be re-simulated to make a change to a single latent variable.

A variety of approaches have been tried to mitigate this problem. For earlier implementations such as the first prototype implementation of Church (Mansinghka, 2009) atop the Blaise system (Bonawitz, 2008), the MIT-Church implementation, and also Monte, ad-hoc attempts were made to control the scope of re-simulation, approximating the Venture notions of PETs and their partition into scaffolds. Similarly, the Shred implementation of Church (Yang et al., 2012) uses techniques from program analysis coupled with just-in-time compilation to try to avoid the re-simulation involved in lightweight implementations of Metropolis-Hastings and also reduce constant-factor overhead.

These approaches exhibit different limitations with respect to generality, scalability, absolute achievable efficiency and runtime predictability. For example, the Blaise implementation of Church and MIT-Church both exhibited quadratic scaling in some cases, and involved substantial runtime overheads. The program analysis and compilation techniques used in Shred incur runtime costs that can be significant in absolute terms and difficult to predict a priori, and additionally impose constraints on the integration of custom primitives into the language. Finally, all of these approaches have only applied directly to single-site Metropolis-Hastings approaches to inference.

Outside of Church-like languages, BLOG (Milch et al., 2007) — the first open-universe probabilistic programming language — is of particular relevance. For example, the original BLOG inference engine was based on infinite contingent Bayesian networks (ICBN) (Milch et al., 2005). ICBNs characterize the valid factorizations of their probability distribution and represent dependencies whose existence is contingent on the values of other variables. They thus provide a data structure for BLOG that serves an analogous function to PETs in Venture. Also, the Gibbs sampling algorithm introduced for BLOG (Arora et al., 2012) exploits a decomposition of a possible world that is related to the scaffolds used to define the scope of inference in Venture. However, BLOG is not higher-order — it does not support random variables that are themselves probabilistic procedures — nor does it provide a programmable inference mechanism. BLOG also does not support collapsed primitives that exhibit exchangeable coupling between applications, primitives that lack calculable probability densities, or primitives that can create and destroy latent variables while providing their own external inference mechanism.

The Figaro system for probabilistic programming (Pfeffer, 2009) shares several goals with Venture, although the differences in language semantics and inference architecture are substantial. Venture is a stand-alone virtual machine, while Figaro is based on an embedded design, implemented as a Scala library. Figaro models are represented as data objects in Scala, and model construction and inference both proceed via calling into a Scala API. This enables Figaro to leverage the mature toolchain for Scala and the JVM, and avoids the need to reimplement difficult but standard programming language features from scratch. It remains to be seen how to apply this machinery to models that depend on higher-order probabilistic procedures, primitives exhibiting exchangeable coupling, and likelihood-free primitives, or how to enable users to extend Figaro with custom model fragments equipped with arbitrary inference schemes. IBAL (Pfeffer, 2001), an ancestor of Figaro

based on an embedding of stochastic choice into the functional language ML, is arguably more similar to Venture in terms of its interface. However, the inference strategies around which IBAL was designed imply restrictions on stochastic choices that are analogous to Infer.NET.

Our hybrid inference primitives embody generalizations of other proposals for global inference in probabilistic programs. The idea of global mean-field inference in probabilistic programs via stochastic optimization, implemented using repeated re-simulation of the program, was first proposed in (Wingate and Weber, 2013). Similarly, a global implementation of particle Gibbs via conditional sequential Monte Carlo, implemented via repeated forward re-simulation, has been independently and concurrently developed in (Wood et al., 2014). Both of these approaches were implemented using the "lightweight" scheme proposed in (Wingate et al., 2011b). They thus lack dependency tracking and will exhibit unfavorable asymptotic scaling. The mean field and conditional SMC schemes developed in this paper are implemented via stochastic regeneration, building on PETs for efficiency and supporting the full SPI. The techniques from this paper can also be used on subsets of the random choices in a probabilistic program and composed with other inference techniques, yielding hybrid inference strategies that may be asymptotically more efficient than homogeneous ones.

To the best of our knowledge, Venture is the first probabilistic programming platform to support a compositional inference programming language with multiple computationally universal primitives. It is also the only probabilistic programming system to support higher-order probabilistic procedures as first class objects, and in particular the implementation of arbitrary higher-order probabilistic procedures with external, per-application latent variables as ordinary primitives. Venture is also the only probabilistic programming system that integrates exact and approximate sampling techniques based on standard Markov chain, sequential Monte Carlo and variational inference.

## 11. Discussion

We have described Venture, along with the key concepts, data structures and algorithms needed to implement a scalable interpreter with programmable inference. Venture includes a common stochastic procedure interface that supports higher-order probabilistic procedures, "likelihood-free" procedures, procedures with exchangeable coupling between their invocations, and procedures that maintain external latent variables and supply external inference code. We have seen how the probabilistic execution traces on which Venture is based generalize key ideas from Bayesian networks, and support partitioning traces into scaffolds corresponding to well-defined inference subproblems. We have defined stochastic regeneration algorithms over scaffolds, and shown how to use them to implement multiple general-purpose inference strategies. We have also given example Venture programs that illustrate aspects of the modeling and inference languages that it provides.

The coverage of Venture in terms of models, inference strategies and end-to-end problems needs to be carefully assessed. It remains to be seen whether the current set of inference strategies are truly sufficient for the full range of problems arising across the spectrum from Bayesian data analysis to large-scale machine learning to real-time robotics. However, expanding the set of inference primitives may be straightforward. One strategy rests on the analogies between PETs and Bayesian networks. For example, it may be possible to define analogues of junction trees for fragments of PETs by conditioning on random choices with existential dependencies. Along similar lines, extensions to the `enumerative_gibbs` inference instruction could leverage the insights from (Koller et al., 1997). Variants based on message passing techniques such as expectation propagation might

also be fruitful. Other inference techniques that we could build using PETs and the SPI include slice sampling and Hamiltonian Monte Carlo. Of these, Hamiltonian Monte Carlo has already seen some use in probabilistic programming (Stan Development Team, 2013; Wingate et al., 2011a). However, these implementations do not apply to arbitrary Venture scopes, where proposals can trigger the resampling of other random choices due to the presence of "likelihood-free" primitives or the brush. Generalizations of Hamiltonian and slice methods that use the auxiliary variable machinery presented in this paper are straightforward and are currently included in development branches of the Venture system.

Although some real-world applications have been successfully implemented using unoptimized prototypes, the performance surface of Venture is largely uncharacterized. This is intentional. Our understanding of probabilistic programming is more limited than our understanding of functional programming when Miranda was introduced (Turner, 1985). Standard graphical models from machine learning correspond to short nested loops that usually can be fully unrolled before execution, and therefore only exercise small subsets of the capabilities of typical expressive languages. It thus seems premature to focus on optimizing runtime performance until better foundations have been established.

The theoretical principles needed to assess the tradeoffs between inference accuracy, asymptotic scaling, memory consumption, and absolute runtime are currently unclear. Point-wise comparisons of runtime are easy to make, but just as easily mislead. Even small changes in problem formulation, dataset size or accuracy can lead to large changes in runtime. The asymptotic scaling of runtime for many probabilistic programming systems is unknown, and for some problems the relevant scale parameters are hard to identify. Systematic comparisons of inference strategies that control for implementation constants and scale thresholds have yet to be performed. Although Venture's asymptotic scaling competitive with hand-optimized samplers in theory, a rigorous empirical assessment is needed. Mathematically rigorous cost models are in our view a precondition for comparative benchmarking and thus could be an important research direction. Additionally, once the relationships between the asymptotic scaling of forward simulation and the asymptotic scaling of typical inference strategies has been characterized, it will be possible to search for effective equivalences between custom inference strategies and model transformations.

Performance engineering research for Venture can build on standard techniques for higher-order languages as well as exploitation of the structure of probabilistic models and inference strategies. Immediate opportunities include runtime specialization via type hints and/or scaffold contents, compilation of the transition operators arising from specific inference instructions, and optimizations to the SPI to minimize copying of data. PETs could be used to support a memory manager that exploits locality of reference along PETs — and conditional independence, more generally — to improve cache efficiency. For example, one could pre-calculate the envelope of a specific inference instruction, and pre-fetch the entire PET fragment. Similar optimizations will be beneficial for high-performance multicore implementations. There are also opportunities for asymptotic savings if deterministic sub-segments of PETs can be compressed; in principle, the memory requirements of inference should scale with the amount of randomness consumed by the model, not by its execution time. Preliminary explorations of some of these ideas have yielded promising results (Perov and Mansinghka, 2012; Wu, 2013) but much work remains to be done. In practice, we believe the judicious migration of performance sensitive regions of a probabilistic program into hand-optimized inference code will be as important as runtime and compiler optimizations.

### 11.1 Debugging and profiling probabilistic programs

Venture makes interactive modeling and inference possible by providing an expressive modeling language with scalable programmable inference. By removing the need to do integrated co-design of probabilistic models, inference algorithms and their implementations, Venture removes one of the main bottlenecks in building probabilistic computing systems. This is far from the only bottleneck, however. Design, validation, debugging and optimization of probabilistic programs — both model programs and inference programs — still requires expertise.

The promise of probabilistic programming is that instead of having to learn multiple non-overlapping fields, future modelers will be able to learn a coherent body of probabilistic programming principles. They should be able to use standard probabilistic programming tools and workflows to navigate the design, validation, debugging and optimization process. This could evolve analogously to how programmers learn to design, test, debug and optimize traditional programs using a mix of intuitive modeling, mathematical analysis and experimentation supported by sophisticated tools.

Venture implements a stochastic semantics for inference that is designed to cohere with Bayesian reasoning via exact sampling in the limit of infinite computation. This facilitates the development of mathematically rigorous debugging strategies. First, the precise gap between exact and approximate sampling can sometimes be measured on small-scale examples; this provides a crucial control for larger-scale debugging. Second, as the amount of computation devoted to inference increases, the quality of the approximation can only increase. As a result, at potentially substantial computational cost, it is always possible to reduce the impact of approximate inference (as distinct from modeling issues or data issues).

There is even a path forward at scales where the true distribution is effectively impossible to sample from or even roughly characterize. For example, the Bernstein-von Mises family of theorems gives general conditions under which posteriors concentrate onto hypotheses that satisfy various invariants. These can be exploited by debugging tools: a tool could increase the amount of synthetic data and inference used to test a probabilistic program until the posterior concentrates. At this point the probabilistic program begins to resemble a problem with a "single right answer", significantly simplifying debugging and testing as compared to regimes where the true distribution can be arbitrarily spread out or multi-modal. Many more tools and techniques will no doubt be needed to tease apart the impact of model mismatch, inadequate or inappropriate data and insufficient inference.

Traditional profilers may also have natural analogs in probabilistic programming. A profiler for a Venture program might track "hot spots" — scaffolds (and their source code locations) where a disproportionate fraction of runtime is spent — as well as "cold spots": scaffolds where transitions are rejected with unusual frequency. Such cold spots might serve as useful warning signs of limited, purely local convergence of inference. As Venture maintains an execution trace, it should be possible to link an execution trace location with the source code location responsible for it, and potentially help probabilistic programmers identify and modify unnecessarily deterministic or constrained model fragments. Both together could help Venture programmers identify the portions of their program that ought to be moved to optimized foreign inference code.

## 11.2 Inference programming

The Venture inference programming language has many known limitations that would be straightforward to address. For example, it currently lacks an iteration construct, a notion of procedural abstraction (so inference strategies can be parameterized and reused), the ability to dynamically evaluate modeling expressions to stochastically generate scope and block names, or an analogue of `eval` that gives inference programs the ability to programmatically execute new directives in the hosting Venture virtual machine. Additionally, its current set of inference primitives cannot themselves be implemented via compound statements in the language; to support this, an analogue of stochastic regeneration could potentially be exposed. Future work will address these issues by reintroducing the full expressiveness of Lisp into the inference expression language and determining what extensions are needed to capture the interactions between modeling and approximate inference.

More substantial inference programming extensions may also be fruitful. For example, it may be fruitful to incorporate support for *inference procedures*: reusable proposal schemes for scaffolds that match pre-specified patterns, where the proposal mechanism is implemented by another Venture program. The scaffold's contents constitute the formal arguments. Each random choice in the border could be mapped to an `OBSERVE`, and the new values for resampling nodes could each come from a `PREDICT`. If inference in the proposal program is assumed to have converged, it should be possible to construct a valid transition operator. This mechanism would enable Venture programmers to use probabilistic modeling and approximate inference to design inference strategies, turning every modeling idiom in Venture into a potential inference tool. A similar mechanism could facilitate the use of custom Venture programs as the skeleton for variational approximations. Finally, the inference programming language needs to be extended to relax the constraints of soundness. Instead of restricting programmers to transition operators that are guaranteed by construction to leave the conditioned distribution on traces invariant, the language should permit experts to introduce arbitrary transition operators on traces.

It will be interesting to develop probabilistic programs that automate aspects of inference while going beyond traditional formulations of interpretation and compilation. For example, it is theoretically possible to develop probabilistic programs that work as inference optimizers. Such programs would take a Venture program as input, including only placeholder `INFER` instructions, and produce as output a new Venture program with `INFER` instructions that are likely to perform better. These programs could also transform the modeling instructions and the instructions introducing the data to improve computational or inferential performance. Depending on architecture, these programs could be viewed as inference planners, compilers, or an integrated combination of the two. Probabilistic models, approximate inference and Bayesian learning could all be deployed to augment engineering of the planning algorithm, compiler transformations or even the objective function used to summarize inference performance. Machine learning schemes for algorithm selection (Xu et al., 2008) and reinforcement-learning-based meta-computation (Lagoudakis and Littman, 2001) can be viewed as natural special cases. Venture also makes it possible to integrate a "control" probabilistic program into an interpreter running another probabilistic program. The control program could intercept and modify the inference instructions in the running Venture program based on approximate, model-based inferences from live performance data, and perhaps influence the machine code generation process invoked by a just-in-time compiler.

Inference programming also supports the integration of "approximate" compilers based on probabilistic modeling and approximately Bayesian learning. Consider a probabilistic program with a specific set of OBSERVEs and PREDICTs. The expressions in the OBSERVEs define a space of possible inputs, each corresponding to a set of literal values, one per observation. The expressions in the PREDICTs define a space of possible outputs, each corresponding to an assignment of a sampled value to each PREDICT. A compiler for Venture might generate an equivalent program that is restricted to precisely the given pattern of inputs and outputs, optimizing the implementation of any INFER instructions in the program given this restriction. It is also possible to use modeling and inference to approximately emulate the program, by writing a probabilistic program that models $p(\{\texttt{PREDICT}s\}|\{\texttt{OBSERVE}s\})$, with parameters and/or structure estimated from data that is generated from the original probabilistic program. This can be viewed as approximate compilation implemented via inference, where the target language is given by the hypothesis space of the model in the emulator program. Both the creation of this kind of emulator and its use as a proposal would be natural to integrate as additional inference instructions.

Probabilistic programming systems should ultimately support the complete spectrum from black-box, truly automatic inference to highly customized inference strategies. So far Venture has focused on the extremes. Other points on the spectrum will require extensions to Venture's instruction language to encode specifications for exact and approximate inference within particular runtime and/or accuracy parameters. Developing a suitable specification language and cost model to enable precise control over the scope of automatic inference is an important challenge for future research.

### 11.3 Conclusion

The similarities between non-probabilistic, Turing-complete, higher-order programming languages are striking given the enormous design space of such languages. Lisp, Java and Python support many of the same programming idioms and can be used to simulate one another without changing the asymptotic order of growth of program runtime. Each of these languages also presents similar foreign interfaces for interoperation with external software. They have all been used for data analysis and for system building, deployed in fields ranging from robotics to statistics. They have also all been used for machine intelligence research grounded in probabilistic modeling and inference. One measure of their flexibility and interchangeability is provided by their use in probabilistic programming research. Each of these languages has been used to implement expressive probabilistic programming languages. For example, multiple versions of Church and Venture have been written in Python, Lisp and Java, and BLOG has been implemented in both Java and Python.

We do not know if it will be possible to attain the flexibility, extensibility and efficiency of Lisp, Java or Python in any single probabilistic programming language, especially due to the complexity of inference. Thus far, most probabilistic programming languages have opted for a narrower scope. Most are not Turing-complete or higher-order, only support a subset of standard approximate inference strategies, and have no notion of inference programming. Translating between sufficiently expressive probabilistic programming languages without distorting the asymptotic scaling of forward simulation will be difficult, especially given the lack of a standard cost model. Providing faithful translations that do not distort the asymptotic scaling of inference — in runtime, in accuracy, or both — will be harder still.

Despite these challenges, it may be possible to develop probabilistic languages that can be used to specify and solve probabilistic modeling and approximate inference problems from many fields.

Ideally such languages would be able to cover the modeling idioms and inference strategies from fields such as robotics, statistics, and machine learning, while also meeting key representational needs in cognitive science and artificial intelligence. We hope Venture, and the principles behind its design and implementation, represent a significant step towards the development of a probabilistic programming platform that is both computationally universal and suitable in practice for general-purpose use.

## References


Harold Abelson and Gerald Jay Sussman. Structure and Interpretation of Computer Programs. 1983.

Christophe Andrieu, Nando De Freitas, Arnaud Doucet, and Michael I Jordan. An introduction to MCMC for machine learning. *Machine learning*, 50(1-2):5–43, 2003.

Christophe Andrieu, Arnaud Doucet, and Roman Holenstein. Particle Markov chain Monte carlo methods. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, 72(3):269–342, 2010.

Nimar S Arora, Stuart J Russell, Paul Kidwell, and Erik B Sudderth. Global seismic monitoring as probabilistic inference. In *NIPS*, pages 73–81, 2010.

Nimar S Arora, Rodrigo de Salvo Braz, Erik B Sudderth, and Stuart Russell. Gibbs Sampling in Open-Universe Stochastic Languages. *arXiv preprint arXiv:1203.3464*, 2012.

James K Baker. Trainable grammars for speech recognition. *The Journal of the Acoustical Society of America*, 65(S1): S132–S132, 1979.

C M Bishop. *Pattern Recognition and Machine Learning*. Springer, 2006.

D. M. Blei, A. Y. Ng, and M. I. Jordan. Latent Dirichlet Allocation. *Journal of Machine Learning Research*, 3:993–1022, 2003.

Keith Bonawitz and Vikash Mansinghka. Monte: An interactive system for massively parallel probabilistic programming, 2009.

Keith A Bonawitz. Composable Probabilistic Inference with Blaise. 2008.

Katalin Csilléry, Michael GB Blum, Oscar E Gaggiotti, and Olivier François. Approximate Bayesian computation (ABC) in practice. *Trends in ecology & evolution*, 25(7):410–418, 2010.

Luc De Raedt and Kristian Kersting. *Probabilistic inductive logic programming*. Springer, 2008.

Pierre Del Moral, Arnaud Doucet, and Ajay Jasra. Sequential Monte Carlo Samplers. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, 68(3):411–436, 2006.

A Doucet, N de Freitas, and N Gordon, editors. *Sequential Monte Carlo Methods in Practice*. Springer-Verlag, 2001.

Robin D Dowell and Sean R Eddy. Evaluation of several lightweight stochastic context-free grammars for RNA secondary structure prediction. *BMC bioinformatics*, 5(1):71, 2004.

David Duvenaud, James Robert Lloyd, Roger Grosse, Joshua B Tenenbaum, and Zoubin Ghahramani. Structure discovery in nonparametric regression through compositional kernel search. *arXiv preprint arXiv:1302.4922*, 2013.

Cameron Freer, Vikash Mansinghka, and Daniel Roy. When are probabilistic programs probably computationally tractable? *NIPS Workshop on Advanced Monte Carlo Methods with Applications*, 2010.

Brendan J Frey. *Bayesian networks for pattern classification, data compression, and channel coding*. PhD thesis, Citeseer, 1997.

N. Friedman and D. Koller. Being Bayesian about network structure: A Bayesian approach to structure discovery. *Machine Learning*, 50:95–126, 2003.

Nir Friedman, Michal Linial, Iftach Nachman, and Dana Pe'er. Using bayesian networks to analyze expression data. *Journal of computational biology*, 7(3-4):601–620, 2000.

A. Gelman, J. B. Carlin, H. S. Stern, and D. B. Rubin. *Bayesian data analysis*. Chapman & Hall, New York, 1995.

Noah Goodman and Joshua Tenenbaum. Probabilistic Models of Cognition, 2013.

Noah D Goodman*, Vikash K Mansinghka*, Daniel M Roy, Keith Bonowitz, and Joshua B Tenenbaum. Church: a language for generative models. In *Uncertainty in Artificial Intelligence*, May 2008.

Peter J Green, Nils Lid Hjort, and Sylvia Richardson. *Highly Structured Stochastic Systems*. Oxford University Press, 2003.

Thomas L Griffiths and Zoubin Ghahramani. Infinite latent feature models and the Indian buffet process. In *NIPS*, volume 18, pages 475–482, 2005.

Roger Grosse, Ruslan R Salakhutdinov, William T Freeman, and Joshua B Tenenbaum. Exploiting compositionality to explore a large space of model structures. *arXiv preprint arXiv:1210.4856*, 2012.

David Heckerman. *A tutorial on learning with Bayesian networks*. Springer, 1998.

Frederick Jelinek, John D Lafferty, and Robert L Mercer. *Basic methods of probabilistic context free grammars*. Springer, 1992.

Mark Johnson, Thomas L Griffiths, and Sharon Goldwater. Adaptor grammars: A framework for specifying compositional nonparametric Bayesian models. *Advances in Neural Information Processing Systems*, 19:641, 2007.

Daphne Koller, David McAllester, and Avi Pfeffer. Effective Bayesian inference for stochastic programs. In *AAAI/IAAI*, pages 740–747, 1997.

Michail G Lagoudakis and Michael L Littman. Learning to select branching rules in the DPLL procedure for satisfiability. *Electronic Notes in Discrete Mathematics*, 9:344–359, 2001.

P. Liang, M. I. Jordan, and D. Klein. Learning programs: A hierarchical Bayesian approach. In *International Conference on Machine Learning (ICML)*, pages 639–646, 2010.

D. J. Lunn, A. Thomas, N. Best, and D. Spiegelhalter. WinBUGS - A Bayesian modelling framework: Concepts, structure, and extensibility. *Statistics and Computing*, 10(4):325–337, October 2000.

Christopher D Manning and Hinrich Schütze. *Foundations of statistical natural language processing*. MIT press, 1999.

Vikash Mansinghka. *Natively Probabilistic Computation*. PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, June 2009.

Vikash Mansinghka*, Tejas D Kulkarni*, Yura N Perov, and Josh Tenenbaum. Approximate bayesian image interpretation using generative probabilistic graphics programs. In *Advances in Neural Information Processing Systems*, pages 1520–1528, 2013.

VK Mansinghka, C Kemp, TL Griffiths, and JB Tenenbaum. Structured priors for structure learning. In *In Proceedings of the 22nd Conference on Uncertainty in Artificial Intelligence (UAI*, 2006.

Paul Marjoram, John Molitor, Vincent Plagnol, and Simon Tavaré. Markov chain Monte Carlo without likelihoods. *Proceedings of the National Academy of Sciences*, 100(26):15324–15328, 2003.

Andrew McCallum, Karl Schultz, and Sameer Singh. FACTORIE: Probabilistic programming via imperatively defined factor graphs. In *Advances in Neural Information Processing Systems (NIPS)*, 2009.

Brian Milch, Bhaskara Marthi, David Sontag, Stuart Russell, Daniel L Ong, and Andrey Kolobov. Approximate inference for infinite contingent Bayesian networks. In *Proc. 10th AISTATS*, pages 238–245, 2005.

Brian Milch, Bhaskara Marthi, Stuart Russell, David Sontag, Daniel L Ong, and Andrey Kolobov. BLOG: Probabilistic models with unknown objects. *Introduction to statistical relational learning*, page 373, 2007.

TP Minka, JM Winn, JP Guiver, and DA Knowles. Infer.NET 2.4. Microsoft Research Cambridge. *See http://research.microsoft.com/infernet*, 2010.

Radford M. Neal. Markov Chain Sampling Methods for Dirichlet Process Mixture Models. *Technical Report*, September 1998.

David J Nott and Peter J Green. Bayesian variable selection and the Swendsen-Wang algorithm. *Journal of computational and Graphical Statistics*, 13(1), 2004.

Songhwai Oh, Stuart Russell, and Shankar Sastry. Markov chain Monte Carlo data association for multi-target tracking. *Automatic Control, IEEE Transactions on*, 54(3):481–497, 2009.

Peter Orbanz and Daniel M Roy. Bayesian Models of Graphs, Arrays and Other Exchangeable Random Structures.

Hanna Pasula, Bhaskara Marthi, Brian Milch, Stuart Russell, and Ilya Shpitser. Identity uncertainty and citation matching. In *Advances in Neural Information Processing Systems*, pages 1401–1408, 2002.

Yura Perov and Vikash Mansinghka. Exploiting conditional independence for efficient, automatic multicore inference for Church, 2012.

Avi Pfeffer. IBAL: A Probabilistic Rational Programming Language. In *Proceedings of the International Joint Conference on Artificial Intelligence*, pages 733–740, 2001.

Avi Pfeffer. Figaro: An object-oriented probabilistic programming language. *Charles River Analytics Technical Report*, page 137, 2009.

David Poole. The independent choice logic for modelling multiple agents under uncertainty. *Artificial intelligence*, 94 (1):7–56, 1997.

Carl Edward Rasmussen. The infinite gaussian mixture model. In *NIPS*, volume 12, pages 554–560, 1999.

CE Rasmussen and CKI Williams. Gaussian processes for machine learning. *Adaptive computation and machine learning*, 2006.

John C Reynolds. Definitional interpreters for higher-order programming languages. In *Proceedings of the ACM annual conference*, pages 717–740. ACM, 1972.

Matthew Richardson and Pedro Domingos. Markov logic networks. *Machine Learning*, 62(1-2):107–136, 2006.

D Roy, V Mansinghka, N Goodman, and J Tenenbaum. A stochastic programming perspective on nonparametric Bayes. In *ICML Nonparametric Bayes Workshop*, volume 22, page 26, 2008.

Stuart J. Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, Englewood Cliffs, NJ, 2nd edition, 2002.

Taisuke Sato and Yoshitaka Kameya. Prism: a language for symbolic-statistical modeling. In *IJCAI*, volume 97, pages 1330–1339. Citeseer, 1997.

Stan Development Team. *Stan Modeling Language Users Guide and Reference Manual, Version 2.0*, 2013. URL `http://mc-stan.org/`.

Robert H Swendsen and Jian-Sheng Wang. Replica monte carlo simulation of spin glasses. *Physical Review Letters*, 57 (21):2607–2609, 1986.

Joshua B Tenenbaum, Charles Kemp, Thomas L Griffiths, and Noah D Goodman. How to grow a mind: structure, statistics and abstraction. *science*, 331(6022):1279–1285, 2011.

Sebastian Thrun, Wolfram Burgard, Dieter Fox, et al. *Probabilistic Robotics*. MIT Press, 2005.

Tina Toni, David Welch, Natalja Strelkowa, Andreas Ipsen, and Michael PH Stumpf. Approximate Bayesian computation scheme for parameter inference and model selection in dynamical systems. *Journal of the Royal Society Interface*, 6 (31):187–202, 2009.

David A Turner. Miranda: A non-strict functional language with polymorphic types. In *Functional Programming Languages and Computer Architecture*, pages 1–16. Springer, 1985.

N. Whiteley, A. Lee, and K. Heine. On the role of interaction in sequential Monte Carlo algorithms. *ArXiv e-prints*, September 2013.

David Wingate and Theo Weber. Automated variational inference in probabilistic programming, 2013.

David Wingate, Noah Goodman, Andreas Stuhlmueller, and Jeffrey M Siskind. Nonstandard interpretations of probabilistic programs for efficient inference. In *Advances in Neural Information Processing Systems*, pages 1152–1160, 2011a.

David Wingate, Andreas Stuhlmueller, and Noah D Goodman. Lightweight implementations of probabilistic programming languages via transformational compilation. In *Proceedings of the International Conference on Artificial Intelligence and Statistics*, 2011b.

Frank Wood, Jan-Willem van de Meent, and Vikash Mansinghka. A new approach to probabilistic programming inference. In *Artificial Intelligence and Statistics*, 2014.

Jeff Wu. Reduced traces and JITing in Church, 2013.

Lin Xu, Frank Hutter, Holger H Hoos, and Kevin Leyton-Brown. SATzilla: Portfolio-based Algorithm Selection for SAT. *J. Artif. Intell. Res.(JAIR)*, 32:565–606, 2008.

Lingfeng Yang, Yi-Ying Yeh, Noah Goodman, and Pat Hanrahan. Just-in-time Compilation of MCMC for Probabilistic Programs, 2012.

Jing Yu, V Anne Smith, Paul P Wang, Alexander J Hartemink, and Erich D Jarvis. Advances to Bayesian network inference for generating causal networks from observational biological data. *Bioinformatics*, 20(18):3594–3603, 2004.