Venture: an extensible platform for probabilistic meta-programming

by

Anthony Lu

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2016

© Massachusetts Institute of Technology 2016. All rights reserved.

Author . . . .
Department of Electrical Engineering and Computer Science
August 16, 2016

Certified by . . . .
Vikash K. Mansinghka
Research Scientist
Thesis Supervisor

Accepted by . . . .
Christopher Terman
Chairman, Masters of Engineering Thesis Committee

Venture: an extensible platform for probabilistic meta-programming

by

Anthony Lu

Submitted to the Department of Electrical Engineering and Computer Science
on August 16, 2016, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

This thesis describes Venture, an extensible platform for probabilistic meta-programming. In Venture, probabilistic generative models, probability density functions, and probabilistic inference algorithms are all first-class objects. Any Venture program that makes random choices can be treated as a probabilistic model defined over the space of possible executions of the program. Such *probabilistic model programs* can also be run while recording the random choices that they make. Modeling and inference in Venture involves two additional classes of probabilistic programs. The first, *probability density meta-programs* partially describe the input-output behavior of probabilistic model programs. The second, *stochastic inference meta-programs* identify probable executions of model programs given stochastic constraints, and typically use density meta-programs as guides. Unlike other probabilistic programming platforms, Venture allows model programs, density meta-programs, and inference meta-programs to be written as user-space code in a single probabilistic programming language. Venture is essentially a Lisp-like higher-order language augmented with two novel abstractions: (i) *probabilistic execution traces*, a first-class object that represents the sequence of random choices that a probabilistic program makes, and (ii) *stochastic procedures*, which encapsulate the probabilistic programs and meta-programs needed to allow simple probability distributions, user-space VentureScript programs, and foreign probabilistic programs to be treated uniformly as components of probabilistic computations. Venture also provides runtime support for stochastic regeneration of execution trace fragments that makes use of the programs and meta-programs of all stochastic procedures invoked during the execution of the original traced program. This thesis describes a new prototype implementation of Venture incorporating these ideas and illustrates the flexibility of Venture by giving concise user-space implementations of primitives and inference strategies that have been built in to Church as well as other probabilistic languages.

Acknowledgments

This thesis owes its existence to my advisor Vikash Mansinghka, whose vision, direction, and mentorship were integral to its success. I also thank my colleagues in the Probabilistic Computing Group, who contributed in many forms. I would especially like to call out Alexey Radul, with whom I have had many insightful discussions and who has also offered a tremendous amount of help on the technical side, starting from when I was first becoming familiar with the group's software; Gregory Marton, who provided vital planning help and emotional support as I was finishing; and Marco Cusumano-Towner, Leo Casarsa, Feras Saad, and Ulrich Schaechtle for providing very helpful feedback throughout the project.

I also thank my family, as well as my friends at MIT, for all their support.

Finally, I am thankful to DARPA for funding this research and my appointment as a research assistant through the Probabilistic Programming for Advancing Machine Learning program.

Contents

1 Introduction **12**

2 Meta-programs for probability densities **14**

2.1 Simulators and densities for standard distributions . . . 14

2.2 Bayes' Rule as a probabilistic meta-program . . . 18

2.3 Optimized marginal densities for hidden Markov models . . . 19

3 Probabilistic execution traces **22**

3.1 Traces as elements of a probability space . . . 26

3.2 An interface to probabilistic execution traces . . . 27

3.3 Stochastic regeneration of trace fragments . . . 30

4 Meta-programs for stochastic inference **31**

4.1 Exact stochastic inference via rejection sampling . . . 33

4.2 Approximate stochastic inference via Metropolis-Hastings . . . 36

4.3 Custom model-specific inference meta-programs . . . 39

4.3.1 Custom Metropolis-Hastings samplers . . . 39

4.4 Scaling of stochastic regeneration for approximate inference . . . 40

5 Stochastic procedures **44**

5.1 Generic stochastic procedures . . . 45

5.2 "Simple" stochastic procedures . . . 48

5.3 "Tail-assessable" stochastic procedures . . . 51

5.4 Tradeoffs between different representations of the Beta-Bernoulli process . . . 52

5.5 Memoization as a user-space stochastic procedure . . . 54

6 Discussion **59**

List of Figures

2.1 A sampler for the standard normal distribution using the Box-Muller transform, linked with an assessor for its log density. 15

2.2 A sampler for the gamma distribution using an explicit rejection loop, together with an assessor for its log density. 17

2.3 A meta-program that computes a posterior density via Bayes' rule, using a Monte Carlo estimator for the marginal density. 19

2.4 Using a Monte Carlo approximation of Bayes' rule to estimate $p(x|y = 4.0)$ in the normal-normal model. 20

2.5 A collection of meta-programs for simulating from a hidden Markov model and assessing joint, marginal, and posterior probability densities over the hidden states and observations. 21

3.1 Traced execution of a model program, followed by inference that modifies the execution trace. 22

3.2 An example tricky-coin model program and a printout of the flat trace data structure for the example program. The `/2/0/1` expressions are addresses; see main text. 24

3.3 The dependency graph structure for the example tricky-coin program in Figure 3.2. Each node is labeled with its address (on the top) and its value in that execution (on the bottom). 25

3.4 The modeling instructions `assume`, `observe`, and `generate` are implemented as macros which expand to applications of the corresponding procedures defined above. `observe` is also defined in terms of a utility procedure that modifies the trace so that it satisfies a given constraint. This procedure relies on primitives for *stochastic regeneration*, described in Section 3.3, to modify the trace and propagate the consequences. 29

3.5 Extract takes a trace and a subproblem, extracts from the trace the values targeted by the subproblem, and returns a log weight corresponding to the ratio between the target and proposal probabilities of the trace with the original values. 32

3.6 Regen takes a trace and an extracted subproblem, proposes new values for the target variables using fresh randomness, and returns a log weight for the ratio between target and proposal probabilities of the trace with the new values. 32

3.7 Restore takes a trace, an extracted subproblem, and the trace fragment returned by extract, and restores the state of the trace from the contents of the trace fragment. 33

3.8 Depiction of stochastic regeneration on the example tricky coin model from Figure 3.2. 34

4.1 Top: A VentureScript program that defines a simple normal-normal model and obtains a posterior sample of x given an observed y using rejection sampling. Bottom: Histograms of samples obtained from 500 independent runs of each procedure, comparing the distribution of x under the prior and the posterior distribution induced by rejection sampling. The true density curves are shown in yellow: the prior $N(0,1)$, and the posterior $N(2, \sqrt{\frac{1}{2}})$. 35

4.2 The rejection sampling meta-program, defined in terms of inference subproblem operations `extract`, `regen`, and `weight_bound`, and the utility `check_consistent` given in Figure 4.3. . . 37

4.3 Definition of a utility procedure to check whether a trace is consistent with all observations. . 37

4.4 Top: A VentureScript program that uses resimulation MH on a simple normal-normal model to obtain an approximate posterior sample of x. Bottom: Histograms of samples obtained by running the Markov chain for different numbers of steps. As the number of steps increases, the distribution approaches the target posterior (yellow). 38

4.5 The resimulation MH meta-program, defined in terms of the inference subproblem operations `extract`, `regen`, and `restore`. 39

4.6 Top: A VentureScript program that uses a custom user-defined inference procedure (shown in Figure 4.7) for random-walk MH with Gaussian drift proposals. Bottom: A plot comparing sample trajectories of the Markov chain obtained from this inference program with sample trajectories from the program in Figure 4.4 which uses independent resimulation proposals. The Gaussian drift proposal is accepted more frequently, suggesting that the custom sampler is more efficiently exploring the space of traces. 41

4.7 A custom meta-program implementing random-walk MH using Gaussian drift proposals, specialized to the normal-normal model from Figure 4.6. 42

4.8 Comparison of inference speed on an HMM model program, as a function of the number of timesteps in the HMM. The model program is shown on the left, as well as a version of the program where the memoized loop is unrolled, showing how the number of steps in the HMM affects the program length. Inference is performed on single-site subproblems, which propose a new value for one element of the state sequence conditioned on all other variables. The plot on the right shows timing for the subproblem construction step and the resimulation step separately. The timing is performed on the unrolled program. 43

5.1 Top: Definition of a beta sampler in terms of a gamma sampler, as a "bare" compound procedure. Bottom left: Definition of a beta sampler using `make_elementary_sp`, which creates a stochastic procedure from a simulator and a density assessor (and possibly additional metadata), implementing the rest of the generic stochastic procedure interface in terms of those procedures. Bottom right: An equivalent procedure constructed using the full interface `make_sp`, showing how `make_elementary_sp` expands into `make_sp`. 49

5.2 Left: Naive definition of the beta-Bernoulli procedure as a bare compound, which draws $\theta \sim \mathrm{Beta}(a, b)$ and returns a closure that flips a coin with weight θ. As with `my_beta_1` from Figure 5.1, this closure has no density meta-program so its output is not constrainable. Right: An equivalent definition using `proc`, a meta-program which takes a *tail-assessable* procedure and generates a constraint kernel for it, which returns the density of the `flip` procedure as its weight. Bottom: Expanded versions of the resulting SPs defined using the full `make_sp` interface. 53

5.3 Top: An example application of a beta-Bernoulli model. A coin with unknown weight is flipped five times. The flips are observed and incorporated one by one and the resulting log likelihood increments are retained. Bottom: Plot of runtime versus log likelihood weight, varying the implementation of the beta-Bernoulli procedure and the amount of inference performed between each observation. Note that the collapsed implementation obtains the true marginal likelihood of the data under the model, while the others produce noisy approximations that become more accurate with better inference. 55

5.4 An implementation of the collapsed beta-Bernoulli model as a stateful procedure that maintains sufficient statistics N, the number of times the procedure was applied, and K, the number of times that the procedure returned true. The implementation uses a generic `counter` data structure for its mutable state, which exposes read, increment, and decrement operations. . . 56

5.5 Supporting definitions for the collapsed beta-Bernoulli procedure. Top: The counter can be emulated in VentureScript using a trace, with increment and decrement operations implemented using inference programming primitives. Bottom: Alternatively, the counter operations can be implemented as methods of a custom Python class and bound into Venture as foreign primitives. 57

5.6 An uncollapsed implementation of the beta-Bernoulli model as a stateful procedure with a conjugate Gibbs kernel. Like the collapsed version from Figure 5.4, it maintains sufficient statistics of its applications, which are used here to sample θ exactly from its conjugate posterior. This Gibbs kernel is provided as a custom meta-program of the SP, which can be invoked as part of an inference program. 58

5.7 Implementation of memoization as a user-space procedure, using an external mutable counter to implement reference counting, and `eval_request` to make traced calls to the original unmemoized SP `f`. 60
5.8 An implementation of an HMM with binary states and observations, using mem to memoize the state sequence. 61

List of Tables

3.1 Index of trace operations, with asymptotic runtime. $|e|$ is the number of sub-operations involved in running the given program e; $|\mathcal{T}|$ is the size of the probabilistic execution trace; $|\mathcal{S}|$ is the size of the subproblem, which is determined by the number of downstream random choices that depend on the proposed value. 27

3.2 Asymptotic runtime of each trace operation in terms of N, the number of nodes in the trace (which approximately corresponds to the length of the program execution); n, the number of nodes involved in the subproblem; and e, the number of dependency edges between nodes involved in the subproblem. 32

5.1 Comparison of the beta-Bernoulli variants. The "runtime" column describes the runtime complexity of doing one round of inference on θ (if applicable) and then simulating or incorporating a new observation, where N is the total number of observations that have been incorporated. 54

1 Introduction

Probabilistic programming is an emerging field that aims to formalize and simplify probabilistic modeling and inference using ideas from programming languages and system software. Several early probabilistic programming languages [5] [9] [13] [11] [3] focused on probabilistic generative modeling, in which programs that make random choices are used to define probabilistic models. In these languages, and also in most languages that have subsequently been developed, probabilistic inference is done by built-in mechanisms in the language that cannot be replaced or extended by user-space programs.

This thesis describes Venture, an extensible platform for probabilistic meta-programming. In Venture, probabilistic generative models, probability density functions, and probabilistic inference algorithms are all first-class objects. Any Venture program that makes random choices can be treated as a probabilistic model defined over the space of possible executions of the program. Such "probabilistic model programs" can also be run while recording the random choices that they make. This kind of "traced execution" is exposed to end users as an ordinary primitive procedure, and is central to Monte Carlo techniques for probabilistic inference, but is also useful for inspection and debugging.

This thesis uses the term "meta-program" to refer to programs that describe, inspect, create, and/or transform the text or executions of other programs. In Venture, we have thus far seen the need for two types of meta-programs:

1. *Probability density meta-programs.* These partially describe the input-output behavior of probabilistic model programs. Densities can sometimes be given by simple analytical formulas, but sometimes themselves consist of sophisticated algorithms. For example, the marginal probability density on the sequence of output symbols from a hidden Markov model [12] can be calculated via a simple dynamic program; in Venture, this program is a valid meta-program that can be associated with a program that samples latent sequences along with the observations that they generate. Bayes' Rule can be viewed as a density meta-program for evaluating posterior densities given two density meta-programs—one for the prior, and one for the likelihood—as well as the (data-dependent) value of the marginal probability of the data under the prior and likelihood densities.

2. *Stochastic inference meta-programs.* These meta-programs find probable executions of model programs given stochastic constraints, and typically use density meta-programs as guides. This thesis includes a rejection sampling inference meta-program that can execute arbitrary model programs subject to "sufficiently stochastic" constraints, i.e., constraints on random choices represented by model programs that have a known marginal probability density with a known bound (as a function of the inputs, for fixed output). This thesis also presents inference meta-programs built using the Metropolis-Hastings recipe, including specialized samplers for specific models as well as a general-purpose approximate

inference algorithm that applies to the same class of inference problems.

Unlike other probabilistic programming platforms, Venture allows model programs, density meta-programs, and inference meta-programs to be written as user-space code in a single probabilistic programming language. In this way, Venture aims to be sufficiently expressive and extensible for general-purpose use. Although Venture has a built-in standard library for modeling and inference, this library is not intended to deliver high performance. Instead, it is designed to facilitate rapid prototyping and run-time reflection for testing and debugging. Once a probabilistic computation has been debugged, it can be incrementally optimized by replacing key components with high-performance native code. This goal was only partially realized in earlier versions of Venture [6].

Venture is essentially a Lisp-like higher-order language augmented with two novel abstractions:

1. *Probabilistic execution traces (PETs)*, sometimes also abbreviated simply as "traces". PETs are a first-class object that represents the sequence of random choices that a probabilistic program makes. PETs serve as the only native form of mutable storage in Venture, and map dynamic "addresses" assigned over the course of program execution to the manifest values taken by the program at those addresses. Venture supports "flat traces" that simply record a mapping from addresses to values, following [14], as well as "dependency graph traces" that can be viewed as extensions of directed graphical models [10] [6] [2]. These traces can be accessed by a uniform interface for (re)generating traces for program fragments that facilitates the development of inference meta-programs. Flat traces and dependency graph traces exhibit different asymptotic scaling behavior and also support different constant-factor optimizations and runtime, making them suitable for different classes of inference strategies.

2. *Stochastic procedures (SPs).* SPs are used to encapsulate simple probability distributions, as well as user-space VentureScript programs and foreign probabilistic objects. An SP consists of a linked collection of programs and meta-programs that collectively describe aspects of a probabilistic program that are important for its use in modeling and inference. For example, SPs are the primary means for linking model programs with their associated density and inference meta-programs. SPs are designed to allow simple probability distributions, user-space VentureScript, and foreign probabilistic programs to be treated uniformly as building blocks of complex probabilistic computations.

This thesis describes a new prototype implementation of Venture incorporating these ideas and illustrates the flexibility of Venture by giving concise user-space implementations of primitives and inference strategies that have been built in to Church [3] as well as other probabilistic languages.

2 Meta-programs for probability densities

Mathematical treatments of probabilistic modeling and inference use multiple representations of probability distributions. The most common choice is to use probability density functions: mathematical functions that calculate the probability density of particular output values (perhaps given input parameters).

In programming terms, density functions can be seen as *incomplete declarative specifications* of the behavior of the code that simulates from a generative process. In Venture, probabilistic programs can be associated with probability density meta-programs that characterize their input-output density. Such an association makes a mathematical assertion about the program's behavior, which subsequent meta-programs such as stochastic inference meta-programs can rely on.

Definition 1 *Let* `s` *be a random procedure from inputs* $x \in \mathcal{X}$ *to random output* $y \in \mathcal{Y}$*. Any such procedure gives rise to a function from inputs* x *to measures on the output space,* $\mu_{(\mathtt{s},x)}$*:*

$$\mu_{(\mathtt{s},x)}(A) = \mathbf{Pr}(\mathtt{s}(x) \in A) \qquad \forall A \subset \mathcal{Y}$$

A density function `p`*, with respect to a base measure* μ_b*, also gives rise to a function from inputs to measures:*

$$\mu_{(\mathtt{p},x)}(A) = \int_{y \in A} \mathtt{p}(y; x) d\mu_b \qquad \forall A \subset \mathcal{Y}$$

A procedure `p` *is a density for* `s` *(with respect to a base measure* μ_b*) if the simulator* `s` *and density* `p` *give rise to the same measure for all input arguments:*

$$\mu_{(\mathtt{s},x)}(A) = \mu_{(\mathtt{p},x)}(A) \qquad \forall x \in \mathcal{X}, A \subset \mathcal{Y}$$

The remainder of this section shows how to associate probabilistic programs with probability density meta-programs in Venture, so that other meta-programs can make use of this information.

2.1 Simulators and densities for standard distributions

Figure 2.1 shows a Venture program representing a standard normal distribution, with a procedure to simulate from the distribution and a procedure to compute its (log) probability density function.

The simulation procedure uses the well known Box-Muller transform, which produces a sample X from

```
// a probabilistic model program
define simulate_std_normal = () ~> {
  u1 = uniform_continuous(0, 1);
  u2 = uniform_continuous(0, 1);
  sqrt(-2 * log(u1)) * cos(2 * 3.14159265 * u2)
};

// a probability density meta-program for the program simulate_std_normal
define assess_std_normal = (x) -> {
  -0.5 * log(2 * 3.14159265) - 0.5 * x * x
};

// a package containing the probabilistic model program and associated
// meta-program(s)
define std_normal = make_elementary_sp(() -> {
  let simulate_std_normal = ${simulate_std_normal};
  let assess_std_normal = ${assess_std_normal};
  dict(
    ['simulate', () -> {
      return (simulate_std_normal())
    }],
    ['log_density', (x) -> {
      return (assess_std_normal(x))
    }])
});
```

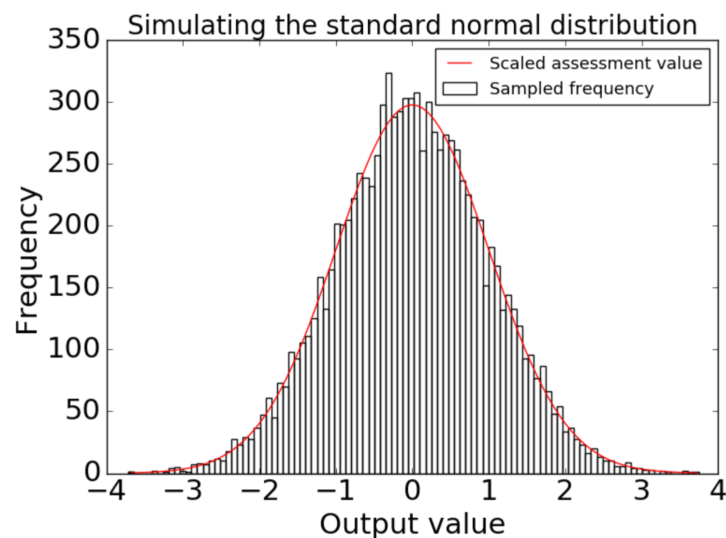

Figure 2.1: A sampler for the standard normal distribution using the Box-Muller transform, linked with an assessor for its log density.

a standard normal distribution:

$$\begin{aligned}
U_1 &\sim \textbf{uniform}(0,1) \\
U_2 &\sim \textbf{uniform}(0,1) \\
X &= \sqrt{-2\log(U_1)}\cos(2\pi U_2)
\end{aligned}$$

This is based on viewing a bivariate unit normal distribution in polar coordinates. Let $R^2 = -2\log U_1$ and $\theta = 2\pi U_2$ be the polar coordinates of a point (x, y) in $\mathbf{R}^2$. Then $p(x, y)$ is given by

$$\begin{aligned}
p(x,y) &= p(R^2)p(\theta)\left|\frac{\partial(X,Y)}{\partial(R^2,\theta)}\right| \\
&= \left(\frac{1}{2}e^{-R^2/2}\right)\left(\frac{1}{2\pi}\right)(2) \\
&= \frac{1}{2\pi}e^{-x^2/2}e^{-y^2/2} \\
&= \left(\frac{1}{\sqrt{2\pi}}e^{-x^2/2}\right)\left(\frac{1}{\sqrt{2\pi}}e^{-y^2/2}\right)
\end{aligned}$$

and in particular the marginal density of $X = \sqrt{R^2}\cos(\theta)$ is the standard normal density

$$p(x) = \frac{1}{\sqrt{2\pi}}e^{-x^2/2}$$

which agrees with the result computed by the density procedure.

As a second example, Figure 2.2 shows a Venture program representing a Gamma distribution, with simulation and density procedures. Here the simulator is a more complex program, implementing a rejection sampling algorithm due to [8] for generating a Gamma random variate y with shape parameter α:

$$\begin{aligned}
d &= \alpha - 1/3 \\
c &= 1/\sqrt{9d} \\
x &\sim \textbf{normal}(0,1) \\
u &\sim \textbf{uniform}(0,1) \\
v &= (1+cx)^3 \\
\textbf{reject unless}\ \log(U) &< \frac{1}{2}x^2 + d - dv + d\log(v) \\
y &= dv
\end{aligned}$$

```
define simulate_std_gamma = (alpha) ~> {
  d = alpha - (1.0/3);
  c = 1.0 / sqrt(9 * d);
  x ~ normal(0, 1);
  v = pow(1 + c * x, 3);
  if (v <= 0) {
    simulate_std_gamma(alpha)
  } else {
    log_bound = 0.5 * x * x + d * (1 - v + log(v));
    u ~ uniform_continuous(0, 1);
    if (log(u) >= log_bound) {
      simulate_std_gamma(alpha)
    } else {
      d * v
    }
  }
};

define assess_std_gamma = (x, alpha) -> {
  (alpha - 1) * log(x) - x - log_gamma_function(alpha)
};

define std_gamma = make_elementary_sp(() -> {
  let simulate_std_gamma = ${simulate_std_gamma};
  let assess_std_gamma = ${assess_std_gamma};
  dict(
    ['simulate', (alpha) -> {
      return (simulate_std_gamma(alpha))
    }],
    ['log_density', (x, alpha) -> {
      return (assess_std_gamma(alpha))
    }])
});
```

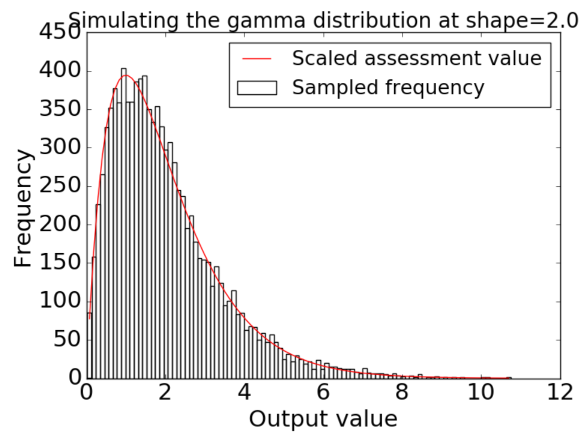

Figure 2.2: A sampler for the gamma distribution using an explicit rejection loop, together with an assessor for its log density.

The rejection step produces an x with density

$$p(x) \propto e^{d\log(v)-dv}, \qquad x \in (-1/c, \infty)$$

which has the property that $y = dv = d(1+cx)^3$ has the standard gamma density

$$p(y) \propto y^{\alpha-1}e^{-y}$$

which agrees with the result computed by the density procedure.

In these example programs, we have seen the primitive `make_elementary_sp` used to associate a simulation procedure with a density. In later sections, we will see how this is useful for interoperating with Venture's built-in tracing and inference facilities. However, it is also possible to write user programs that directly make use of simulators with densities.

2.2 Bayes' Rule as a probabilistic meta-program

Some forms of Bayes' Rule can be understood as probabilistic density meta-programs. Recall that Bayes' rule gives a formula for the conditional probability density

$$p(x|y) = \frac{p(x)p(y|x)}{p(y)}$$

in terms of the prior $p(x)$, the likelihood $p(y|x)$, and the marginal probability of the data, $p(y)$. Often, $p(y)$ is not given in closed form, but can be approximated with Monte Carlo integration over x:

$$\begin{aligned} p(y) &= \int p(x)p(y|x)dx \\ &\approx \frac{1}{N}\sum_{i=1}^{N} p(y|x_i), \qquad x_i \sim p_x \end{aligned}$$

Figure 2.3 shows a Venture program implementing the Monte Carlo approach. The prior distribution is represented via a stochastic procedure that generates samples and has an associated density; the Bayes' rule meta-program uses samples from this procedure to form the Monte-Carlo estimate. The likelihood distribution is represented by another stochastic procedure that is only required to have a probability density.

To illustrate, consider the following simple normal-normal model:

$$x \sim \textbf{normal}(0, 1)$$

$$y \sim \textbf{normal}(x, 1)$$

```
// compute p(y) = integral of p(x) p(y | x)
define marginal_density_estimate = (prior, likelihood, n_samples) -> {
  (y) -> {
    estimates <- for_each(arange(n_samples), (i) -> {
      x <~ invoke_metaprogram_of_sp(prior, 'simulate', []);
      logp <- invoke_metaprogram_of_sp(likelihood, 'log_density', [y, x]);
      return (logp)
    });
    return (logsumexp(estimates) - log(n_samples))
  }
};

// p(x|y) = p(x) p(y|x) / p(y)
define bayes_rule_estimate = (prior, likelihood, y, n_samples) -> {
  (x) -> {
    logp_x <- invoke_metaprogram_of_sp(prior, 'log_density', [x]);
    logp_y_given_x <- invoke_metaprogram_of_sp(likelihood, 'log_density', [y, x]);
    logp_y <- marginal_density_estimate(prior, likelihood, n_samples)(y);
    return (logp_x + logp_y_given_x - logp_y)
  }
};
```

Figure 2.3: A meta-program that computes a posterior density via Bayes' rule, using a Monte Carlo estimator for the marginal density.

where we observe the value of y and are interested in the posterior distribution $p(x|y)$. We will use this as a running example to demonstrate various inference meta-programs; this problem, though simple, serves as a useful minimal test case to check how well various inference strategies work.

Figure 2.4 shows the Bayes rule meta-program applied to this problem with an observed value $y = 4.0$, estimating the density using $N = 500$ samples.

2.3 Optimized marginal densities for hidden Markov models

For restricted classes of probability models, it is often possible to give optimized algorithms for evaluating marginal probability densities. Some of these, such as the forward-backward algorithm for Hidden Markov Models [12], were originally viewed as landmark results. Venture makes it possible to represent these algorithmic innovations as ordinary user-space meta-programs and attach them to stochastic procedures. This way, the optimized algorithms can be used whenever the probability density of the stochastic procedure is requested.

Figure 2.5 shows a collection of simulator and density procedures describing the HMM model, including a procedure for simulating sequences of hidden states and observations; a procedure for assessing the joint probability of the hidden states and observations; a procedure for computing the exact marginal probability of the observations, using the forward algorithm; and a procedure that uses the two density procedures to directly compute posterior probabilities of the states given the observations.

```
// like std_normal, but with a non-zero mean
define unit_normal = make_elementary_sp(() -> {
  let simulate_std_normal = ${simulate_std_normal};
  let assess_std_normal = ${assess_std_normal};
  dict(
    ['simulate', (mean) -> {
      return (mean + simulate_std_normal())
    }],
    ['log_density', (x, mean) -> {
      return (assess_std_normal(x - mean))
    }])
});

define normal_normal_posterior_density =
  bayes_rule_estimate(
    prior: std_normal,
    likelihood: unit_normal,
    y: 4.0, n_samples: 500);
```

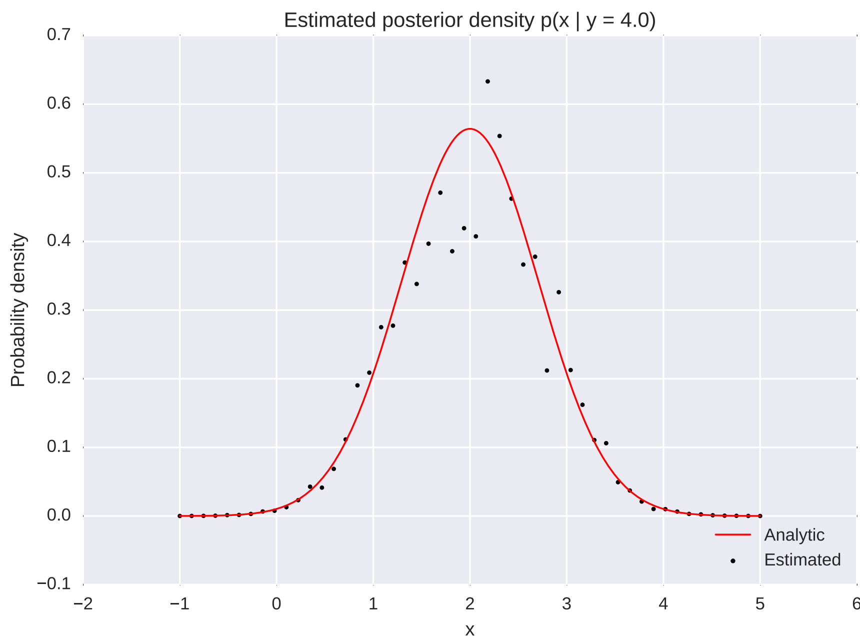

Figure 2.4: Using a Monte Carlo approximation of Bayes' rule to estimate $p(x|y = 4.0)$ in the normal-normal model.

```
// Joint simulator for a discrete HMM

define simulate_hmm = (T, O, N) ~> {
  simulate_hmm_loop(T, O, 0, [], [], N);
};

define simulate_hmm_loop = (T, O, prev, states, data, N) ~> {
  if (N == 0) {
    (reverse(states), reverse(data))
  } else {
    next = categorical(row(T, prev));
    out = categorical(row(O, next));
    simulate_hmm_loop(T, O, next, pair(next, states), pair(out, data), N - 1);
  }
};

// Joint assessor for a discrete HMM

// Explicitly does the computation that tracing the state sequence
// and assessing pointwise would have produced.
define assess_hmm_joint_density = (states, data, T, O, N) -> {
  if (length(states) == length(data) && length(states) == N) {
    hmm_joint_density_loop(0, states, data, T, O)
  } else {
    0 - infinity
  }
};

define hmm_joint_density_loop = (prev, states, data, T, O) -> {
  if (is_pair(states)) {
    log_step_p = assess(categorical, first(states), row(T, prev));
    log_out_p = assess(categorical, first(data), row(O, first(states)));
    log_step_p + log_out_p + hmm_joint_density_loop(first(states), rest(states), rest(data), T, O)
  } else {
    0
  }
};

// Forward filtering computation for a discrete HMM.

// Computes the probability of the data, and an explicit
// representation of the marginal at each time step conditioned on the
// data up to it (inclusive).
define assess_hmm_density_of_observations = (data, T, O) -> {
  forward_filter_loop(zero_with_prob_1, 0, [], data, T, O)
};

define forward_filter_loop = (prev, data_prob, filtered_probs, data, T, O) -> {
  if (is_pair(data)) {
    candidates = prev * T;
    cur = dot(candidates, col(O, first(data)));
    forward_filter_loop(cur, data_prob + log(sum(cur)), pair(cur, filtered_probs), second(data), T, O)
  } else {
    (data_prob, reverse(filtered_probs))
  }
};

// Posterior assessment

// Consists of scaling joint assessment by the probability of the
// data, the latter computed by forward filtering.
define assess_hmm_posterior_density = (data, T, O) -> {
  (data_prob, _) = assess_hmm_density_of_observations(data, T, O);
  (states) -> {
    assess_hmm_joint_density(states, data, T, O, length(data)) - data_prob
  }
};
```

Figure 2.5: A collection of meta-programs for simulating from a hidden Markov model and assessing joint, marginal, and posterior probability densities over the hidden states and observations.

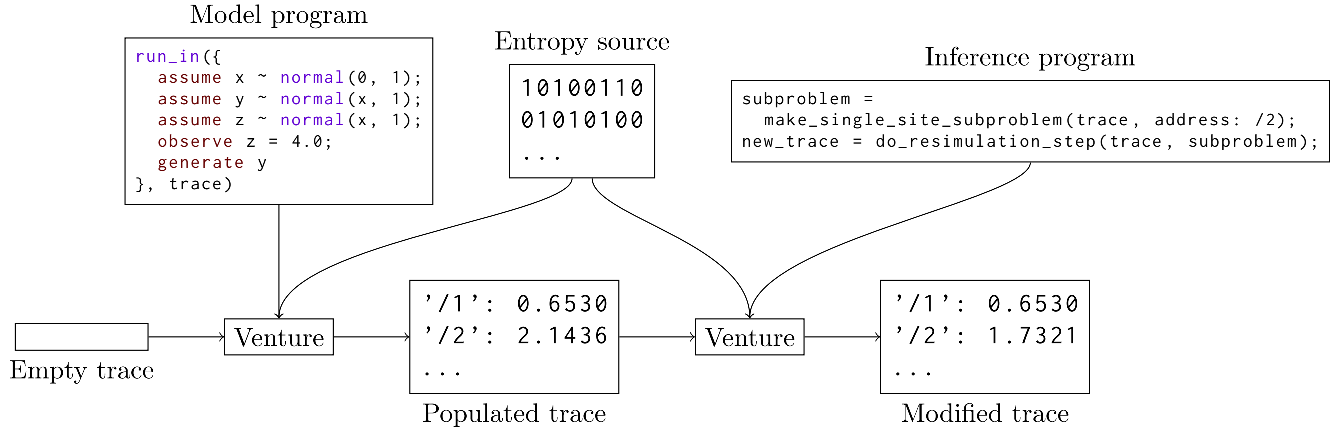

Figure 3.1: Traced execution of a model program, followed by inference that modifies the execution trace.

3 Probabilistic execution traces

For larger probabilistic programs that make many random choices, we are often interested in going beyond characterizing their input-output behavior with a density procedure. We would instead like to capture information about all random choices that occur in a program, so that we can perform inference about parts of the program conditioning on other parts.

Probabilistic execution traces (PETs), sometimes also abbreviated as "traces", are a new class of first-class object that represents the sequence of random choices that a probabilistic program makes. PETs explicitly represent the manifest behavior of a probabilistic program on a specific execution. All PETs contain a mapping from dynamic "addresses", each corresponding to distinguishable locations in the execution of some probabilistic program, to the manifest values taken by the program at those addresses. They thus contain complementary information to the source code of the program, which implicitly represents the potential behavior of the program. Put differently, the space of possible behaviors of the program corresponds to the space of PETs that could be generated by tracing its execution.

This idea of "traced execution" is central to a core capability of Venture: the ability to treat any probabilistic program as a probabilistic model defined over the space of possible executions of the program. A probabilistic model program defines a probabilistic model over the space of possible PETs that can result from tracing its execution. Venture exposes "traced execution" as an ordinary primitive procedure in the language, allowing users to run a Venture program and record the sequence of random choices that the program makes. The values in a PET can be mutated via a simple interface that is reflected back into user-space. The ability to mutate the values in a PET is central to Monte Carlo techniques for probabilistic inference, but is also useful for debugging. Figure 3.1 shows an example of running a model program, capturing its execution trace, and performing inference by exploring alternative execution histories.

Venture supports two kinds of traces:

1. *Flat traces.* These record a mapping from addresses to values, following [14]. Because they do not represent dependency information, they consume less memory and support a broad class of constant-factor performance optimizations. However, because they do not represent dependency information, it is difficult to determine what values in a flat trace would be potentially invalidated if some other value is changed. The simplest general policy is to always re-execute the entire traced program after changing even a single value. This policy results in unfavorable asymptotic scaling for key inference operations, especially for local sampling and gradient-based optimization algorithms. See Figure 3.2 for an example model program and a representation of its corresponding flat trace.

2. *Dependency graph traces.* These can be viewed as extensions of directed graphical models [10] [6] [2]. Because they contain dependency information, they can be used to re-execute fragments of a traced program without re-executing the entire program. See Figure 3.3 for an example.

In traced execution, an addressing scheme is used to assign a name to each random choice made during a program's execution, similar to [14]. In our scheme, each subcomputation that yields a result is assigned a unique address. An address is one of the following:

- *Toplevel address.* Each toplevel modeling command (`assume`, `observe`, or `generate`) triggers a traced subcomputation which is associated with an increasing sequential number starting from 1. For example, in the program in Figure 3.2, the address `/1` refers to the `assume`d expression `bernoulli(0.1)`, and the address `/3` refers to the `observe`d expression `bernoulli(weight)`.

- *Subexpression address.* This corresponds to traversing an edge in the abstract syntax tree. A subexpression address consists of a base address followed by a numeric index. When the base address refers to the result of a procedure application, an index of 0 selects the operator being applied, an index of 1 selects the first operand, etc. For example, `/1/0` refers to the operator of the first toplevel expression in the program, and `/1/1` to its argument.

- *Request address.* Some stochastic procedures, when applied, will "request" the execution of another program fragment. The most common instance of this is compound procedures, which request the execution of their procedure body. When such a procedure makes an execution request, it gives the request a unique ID, so that the requested subprogram is assigned the address `r(sp_address, request_id)` where `sp_address` is the address where the requesting procedure was created or defined. In the case of compound procedures, the request ID is the address of the call site, so that the address is analogous to a call stack address. For example, `r(/1, /2)` refers to the execution of the body of the procedure defined at `/1`, due to an application of the procedure at `/2`.

```
{
  assume coin_is_tricky ~ bernoulli(0.1);
  assume weight =
    if (coin_is_tricky) { beta(1.0, 1.0) }
    else { 0.5 };
  observe bernoulli(weight) = 1;
  clone_trace()
}
```


```
{'global_env': [{'coin_is_tricky': '/1', 'weight': '/2'}],
 'observations': {'/3': 1.0},
 'requests': {'/1': (['bernoulli', 0.1], [{}, 'global_env']),
              '/2': ([['biplex',
                       'coin_is_tricky',
                       ['make_csp',
                        ['quote', []],
                        ['quote', ['beta', 1.0, 1.0]]],
                       ['make_csp', ['quote', []], ['quote', 0.5]]]],
                     [{}, 'global_env']),
              '/3': (['bernoulli', 'weight'], [{}, 'global_env']),
              'r(/2/0/2, /2)': (['beta', 1.0, 1.0], [{}, {}, 'global_env'])},
 'results': {'/1': 1.0,
             '/1/0': 'a procedure',
             '/1/1': 0.1,
             '/2': 0.6468079632235574,
             '/2/0': 'a procedure',
             '/2/0/0': 'a procedure',
             '/2/0/1': 1.0,
             '/2/0/2': 'a procedure',
             '/2/0/3': 'a procedure',
             '/3': 1.0,
             '/3/0': 'a procedure',
             '/3/1': 0.6468079632235574,
             'r(/2/0/2, /2)': 0.6468079632235574,
             'r(/2/0/2, /2)/0': 'a procedure',
             'r(/2/0/2, /2)/1': 1.0,
             'r(/2/0/2, /2)/2': 1.0}}
```


Figure 3.2: An example tricky-coin model program and a printout of the flat trace data structure for the example program. The `/2/0/1` expressions are addresses; see main text.

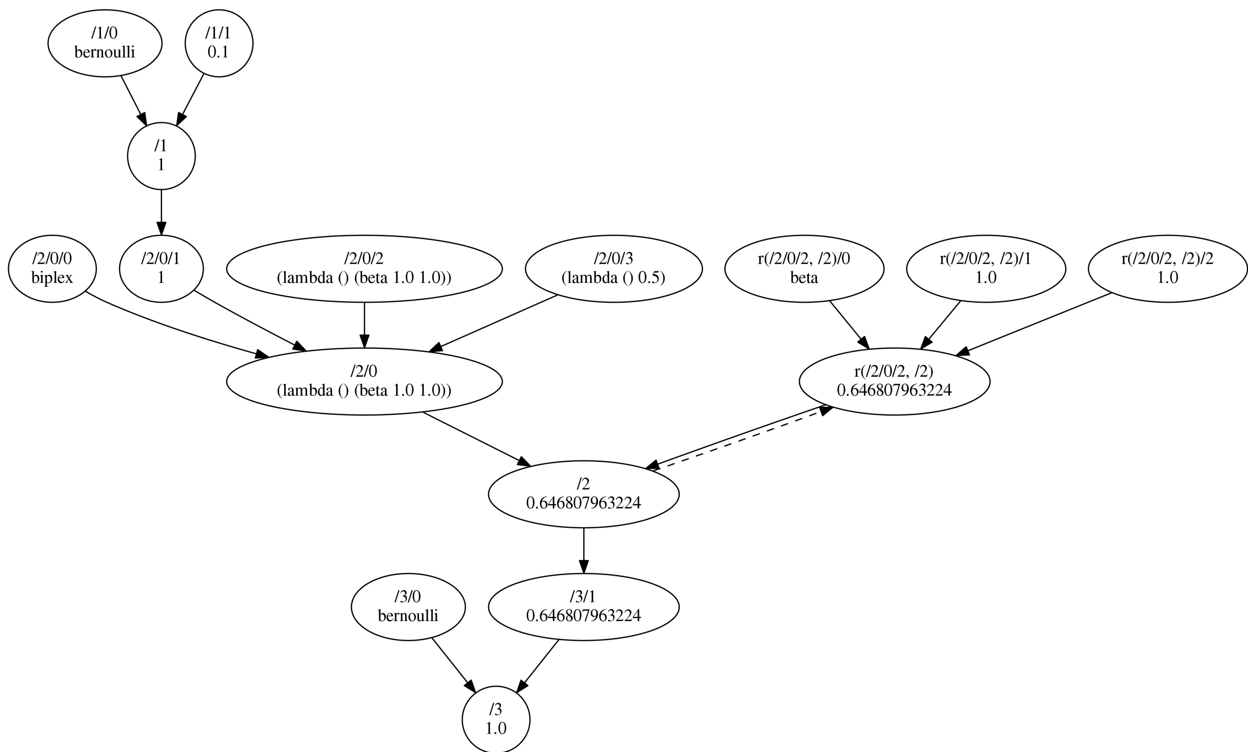

Figure 3.3: The dependency graph structure for the example tricky-coin program in Figure 3.2. Each node is labeled with its address (on the top) and its value in that execution (on the bottom).

In the dependency graph trace, nodes in the dependency graph are in one-to-one correspondence with addresses. Thus, each node represents the result of a sub-computation in the execution of the program.

It is important to note that PETs also serve as the only native form of mutable storage in Venture. New storage locations are allocated by tracing the execution of a random choice, and can be mutated for purposes other than performing probabilistic inference. For example, exchangeably coupled primitives can use traces to store and update sufficient statistics. This allows user-space VentureScript programs to emulate the behavior of foreign primitives that make use of mutating operations in their native implementation.

3.1 Traces as elements of a probability space

An execution trace represents a point ρ in a sample space Ω of all possible program executions. Formally, we define Ω to be the space of mappings from all possible stochastic procedure applications to results of those applications:

$$\Omega = \{\rho : (\texttt{addr}, \texttt{sp}, \texttt{inputs}) \mapsto \texttt{output}\}$$

Given ρ, each program subcomputation that yields a result corresponds to a random variable $f : \Omega \mapsto V$ which maps points in the sample space Ω to a space of possible values V. The value of $f(\rho)$ is determined by the result of the subcomputation corresponding to f when the program is executed conditional on ρ, that is, using ρ to determine the results of all stochastic choices.

Because traces are first-class objects in Venture, it is possible to have expressions corresponding to random variables that themselves take on values in Ω. This allows stochastic inference programs to be expressed as programs that output random execution traces of other programs, which in general sample from a distribution that is different from the unconditioned distribution of the model program. Often, said different distribution is an approximation to a conditional distribution of interest.

Note that in the above definition, the elements of Ω are abstract infinite-dimensional objects that represent potential program executions, describing the result of every possible random choice that a program could make. In contrast, the PET obtained by traced execution of a Venture program corresponds to a realized execution of that program, containing the finite number of random choices made by the program (assuming the program halts). These two notions of traces—roughly, execution "futures" and execution "histories"—are different but closely related. Given an execution future ρ and the source code of a program, there is a corresponding execution history, which is obtained by executing the program using ρ as the source of randomness for all stochastic choices that occur and recording the results of those choices. In other words, the execution history is the restriction of ρ to the set of random choices that are actually realized during the program's execution. Conversely, given an execution history obtained by tracing the execution of a program, it is possible to construct an execution future by augmenting it with a random number generator state, which is used to generate the results of any random choices other than the ones realized during the

Operation	Runtime
`next_base_address()`	$O(1)$
`global_env()`	$O(1)$
`eval_request(addr, exp, env)`	$O(\|e\|)$
`value_at(addr)`	$O(1)$
`bind_global(symbol, addr)`	$O(1)$
`find_symbol(env, symbol)`	$O(1)$
`register_observation(addr, val)`	$O(1)$
`get_observations()`	$O(1)$
`set_value_at(addr, val)`	$O(1)$
`clone_trace()`	$O(\|\mathcal{T}\|)$
`single_site_subproblem(addr)`	$O(\|\mathcal{S}\|)$

Table 3.1: Index of trace operations, with asymptotic runtime. $|e|$ is the number of sub-operations involved in running the given program e; $|\mathcal{T}|$ is the size of the probabilistic execution trace; $|\mathcal{S}|$ is the size of the subproblem, which is determined by the number of downstream random choices that depend on the proposed value.

program's execution.

Together, these two trace notions provide a framework for modifying the execution history of a program and re-running the program in light of those modifications. Venture provides a version of this functionality in the form of *stochastic regeneration* (described in Section 3.3). Stochastic regeneration can be seen as an operation that extracts part of an execution history and turns it into a representation of a possible execution future, and then samples a new execution future which can then be incorporated back into the history.

In the remainder of this thesis, PETs and traces will be used to refer to realized execution histories and the data structures that represent them. Trace "futures" currently have no explicit representation, outside of the objects used in stochastic regeneration; further unifying these notions is interesting ground for future work but beyond the scope of this thesis.

3.2 An interface to probabilistic execution traces

This section briefly describes the key operations supported by probabilistic execution traces in a prototype implementation of Venture. In the below, the trace object is denoted $\mathcal{T}$.

- $a \leftarrow$ `next_base_address`$(\mathcal{T})$

  Get the next unallocated base address in the trace, and increment the next address pointer.

- $\Gamma \leftarrow$ `global_env`$(\mathcal{T})$

  Get the trace's global environment.

- `eval_request`$(\mathcal{T}$, `base_address`: a, `expression`: e, `environment`: $\Gamma)$

  Execute the program fragment denoted by the expression in the environment and record all intermediate results, storing the result at the given base address.

- $v \leftarrow$ `value_at`($\mathcal{T}$, `address`: a)

  Get the value recorded in the trace at the given address.

- `bind_global`($\mathcal{T}$, `symbol`: s, `address`: a)

  Bind the symbol in the trace's global environment to the result at the given address.

- $a \leftarrow$ `find_symbol`($\mathcal{T}$, `environment`: Γ, `symbol`: s)

  Look up the symbol in the given environment and return the corresponding address.

- `register_observation`($\mathcal{T}$, `address`: a, `value`: v)

  Record that the result at the given address has been observed to be the given value. On its own, this has no effect on the execution trace, and in particular does not modify the value stored at that address to be consistent with the observation. However, various meta-programs will check for inconsistent traces and treat them as having zero probability (for example, by rejecting a transition that results in such a trace).

- { (`address`: a, `value`: v) } $\leftarrow$ `get_observations`($\mathcal{T}$)

  Get the list of observations that have been recorded. This can be used in conjunction with `value_at` to check whether the trace is consistent with all the observations.

- `set_value_at`($\mathcal{T}$, `address`: a, `value`: v)

  Modify the value stored for the result at address. This is mainly intended for low level inference programs rather than end users.

- $\mathcal{T}^* \leftarrow$ `clone_trace`($\mathcal{T}$)

  Create a copy of the current trace.

- $\mathcal{S} \leftarrow$ `single_site_subproblem`($\mathcal{T}$, `address`: a)

  Define a partition of the trace into subsets that will be regenerated and preserved when a single random choice (stochastic procedure application) is regenerated. This corresponds roughly to the notion of a "scaffold" from [6, 2].

The asymptotic runtimes of these operations in the trace data structure defined in our prototype are given in Table 3.1.

Using these trace primitives, it is possible to define modeling instructions as well as inference procedures and utilities. See Figure 3.4 for implementations of the common modeling instructions; inference programming will be covered in Section 4.

In our prototype, traces also provide the interface by which key behaviors of stochastic procedures can be invoked. Examples include applying stochastic procedures; assessing log densities; and simulating kernels. These implementation details are beyond the scope of this thesis.

```
define _assume = (symbol, expr) -> {
  // execute the program denoted by expr (in the ambient trace)
  // and bind the result to symbol in the global environment
  addr <- next_base_address();
  env <- global_env();
  value <- eval_request(addr, expr, env);
  bind_global(symbol, addr);
  value
};

define _observe = (expr, value) -> {
  if (is_symbol(expr)) {
    // look up the symbol and constrain its target
    env <- global_env();
    addr <- find_symbol(env, expr);
    register_observation(addr, value);
    incorporate_constraint(addr, value)
  }
  else {
    // execute the program denoted by expr and constrain the result
    addr <- next_base_address();
    env <- global_env();
    eval_request(addr, expr, env);
    register_observation(addr, value);
    incorporate_constraint(addr, value)
  }
};

define _generate = (expr) -> {
  // execute the program denoted by expr and return the result
  addr <- next_base_address();
  env <- global_env();
  value <- eval_request(addr, expr, env);
  value
};

define incorporate_constraint = (addr, value) -> {
  // modify the trace so that the result recorded at addr
  // is consistent with the observed value
  oper <- value_at(subexpression(0, addr));
  kernel <- constraint_kernel_of_sp_at(addr, oper, value);
  subproblem <- single_site_subproblem(addr, kernel);
  package <- extract(subproblem);
  let (_weight, trace_fragment) = package;
  regen(subproblem, trace_fragment)
};
```

Figure 3.4: The modeling instructions `assume`, `observe`, and `generate` are implemented as macros which expand to applications of the corresponding procedures defined above. `observe` is also defined in terms of a utility procedure that modifies the trace so that it satisfies a given constraint. This procedure relies on primitives for *stochastic regeneration*, described in Section 3.3, to modify the trace and propagate the consequences.

3.3 Stochastic regeneration of trace fragments

Traces can be accessed by a uniform interface for stochastically (re)generating traces for program fragments that facilitates the development of inference meta-programs. Flat traces and dependency graph traces exhibit different asymptotic scaling behavior (see Table 3.2) and also support different constant-factor optimizations and runtime, making them suitable for different classes of inference strategies. Because these operations partially determine the requirements for stochastic procedures, this chapter briefly describes each of the key stochastic regeneration operations. See [6] and [2] for additional perspectives on stochastic regeneration.

Stochastic regeneration operates on *subproblems*, which are partitions of the trace into subsets of random choices to be regenerated and preserved, together with proposal distributions for the random choices being regenerated. Conceptually, a subproblem $\mathcal{S}$ assigns each random choice in the trace into one of three sets $(\mathbf{R}_\mathcal{S}, \mathbf{C}_\mathcal{S}, \mathbf{O}_\mathcal{S})$, where

1. $\mathbf{R}_\mathcal{S}$ is the "regenerated" set, consisting of random choices that are being proposed or values that may change as a result of the proposal. The subproblem induces a proposal distribution $q(\mathbf{R}_\mathcal{S})$.

2. $\mathbf{C}_\mathcal{S}$ is the "constrained" set, consisting of random choices that depend on values in $\mathbf{R}_\mathcal{S}$; these are the children of $\mathbf{R}_\mathcal{S}$ in the dependency graph. A node becomes part of $\mathbf{C}_\mathcal{S}$ if its inputs may change as a result of the proposal but it can assess the density or likelihood ratio of producing the same output given the new inputs. These nodes are relevant for assessing the likelihood of a proposal.

3. $\mathbf{O}_\mathcal{S}$ are other nodes in the trace that do not participate in the proposal.

A subproblem targets the "local posterior" distribution $p(\mathbf{R}_\mathcal{S}|\mathbf{C}_\mathcal{S}, \mathbf{O}_\mathcal{S}) \propto p(\mathbf{R}_\mathcal{S}|\mathbf{O}_\mathcal{S})p(\mathbf{C}_\mathcal{S}|\mathbf{R}_\mathcal{S}, \mathbf{O}_\mathcal{S})$, and induces a weight function corresponding to the ratio between the target and proposal densities

$$w = \log \frac{p(\mathbf{R}_\mathcal{S}, \mathbf{C}_\mathcal{S}|\mathbf{O}_\mathcal{S})}{q(\mathbf{R}_\mathcal{S}, \mathbf{C}_\mathcal{S})} = \log \frac{p(\mathbf{R}_\mathcal{S}|\mathbf{C}_\mathcal{S}, \mathbf{O}_\mathcal{S})}{q(\mathbf{R}_\mathcal{S}, \mathbf{C}_\mathcal{S})} + constant.$$

Note that this treatment only addresses independent proposals q. Integrating non-independent proposals into this framework is left for future work; see [6] for a possible approach.

In practice, a subproblem is represented as an ordered mapping from trace addresses to kernels, which are stochastic procedure meta-programs. Kernels implement an interface very similar to subproblems, but represent a proposal distribution on an individual stochastic procedure application. Nodes in $\mathbf{R}$ and $\mathbf{C}$ are distinguished by the latter being associated with *constraint kernels*, a special type of kernel whose proposal distribution always produces the same output value. Kernels of stochastic procedures are described in more detail in Section 5.

The operations for interacting with subproblems are described below. In the following descriptions, ρ denotes the original values in the trace and ξ denotes the proposed values, and the notation $\mathbf{R}_\mathcal{S}(\rho)$ means

the elements of ρ corresponding to nodes in $\mathbf{R}_\mathcal{S}$ (and similarly $\mathbf{R}_\mathcal{S}(\xi)$).

- (`weight`: w, `trace_fragment`: t) $\leftarrow$ `extract`($\mathcal{T}$, `subproblem`: $\mathcal{S}$)

  Mutate the trace to remove the values $\mathbf{R}_\mathcal{S}(\rho)$ corresponding to the random choices targeted by the subproblem $\mathcal{S}$. Extracts these values and stores them in a newly allocated trace fragment. Returns the (log) weight corresponding to the ratio between the target and proposal probability densities of the original completed trace

  $$w = \log \frac{p(\mathbf{R}_\mathcal{S}(\rho), \mathbf{C}_\mathcal{S} | \mathbf{O}_\mathcal{S})}{q(\mathbf{R}_\mathcal{S}(\rho), \mathbf{C}_\mathcal{S})}.$$

  `extract` is illustrated schematically in Figure 3.5.

- (`weight`: w) $\leftarrow$ `regen`($\mathcal{T}$, `subproblem`: $\mathcal{S}$)

  Samples new values $\mathbf{R}_\mathcal{S}(\xi)$ for the random choices targeted by the subproblem $\mathcal{S}$, according to the proposal distribution $q(\mathbf{R}_\mathcal{S})$. Returns the (log) weight corresponding to the ratio between the target and proposal probability densities for the new values

  $$w = \log \frac{p(\mathbf{R}_\mathcal{S}(\xi), \mathbf{C}_\mathcal{S} | \mathbf{O}_\mathcal{S})}{q(\mathbf{R}_\mathcal{S}(\xi), \mathbf{C}_\mathcal{S})}.$$

  Together with the weight returned by `extract`, this enables computing the acceptance ratio for Metropolis-Hastings transitions between ρ and ξ based on independent proposals q.

  `regen` is illustrated schematically in Figure 3.6.

- `restore`($\mathcal{T}$, `subproblem`: $\mathcal{S}$, `trace_fragment`: t)

  Returns the values in t into the trace $\mathcal{T}$. This can be used for rejecting transitions. `restore` is illustrated schematically in Figure 3.7.

- $w^* \leftarrow$ `weight_bound`($\mathcal{T}$, `subproblem`: $\mathcal{S}$)

  Returns an upper-bound on the regeneration weight of $\mathcal{T}$, suitable for use with rejection sampling.

These operations are implemented by delegating to the corresponding operations of the kernels that are contained in the subproblem.

Figure 3.8 walks through an example of stochastic regeneration on the trick coin example.

4 Meta-programs for stochastic inference

Most probabilistic programming languages come with built-in, immutable meta-programs for performing inference, namely identifying probable executions of a probabilistic model program in light of a set of user-

Operation	Flat trace	Dependency graph trace
`single_site_subproblem`	$O(N)$	$O(n+e)$
`extract`	$O(N)$	$O(n)$
`regen`	$O(N)$	$O(n)$
`restore`	$O(N)$	$O(n)$

Table 3.2: Asymptotic runtime of each trace operation in terms of N, the number of nodes in the trace (which approximately corresponds to the length of the program execution); n, the number of nodes involved in the subproblem; and e, the number of dependency edges between nodes involved in the subproblem.

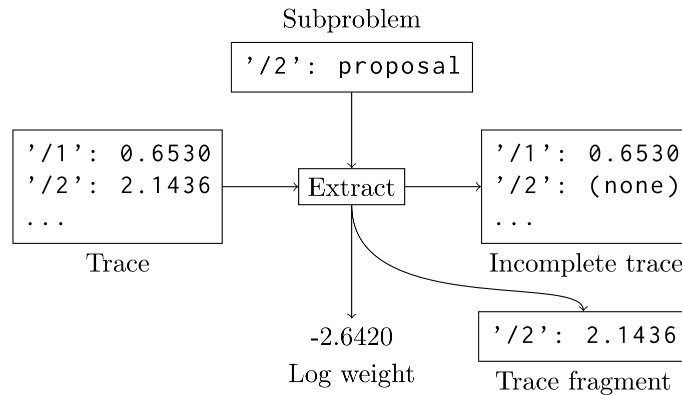

Figure 3.5: Extract takes a trace and a subproblem, extracts from the trace the values targeted by the subproblem, and returns a log weight corresponding to the ratio between the target and proposal probabilities of the trace with the original values.

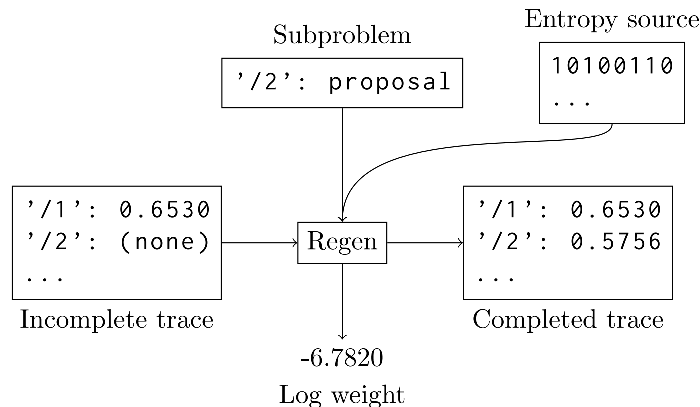

Figure 3.6: Regen takes a trace and an extracted subproblem, proposes new values for the target variables using fresh randomness, and returns a log weight for the ratio between target and proposal probabilities of the trace with the new values.

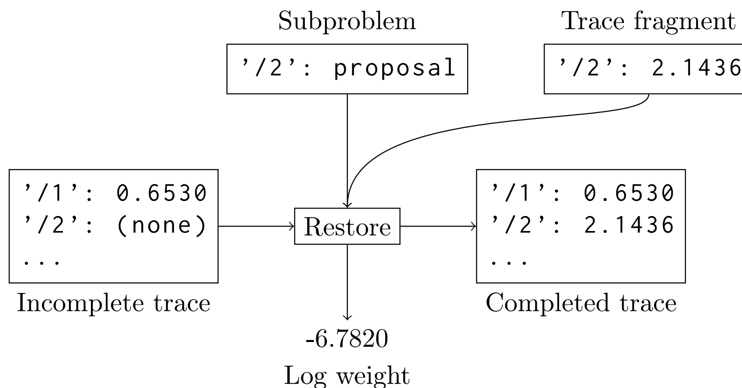

Figure 3.7: Restore takes a trace, an extracted subproblem, and the trace fragment returned by extract, and restores the state of the trace from the contents of the trace fragment.

specified constraints. In Venture, this is kind of computation is called *stochastic inference*. These meta-programs can be written as ordinary user-space VentureScript programs that make use of the built-in meta-programming facilities.

4.1 Exact stochastic inference via rejection sampling

This section describes a rejection sampling inference meta-program that can execute arbitrary model programs subject to "sufficiently stochastic" constraints. "Sufficiently stochastic" constraints are those that constrain random choices represented by model programs that have a known marginal probability density with a known bound (as a function of the inputs, for fixed output). This rejection sampling meta-program is demonstrated on a simple normal-normal model in Figure 4.1, showing that the results of rejection sampling match the analytical posterior distribution.

Figure 4.2 shows the implementation of the rejection sampling meta-program, which makes use of the stochastic regeneration primitives discussed in Section 3.3. In particular, recall that $\mathbf{regen}(\mathcal{T}, \mathcal{S})$ generates new values $\mathbf{R}_\mathcal{S}(\xi)$ according to a proposal distribution $q(\mathbf{R}_\mathcal{S}, \mathbf{C}_\mathcal{S})$, and returns a weight

$$w = \log \frac{p(\mathbf{R}_\mathcal{S}(\xi), \mathbf{C}_\mathcal{S}|\mathbf{O}_\mathcal{S})}{q(\mathbf{R}_\mathcal{S}(\xi), \mathbf{C}_\mathcal{S})}.$$

Furthermore, $\mathbf{weight_bound}(\mathcal{T}, \mathcal{S})$ computes a (conservative) upper bound w^* such that

$$w^* \geq \log \frac{p(\mathbf{R}_\mathcal{S}(\xi), \mathbf{C}_\mathcal{S}|\mathbf{O}_\mathcal{S})}{q(\mathbf{R}_\mathcal{S}(\xi), \mathbf{C}_\mathcal{S})}, \text{ for all possible values of } \mathbf{R}_\mathcal{S}(\xi)$$

With these two operations, it is possible to outline a simple rejection sampling algorithm for sampling from the subproblem-local posterior $p(\mathbf{R}_\mathcal{S}|\mathbf{C}_\mathcal{S}, \mathbf{O}_\mathcal{S})$:

```
// run a model program, returning a trace

trace = run_in({
  assume coin_is_tricky ~ bernoulli(0.1);
  assume weight =
    if (coin_is_tricky) { beta(1.0, 1.0) }
    else { 0.5 };
  observe bernoulli(weight) = 1;
  clone_trace()
}, graph_trace());

render_graph(trace);

// select a random choice and construct
// a subproblem around it

subproblem = run_in({
  single_site_subproblem(/1)
}, trace);

render_subproblem(trace, subproblem);

// extract the existing values
// from the trace

(old_weight, trace_fragment) = run_in({
  extract(subproblem)
}, trace);

render_subproblem(trace, subproblem);
render_fragment(trace_fragment);

// regenerate the trace with new proposed
// values

new_weight = run_in({
  regen(subproblem)
}, trace);

render_subproblem(trace, subproblem);

// the new trace

render_graph(trace);

// query

run_in({
  generate coin_is_tricky  // 0
}, trace);
```

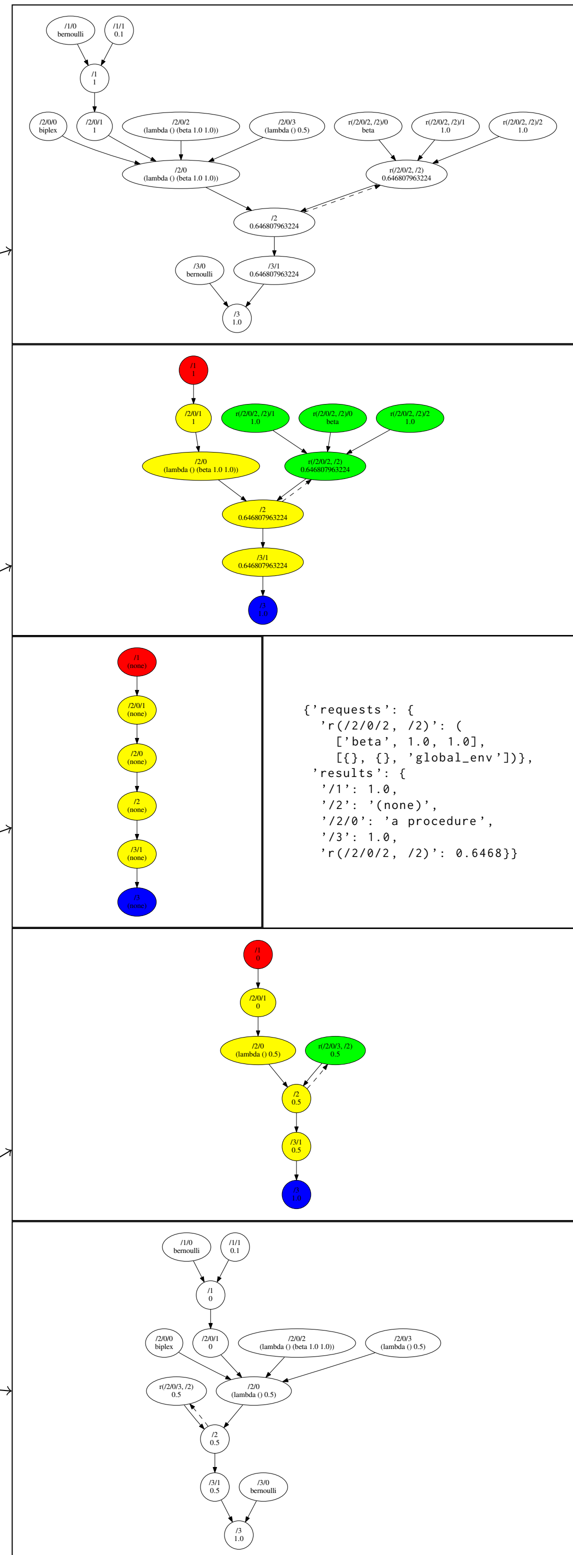

Figure 3.8: Depiction of stochastic regeneration on the example tricky coin model from Figure 3.2.

```
define example_prior = () ~> {
  run_in({
    assume x ~ normal(0, 1);
    assume y ~ normal(x, 1);
    observe y = 4.0;
    generate x
  }, flat_trace())
};
define example_posterior = () ~> {
  run_in({
    assume x ~ normal(0, 1);
    assume y ~ normal(x, 1);
    observe y = 4.0;
    subproblem <- single_site_subproblem(/1);
    rejection_sample(subproblem);
    generate x
  }, flat_trace())
};
```

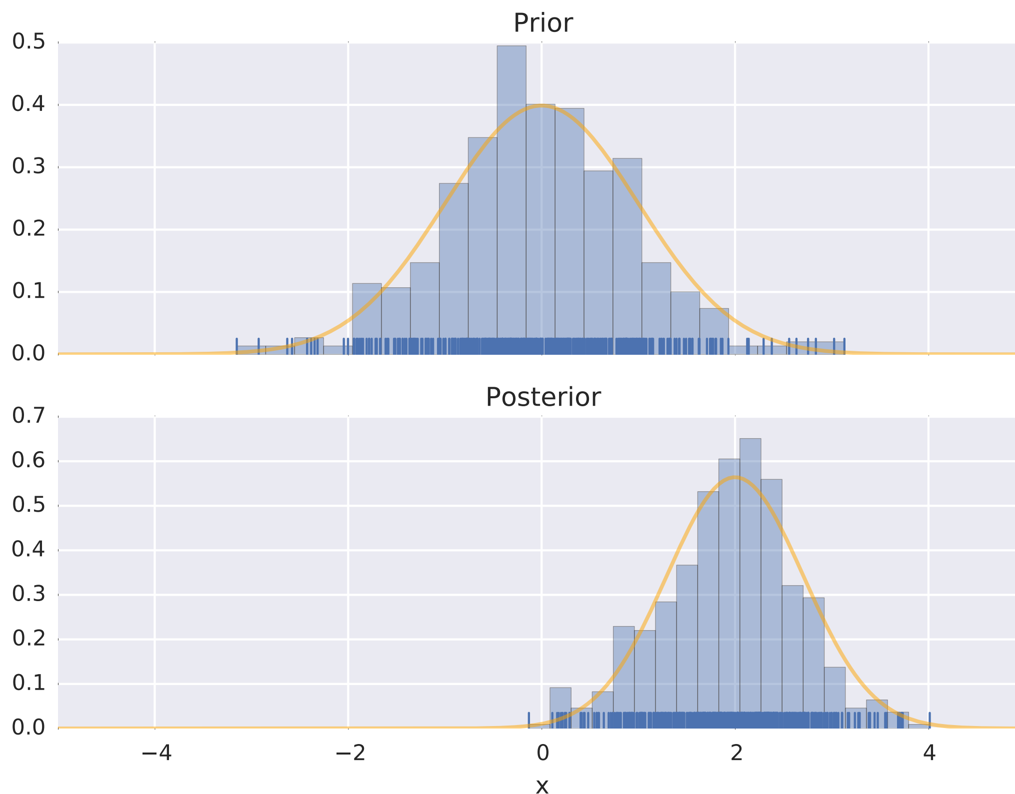

Figure 4.1: Top: A VentureScript program that defines a simple normal-normal model and obtains a posterior sample of x given an observed y using rejection sampling. Bottom: Histograms of samples obtained from 500 independent runs of each procedure, comparing the distribution of x under the prior and the posterior distribution induced by rejection sampling. The true density curves are shown in yellow: the prior $N(0, 1)$, and the posterior $N(2, \sqrt{\frac{1}{2}})$.

1. Compute a weight bound w^* for the subproblem $\mathcal{S}$.

2. Repeatedly:

   (a) Regenerate $\mathcal{S}$, populating the trace with new values ξ and obtaining weight w.

   (b) With probability $\exp(w - w^*) = \exp(w)/\exp(w^*)$, accept ξ. Otherwise, reject and try again.

The meta-program in Figure 4.2 follows this template, with the addition that it checks whether the proposed values ξ are consistent with the `observe` statements registered in the trace; if any are violated, the proposal is immediately rejected. This ensures that a rejection sampling step always samples conditional on the observations, even if the selected subproblem would otherwise cause an observed expression to be resimulated. This check is performed using a utility procedure `check_consistent`, which has a simple definition shown in Figure 4.3.

In the case of the normal-normal example:

1. $\mathbf{R}_\mathcal{S}$ is the variable x and $\mathbf{C}_\mathcal{S}$ is the observed variable y;

2. the proposal distribution $q(\mathbf{R}_\mathcal{S})$ is just the prior distribution of x, $N(0, 1)$;

3. the regeneration weight reduces to the log likelihood $\log N(y; x, 1)$;

4. and the weight bound is the maximum of the density function of the likelihood,

$$\max_x \log N(y; x, 1) = \log N(y; y, 1) = -\frac{1}{2} \log \pi - \log \sigma.$$

Thus, the rejection sampling meta-program recovers a valid and straightforward inference algorithm for this model by assembling together meta-programs (e.g. simulator and density) provided by the `normal` SPs that comprise the model program.

4.2 Approximate stochastic inference via Metropolis-Hastings

This section describes an inference meta-program built from the Metropolis-Hastings recipe, which applies to the same class of inference problems as the previous rejection sampling meta-program. Unlike rejection sampling, which returns an exact sample from the target conditional distribution $p(\mathbf{R}_\mathcal{S}|\mathbf{C}_\mathcal{S}, \mathbf{O}_\mathcal{S})$ on each invocation, the output distribution from this meta-program asymptotically converges to the conditional distribution under repeated applications. This meta-program is demonstrated on the normal-normal example model in Figure 4.4, showing that the sampling distribution approaches the posterior distribution with more iterations.

Figure 4.5 shows the implementation of the Metropolis-Hastings meta-program. Similar to the rejection meta-program, it makes use of stochastic regeneration to propose new samples, but unlike rejection, it does

```
define rejection_sample = (subproblem) -> {
  bound <- weight_bound(subproblem);
  rejection_sample_given_bound(subproblem, bound)
};
define rejection_sample_given_bound = (subproblem, bound) -> {
  package <- extract(subproblem);
  let (old_weight, trace_fragment) = package;
  new_weight <- regen(subproblem, trace_fragment);
  consistent <- check_consistent();
  if (consistent && log_flip(new_weight - bound)) {
    // accept
    pass
  }
  else {
    // try again
    rejection_sample_given_bound(subproblem, bound)
  }
};
```

Figure 4.2: The rejection sampling meta-program, defined in terms of inference subproblem operations `extract`, `regen`, and `weight_bound`, and the utility `check_consistent` given in Figure 4.3.

```
define check_consistent = () -> {
  observations <- get_observations();
  is_consistent <- for_each(keys(observations), (addr) -> {
    obs_val = lookup(observations, addr);
    cur_val <- value_at(addr);
    return (obs_val == cur_val)
  }
  return (all(is_consistent))
};
```

Figure 4.3: Definition of a utility procedure to check whether a trace is consistent with all observations.

```
define example_resimulation = (steps) ~> {
  run_in({
    assume x ~ normal(0, 1);
    assume y ~ normal(x, 1);
    observe y = 4.0;
    repeat(steps, {
      subproblem <- single_site_subproblem(/1);
      resimulation_step(subproblem)
    });
    generate x
  }, flat_trace())
};
```

Figure 4.4: Top: A VentureScript program that uses resimulation MH on a simple normal-normal model to obtain an approximate posterior sample of x. Bottom: Histograms of samples obtained by running the Markov chain for different numbers of steps. As the number of steps increases, the distribution approaches the target posterior (yellow).

```
define resimulation_step = (subproblem) -> {
  package <- extract(subproblem);
  let (old_weight, old_trace_fragment) = package;
  new_weight <- regen(subproblem, old_trace_fragment);
  consistent <- check_consistent();
  if (consistent && log_flip(new_weight - old_weight)) {
    // accept
    pass
  }
  else {
    // reject
    extract(subproblem);
    restore(subproblem, old_trace_fragment)
  }
};
```

Figure 4.5: The resimulation MH meta-program, defined in terms of the inference subproblem operations `extract`, `regen`, and `restore`.

not require an absolute upper bound on the regeneration weight. Instead, it makes use of the ratio (or difference, in log-space) between the weights returned by `extract` and `regen`:

$$\begin{aligned} w_\xi - w_\rho &= \log \frac{p(\mathbf{R}_\mathcal{S}(\xi), \mathbf{C}_\mathcal{S} | \mathbf{O}_\mathcal{S})}{q(\mathbf{R}_\mathcal{S}(\xi), \mathbf{C}_\mathcal{S})} - \log \frac{p(\mathbf{R}_\mathcal{S}(\rho), \mathbf{C}_\mathcal{S} | \mathbf{O}_\mathcal{S})}{q(\mathbf{R}_\mathcal{S}(\rho), \mathbf{C}_\mathcal{S})} \\ &= \log \frac{p(\mathbf{R}_\mathcal{S}(\xi), \mathbf{C}_\mathcal{S} | \mathbf{O}_\mathcal{S}) q(\mathbf{R}_\mathcal{S}(\rho), \mathbf{C}_\mathcal{S})}{p(\mathbf{R}_\mathcal{S}(\rho), \mathbf{C}_\mathcal{S} | \mathbf{O}_\mathcal{S}) q(\mathbf{R}_\mathcal{S}(\xi), \mathbf{C}_\mathcal{S})} \end{aligned}$$

This can be used as an acceptance ratio, producing a Markov chain that is stationary on the target distribution $p(\mathbf{R}_\mathcal{S}, \mathbf{C}_\mathcal{S} | \mathbf{O}_\mathcal{S})$. The implementation in Figure 4.5 follows this recipe. Note that once again `check_consistent` is used to reject if any observations are violated. Also, in the case that a transition is rejected, `restore` is used to return the trace back to its original state.

4.3 Custom model-specific inference meta-programs

Because inference can be written in user-space, Venture makes it possible to write custom inference algorithms as ordinary user-space programs. Users can adapt inference to the needs of the problem at hand, starting with general-purpose inference meta-programs but optimizing once they have developed test cases and better understand the functional requirements in terms of accuracy and runtime cost.

4.3.1 Custom Metropolis-Hastings samplers

While Venture provides a Metropolis-Hastings primitive based on stochastic regeneration proposals on subproblems, users can also write their own Metropolis-Hastings procedures, circumventing Venture's built-in

subproblem machinery. Custom procedures can be more efficient by avoiding the interpretive overhead of a generic regeneration strategy, or make use of analyses such as cancellations or simplifications in the acceptance ratio.

Figure 4.6 shows an example application of one such a custom inference procedure, which implements a Metropolis-Hastings sampler using a custom Gaussian drift proposal specialized to our running example of a normal-normal model. In this case, the custom procedure is both more computationally efficient, because its code is specialized to the model, and more effective at inference because the proposal distribution is a better fit for the inference problem.

The implementation of the custom Gaussian drift procedure is shown in Figure 4.7. This implementation takes advantage of a cancellation in the acceptance ratio that comes from the fact that the Gaussian drift proposal is symmetric:

$$\frac{p(x_\xi|y)q(x_\xi \to x_\rho)}{p(x_\rho|y)q(x_\rho \to x_\xi)} = \frac{p(x_\xi|y)}{p(x_\rho|y)} = \frac{p(x_\xi)p(y|x_\xi)}{p(x_\rho)p(y|x_\rho)}$$

This simplified ratio can then be computed directly, by querying the log density of the `normal` procedure from within Venture.

4.4 Scaling of stochastic regeneration for approximate inference

Flat traces and dependency graph traces implement different algorithms for stochastic regeneration; thus, inference strategies based on stochastic regeneration exhibit different asymptotic scaling behavior and constant-factor runtime on the two types of traces.

Figure 4.8 compares the runtime per iteration of single-site resimulation-based inference on a Hidden Markov Model program, as a function of the number of timesteps in the HMM.

Each iteration of inference involves two types of operations: subproblem construction, where the nodes belonging to the regenerated and constrained sets are identified; and stochastic regeneration itself, which replays the parts of the program corresponding to those nodes.

In the flat trace, both operations require a full traversal of the trace and are thus linear in the size of the trace. In the first pass, each subexpression in the program is traversed, in program execution order, marking all values that depend on the values being proposed (i.e., the regenerated set $\mathbf{R}_\mathcal{S}$). Regeneration is done in a second pass, simulating a new execution of the program where random choices in $\mathbf{R}_\mathcal{S}$ are resampled using fresh randomness. Although it seems possible to merge the two passes so that the regeneration step does not require traversing the whole program again, the overall algorithm would still exhibit the same linear scaling.

With the dependency graph trace, the regenerated set $\mathbf{R}_\mathcal{S}$ can instead be identified using a depth-first search that follows dependency edges from regenerated nodes to the nodes that depend on them. Once the

```
define example_drift = (steps) ~> {
  run_in({
    assume x ~ normal(0, 1);
    assume y ~ normal(x, 1);
    observe y = 4.0;
    repeat(steps, {
      custom_drift_mh_step();
    });
    generate x
  }, flat_trace())
};
```

Figure 4.6: Top: A VentureScript program that uses a custom user-defined inference procedure (shown in Figure 4.7) for random-walk MH with Gaussian drift proposals. Bottom: A plot comparing sample trajectories of the Markov chain obtained from this inference program with sample trajectories from the program in Figure 4.4 which uses independent resimulation proposals. The Gaussian drift proposal is accepted more frequently, suggesting that the custom sampler is more efficiently exploring the space of traces.

```
define custom_drift_mh_step = () -> {
  // read the current values of x and y from the trace
  x <- value_at(/1);
  y <- value_at(/2);

  // Gaussian drift proposal
  let x_proposed ~ normal(x, 0.5);

  // evaluate the density of the current trace
  term1 <- log_density_of_sp(normal, x, [0, 1]);
  term2 <- log_density_of_sp(normal, y, [x, 1]);
  let old_weight = term1 + term2;

  // evaluate the density of the proposed trace
  term3 <- log_density_of_sp(normal, x_proposed, [0, 1]);
  term4 <- log_density_of_sp(normal, y, [x_proposed, 1]);
  let new_weight = term3 + term4;

  if (log_flip(new_weight - old_weight)) {
    // accept
    set_value_at(/1, x_proposed);
  }
  else {
    // reject
    pass;
  }
};
```

Figure 4.7: A custom meta-program implementing random-walk MH using Gaussian drift proposals, specialized to the normal-normal model from Figure 4.6.

```
define example_hmm_memoized = {
  assume x = mem((t) -> {
    if (t == 0) { flip() }
    else {
      flip(if (x(t - 1)) { 0.9 } else { 0.1 })
    }
  });
  assume y = (t) -> {
    flip(if (x(t)) { 0.7 } else { 0.3 })
  };
};

define example_hmm_unrolled = {
  assume x1 = flip();
  assume x2 = flip(if (x1) { 0.7 } else { 0.3 });
  assume x3 = flip(if (x2) { 0.7 } else { 0.3 });
  assume x4 = flip(if (x3) { 0.7 } else { 0.3 });
  assume x5 = flip(if (x4) { 0.7 } else { 0.3 });

  assume y1 = flip(if (x1) { 0.9 } else { 0.1 });
  assume y2 = flip(if (x2) { 0.9 } else { 0.1 });
  assume y3 = flip(if (x3) { 0.9 } else { 0.1 });
  assume y4 = flip(if (x4) { 0.9 } else { 0.1 });
  assume y5 = flip(if (x5) { 0.9 } else { 0.1 });
};
```

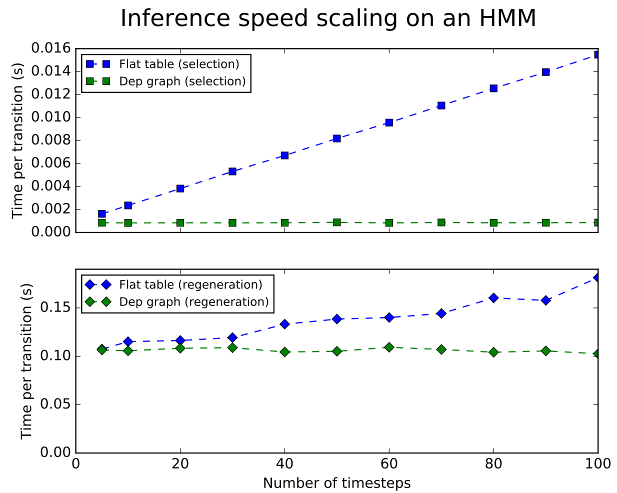

Figure 4.8: Comparison of inference speed on an HMM model program, as a function of the number of timesteps in the HMM. The model program is shown on the left, as well as a version of the program where the memoized loop is unrolled, showing how the number of steps in the HMM affects the program length. Inference is performed on single-site subproblems, which propose a new value for one element of the state sequence conditioned on all other variables. The plot on the right shows timing for the subproblem construction step and the resimulation step separately. The timing is performed on the unrolled program.

subproblem is identified, regeneration can be done using a traversal that only touches the nodes involved in the subproblem (and their parents). Therefore, each iteration of inference only scales as the size of the subproblem $|\mathcal{S}|$. However, this comes at the cost of somewhat more implementation complexity and memory overhead.

5 Stochastic procedures

Stochastic procedures (SPs) are used to encapsulate simple probability distributions, as well as user-space VentureScript programs and foreign probabilistic objects. They consist of a linked collection of programs and meta-programs that collectively describe aspects of a probabilistic program that are important for its use in modeling and inference. For example, they are the primary means for linking model programs with their associated density and inference meta-programs. SPs are designed to allow simple probability distributions, user-space VentureScript, and foreign probabilistic programs to be treated uniformly as building blocks of complex probabilistic computations.

Church and other previous probabilistic programming languages have focused on simple probability distributions that are completely specified by scalar-valued simulator procedures and analytical probability density functions. BayesDB [7] allows a larger class of probabilistic objects to be treated as encapsulated primitives—namely, *generative population models*, which describe multivariate joint distributions that can be simulated (or have their densities evaluated) conditionally and unconditionally. The class of probabilistic objects that Venture encapsulates includes both of these classes as proper subsets.

This thesis focuses on three kinds of stochastic procedures. These include (i) a generic interface that is suitable for encapsulating a broad class of probabilistic computations, including exchangeable random processes, likelihood-free primitives, and (at least in principle) interpreters and/or inference engines for other probabilistic programming languages; (ii) a simpler interface for stochastic procedures that makes it easy to encapsulate probabilistic objects that have known probability density functions; and (iii) a variant of the simple interface that can be used when the procedure is known to be "sufficiently stochastic", i.e., has an output that is itself a random choice with a known, bounded probability density.

The key methods associated with each of these types can be summarized as follows:

1. *Generic stochastic procedures.* These include (i) a probabilistic program `apply` for stochastically generating outputs, (ii) a "`proposal kernel`" probabilistic inference meta-program for proposing a new output and evaluating its marginal density ratio with the old output, and (iii) a specialized "`constraint kernel`" probabilistic inference meta-program that resamples internal randomness subject to the constraint that the output value matches a given constraint. These three capabilities allow SPs to encapsulate arbitrary probabilistic programs that come with their own internal latent variables, plus custom simulation, inference, density evaluation, and constraint enforcement mechanisms.

2. *"Simple" stochastic procedures.* When a stochastic procedure has a known marginal probability density on outputs, and when resimulation is a valid proposal strategy for inference, it is possible to provide a baseline implementation of the generic SP interface. This simplified interface also handles SPs that have internal mutation inducing an exchangeable coupling between the random variables corresponding

to repeated outputs.

3. *"Tail-assessable" stochastic procedures.* Some stochastic procedures—such as programs representing complex probabilistic generative models with noisy data likelihoods—have an intractable marginal output density but do induce a "sufficiently stochastic" likelihood conditioned on their internal latent variables. This is a sufficiently important special case that VentureScript uses it for its default stochastic procedure construction syntax.

It is helpful to think of stochastic procedures by analogy to software packages in a UNIX environment. These packages often include combinations of source code, binaries, test cases, and documentation. The presence of all these components—going considerably beyond pre-compiled binaries—facilitates the recombination of software by end-users, especially software developers and system administrators who are familiar with the associated conventions. In the case of Venture, inference meta-programs and end-to-end applications can play the role of the end-user, constructing new stochastic procedures dynamically, modifying the source code and/or runtime behavior, and both using and building models of the results.

5.1 Generic stochastic procedures

A stochastic procedure defines a joint distribution $p(\{y_i\}, \mathbf{z} | \{\mathbf{x}_i\})$, where $\mathbf{x}_i$ are the inputs to the i-th application of the procedure, y_i is the output of the i-th application, and $\mathbf{z}$ represents any latent random choices made during the applications of the SP other than the outputs y_i.

The y_i need not be independent, or even conditionally independent given $\mathbf{z}$; this allows SPs that represent exchangeably-coupled processes, such as collapsed conjugate models. However, Venture's stochastic regeneration algorithms assume that the applications are exchangeable, so that they can be regenerated in a different order while retaining the same distributional properties.

Stochastic procedures can have mutable internal state, which can be used to record results of latent random choices or to implement exchangeable coupling by maintaining sufficient statistics of previous applications. While PETs are the only native form of mutable storage directly available in Venture, it is possible to bind foreign procedures that use mutable state internally or expose sources of mutable state that user-defined SPs can use.

An SP can facilitate inference programming by providing one or more *kernels*, which operate on subsets of its random state space $(\{y_i\}, \mathbf{z})$. Kernels are analogous to trace subproblems and have a very similar contract, allowing random choices made during SP applications to be extracted and proposed anew.

General stochastic procedures can be created by the operation `make_sp`:

- $\mathcal{P} \leftarrow$ `make_sp(metadict_expression:` e`)`

Create a stochastic procedure from the collection of meta-programs specified by e, which should evaluate to a dictionary of method names and values.

Recognized methods include:

- $y \leftarrow$ `apply`($\mathcal{P}$, `trace_handle`: $\mathcal{H}$, `application_id`: i, `inputs`: $\mathbf{x}$)

  Apply the procedure to the inputs $\mathbf{x}$, producing output y and possibly recording latent choices z. This induces a distribution $p(y_i, z_i|\mathbf{x}_i, \mathcal{P})$.

  The trace handle $\mathcal{H}$ is an interface to the surrounding trace, which can be used to make call-backs to the traced execution engine; see the trace handle methods below.

- $l \leftarrow$ `log_density`($\mathcal{P}$, `output`: y, `inputs`: $\mathbf{x}$)

  (Optional) Evaluate the log density $\log p(y|\mathbf{x}, \mathcal{P})$.

  Not all stochastic procedures implement this method; those that do are called *assessable* procedures.

- $\mathcal{K} \leftarrow$ `proposal_kernel`($\mathcal{P}$, `trace_handle`: $\mathcal{H}$, `application_id`: i)

  Construct a kernel which operates on the results of a previous application of $\mathcal{P}$ (given by the application id i), representing a proposal distribution q that samples from the space of possible results of that application.

  The support of q should include the support of p (restricted to the subset of the SP's random state that is associated with the selected application). A standard choice is $q = p$, i.e., proposing from the prior.

  The kernel $\mathcal{K}$ should be a dictionary of method names and values that conforms to the kernel interface (described below).

- $\mathcal{K} \leftarrow$ `constraint_kernel`($\mathcal{P}$, `trace_handle`: $\mathcal{H}$, `application_id`: i, `constraint_value`: y)

  Construct a kernel which operates on the results of a previous application of $\mathcal{P}$ (given by the application id i), representing a proposal whose marginal distribution on outputs is a delta distribution at the value y. In other words, the constraint kernel resamples internal randomness (if any) subject to a constraint on the output value.

  For SPs with no latent randomness, this should deterministically return y. This is mainly useful for the kernel's regeneration weight, which reduces to the log density in this case.

A kernel $\mathcal{K}$ represents a proposal distribution on a subset of random choices made by the SP, analogous to a subproblem of a trace. Kernels have the following methods:

- (`log_weight`: w, `trace_fragment`: $\mathcal{F}$) $\leftarrow$ `extract`($\mathcal{K}$, `output`: y, `inputs`: $\mathbf{x}$)

  Extract the values of the target random choices from the SP's internal state. Return a log weight w corresponding to the ratio between the target and proposal probabilities of the original values (including any extracted latents)

  $$w = \log \frac{p(y, z|\mathbf{x}, \mathcal{P})}{q(y, z|\mathbf{x}; \mathcal{P})},$$

  along with a trace fragment $\mathcal{F}$ which is an object that can be used to restore the SP's state as it existed before extracting.

- (`log_weight`: w, `output`: y) $\leftarrow$ `regen`($\mathcal{K}$, `inputs`: $\mathbf{x}$)

  Propose new values for the target random choices using fresh randomness. Return a log weight w for the ratio between the target and proposal probabilities of the new values

  $$w = \log \frac{p(y, z|\mathbf{x}, \mathcal{P})}{q(y, z|\mathbf{x}; \mathcal{P})},$$

  along with the new output value y.

- (`output`: y) $\leftarrow$ `restore`($\mathcal{K}$, `inputs`: $\mathbf{x}$, `trace_fragment`: $\mathcal{F}$)

  Restore the SP's state using a trace fragment returned by `extract`.

- $w^* \leftarrow$ `weight_bound`($\mathcal{K}$, `inputs`: $\mathbf{x}$, `input_mask`: m)

  Return an upper bound on the regeneration weight w of this kernel, which is used to compute the rejection bound for a subproblem.

  The input mask m is a list of boolean values, the same length as $\mathbf{x}$, which indicates, for each input argument, whether it should be considered fixed or varying when computing a bound.

The trace handle $\mathcal{H}$ provides the following methods for interacting with the surrounding trace:

- $a \leftarrow$ `request_address`($\mathcal{H}$, `request_id`: i)

  Return a trace address derived from the given request ID, suitable for using with the below methods. The request ID can be any Venture value used by the SP to identify the subprogram executions it has requested. For example, compound procedures request the execution of their body using the address of the call site (i.e., the `application_id`) as the request ID, and `mem` (described in Section 5.5) uses the value of the input argument as a request ID.

- $z \leftarrow$ `eval_request`($\mathcal{H}$, `request_id`: i, `expression`: e, `environment`: Γ)

  Execute the program fragment denoted by the expression in the environment. This is an interface to the `eval_request` method of the ambient trace, allowing SPs to make callbacks to the traced interpreter.

- $z \leftarrow$ `value_at`($\mathcal{H}$, `request_id`: i)

  Get the value recorded in the trace at the given address. This is an interface to the `value_at` method of the ambient trace.

- `uneval_request`($\mathcal{H}$, `request_id`: i)

  Undo the execution of a request. This traverses the program fragment in reverse execution order, calling `extract` on all SP applications.

- $z \leftarrow$ `restore_request`($\mathcal{H}$, `request_id`: i)

  Restore an unevaluated request, traversing the program fragment in forward order and calling `restore` on all SP applications.

Execution requests provide an integrated mechanism for SPs to make callbacks to the interpreter, enabling the implementation of higher-order procedures such as `mem` (see Section 5.5), `eval`, `apply`, and `map`. Importantly, these callbacks are executed by the same traced interpreter as the original SP application, so the dependencies and random choices made during their execution are exposed and available for trace metaprograms to inspect. This allows, for example, dependency information to be propagated through the bodies of compound procedures and the applications of higher-order procedures.

5.2 "Simple" stochastic procedures

When a stochastic procedure has a known marginal probability density on outputs, and when resimulation is a valid proposal strategy for inference, it is possible to provide a baseline implementation of the generic SP interface.

Such an SP may be created with the `make_elementary_sp` procedure:

- $\mathcal{P} \leftarrow$ `make_elementary_sp`(`metadict_expression`: e)

  Create a stochastic procedure from the collection of meta-programs specified by e. That collection is expected to implement the below simplified interface, which assumes that the SP has no "latent" structure and represents a distribution $p(y|\mathbf{x})$ with procedures for simulating and assessing the density.

The interface expected by `make_elementary_sp` is:

- $y \leftarrow$ `simulate`($\mathcal{P}$, `inputs`: $\mathbf{x}$)

  Return a sample $y \sim p(\cdot|\mathbf{x})$.

- $l \leftarrow$ `log_density`($\mathcal{P}$, `output`: y, `inputs`: $\mathbf{x}$)

  Evaluate the log density $\log p(y|\mathbf{x})$.

```
define my_beta_1 = (a, b) -> {
  let y ~ gamma(a, 1);
  let z ~ gamma(b, 1);
  y/(y + z)
};
```

```
define my_beta_2 = make_elementary_sp(() -> {
  dict(
    ['simulate', (a, b) -> {
      let y ~ gamma(a, 1);
      let z ~ gamma(b, 1);
      return (y/(y + z))
    }],
    ['log_density', (x, a, b) -> {
      let numerator =
        (a - 1) * log(x) + (b - 1) * log(1 - x);
      let denominator = log_beta_function(a, b);
      return (numerator - denominator)
    }])
});
```

```
define my_beta_2_expanded = make_sp(() -> {
  dict(
    ['state', nil],
    ['apply', (trace_handle, app_id, a, b) -> {
      let y ~ gamma(a, 1);
      let z ~ gamma(b, 1);
      return (y/(y + z))
    }],
    ['proposal_kernel', (trace_handle, app_id) -> {
      return (dict(
        ['extract', (x, a, b) -> {
          let trace_fragment = x;
          return (pair(0, trace_fragment))
        }],
        ['regen', (a, b) -> {
          let y ~ gamma(a, 1);
          let z ~ gamma(b, 1);
          return (pair(0, y/(y + z)))
        }],
        ['restore', (a, b, trace_fragment) -> {
          let x = trace_fragment;
          return (x)
        }]))
    }],
    ['constraint_kernel', (handle, app_id, val) -> {
      return (dict(
        ['extract', (x, a, b) -> {
          let numerator =
            (a - 1) * log(x) + (b - 1) * log(1 - x);
          let denominator = log_beta_function(a, b);
          let weight =
            if (x == val) {
              (numerator - denominator)
            } else { log(0) };
          let trace_fragment = x;
          return (pair(weight, trace_fragment))
        }],
        ['regen', (a, b) -> {
          let x = val;
          let numerator =
            (a - 1) * log(x) + (b - 1) * log(1 - x);
          let denominator = log_beta_function(a, b);
          let weight = (numerator - denominator);
          return (pair(weight, x))
        }],
        ['restore', (a, b, trace_fragment) -> {
          let x = trace_fragment;
          return (x)
        }]))
    }])
});
```

Figure 5.1: Top: Definition of a beta sampler in terms of a gamma sampler, as a "bare" compound procedure. Bottom left: Definition of a beta sampler using `make_elementary_sp`, which creates a stochastic procedure from a simulator and a density assessor (and possibly additional metadata), implementing the rest of the generic stochastic procedure interface in terms of those procedures. Bottom right: An equivalent procedure constructed using the full interface `make_sp`, showing how `make_elementary_sp` expands into `make_sp`.

- $l^* \leftarrow$ `log_density_bound`($\mathcal{P}$, `output`: y, `inputs`: $\mathbf{x}$, `input_mask`: m)

  Evaluate an upper bound on the log density $\log p(y|\mathbf{x})$.

  The input mask m is a list of boolean values, the same length as $\mathbf{x}$, which indicates, for each input argument, whether it should be considered fixed or varying when computing a bound.

- `incorporate`($\mathcal{P}$, `output`: y, `inputs`: $\mathbf{x}$)

  Optional: For SPs that maintain sufficient statistics of their applications, record the input-output pair $(\mathbf{x}, y)$. This is called at the end of every `apply`, `regen`, and `restore`.

- `unincorporate`($\mathcal{P}$, `output`: y, `inputs`: $\mathbf{x}$)

  Optional: Inverse of `incorporate`: remove the input-output pair $(\mathbf{x}, y)$, as if that application had never happened. This is called at the beginning of every `extract`.

The generic SP interface can be defined using these methods: roughly, `apply` calls `simulate`; the proposal kernel implements resimulation proposals using `simulate`, and the constraint kernel returns a likelihood weight using `log_density`. If `incorporate` and `unincorporate` are defined, they are called at the end of every `apply`/`regen`/`restore` and at the beginning of every `extract`, respectively.

Figure 5.1 gives an example of a stochastic procedure that samples from the beta distribution, first as a "bare" compound procedure, and then as a procedure defined using `make_elementary_sp` by providing both `simulate` and `log_density`. Although both procedures have the same output behavior when invoked directly to generate samples, they are treated differently by tracing and inference meta-programs:

- The bare compound procedure `my_beta_1` is treated as the sum of its parts, with each call to `gamma` in the body recorded as a separate random choice, and the output of the beta procedure as a value computed from the two gamma samples. The procedure lacks any interesting auxiliary meta-programs, notably an output density or constraint kernel, so its output cannot be constrained to a given value.

- In contrast, `my_beta_2` is treated as if it were a single unit. Even though its simulator body still makes calls to `gamma`, these details are encapsulated inside the procedure rather than exposed to the trace. Instead, it has a `log_density` which characterizes the marginal distribution of outputs, allowing it to accept constraints on its output and report the relative likelihood of obtaining the output value from the given inputs, just like built-in primitives that come with density functions.

In this way, stochastic procedures provide a means of abstraction for traced programs, in the same way that "normal" procedures do for untraced programs.

5.3 "Tail-assessable" stochastic procedures

The stochastic procedures described in the previous section are "assessable" stochastic procedures, so called because they come with a tractable procedure for assessing their marginal probability density on outputs.

Some stochastic procedures—such as programs representing complex probabilistic generative models with noisy data likelihoods—have an intractable marginal output density but do induce a "sufficiently stochastic" likelihood conditioned on their internal latent variables. This is a sufficiently important special case that VentureScript uses it for its default stochastic procedure construction syntax.

A procedure f is called "tail-assessable" if either

- f is assessable, or
- f factors as $f(x) = g(h(x))$, where g is tail-assessable.

This class of procedures is interesting because, even though the marginal density is intractable, it is still possible to implement a constraint kernel that allows conditioning on the output y in rejection sampling, importance sampling, and Metropolis-Hastings style proposals.

To see how a constraint kernel can be constructed for the second case, consider the joint space (z, y), whose target distribution is given by

$$p(z, y|x) = p_h(z|x)p_g(y|z, x),$$

where p_h and p_g are the probability densities induced by h and g respectively (note that these densities are not necessarily tractable to compute). Then define the following proposal distribution on (z, y):

$$q(z, y|x; y_{obs}) = p_h(z|x)\mathbf{1}(y = y_{obs})$$

It is easy to simulate from this distribution: sample a new $z \sim h(x)$, and set $y = y_{obs}$. The density ratio is

$$\frac{p(z, y|x)}{q(z, y|x; y_{obs})} = \frac{p_h(z|x)p_g(y|z, x)}{p_h(z|x)\mathbf{1}(y = y_{obs})} = \frac{p_g(y|z, x)}{\mathbf{1}(y = y_{obs})}$$

If the density p_g is tractable to compute (that is, g is assessable), then we are done—we have a procedure to propose samples from the joint space (z, y), and we can evaluate the ratio between our proposal distribution and the target distribution. If not, g is by assumption tail-assessable, so we can recursively construct a constraint kernel for its output as well. Wherever the recursion terminates, we can simulate and evaluate the density ratio (possibly on an extended space $(z, y, w, \ldots)$, if g also has internal latents).

In VentureScript, it is possible to construct compound procedures with these semantics using the `proc` form:

- $\mathcal{P} \leftarrow$ `proc`(`parameters`: p, `body`: e)

  Create a stochastic procedure with the given parameters and body. This is a probabilistic analogue of `lambda` from Lisp-like languages.

The syntax is

```
proc (x) {
  g(h(x))
}
```

where in general there may be zero or more parameters (here, x is the only parameter), and the body is an expression that has a procedure call in tail position, whose operator (here, g) is a constrainable procedure.

Figure 5.2 shows two stochastic procedure definitions implementing the same beta-bernoulli process, one defined as a "bare" compound procedure and one defined as a tail-assessable procedure using `proc`, along with the equivalent expanded definitions that use the general `make_sp` interface, showing the differences. The version defined as a bare compound procedure has no density or constraint kernel meta-program, so its output is not constrainable. In contrast, the version defined using `proc` can be constrained by delegating to the density meta-program of the `bernoulli` SP called in its procedure body.

5.4 Tradeoffs between different representations of the Beta-Bernoulli process

The stochastic procedure interface makes it easy to represent a single stochastic process in multiple different ways. Probabilistic programmers can choose to expose random variables to Venture; represent them but keep them foreign; or collapse them out via marginalization. Probabilistic programmers can also choose different boundaries between VentureScript and faster (but potentially more opaque) native code. This flexibility may ultimately have significant implications for the achievable performance of inference.

The examples in this section explores this degree of freedom using the Beta-Bernoulli process, which can be thought of in terms of a single latent coin weight that yields a countably infinite sequence of coin flips. The most straightforward implementations of this process are the versions in Figure 5.2, consisting of a compound procedure that closes over a variable `theta` representing the coin weight.

Additionally, we consider two optimized implementations of the Beta-Bernoulli process. The first, shown in Figure 5.4, is a "collapsed" representation in which the latent coin weight is integrated out. The procedure maintains sufficient statistics of previous coin flips as it is applied, and uses them to simulate (and assess the density of) the next coin flip from its posterior predictive distribution, exploiting the conjugacy of the Beta and Bernoulli distributions.

The second implementation, shown in Figure 5.6, is an uncollapsed representation, so it samples the latent coin weight rather than integrating it out. However, it also keeps sufficient statistics, and makes use

```
define make_beta_bern_lambda = (a, b) -> {
  let theta ~ beta(a, b);
  () ~> {
    flip(theta)
  }
};

// example usage
assume coin = make_beta_bern_lambda(1, 1);
// <procedure>
generate coin();
// true
observe coin() = true;
// error: cannot constrain
```

```
define make_beta_bern_proc = (a, b) -> {
  let theta ~ beta(a, b);
  proc () {
    flip(theta)
  }
};

// example usage
assume coin = make_beta_bern_proc(1, 1);
// <procedure>
generate coin();
// true
observe coin() = true;
// weight: -0.0957080346425
```

```
define make_beta_bern_lambda_expanded = (a, b) -> {
  let theta ~ beta(a, b);
  let env = get_current_environment();
  make_sp((theta) -> {
    let exp = [| flip(theta) |];
    dict(['state', nil],
         ['apply', (trace_handle, app_id) -> {
           with(trace_handle, {
             let extended_env = extend_environment(
               $env, 'theta', address_of(theta));
             addr <- request_address(app_id);
             eval_request(addr, exp, extended_env)
           })
         }],
         ['proposal_kernel', (trace_handle, app_id) -> {
           return (dict(
             ['extract', (x) -> {
               with(trace_handle, {
                 addr <- request_address(app_id);
                 uneval_request(addr);
                 return (pair(0, nil))
               })
             }],
             ['regen', () -> {
               with(trace_handle, {
                 let extended_env = extend_environment(
                   $env, 'theta', address_of(theta));
                 addr <- request_address(app_id);
                 x <- eval_request(addr, exp, extended_env);
                 return (pair(0, x))
               })
             }],
             ['restore', (_) -> {
               with(trace_handle, {
                 addr <- request_address(app_id);
                 x <- restore_request(addr);
                 return (x)
               })
             }]))
         }])
  })
};
```

```
define make_beta_bern_proc_expanded = (a, b) -> {
  let theta ~ beta(a, b);
  env = get_current_environment();
  make_sp((theta) -> {
    let oper_exp = [| flip |];
    let opand_exp = [| theta |];
    dict(['state', nil],
         ['apply', (trace_handle, app_id) -> {
           with(trace_handle, {
             let ext_env = extend_environment(
               $env, 'theta', address_of(theta));
             oper_addr <- request_address(pair(app_id, 0));
             opand_addr <- request_address(pair(app_id, 1));
             operator <- eval_request(oper_addr, oper_exp, ext_env);
             operand <- eval_request(opand_addr, opand_exp, ext_env);
             result_addr <- request_address(app_id);
             output <- apply_sp(result_addr, operator, [operand]);
             return (output)
           })
         }],
         ['proposal_kernel', (trace_handle, app_id) -> {
           // same; omitted
         }],
         ['constraint_kernel', (trace_handle, app_id, val) -> {
           return (dict(
             ['extract', (x) -> {
               with(trace_handle, {
                 oper_addr <- request_address(pair(app_id, 0));
                 opand_addr <- request_address(pair(app_id, 1));
                 operator <- value_at(oper_addr);
                 operand <- value_at(opand_addr);
                 result_addr <- request_address(app_id);
                 kernel <- constraint_kernel_of_sp_at(
                   result_addr, operator, val);
                 extract_kernel(kernel, x, [operand])
               })
             }],
             ['regen', () -> {
               with(trace_handle, {
                 oper_addr <- request_address(pair(app_id, 0));
                 opand_addr <- request_address(pair(app_id, 1));
                 operator <- value_at(oper_addr);
                 operand <- value_at(opand_addr);
                 result_addr <- request_address(app_id);
                 kernel <- constraint_kernel_of_sp_at(
                   result_addr, operator, val);
                 regen_kernel(kernel, [operand])
               })
             }],
             ['restore', (trace_fragment) -> {
               with(trace_handle, {
                 oper_addr <- request_address(pair(app_id, 0));
                 opand_addr <- request_address(pair(app_id, 1));
                 operator <- value_at(oper_addr);
                 operand <- value_at(opand_addr);
                 result_addr <- request_address(app_id);
                 kernel <- constraint_kernel_of_sp_at(
                   result_addr, operator, val);
                 restore_kernel(kernel, [operand], trace_fragment)
               })
             }]))
         }])
  })
};
```

Figure 5.2: Left: Naive definition of the beta-Bernoulli procedure as a bare compound, which draws $\theta \sim \text{Beta}(a, b)$ and returns a closure that flips a coin with weight θ. As with `my_beta_1` from Figure 5.1, this closure has no density meta-program so its output is not constrainable. Right: An equivalent definition using `proc`, a meta-program which takes a *tail-assessable* procedure and generates a constraint kernel for it, which returns the density of the `flip` procedure as its weight. Bottom: Expanded versions of the resulting SPs defined using the full `make_sp` interface.

Variant	Code	LOC	$\theta \sim$	$x \sim$	$w(x)$	Runtime
`make_beta_bern_lambda`	Figure 5.2	6	$p(\theta)$	$p(x\|\theta)$	N/A	$O(N)$
`make_beta_bern_proc`	Figure 5.2	6	$p(\theta)$	$p(x\|\theta)$	$p(x\|\theta)$	$O(N)$
`make_beta_bern_collapsed`	Figure 5.4	25	N/A	$E_\theta[p(x\|\theta)]$	$E_\theta[p(x\|\theta)]$	$O(1)$
`make_beta_bern_uc_gibbs`	Figure 5.6	31	$p(\theta\|x)$	$p(x\|\theta)$	$p(x\|\theta)$	$O(1)$

Table 5.1: Comparison of the beta-Bernoulli variants. The "runtime" column describes the runtime complexity of doing one round of inference on θ (if applicable) and then simulating or incorporating a new observation, where N is the total number of observations that have been incorporated.

of the Beta-Bernoulli conjugacy in a different way, by providing a custom meta-program that resamples the latent coin weight from its conjugate posterior on demand.

Table 5.1 presents a summary of the representations that are described and compared, and Figure 5.3 compares the runtime and a measure of accuracy on an example.

5.5 Memoization as a user-space stochastic procedure

Memoization is a canonical application of higher-order procedures. A memoizer is a procedure that takes a procedure as input and returns a new procedure that internally caches values as they are computed. This can save computation time, especially for recursive procedures, at the cost of space. In the setting of probabilistic programming, memoization can be used as a building block for many interesting semi-parametric Bayesian models [3]. However, because memoization is a higher-order procedure, it does not fit within the simpler template of "elementary random procedures" from early implementations of Church [3, 14]. This section illustrates the flexibility of Venture and the stochastic procedure interface by giving a user-space implementation of memoization.

The implementation of the memoizer is given in Figure 5.7. When the memoized SP is applied, it first looks up the input argument and returns the existing value if there is any, and otherwise delegates to the unmemoized SP.

The implementation relies on execution requests, described earlier, which allows it to make traced calls to the original unmemoized SP. This is done by requesting the execution of a program fragment of the form `f(x)`, in an environment where `f` is bound to the address of the original SP.

Using execution requests allows memoized SPs to hook into the dependency tracking mechanism, so that for example, when a proposal is made to a random choice in the body of a memoized SP that affects the value that would be returned, all applications of the SP with the same input argument are notified so that they can propagate the new value. This behavior can be controlled by the SP defining an additional kernel `propagating_kernel`, which is used as an optimization by the dependency trace implementation.

Figure 5.8 shows an example model program using memoization to define the hidden state sequence for an HMM. `x(t)` is a memoized value that refers to the state at the t-th timestep. In this way, memoization

```
define example_beta_bern = (make_beta_bern, infer) -> {
  run_in({
    assume coin = ${make_beta_bern}(0.5, 0.5);
    let dataset = [
      true, true, true, true, true,
      false, false, true, true,
      false, true, false, true
    ];
    weights <- for_each(dataset, (x) -> {
      infer();
      weight <- { observe coin() = x };
      return (weight)
    });
    return (sum(weights))
  }, flat_trace())
};
```

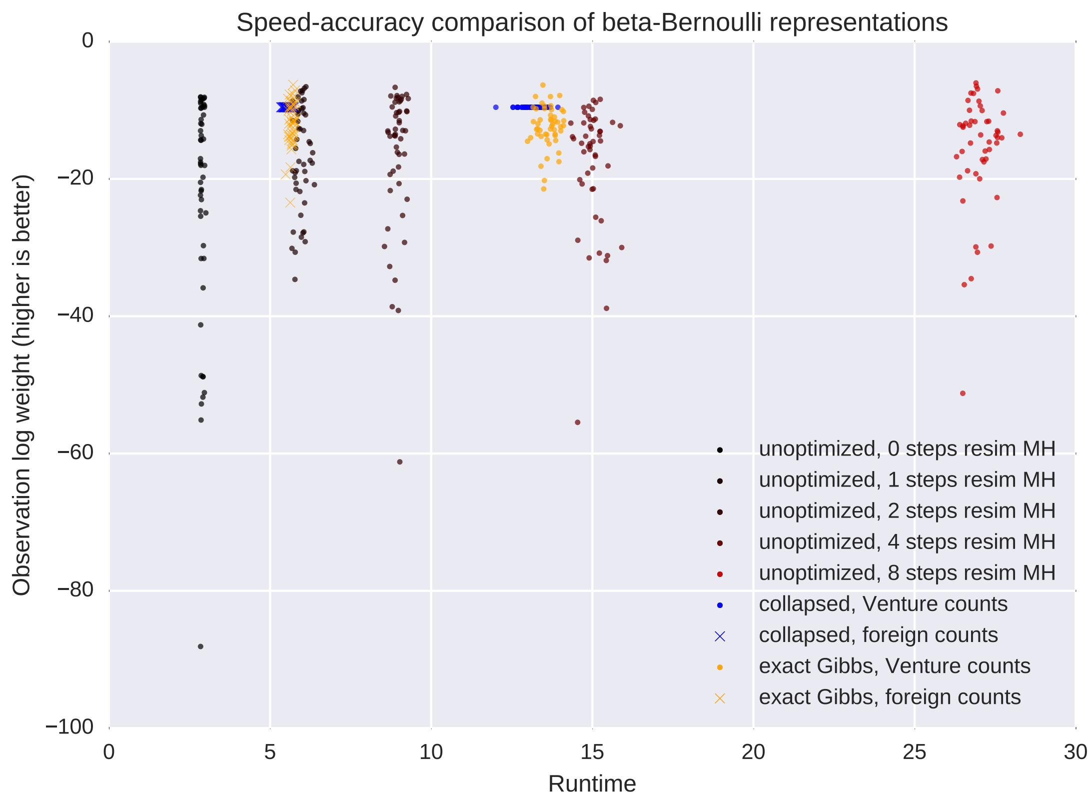

Figure 5.3: Top: An example application of a beta-Bernoulli model. A coin with unknown weight is flipped five times. The flips are observed and incorporated one by one and the resulting log likelihood increments are retained. Bottom: Plot of runtime versus log likelihood weight, varying the implementation of the beta-Bernoulli procedure and the amount of inference performed between each observation. Note that the collapsed implementation obtains the true marginal likelihood of the data under the model, while the others produce noisy approximations that become more accurate with better inference.

```
define make_beta_bern_collapsed = (a, b) -> {
  make_elementary_sp((a, b) -> {
    dict(['state', counter(['N', 'K'])],
         ['simulate', () ~> {
           N <- counter_get('N');
           K <- counter_get('K');
           return (flip((a + K) / (a + b + N)))
         }],
         ['log_density', (x) -> {
           N <- counter_get('N');
           K <- counter_get('K');
           let numerator = if (x) { log(a + K) } else { log(b + N - K) };
           let denominator = log(a + b + N);
           return (numerator - denominator)
         }],
         ['incorporate', (x) -> {
           counter_incr('N');
           if (x) { counter_incr('K') } else { pass };
         }],
         ['unincorporate', (x) -> {
           counter_decr('N');
           if (x) { counter_decr('K') } else { pass };

         }])
  })
};
```

Figure 5.4: An implementation of the collapsed beta-Bernoulli model as a stateful procedure that maintains sufficient statistics N, the number of times the procedure was applied, and K, the number of times that the procedure returned true. The implementation uses a generic `counter` data structure for its mutable state, which exposes read, increment, and decrement operations.

```
// pure VentureScript counter implementation

define counter = (keys) -> {
  run_in({
    for_each(keys, (key) -> {
      assume $key = poisson(1);
      counter_set(key, 0)
    });
    clone_trace()
  }, flat_trace())
};

define counter_set = (key, val) -> {
  env <- global_env();
  addr <- find_symbol(env, key);
  set_value_at(addr, val);
};

define counter_get = (key) -> {
  env <- global_env();
  addr <- find_symbol(env, key);
  value_at(addr)
};

define counter_incr = (key) -> {
  val <- counter_get(key);
  counter_set(key, val + 1);
};

define counter_decr = (key) -> {
  val <- counter_get(key);
  counter_set(key, val - 1);
};
```

```
## foreign Python counter implementation

class CounterState(object):
  def __init__(self):
    self.counts = {}

  def get(self, key):
    return self.counts.get(key, 0)

  def set(self, key, val):
    self.counts[key] = val

  def incr(self, key):
    self.counts[key] = self.counts.get(key, 0) + 1

  def decr(self, key):
    self.counts[key] = self.counts.get(key, 0) - 1

register_state_type("counter", CounterState, {
  "counter_get": trace_action("get", [t.Symbol], t.Number),
  "counter_set": trace_action("set", [t.Symbol, t.Number], t.Nil),
  "counter_incr": trace_action("incr", [t.Symbol], t.Nil),
  "counter_decr": trace_action("decr", [t.Symbol], t.Nil),
})
```

Figure 5.5: Supporting definitions for the collapsed beta-Bernoulli procedure. Top: The counter can be emulated in VentureScript using a trace, with increment and decrement operations implemented using inference programming primitives. Bottom: Alternatively, the counter operations can be implemented as methods of a custom Python class and bound into Venture as foreign primitives.

```
define make_beta_bern_uc_gibbs = (a, b) -> {
  make_elementary_sp((a, b) -> {
    dict(['state', counter(['N', 'K', 'theta'])],
         ['initialize', (trace_handle) -> {
           let theta ~ beta(a, b);
           counter_set('theta', theta);
         }],
         ['simulate', () -> {
           theta <- counter_get('theta');
           return (flip(theta))
         }],
         ['log_density', (x) -> {
           theta <- counter_get('theta');
           return (if (x) { log(theta) } else { log(1 - theta) })
         }],
         ['incorporate', (x) -> {
           counter_incr('N');
           if (x) { counter_incr('K') } else { pass };
         }],
         ['unincorporate', (x) -> {
           counter_decr('N');
           if (x) { counter_decr('K') } else { pass };
         }],
         ['infer_latent_conjugate_gibbs', () -> {
           N <- counter_get('N');
           K <- counter_get('K');
           let new_theta ~ beta(a + K, b + N - K);
           counter_set('theta', new_theta);
         }])
  })
};

// custom meta-program usage example
define example_using_custom_metaprogram = () -> {
  run_in({
    assume coin = make_beta_bern_uc_gibbs(0.5, 0.5);
    observe coin() = true;
    // ... add more observations
    sp_address <- find_symbol('coin');
    sp <- value_at(sp_address);
    invoke_metaprogram_of_sp(sp, 'infer_latent_conjugate_gibbs', [])
  }, flat_trace())
};
```

Figure 5.6: An uncollapsed implementation of the beta-Bernoulli model as a stateful procedure with a conjugate Gibbs kernel. Like the collapsed version from Figure 5.4, it maintains sufficient statistics of its applications, which are used here to sample θ exactly from its conjugate posterior. This Gibbs kernel is provided as a custom meta-program of the SP, which can be invoked as part of an inference program.

defines an infinite lazy sequence of random variables, which is instantiated only as necessary.

6 Discussion

This thesis described a new prototype implementation of Venture, a probabilistic meta-programming platform that addresses fundamental limitations in expressiveness of other probabilistic programming languages. With Venture, meta-programs for describing probability density functions and performing inference can be written as ordinary user-space code. Also, the language is sufficiently expressive to allow foreign primitives and in-language compound procedures to represent what appears to be the same broad class of probabilistic objects. These features were only partially realized in earlier versions of Venture. The resulting flexibility has been illustrated by showing how to give user-space implementations of objects and operations that had to be built in to other probabilistic languages.

One limitation of Venture is that it currently lacks both (i) a simple, meta-circular interpreter and (ii) a formal semantics. Both of these are complementary approaches to building incomplete models of the behavior of Venture. Each could serve to clarify the design of the language, aid in performance engineering, and support the development of more sophisticated meta-programs for modeling and inference. Both approaches could also help reveal design flaws and implementation bugs, and provide a concrete basis on which to evaluate potential remedies. A meta-circular implementation of dependency traces and stochastic regeneration also could lead to powerful tools for debugging. Consider that a Venture programmer could then use modeling and inference to provide machine assistance for finding unlikely edge cases that are indicative of bugs, by asking "what random choices could have led this inference meta-program to manifest this particular model program trace?"

Performance engineering is a key area for future work. The current prototype implementation of VentureScript is slower than several previous versions of Venture. This is not a fundamental barrier for practical applications, but it does impede applied work due to slow development cycles. The fact that VentureScript is higher-order and maintains symmetries between native code and VentureScript means that incremental compilation—perhaps with a VentureScript-to-VentureScript transformation pass in addition to a VentureScript-to-native-code compiler—could be an appealing tactic. For example, many models have relatively static structure, so that Venture's generic subproblem construction and regeneration algorithms always perform the similar traversal on the nodes in the trace graph. A specializing compiler could eliminate that traversal, resulting in a more efficient program is specialized to that model and inference subproblem, using lower-level trace operations to directly read and mutate the trace and compute the appropriate likelihood ratio.

Another key area for future work is in the development of domain-specific languages and libraries for probabilistic modeling and inference. VentureScript, currently the only built-in language, implements specific

```
define mem = (f) -> {
  make_sp((f) -> {
    dict(['state', counter()],
         ['apply', (trace_handle, app_id, x) -> {
           ref_count <- counter_get(x);
           y <- with(trace_handle, {
             addr <- request_address(x);
             if (ref_count > 0) {
               value_at(addr)
             }
             else {
               let exp = [| f($x) |];
               let env = extend_environment(
                 get_empty_environment(), 'f', address_of(f));
               eval_request(addr, exp, env)
             }
           });
           counter_incr(x);
           return (y)
         }],
         // ... kernel definitions broken out and shown below
        )
  })
};
```

```
        // ...
        ['proposal_kernel', (trace_handle, app_id) -> {
          return (dict(
            ['extract', (y, x) -> {
              counter_decr(x);
              ref_count <- counter_get(x);
              with(trace_handle, {
                if (ref_count == 0) {
                  addr <- request_address(x);
                  uneval_request(addr);
                } else { pass };
                return (pair(0, nil))
              })
            }],
            ['regen', (x) -> {
              ref_count <- counter_get(x);
              y <- with(trace_handle, {
                addr <- request_address(x);
                if (ref_count > 0) {
                  value_at(addr)
                }
                else {
                  let exp = [| f($x) |];
                  let env = extend_environment(
                    get_empty_environment(),
                    'f', address_of(f));
                  eval_request(addr, exp, env)
                }
              });
              counter_incr(x);
              return (pair(0, y))
            }],
            ['restore', (x, trace_fragment) -> {
              ref_count <- counter_get(x);
              y <- with(trace_handle, {
                addr <- request_address(x);
                if (ref_count > 0) {
                  value_at(addr)
                }
                else {
                  restore_request(addr)
                }
              });
              counter_incr(x);
              return (pair(0, y))
            }]))
        }],
        // ...
```

```
        // ...
        ['propagating_kernel', (trace_handle, app_id, parent) -> {
          addr <- with(trace_handle, {
            x <- value_at(subexpression(1, app_id));
            request_address(x)
          });
          if (addr == parent) {
            return (dict(
              ['extract', (y, x) -> {
                let trace_fragment = y;
                return (pair(0, trace_fragment))
              }],
              ['regen', (x) -> {
                y <- with(trace_handle, {
                  value_at(addr)
                });
                return (pair(0, y))
              }],
              ['restore', (x, trace_fragment) -> {
                let y = trace_fragment;
                return (y)
              }]))
          }
          else {
            return (nil)
          }
        }]
        // ...
```

Figure 5.7: Implementation of memoization as a user-space procedure, using an external mutable counter to implement reference counting, and `eval_request` to make traced calls to the original unmemoized SP `f`.

```
define example_memoized_hmm = () -> {
  run_in({
    assume x = mem((t) -> {
      if (t == 0) { flip() }
      else {
        flip(if (x(t - 1)) { 0.9 } else { 0.1 })
      }
    });
    assume y = (t) -> {
      flip(if (x(t)) { 0.7 } else { 0.3 })
    };
    generate list(y(1), y(2), y(4), y(5), y(7), y(8))
  }, blank_trace())
};
```

Figure 5.8: An implementation of an HMM with binary states and observations, using mem to memoize the state sequence.

hypotheses about how to compositionally specify models and inference schemes. It provides a way for users to annotate models with meta-data, construct meaningful subproblems using this meta-data, and solve these subproblems using combinations of generic inference library routines and custom inference tactics. VentureScript relies heavily on the conventions of the stochastic procedure interface, but is only one point in the design space, and also these conventions themselves may not be optimal. At minimum, the conventions need to be expanded to define best practices for adding other meta-programs such as gradient information, which are key for optimization-based inference as well as variational techniques. It will also be important to explore static checking for inference programs, and to attempt to develop inference libraries that come with appropriate formal guarantees of soundness or completeness.

Another area for future work is in the extensions needed to use VentureScript inference programs on models defined in foreign probabilistic programming languages. Consider Stan [1] models. After compilation, the Stan runtime effectively maintains an internal trace that is equipped with a fast gradient with respect to the latent parameters. In principle, this could be packaged as a stochastic procedure, or perhaps even a custom trace type. VentureScript could then be used to extend Stan models with discrete latent variables, latent data structures, and objects from Bayesian non-parametrics. VentureScript could also be used to specify hybrid inference strategies that build on both general-purpose Monte Carlo methods from Venture and optimized gradient-based inference from the Stan runtime. In fact, the authors are currently involved in the early stages of developing a new version of Picture [4] that integrates with Venture in this way.

This thesis has shown that it is possible to build a probabilistic programming platform in which a single higher-order language is used to represent random variables, density functions, and inference algorithms. The key ideas involve augmenting a Lisp to support (i) reifying and reflecting on a program's behavior and (ii) linking probabilistic programs and meta-programs into a single package. This focus on reflective meta-

programming leads to greater flexibility and extensibility as compared to other probabilistic programming platforms. It also may suggest new ways to understand the fundamental technical challenges of probabilistic programming. Consider the UNIX family of operating systems. UNIX is often used to assemble and maintain collections of interacting programs, including programs that monitor and modify the runtime behavior of one another. These programs communicate via mechanisms such as the shared filesystem as well as explicit callbacks. Key aspects of the internal behavior and mutable state from a running UNIX program can be inspected by reflective mechanisms such as `strace` and `gdb`. Several of these mechanisms have close parallels to the facilities for reflection and meta-programming in Venture, with heap-allocated memory in UNIX playing a role that is analogous to the probabilistic execution trace. Ideas from UNIX performance engineering may thus be relevant for Venture. It also may be that ideas from operating systems and other forms of system software, not just programming languages and mono-lingual programming systems, can help us formalize and automate key aspects of probabilistic modeling and inference.

References

[1] Bob Carpenter, Andrew Gelman, Matt Hoffman, Daniel Lee, Ben Goodrich, Michael Betancourt, Michael A. Brubaker, Jiqiang Guo, Peter Li, and Allen Riddell. Stan: A probabilistic programming language. *Journal of Statistical Software*, 2016.

[2] Yutian Chen, Vikash Mansinghka, and Zoubin Ghahramani. Sublinear-time approximate mcmc transitions for probabilistic programs. 2014.

[3] Noah D. Goodman, Vikash K. Mansinghka, Daniel M. Roy, Keith Bonawitz, and Joshua B. Tenenbaum. Church: a language for generative models. In David A. McAllester and Petri Myllymäki, editors, *UAI 2008, Proceedings of the 24th Conference in Uncertainty in Artificial Intelligence, Helsinki, Finland, July 9-12, 2008*, pages 220–229. AUAI Press, 2008.

[4] Tejas D. Kulkarni, Pushmeet Kohli, Joshua B. Tenenbaum, and Vikash K. Mansinghka. Picture: A probabilistic programming language for scene perception. In *IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2015, Boston, MA, USA, June 7-12, 2015*, pages 4390–4399. IEEE Computer Society, 2015.

[5] David J. Lunn, Andrew Thomas, Nicky Best, and David J. Spiegelhalter. WinBUGS - A bayesian modelling framework: Concepts, structure, and extensibility. *Statistics and Computing*, 10(4):325–337, 2000.

[6] Vikash K. Mansinghka, Daniel Selsam, and Yura N. Perov. Venture: a higher-order probabilistic programming platform with programmable inference. *CoRR*, abs/1404.0099, 2014.

[7] Vikash K. Mansinghka, Richard Tibbetts, Jay Baxter, Patrick Shafto, and Baxter Eaves. BayesDB: A probabilistic programming system for querying the probable implications of data. *CoRR*, abs/1512.05006, 2015.

[8] George Marsaglia and Wai Wan Tsang. A simple method for generating gamma variables. *ACM Trans. Math. Softw.*, 26(3):363–372, September 2000.

[9] Brian Milch, Bhaskara Marthi, Stuart J. Russell, David Sontag, Daniel L. Ong, and Andrey Kolobov. BLOG: probabilistic models with unknown objects. In Leslie Pack Kaelbling and Alessandro Saffiotti, editors, *IJCAI-05, Proceedings of the Nineteenth International Joint Conference on Artificial Intelligence, Edinburgh, Scotland, UK, July 30 - August 5, 2005*, pages 1352–1359. Professional Book Center, 2005.

[10] Brian Milch, Bhaskara Marthi, David Sontag, Stuart J. Russell, Daniel L. Ong, and Andrey Kolobov. Approximate inference for infinite contingent bayesian networks. In Robert G. Cowell and Zoubin

Ghahramani, editors, *Proceedings of the Tenth International Workshop on Artificial Intelligence and Statistics, AISTATS 2005, Bridgetown, Barbados, January 6-8, 2005*. Society for Artificial Intelligence and Statistics, 2005.

[11] Avi Pfeffer. IBAL: A probabilistic rational programming language. In Bernhard Nebel, editor, *Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence, IJCAI 2001, Seattle, Washington, USA, August 4-10, 2001*, pages 733–740. Morgan Kaufmann, 2001.

[12] Lawrence R. Rabiner. Readings in speech recognition. chapter A Tutorial on Hidden Markov Models and Selected Applications in Speech Recognition, pages 267–296. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1990.

[13] Taisuke Sato and Yoshitaka Kameya. PRISM: A language for symbolic-statistical modeling. In *Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence, IJCAI 97, Nagoya, Japan, August 23-29, 1997, 2 Volumes*, pages 1330–1339. Morgan Kaufmann, 1997.

[14] David Wingate, Andreas Stuhlmüller, and Noah D. Goodman. Lightweight implementations of probabilistic programming languages via transformational compilation. In Geoffrey J. Gordon, David B. Dunson, and Miroslav Dudík, editors, *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics, AISTATS 2011, Fort Lauderdale, USA, April 11-13, 2011*, volume 15 of *JMLR Proceedings*, pages 770–778. JMLR.org, 2011.