Probabilistic programming with programmable inference

Probabilistic Programming with Programmable Inference

Vikash K. Mansinghka
Massachusetts Institute of Technology
USA
vkm@mit.edu

Ulrich Schaechtle
Massachusetts Institute of Technology
USA
ulli@mit.edu

Shivam Handa
Massachusetts Institute of Technology
USA
shivam@mit.edu

Alexey Radul
Massachusetts Institute of Technology
USA
axch@mit.edu

Yutian Chen
Google DeepMind
UK
yutian.chen.research@gmail.com

Martin Rinard
Massachusetts Institute of Technology
USA
rinard@csail.mit.edu

Abstract

We introduce inference metaprogramming for probabilistic programming languages, including new language constructs, a formalism, and the first demonstration of effectiveness in practice. Instead of relying on rigid black-box inference algorithms hard-coded into the language implementation as in previous probabilistic programming languages, inference metaprogramming enables developers to 1) dynamically decompose inference problems into subproblems, 2) apply inference tactics to subproblems, 3) alternate between incorporating new data and performing inference over existing data, and 4) explore multiple execution traces of the probabilistic program at once. Implemented tactics include gradient-based optimization, Markov chain Monte Carlo, variational inference, and sequental Monte Carlo techniques. Inference metaprogramming enables the concise expression of probabilistic models and inference algorithms across diverse fields, such as computer vision, data science, and robotics, within a single probabilistic programming language.

CCS Concepts • **Mathematics of computing → Probabilistic inference problems**; • **Software and its engineering →** *Formal language definitions*;
Keywords Probabilistic Programming, Semantics, Inference

ACM Reference Format:
Vikash K. Mansinghka, Ulrich Schaechtle, Shivam Handa, Alexey Radul, Yutian Chen, and Martin Rinard. 2018. Probabilistic Programming with Programmable Inference. In *Proceedings of 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'18).* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3192366.3192409

PLDI'18, June 18–22, 2018, Philadelphia, PA, USA

ACM ISBN 978-1-4503-5698-5/18/06.
https://doi.org/10.1145/3192366.3192409

1 Introduction

Probabilistic modeling and inference are becoming central computational tools across a broad range of fields [10, 12, 21, 29, 35, 42]. To better support this increasingly prominent class of computations, researchers have designed and implemented probabilistic programming languages, which provide direct support for defining probabilistic models and performing probabilistic inference within the language [2, 13–16, 20, 24, 27, 43, 45]. This stands in contrast to standard practice in probabilistic modeling and inference, in which programmers directly implement models and inference algorithms using ordinary programming language constructs.

Practitioners today know that no one approach works well (or even at all) for all problems. Effective inference requires matching the characteristics of the inference algorithm to the functional requirements, modeling assumptions, and even the data. Over the last several decades the field has developed a wide range of inference algorithms as well as a body of knowledge that matches these algorithms to problems from a diverse range of fields. Prominent themes include particle methods that explore multiple solutions at once and hybrid strategies that break inference problems down into subproblems and apply specialized inference algorithms as appropriate to each subproblem [1, 9, 46].

Current probabilistic programming languages, however, typically provide a small set of black-box inference strategies hard-coded into the implementation. This limitation renders current probabilistic programming languages unsuitable for most potential applications of probabilistic inference.

1.1 Programmable Inference

We introduce novel inference metaprogramming constructs for custom inference algorithms. These constructs enable probabilistic programmers to 1) dynamically decompose inference problems into subproblems, 2) apply exact and/or approximate inference tactics to fully or partially solve these inference subproblems, 3) alternate between incorporating new data and performing inference over existing data, and 4) explore multiple execution traces of the probabilistic program at once.

We have implemented these constructs in the context of the Venture probabilistic programming language [24]. Experience using these constructs to implement probabilistic modeling and inference algorithms in a range of fields, including probabilistic robotics [7, 31], computer vision [26], inverse planning [6], geophysics [36], and data science [5, 25, 38] highlights the effectiveness of inference metaprogramming in this context.

1.2 Inference Metaprogramming Concepts

Most probabilistic programming languages support two types of code: *generative code* and *observation code*. The stochastic choices in the generative code define a probability distribution over *program execution traces*. These traces record all the stochastic choices that the program makes when it runs. Observation code links these stochastic choices to the observed data. Together, the generative and the observation code can be viewed as inducing a Bayesian posterior distribution that concentrates on program execution traces that are likely in light of the prior modeling assumptions encoded in the generative code and the observed data.

We introduce *inference code* for writing *inference metaprograms*, which explictly specify how to find generative code executions that are likely in light of the constraints from the observation code. While it is sometimes possible for metaprograms to sample program executions from the exact Bayesian posterior, in practice most metaprograms deliver approximate samples that nevertheless satisfy the functional requirements of the application.

Inference metaprogramming constructs enable developers to identify subproblems by selecting *target* stochastic choices from the generative code. Corresponding *absorbing* stochastic choices insulate the remaining computation from changes to the target stochastic choices. *Resampling* constructs select likely traces from sets of traces that execute the program together in lock step. All of this code interoperates seamlessly with standard code from the underlying probabilistic programming language.

Programmable inference enables programmers to express, evaluate, and deploy a broad range of probabilistic inference and modeling algorithms, including state of the art inference algorithms in widespread use as well as novel custom/hybrid algorithms that deliver accuracy and performance tradeoffs tailored specifically for the problem at hand.

1.3 Soundness

Several notions of soundness are relevant in practice. Exact Bayesian inference algorithms generate traces according to the exact Bayesian posterior distribution [3, 13]. The stochastic choices define the prior and the constraints define the posterior given the prior. Maximum a posteriori (MAP) inference delivers traces that maximize the posterior probability density [19]. Maximum likelihood inference delivers a trace that maximizes the likelihood of the data, without taking the prior probability density of the trace into account [23].

Obtaining any of these notions of soundness is computationally intractable for many problems of interest [4]. Therefore, practitioners routinely use either 1) asymptotically sound techniques, such as Markov chain and sequential Monte Carlo inference [1, 2, 9, 13, 43, 45], or 2) unsound techniques, such as variational inference [20, 44, 46]. Asymptotically sound algorithms generate traces from a distribution that converges to the exact Bayesian posterior as some parameter increases without limit. In Markov chain Monte Carlo algorithms, this number is the number of Markov chain iterations, while in sequential Monte Carlo algorithms, this number is the number of independent particles or replicates. Unsound algorithms are designed to have some conceptual connection to the Bayesian posterior that is intended to capture the key properties needed for the given application.

These soundness questions apply to both 1) the tactics used for inference subproblems and 2) the inference metaprogram taken as a whole. Venture's built-in library of inference tactics includes sound, asymptotically sound, and unsound algorithms. These tactics can be used to create sound, asymptotically sound, or unsound inference metaprograms.

1.4 Contributions

This paper makes the following contributions:

- **Inference Metaprogramming:** It introduces the concept of *inference metaprogramming*, which enables probabilistic programmers to specify customized probabilistic inference algorithms as appropriate for the probabilistic inference problem at hand.
- **Language Constructs:** It presents novel inference metaprogramming constructs that interoperate seamlessly with standard probabilistic programming constructs. These new constructs enable developers to dynamically decompose inference problems into subproblems, apply custom inference tactics to subproblems, and explore multiple execution traces concurrently.
- **Case Studies:** It presents case studies that highlight the range of application areas and custom inference strategies that inference metaprogramming supports. A key empirical observation is that different inference metaprograms and strategies are appropriate for different inference problems and probabilistic programs.
- **Formalism:** It presents a precise formal characterizion of key concepts in inference metaprogramming.

Inference metaprogramming comprises a fundamental change to the conceptual model of probabilistic programming. The inference algorithms used by practitioners vary widely across fields, problems, and even problem instances, but probabliistic programming languages previously only supported a small set of built-in algorithms. Inference metaprogramming thus eliminates a key obstacle to the widespread adoption of probabilistic programming in practice.

```
// Generative code defining a two-variable
// probabilistic model and associating labels
// to the choices.
assume x = normal(0, 1) #latents:"x";
assume y = normal(0, 1) #latents:"y";
```

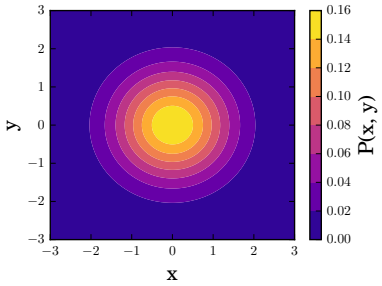
(a) Generative code

```
// Observation code defining a likelihood model, based
// on a Gaussian latent variable, and an observed
// value for this variable.
observe normal(x + y, 1.0) = 3;
```

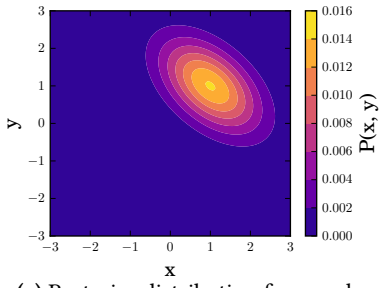
(b) Observation code

```
infer resample(100);
infer reset_to_prior();
repeat(1000, {
  infer gradient_ascent(
    minimal_subproblem(/?latents/*), 0.01)});
repeat(100, {
  infer resimulation_mh(
    minimal_subproblem(
        random_singleton(/?latents/*))))});
```

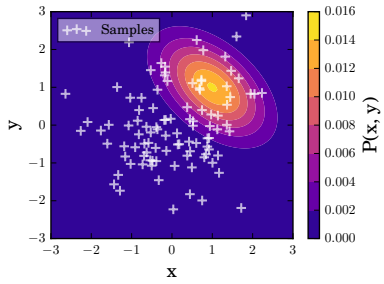
(c) Inference metaprogram



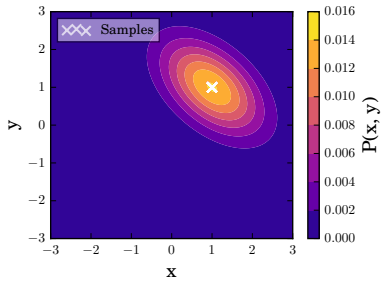
(d) Prior distribution for x and y



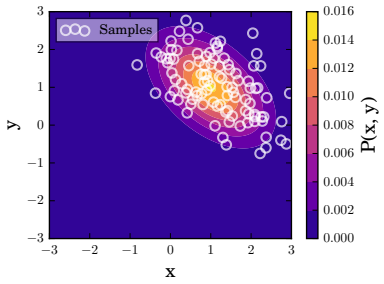
(e) Posterior distribution for x and y



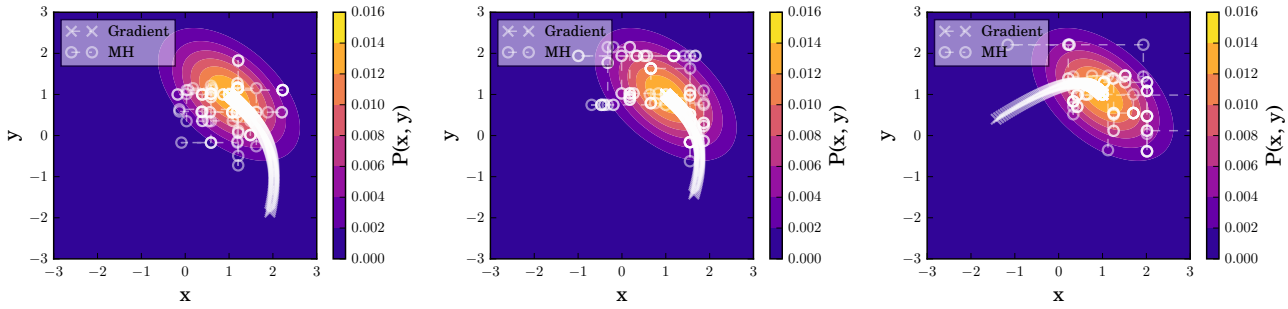
(f) Traces sampled from the prior distribution



(g) Traces after gradient steps



(h) Traces after gradient and SVMH steps



(i) Sequences of trace values for three of the 100 traces, showing how individual traces evolve under the inference metaprogram

Figure 1. Inference metaprogramming example. The generative code (1a) defines a prior distribution over x and y. The observation code (1b) defines the posterior distribution over x and y given the prior. Figure 1d graphically presents the prior; Figure 1e graphically presents the posterior. The goal of probabilistic inference is to convert samples from the prior (1f) into samples from the posterior. The inference metaprogram (1c) first performs gradient steps to move samples from the prior towards the peak of the posterior (1g), then Single-Variable Metropolis-Hastings (SVMH) steps to spread out the samples to more accurately represent the posterior (1h). Figure 1i presents the time evolution of three of the traces. The white crosses represent gradient steps; the white circles represent SVMH steps.

2 Example

We next present an example that illustrates inference metaprogramming in Venture. The goal is to obtain likely values for two latent model variables `x` and `y` that explain a (noisy) observation that the sum of the two variables equals three.

Generative Code: Venture programs often start with a sequence of stochastic choices that define the values of latent model variables. We call this code *generative* code. Figure 1a presents the generative code in our example. Each possible execution of the generative code corresponds to a complete setting of all the latent variables in an associated probabilistic generative model. The `assume` statements in Figure 1a execute stochastic choices that define the values of the `x` and `y` latent model variables. They also associate the stochastic choices with labels: `/?latents/*` names both stochastic choices, `/?latents/?x` names the `x` choice, `/?latents/?y` names the `y` choice, and `random_singleton(/?latents/*)` names a single randomly selected `x` or `y` choice. The inference code uses the labels to identify stochastic choices to include in subproblems.

Together, the stochastic choices in Figure 1a define a conceptual prior probability distribution over possible program executions. Figure 1d graphically presents this prior for the `x` and `y` latent model variables. As illustrated in Figure 1f, executions of the generative code sample from this prior.

Observation Code: Venture programs often next define a likelihood model by executing code that makes additional stochastic choices and associates observed data values with these choices. We call this code *observation* code. Figure 1b presents the observation code in our example. The `observe` statement adds a new stochastic choice — from a Gaussian distribution with mean `x+y` and standard deviation `1.0` — and associates this choice with the observed value 3. Together, the generative and observation code induce a posterior distribution on program execution traces (Figure 1e).

Probabilistic Inference: The next step is to perform probabilistic inference to (typically approximately) convert samples from the prior into samples from the posterior with the goal of obtaining values for the latent model variables that acceptably explain the observed data.

Inference Metaprogram: Figure 1c presents the inference metaprogram. Over the course of its execution, this inference metaprogram evolves a set of traces inside the Venture runtime from an initial set drawn from the prior into sets of traces drawn from different approximations to the posterior.

The metaprogram first executes `resample(100)` command. This command takes the current set of executing traces, each of which has a weight that captures the likelihood of that trace relative to the other executing traces, samples 100 traces randomly from this set in proportion to their weights, then replaces the current set with the sampled set. Here there is only one trace when the resample command executes, so the command simply creates 100 replicas of that trace.

The following `reset_from_prior()` command replaces each of these 100 traces with a new trace drawn from the prior, updating the weights to correspond to the likelihood of the observations under the new values, i.e., the log probability density of the `normal(x+y,1.0)` distribution at the value 3. The white crosses in Figure 1f present the `x`, `y` values from this set of traces overlaid on the prior. The samples match the prior but not the posterior.

The remainder of the metaprogram updates the set of traces by executing two different probabilistic inference algorithms in sequence. The first uses gradient ascent to update the values of `x` and `y` in a direction that locally increases the probability density of the execution traces. The algorithm selects stochastic choices, creates an inference subproblem from those choices, and applies a gradient search inference tactic to update the subtrace internal to that subproblem.

Figure 1g shows the resulting set of traces, again with the white crosses presenting the `x`, `y` values of each trace. These traces concentrate in the center of the probability density of the posterior distribution. Note that the inference metaprogram executed so far is highly unsound from the subjective Bayesian viewpoint, i.e., the inferred trace distribution does not correspond closely to the full posterior distribution. Also note that for some applications, this kind of unsound approximation is more useful than a sound result in which traces are randomly drawn from the full posterior distribution — some applications are best served with just the single highest probability density trace selected from a large set of candidate traces.

The second algorithm executes after the first. It repeatedly applies a variant of the Metropolis-Hastings algorithm that 1) randomly selects a single stochastic choice from among the `/?latents/*` choices, 2) constructs a minimal subproblem from this choice (i.e., the smallest subproblem that includes this choice, see Section 4), 3) proposes a new subtrace for this subproblem by reexecuting the program fragment corresponding to this subproblem, and 4) accepts or rejects the new subtrace according to the Metropolis-Hastings rule. We call this algorithm *Single-Variable Metropolis-Hastings* (SVMH). Figure 1h shows the resulting set of traces. The set now represents a good approximation to the full posterior distribution, not just its highest probability density regions.

Figures 1i presents three examples of the `x`, `y` trajectories generated by the execution of the inference metaprogram in Figure 1c. Each trajectory starts with a sequence of white crosses presenting the sequence of `x` and `y` values generated during the gradient ascent inference algorithm. The sequence shows the path from each different sampled prior curving into the center of the posterior probability density function.

This sequence is followed by a sequence of white circles presenting the sequence of `x` and `y` values from the Metropolis-Hastings algorithm. These sequences show the traces spreading out to more accurately represent the posterior via the Metropolis-Hastings algorithm.

For the gradient-ascent algorithm, the specified subproblem includes the stochastic choices for both latent variables `x` and `y`. These are the *target* stochastic choices for the subproblem. The *absorbing* stochastic choice is the choice made by the expression `normal(x+y,1.0)`. We call this choice the absorbing choice because it absorbs the changes to `x` and `y` — the inference does not change the value of this choice, which is constrained to equal 3 in all executions.

For the SVMH algorithm, the subproblem is randomly generated at each step, with the target set corresponding to either the choice that generated the value of `x` or the choice that generated the value `y`. The value of the other `x`, `y` choice remains unchanged because it lies outside the subproblem. The absorbing choice is again the stochastic `normal(x+y,1.0)` choice.

3 Case Studies

Inference metaprogramming in Venture has been used to solve problems in diverse fields, including probabilistic robotics [7, 31], computer vision [26], inverse planning [6], geophysics [36], and data science [5, 25, 38].

These applications have diverse requirements. For example, the robotics application from [7, 31] is an instance of simultaneous localization and mapping (SLAM), in which a stream of control signals and sensor data is interpreted via inference in a probabilistic model derived from quantitative models of the robot's motor system and sensors. Accurate online inference requires small updates for each new sensor reading; these can be done via sequential Monte Carlo with custom Metropolis-Hastings updates. In contrast, the application to automatic Bayesian model discovery for multivariate data tables [5] uses a prior over a broad class of probabilistic model structures and parameters. The inference metaprogram for solving this problem implements a Markov chain Monte Carlo algorithm for searching this space to find plausible models given the data. In this metaprogram, the subproblem decomposition is itself dynamic, because the model structure changes over the course of model discovery.

This paper uses four case studies to illustrate the range of inference algorithms that can be expressed via inference metaprogramming. Each case study presents a probabilistic model via generative code, an inference problem defined by additional observation code, and multiple inference metaprograms that implement approximation algorithms for solving each model and/or inference problem.

3.1 Modeling Time Series Data

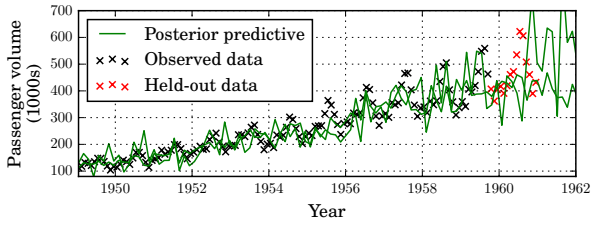
We present a state-of-the-art application of probabilistic programming: Bayesian program synthesis for automatically discovering models of time-series data [39]. The model takes the form of a program, in the language of Gaussian process models [23, 33], that models the input time series as a composition of primitive structural components such as linear trends, periodic variation, white noise, and change points.

```
repeat(n, { infer resimulation_mh(
        minimal_subproblem(random_singleton(/*))))});
```

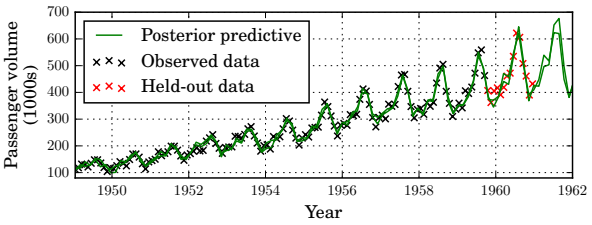
Figure 2. Generic SVMH algorithm for automatic model discovery from time-series data.

```
repeat(n, { repeat(5, { infer resimulation_mh(
   minimal_subproblem(random_singleton(/?structure/*))))});
  infer lbfgs_optimize(minimal_subproblem(/?hypers/*))});
```

Figure 3. Custom inference metaprogram for automatic model discovery from time-series data based on SVMH with block updates and LBFGS gradient-based optimization updates for numerical hyperparameters.



(a) Results for automatic time-series model discovery with a single-variable Metropolis-Hastings inference metaprogram from Figure 2.



(b) Results for automatic time-series model discovery with a custom metaprogram (Figure 3) that uses block updates and an LBFGS gradient-based search to simultaneously update all numerical hyperparameters.

Figure 4. Discovered time-series models with different inference metaprograms.

Generative Code: The generative code (not shown) uses a probabilistic context-free grammar to generate a random Gaussian process covariance kernel. This kernel is specified using a data structure that can be thought of as a program in the domain-specific language of Gaussian process models. The primitives in this language include a constant kernel (i.e., a constant covariance function), a linear kernel, a white noise kernel, a squared exponential kernel, and a periodic kernel. The language combines these primitives with kernel multiplication, addition, and a change-point kernel switch function that smoothly replaces one kernel with another. Each model in this language defines a function from time to the value of the time series at each point in time.

All of the kernels have hyperparameters that determine quantitative characteristics such as the location of a change point or the amplitude of white noise. The generative code

that implements the probabilistic context-free grammar defines a prior probability distribution over Gaussian process time-series models. The generative code makes two classes of stochastic choices: *discrete structural choices* that determine the structure of the generated model (each of these selects a production from the probabilistic context-free grammar as the generative code builds the model) and *numerical choices* that generate continuous kernel hyperparameters.

Observation Code: The observation code (not shown) reads the observed time-series data, then executes `observe` statements that constrain the model to produce the same values as the observed time-series data.

Inference Code: Figure 2 presents a generic SVMH metaprogram (Section 2) for solving this problem. Figure 3 presents a custom metaprogram that alternates between iteratively applying the SVMH algorithm to a randomly selected structural choice and applying a Limited Memory Broyden-Fletcher-Goldfarb-Shanno (LBFGS) gradient-based search to simultaneously optimize all the hyperparameters. The code uses `/?structure/*` to name all of the discrete structural choices and `/?hypers/*` to name all of the continuous hyperparameter choices.

It is not possible to apply the LBFGS algorithm to optimize the discrete structural stochastic choices; the algorithm works only for numerical choices with computable gradients. This fact highlights how inference metaprogramming supports specialized inference algorithms that apply only to selected subproblems within the larger inference problem.

Results: These metaprograms have been used to obtain models for a variety of time-series data, including solar radiation, unemployment, and sulphuric acid production data [39]. We present results for a data set of monthly totals of international airline passengers from 1949 through 1961. We train on the first 90% of the data, holding out the remaining 10% to evaluate the prediction accuracy. We run each metaprogram repeatedly for 30 minutes. Figures 4a and 4b present the two best (i.e., least error on training data) time-series models obtained from the two metaprograms. These models show that the input time-series data contain both a component that linearly increases with time (as air traffic increases over time) and a periodic component corresponding to seasonal variations in traffic (more in summer, less in winter).

The second metaprogram (Figure 3), by combining SVMH for discrete choices and block gradient-based optimization for continuous choices, sometimes succeeds in producing accurate models. Its best models are more accurate than than the best models from the first metaprogram (Figure 2) — they match the training data more accurately and generalize better (they more accurately predict the held out data). This case study highlights how custom inference metaprograms can increase the probability of obtaining adequate approximations to computationally intractable probabilistic modeling and inference problems within a given runtime budget.

This case study can be viewed as an alternative implementation of the Automatic Bayesian Covariance Discovery (ABCD) approach from the Automated Statistican system [22]. This Venture program is both more accurate than the original ABCD system and is implemented in over fifty times fewer lines of code [38]. This case study thus highlights the expressive power of inference metaprogramming in concisely describing state-of-the-art algorithms for soving complex probabilistic modeling and inference problems.

3.2 Linear Regression With Outlier Detection

We use inference metaprogramming to solve the problem of simultaneously 1) fitting a line to data while 2) inferring which data points are outliers and should not affect the line.

Generative Code: Figure 5 presents the generative code, which uses `assume` statements to build the model. These statements define values of latent model variables (such as `slope` and `intercept`) by sampling from specified prior distributions. For example, `slope = normal(0,1)` sets `slope` to a random value drawn from a normal distribution with mean `0` and standard deviation `1`. The program randomly generates `slope` and `intercept` for the linear regression, `noise` from a gamma distribution, then sets `outlier_noise` and `outlier_probability` to constant values.

The procedure `model(i,x)` also contains generative code that accepts a data point identifier `i` and input value `x` as arguments and generates an observable `y` value as output. This procedure first uses a Bernoulli stochastic choice, via `is_outlier(i)`, to decide whether each data point `x` should be included in the regression or is an outlier that should be excluded. It then makes a normal-distributed stochastic choice to generate a value `y` for that data point based on its `x` value. If `is_outlier(i) == True`, i.e. `x` is an outlier, this `normal` distribution has mean `0` and standard deviation `25`. Otherwise, the mean is given by the regression line, and the standard deviation is given by the `noise` variable.

The generative code includes labels that make it possible to write custom inference metaprograms that exploit the structure in this problem. Stochastic choices that generate numerical parameters are tagged with the `parameters` label; choices that generate the binary outlier indicators for each data point are tagged with the `outlierindicators` label. These labels reflect the fact that there are two kinds of random choices: numerical choices for the parameters of the regression line and discrete choices that determine if each point should be included in or excluded from the regression. In this situation it is appropriate to use different inference strategies for these different choices. Venture's labeling constructs make it possible to separate these choices into different subproblems, then apply an appropriate inference technique to each subproblem. The choices are also tagged with more specific labels. For example, the outlier assignment choices are labeled with unique integer identifiers corresponding to the identifier for the data point they are

```
assume slope     = normal(0, 1) #parameters:0;
assume intercept = normal(0, 1) #parameters:1;
assume noise     = gamma(1,1)   #parameters:2;
assume outlier_noise   = 25;
assume outlier_probability = 0.1;
assume is_outlier = mem((i) ~> {
  bernoulli(outlier_probability) #outlierindicators:i });

assume model = (i, x) ~> {
    if (is_outlier(i)) {
      normal(0, outlier_noise)
    } else {
      normal(slope * x + intercept, noise)}};
```

Figure 5. Generative code for linear regression with outlier detection.

```
for_each (arange(size(data_xs)), (i) -> {
  observe model($i, ${data_xs[i]}) = data_ys[i]});
repeat(n, { infer resimulation_mh(
    minimal_subproblem(random_singleton(/*))))});
```

Figure 6. Observation and inference code for a generic single-variable Metropolis-Hastings algorithm.

```
for_each (arange(size(data_xs)), (i) -> {
  observe model($i, ${data_xs[i]}) = data_ys[i]});
repeat(n, { repeat(m, {
  infer resimulation_mh(minimal_subproblem(
    random_singleton(/?parameters/*))))});
  for_each (arange(size(data_xs)), (i) -> {
    infer gibbs(minimal_subproblem(
      /?outlierindicators==i)))})});
```

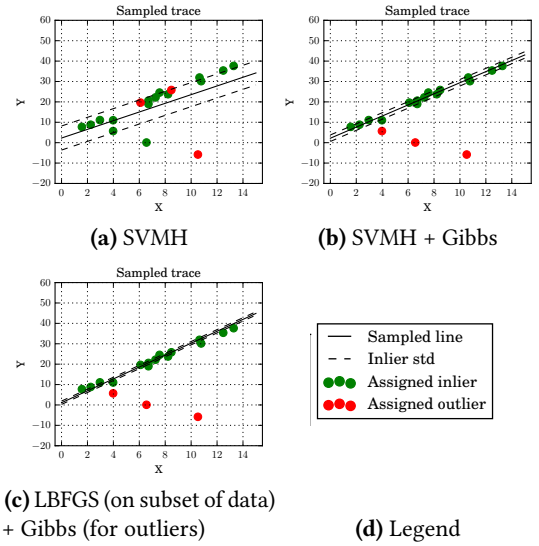
Figure 7. Observation and inference code that uses Gibbs sampling on the outlier indicator variables.

```
define indices = shuffle(size(data_xs));
for_each (arange(k), (i) -> {
  j = indices[i];
  observe model($j, ${data_xs[j]}) = data_ys[j]; });
infer lbfgs_optimize(minimal_subproblem(/?parameters/*));
for_each (arange(k, size(data_xs)), (i) -> {
  j = indices[i];
  observe model($j, ${data_xs[j]}) = data_ys[j] });
for_each (arange(size(data_xs)), (i) -> {
  infer gibbs(minimal_subproblem(
    /?outlierindicators==i))});
```

Figure 8. Observation and inference code for a custom algorithm that fits the regression models to a subset of the data and then updates each outlier indicator once.

associated with (here, `i`). These labels enable outlier indicators for specific data points to be the targets of subproblems for custom inference metaprograms.

Observation Code and Inference Code: We next discuss three different observation code blocks and inference metaprograms for this problem. The first (Figure 6) incorporates the data via `observe` statements `observe model(i,x)=y`, where `x`, `y` is an observed data point with identifier `i`. The constraint `model(i,x)=y` links the stochastic choices that



(a) SVMH **(b)** SVMH + Gibbs

(c) LBFGS (on subset of data) + Gibbs (for outliers) **(d)** Legend

Figure 9. Linear regression in the presence of outliers with different inference strategies.

determine the predicted `y` value to the observed `y` value. It then applies the SMVH algorithm from Section 2.

The second metaprogram (Figure 7) performs `n` steps of a custom inference strategy that improves on the metaprogram in Figure 6 in two ways. First, it alternates between performing `m` SVMH inference steps on the `parameters` stochastic choices and performing inference on the `outlierindicators` stochastic choices. Second, it uses Gibbs sampling to update each of the `outlierindicators`.

The SVMH inference steps resample the numerical `slope`, `intercept`, and `noise` choices, holding the remaining choices constant — the predicted `y` values and outlier decisions do not change during the inference iterations. The goal is to find `slope`, `intercept`, and `noise` choices that better explain the predicted `y` values and current outlier decisions. The target set of each subproblem is the randomly selected `slope`, `intercept`, or `noise` choice (as identified by the `random_singleton(/?numerical/*)` expression.) The absorbing set contains all of the `normal` choices executed by the `model` procedure that apply the linear model to obtain the predicted `y` values (Figure 5).

The Gibbs steps use the `/?outlierindicators/?i` labels to generate and exactly solve a subproblem for each observed data point `x`, `y`. Unlike the generic reexecution proposal from Figure 6, this tactic exactly solves the subproblem by enumerating both possible values for `is_outlier(i)`. It thus takes the fit between each data point and the regression model into account when proposing updates.

Each Gibbs step potentially changes only the outlier decision for the current point, holding the predicted `y` values, the `slope`, `intercept`, and `noise` values, and the other outlier decisions constant. The goal is simply to find an outlier decision for the current point that better explains all of these other choices.

The final inference metaprogram (Figure 8) interleaves observations and inference. It starts by invoking the `model(i,x)` procedure to include k randomly selected data points in the model. It then applies the LBFGS algorithm (discussed in Section 3.1) to only those k points to infer a linear regression line for those points. It then completes the model for the remaining observed data points and applies a Gibbs step to each of the observed data points to obtain outlier decisions that better explain the inferred linear regression line.

Results: Figure 9 presents results from executions with the three inference metaprograms (with k=4 for the third metaprogram). Each graph presents the observed data points, with green points included in the regression and red points excluded. The black line is the inferred regression line (`slope` and `intercept`) and the two dashed lines indicate the inferred noise (`noise`) around the regression line. Both the custom metaprograms correctly identify outliers and deliver an accurate regression line with small predicted noise. In contrast, the generic single-variable Metropolis-Hastings algorithm incorrectly classifies five points and delivers a less accurate regression line.

3.3 Inference In A Hidden Markov Model

This case study is based on the problem of inferring hidden states in a hidden Markov model. Hidden Markov models have been the basis of practical applications in fields such as speech recognition, genetics, natural language processing, and quantitative finance. In this problem class, there is a probabilistic state machine whose state is hidden from the probabilistic programmer. The observed data is a sequence of output symbols that are assumed to be stochastically emitted based on the hidden state that the machine is in at each step of the sequence.

Generative Code: The generative code (not shown) defines the initial state distribution, the transition probabilities between states, and the probability with which each state emits each output symbol. It also defines functions for generating a hidden state sequence and observation sequence according to these probabilities.

Interleaved Observation Code and Inference Code: This case study illustrates the use of inference metaprograms that interleave the observation of data with the application of inference tactics. The first inference metaprogram (Figure 10) replicates the trace `number_traces` times (typically tens of times). It then adds the observations in sequential order, with each trace executing the generative code to sample a hidden state from the prior for each new observation. Each trace calculates an updated trace weight based on the match with the observed data. After the metaprogram loads all of the observations, it executes a resampling inference operation to sample one trace from the current set of traces with probability proportional to the weights for each of the current traces. This inference metaprogram can be viewed as implementing an importance sampling algorithm.

```
infer resample(number_traces);
infer reset_to_prior();
for_each (arange(size(training_data)), (i) -> {
  observe emission(get_state(integer($i)))
          = training_data[i]; });
infer resample(1);
```

Figure 10. Observation code and importance resampling inference code for inferring states in a hidden Markov model.

```
infer resample(number_traces);
infer reset_to_prior();
for_each (arange(size(training_data)), (i) -> {
  observe emission(get_state(integer($i)))
          = training_data[i];
  infer resample(number_traces); });
infer resample(1);
```

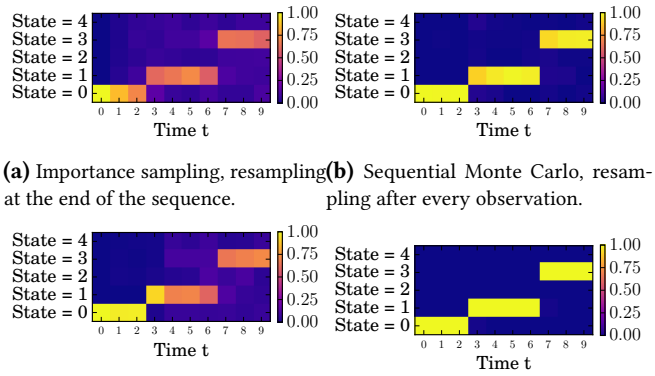
Figure 11. Observation code and sequential Monte Carlo inference code with resampling after every observation.

```
infer resample(number_traces);
infer reset_to_prior();
for_each (arange(k), (i) -> {
    observe emission(get_state(integer($i)))
            = training_data[i]; });
infer resample(number_traces);
for_each (arange(5, 10), (i) -> {
    observe emission(get_state(integer($i)))
            = training_data[i]; });
infer resample(1);
```

Figure 12. Observation code and inference code for sequential Monte Carlo with two resampling steps.

The second inference metaprogram (Figure 11) applies a resampling operation after every observation is added. This corresponds to a sequential importance samping with resampling (SIR) algorithm, a widely used instance of the broader class of sequential Monte Carlo algorithms. The third inference metaprogram (Figure 12) applies only two resampling steps: once after the fifth observation is included, and a second resampling step after all observations have been added. This resampling step replaces the initial traces with new traces that assign high likelihood to the first few states so that computational effort is focused on traces that give good explanations of these first states. This is a simple example of a custom sequential Monte Carlo techinique that cannot be expressed in probabilistic programming languages based on black-box sequential Monte Carlo. This third metaprogram spends less computational effort resampling, at the cost of a reduction in accuracy relative to the second metaprogram.

Results: We ran each inference metaprogram with a range of numbers of traces, such that runtimes varied from 15 to 25 seconds. Figure 13 presents the results. The X axis of each graph plots the time; the Y axis plots the inferred probability estimate that the Markov model is in the corresponding state at that time. We derive the estimates by aggregating results from all of the runs, specifically by counting the number of



(a) Importance sampling, resampling at the end of the sequence.

(b) Sequential Monte Carlo, resampling after every observation.

(c) Sequential Monte Carlo, two resampling steps.

(d) Ground truth probability

Figure 13. Estimates of hidden state probabilities in a hidden Markov model for different inference metaprograms. Ground truth probabilities are included for comparison. Each plot shows estimates of the probability that the hidden Markov model is in each state after each time step. Each column corresponds to a time step; each row corresponds to a state. The shading indicates the probability.

executions that placed the Markov model in each state at each time step.

The results show that importance sampling (Figure 10) delivers the least accurate estimates. Sequential Monte Carlo algorithm resampling after every step (Figure 11) delivers the best accuracy. A custom sequental Monte Carlo algorithm with resampling after two specific steps in the sequence (Figure 12) delivers intermediate accuracies but runs faster than the second metaprogram.

3.4 Inference In A Bayesian Network Model

This case study is a synthetic example designed to share characteristics with a broad class of Bayesian network inference problems. The network topology is a bipartite structure based on Bayesian network models for medical diagnoses. In these networks, a set of unobservable variables representing diseases have causal links that activate variables representing observable symptoms or measurements. The diseases have low prior probabilities. The inference problem is to sample from the probability distribution over diseases given the observable measurements. These kinds of inference problems can be challenging because there are often multiple logically possible explanations of the data.

Generative Code: The generative code (Figure 14) first samples hidden disease variables according to their prior probabilities. It then defines samplers for the noisy-OR distribution that defines the probability with which a symptom is present, given the presence or absence of each parent. Finally, it defines a variable for each symptom. It labels each disease according to the symptoms that it can cause, which allows inference metaprograms to define subproblems that contain

```
assume cause_prior = 0.01;

assume disease_1 ~ bernoulli(cause_prior) #symptomA:0;
assume disease_2 ~ bernoulli(cause_prior) #symptomA:1;
assume disease_3 ~ bernoulli(cause_prior)
  #symptomA:2 #symptomB:0;

assume disease_4 ~ bernoulli(cause_prior) #symptomB:1;
assume disease_5 ~ bernoulli(cause_prior) #symptomB:2;
assume disease_6 ~ bernoulli(cause_prior) #symptomB:3;

assume get_causal_link = (parent) -> {
  if (parent) { 0.01 } // Probability of a leak
  else { 1. }
};
assume get_effect = (parents, effect) -> {
  if (size(parents) == 0) { effect } else {
    get_effect(rest(parents), effect *
      get_causal_link(first(parents)))
  }
};
assume noisy_or = (parents) ~> {
    p_spontaneous = 0.001;
    bernoulli(1 - ((1-p_spontaneous) *
      get_effect(to_list(parents), 1)))
};

assume symptom_A = noisy_or(
                    [disease_1, disease_2, disease_3]);
assume symptom_B = noisy_or(
                    [disease_3, disease_4,
                    disease_5, disease_6]);
```

Figure 14. Generative code for a Bayesian network model.

```
observe symptom_A = True;
observe symptom_B = True;
```

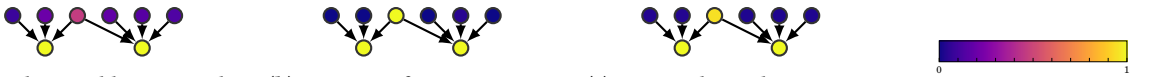
Figure 15. Observation code for a Bayesian network model.

```
repeat(n, {
  infer gibbs(minimal_subproblem(/?symptomA/*));
  infer gibbs(minimal_subproblem(/?symptomB/*));
});
```

Figure 16. Custom blocked Gibbs inference metaprogram for a Bayesian network model.

groups of disease variables that are most strongly coupled, i.e., those disease variables that compete to probabilistically explain a given symptom.

Observation Code: The observation code (Figure 15) is common across all three inference metaprograms. This code constrains the stochastic choice representing each symptom to take on the value `True`. Note that this pattern of data is best explained by the presence of disease #3 alone. This is because 1) disease #3 is sufficient to explain both symptoms (as the model parameters are such that each disease causes its associated symptoms with high probability, and 2) all diseases are assumed to be rare a priori under the generative code, so explanations that invoke two diseases simultaneously are significantly less likely.



(a) Single-variable Metropolis-Hastings **(b)** Custom inference metaprogram. **(c)** Ground truth posterior probabilities. **(d)** Legend: $P(\text{node} = \text{on})$

Figure 17. Inferring probable hidden causes of data according to a Bayesian network model. Panels (a) and (b) show estimates of the marginal posterior probabilities from two different inference metaprograms; panel (c) shows the ground truth. The custom inference metaprogram uses multiple-variable blocked Gibbs sampling with subproblems that match the dependence structure of the Bayesian network.

Inference Code: The first metaprogram implements the generic SVMH algorithm (Section 2). The second implements custom blocked Gibbs sampling that repeatedly 1) samples values for all three causes of symptom A from their posterior distribution given the current setting of all other variables, and then 2) samples values for all four causes of symptom B from their posterior distribution given the new values for causes of symptom A from step 1. The third disease appears in both subproblems since it influences both symptoms.

Results: Figures 17a and 17b present the results from each of the inference metaprograms, compared to ground truth in FIgure 17c. The number of repetitions of both inference metaprograms are chosen to deliver executions that finish in under 20 seconds. We obtain the output probabilities in Figures 17a and 17b by averaging over all traces from each inference metaprogram. The custom inference metaprogram (Figure 16) is significantly more accurate for two reasons. First, it simultaneously updates all variables that are coupled, and second, it updates these variables from their exact posterior. It can thus easily shift which disease is being used to explain each symptom, and find the state where disease #3 is present but no other diseases are present. In contrast, the SVMH algorithm has difficulty switching diseases off, and thus overestimates the probability that two diseases are present and independently causing the observed symptoms.

4 Inference Metaprogramming Formalism

We next precisely define several central concepts in inference metaprogramming. These concepts include subproblem identification, target and absorbing sets, requirements that subproblem inference algorithms must satisfy. While we expect these concepts to be reflected in implementations of probabilistic programming languages that support subproblem inference, the specific realization of the concepts may vary across implementations. We present these concepts with formalisms of two core Venture languages, the *generative* language (which includes only assume and observe statements), and the *inference* language (which also includes infer statements). Full formalization details are available [17].

4.1 Core Venture Languages

We formalize the basic concept of valid Venture traces using the core Venture generative language in Figure 18. A program in this language is an ordered list of assume and observe statements.

Expressions are derived from the untyped lambda calculus augmented with the standard probabilistic programming construct $\text{Dist}(e, l(e_l))$ (a stochastic choice drawn from the distribution Dist with parameter e). Venture allows the programmer to label each stochastic choice using a label $l(e_l)$, which may later be used to specify the stochastic choices in a subproblem.

4.2 Valid Traces

When a generative program executes, it builds up a trace that records the stochastic choices, executed statements, and values of computed expressions. The trace also includes unique identifiers $id \in ID$ for each executed statement, definition, or computed expression. These *id*s are later used to derive a dependence graph used to define subproblems. We formalize traces as unrolled program traces $t \in T$ (Figure 19). Each program p has a set of valid traces $t \in \text{Traces}(p)$ that correspond to valid executions of the program. Each expression, definition, and statement within a trace is augmented with its corresponding computed value and a unique id.

We define the execution, including the generation of valid traces $t \in T$, with a relation $\langle p, \sigma_v, \sigma_{id} \rangle \Rightarrow_s t$, which executes a program p in environment σ_v, σ_{id} to obtain a trace t, where $\sigma_v : Vars \to Vals$ and $\sigma_{id} : Vars \to ID$ [17]. Note that the generative language, which does not include infer statements, defines the set of valid traces — infer statements only mutate one valid trace into another valid trace but do not change the set of valid traces.

4.3 Dependence Graphs $\langle \mathcal{N}, \mathcal{D}, \mathcal{E} \rangle$

Each trace $t \in T$ has a corresponding dependence graph $\langle \mathcal{N}, \mathcal{D}, \mathcal{E} \rangle = \text{Graph}(t)$ that makes the data and existential dependences in t explicit. We use the dependence graph to define valid subproblems. Here $\mathcal{N} : ID \to aE \cup aD \cup aS$ maps an $id \in ID$ to a corresponding augmented expression $ae \in aE$, augmented definition $ad \in aD$, or augmented statement $as \in aS$. dom $\mathcal{N}$ is the set of nodes in the graph.

$\mathcal{D} \subseteq ID \times ID$ are the data dependence edges; $\mathcal{E} \subseteq ID \times ID$ are the existential edges. There is a data dependence edge $\langle id_1, id_2 \rangle \in \mathcal{D}$ if the value of the augmented expression $\mathcal{N}(id_2)$ depends directly on the augmented expression $\mathcal{N}(id_1)$. There is an existential edge $\langle id_1, id_2 \rangle \in \mathcal{E}$ if the value of the augmented expression $\mathcal{N}(id_1)$ controls whether or not the augmented expression or augmented definiton $\mathcal{N}(id_2)$ executes. For example, if $\mathcal{N}(id_1)$ identifies the condition in an if expression and $\mathcal{N}(id_2)$ identifies an expression in the executed then or else branch, then $\langle id_1, id_2 \rangle \in \mathcal{E}$.

4.4 Valid Subproblems

Given a trace t with dependence graph $\langle \mathcal{N}, \mathcal{D}, \mathcal{E} \rangle = \text{Graph}(t)$, a valid subproblem $\mathcal{S} \subseteq \mathcal{N}$ must satisfy two properties: 1) there are no outgoing existential edges and 2) all outgoing data dependence edges must terminate at a stochastic choice (either in a $\text{Dist}(e, l(e_l))$ expression or as part of an observe statement). Conceptually, the second property ensures that any changes made by the subproblem inference tactic can be absorbed by an unchanged stochastic choice to generate a valid trace after subproblem inference. Because the changes are absorbed by the stochastic choice, the new valid trace remains unchanged outside the subproblem.

The first property ensures that if the control flow changes during subproblem inference, any expressions or definitions that depend on this control flow lie within the subproblem and can be removed by the subproblem inference tactic to deliver a valid trace after subproblem inference. We formalize these two properties as follows:

- $\forall id \in \text{dom}\, \mathcal{S}. \langle id, id_o \rangle \in \mathcal{E} \implies id_o \in \mathcal{S}$
- $\forall id \in \text{dom}\, \mathcal{S}.\ \langle id, id_o \rangle \in \mathcal{D} \wedge id_o \in \text{dom}\, \mathcal{N} - \text{dom}\, \mathcal{S} \implies \mathcal{N}(id_o) = \text{Dist}(*, l(*)) \vee \mathcal{N}(id_o) = \text{observe}(\text{Dist}(*) = c)$

The input boundary $\mathcal{B} \subseteq \text{dom}\, \mathcal{N} - \text{dom}\, \mathcal{S}$ of a subproblem $\mathcal{S}$ is the set of nodes on which the subproblem directly depends, i.e., $\mathcal{B} = \{id_b \mid id_b \in \text{dom}\, \mathcal{N} - \text{dom}\, \mathcal{S} \wedge \exists id_i \in \text{dom}\, \mathcal{S}. \langle id_b, id_i \rangle \in \mathcal{D} \cup \mathcal{E}\}$.

The absorbing set $\mathcal{A} \subseteq \text{dom}\, \mathcal{N} - \text{dom}\, \mathcal{S}$ of a subproblem $\mathcal{S}$ is the set of stochastic choice or executed observe statements whose value directly depends on the nodes in the subproblem, i.e., $\mathcal{A} = \{id_a \mid id_a \in \text{dom}\, \mathcal{N} - \text{dom}\, \mathcal{S} \wedge \exists id_i \in \text{dom}\, \mathcal{S}. \langle id_i, id_a \rangle \in \mathcal{D}\}$.

4.5 Infer Statements

The core Venture inference language enables the programmer to intersperse infer statements within the sequence of assume and observe statements. Each infer statement has the form $\text{infer}\ (\langle l(e_1), \ldots, l(e_n) \rangle, \text{SS}, \text{IT})$, where $\langle l(e_1), \ldots, l(e_n) \rangle$ is a set of label expressions that identify stochastic choices in the subproblem, SS is a subproblem selection strategy, and IT is an inference tactic.

After evaluating the label expressions $\langle l(e_1), \ldots, l(e_n) \rangle$ to label values $\langle l(v_1), \ldots, l(v_n) \rangle$, a subproblem selection strategy $\mathcal{S} = \text{SS}(\langle l(v_1), \ldots, l(v_n) \rangle, t)$ takes the label values and a trace t and returns a valid subproblem $\mathcal{S}$ as defined above. Here the target set is the set of all stochastic choices in t with a label $l(v) \in \{l(v_1), \ldots, l(v_n)\}$. We also define the minimum subproblem selection strategy, which selects the smallest valid subproblem $\mathcal{S}$ that contains all the target stochastic choices. Note that, in general, a valid subproblem selection strategy may select a subproblem that contains some, all, or none of the target stochastic choices.

An inference tactic $t' = \text{IT}(t, \mathcal{S})$ takes as input a trace t and a subproblem $\mathcal{S}$ and produces an output trace t'. Here $\mathcal{S}$ must be a valid subproblem for t as defined above in Section 4.4.

$$
\begin{array}{rcl}
c \in Const & := & \mathbb{R} \cup \mathbb{I} \cup \mathbb{B} \\
Vars & := & x_i \\
e \in E & := & x \mid c \mid \text{Dist}(e, l(e_l)) \\
 & \mid & \ominus e \mid \oplus\ e_1\ e_2 \\
 & \mid & \text{let}\ d\ \text{in}\ e \\
 & \mid & \text{if}\ e_\phi\ \text{then}\ e_1\ \text{else}\ e_2 \\
 & \mid & \lambda.x\ e \mid (e_1\ e_2) \\
 & \mid & \langle e_1 \ldots e_k \rangle \mid e[n] \\
d \in D & := & x = e \\
s \in S & := & \text{assume}\ d \\
 & \mid & \text{observe}(\text{Dist}(e) = c) \\
p \in P & := & \emptyset \mid s; p
\end{array}
$$

Figure 18. Core Generative Language

$$
\begin{array}{rcl}
v \in Vals & := & \mathbb{R} \cup \mathbb{I} \cup \mathbb{B} \cup \langle v_1, v_2 \ldots v_n \rangle \cup \langle \lambda\ x.e, \sigma_v, \sigma_{id} \rangle \\
id \in ID & := & id_i \\
ae \in aE & := & (x(id_v) : v)\#id \mid (c : c)\#id \\
 & \mid & (\text{Dist}(ae, l(ae_l)) : v)\#id \\
 & \mid & (\ominus\ ae : v)\#id \mid (\oplus\ ae_1\ ae_2 : v)\#id \\
 & \mid & (\text{if}\ ae_\phi\ \text{then}\ ae_1\ \text{else}\ e_2 : v)\#id \\
 & \mid & (\text{if}\ ae_\phi\ \text{then}\ e_1\ \text{else}\ ae_2 : v)\#id \\
 & \mid & (\text{let}\ ad\ \text{in}\ ae_r : v)\#id \\
 & \mid & (\lambda\ x.e : \langle \lambda\ x.e, \sigma_v, \sigma_{id} \rangle)\#id \\
 & \mid & ((ae_1\ ae_2) = ae_3 : v)\#id \\
 & \mid & (\langle ae_1, ae_2, \ldots ae_k \rangle : v)\#id \mid (ae[n] : v)\#id \\
ad \in aD & := & (x = ae)\#id \\
as \in aS & := & (\text{assume}\ ad : \bot)\#id \\
 & \mid & (\text{observe}(\text{Dist}(ae) = c) : \bot)\#id \\
t \in T & := & \emptyset \mid as; t
\end{array}
$$

Figure 19. Unrolled Program Traces $t \in T$

The output trace t' must 1) be a valid trace of the program that generated t that 2) changes only the subproblem $\mathcal{S}$. We formalize these constraints as follows:

- $t' \in \text{Traces}(\text{Program}(t))$
- $\mathcal{S} \vdash t \equiv t'$

where $\langle N, D, E \rangle = \text{Graph}(t)$, $\langle N', D', E' \rangle = \text{Graph}(t')$, and $\text{Program}(t)$ is the Venture Generative Program that generated t. A trace t contains enough information to recover the program p that generated the trace, conceptually by rerolling the execution that generated the trace.

$\mathcal{S} \vdash t \equiv t'$ indicates that the traces t and t' are equivalent (i.e., have the same values and same structure) except for expressions or definitions in the subproblem $\mathcal{S}$. This condition ensures that portions of the trace t outside the subproblem do not change. Conceptually, the check recursively iterates through the structure of t and t', checking structural similarity and value equality in the parts of the trace which are not within the subproblem. Captured environments σ_v, σ_{id} outside $\mathcal{S}$ may change, i.e., we consider $\langle \lambda\ x.e, \sigma_v, \sigma_{id} \rangle$ equivalent to $\langle \lambda\ x.e, \sigma'_v, \sigma'_{id} \rangle$ even if $id \notin \text{dom}\, \mathcal{S}$.

5 Related Work

We discuss related work in two fields: probabilistic programming languages and approximate probabilistic inference.

Probabilistic Programming Languages: Researchers have developed a range of probabilistic programming languages. Each implementation typically comes with one or a few black box inference strategies. Prominent examples of language/inference strategy pairs include Stan [2] with Hamiltonian Monte Carlo inference [1]; WebPPL [14] with Single-Variable Metropolis-Hastings inference, Black Box variational inference, and Hamiltonian Monte Carlo inference; and Anglican [43] with particle Gibbs sampling. The Augur compiler generates efficient compiled implementations of Markov Chain Monte Carlo inference algorithms [18, 45].

Venture differs from all previous probabilistic programming languages in its support for inference metaprogramming. The current Venture prototype provides tactics that implement each major approximate inference approach from the above languages, but extended so that they apply to broad classes of user-defined subproblems. Example tactics include gradient ascent; LBFGS-based optimization; Hamiltonian Monte Carlo; partial mean field variational inference; slice sampling; Gibbs sampling for discrete variables; Metropolis-Hastings with a proposal based on re-executing the program; rejection sampling; and particle Gibbs sampling. See [24] for details on how to implement these tactics, and [17] for additional formalism describing an interface for new tactics. Inference code can also implement particle methods by resampling traces based on their weights, and interleave inference code with generative code for extending a model and observation code for incorporating observations.

Some other languages provide mechanisms for customizing aspects of inference. For example, LibBi [30] and Gen [**?**] support user-space definition of custom proposal distributions, and Figaro [**?**], Edward [**?**], and Pyro [**?**] support additional constructs. These capabilities can either be implemented as inference metaprogramming tactics or as new metaprogramming constructs that interoperate with the constructs introduced here. Additional inference metaprogramming techniques have been inspired by the constructs described here, such as the use of probabilistic programs as Metropolis-Hastings proposals [**?**], and incremental inference for probabilistic programs [**?**].

After inference metaprogramming was introduced in Venture, other probabilistic languages, e.g. Turing [11] and PyMC3 [37], have added limited support for custom inference in terms of subproblems. Some aspects of inference metaprogramming were inspired by Blaise [**?**], the runtime system used to implement the first version of Church [13]. However, Blaise was not a probabilistic programming language in itself, and lacked the concepts of generative code, observation code, and inference code.

Probabilistic Inference: Custom inference strategies, implemented in traditional programming languages, are now a standard approach for solving probabilistic modeling and inference problems. Proposed strategies include decomposing multivariate state vectors into components that are then updated separately, simulateously resampling highly correlated variables, and applying specialized tactics to appropriately defined subproblems [1]. Proposed hybrid strategies include combinations of Markov chain Monte Carlo techniques with other approaches to inference such as sequential Monte Carlo, variational inference, and gradient-based optimization [1]. Venture supports all of these general approaches (as well as their compositional application to different subproblems). The interface Venture provides for inference tactics is also expressive enough in principle to support message-passing-based approaches to inference [19, 28] and to implement more flexible operator-based variational techniques [32]. One goal of the Venture inference strategy constructs presented in this paper is to make all of these strategies, as well as new custom inference metaprograms, compositionally available in probabilistic programming languages.

6 Conclusion

This paper introduces inference metaprogramming, a fundamental new development in probabilistic programming. Inference metaprogramming makes it possible for probabilistic programmers to implement custom inference algorithms that exploit the structure of each problem, and enables a single probabilistic language to express state-of-the-art solutions to problems from multiple fields. We anticipate that inference metaprogramming will enable the adoption of probabilistic programming for real-time and large-scale applications, once implementations have matured. Inference metaprogramming also opens up new research directions, such as the development of new (meta)metaprograms for profiling, testing, analyzing, and verifying the soundness of metaprograms that carry out approximate inference. Finally, by simplifying the use of probabilistic modeling and inference, we anticipate that inference metaprogramming will help researchers explore both Bayesian and non-Bayesian formulations of the central problems of artificial intelligence.

Acknowledgements. The authors acknowledge Daniel Selsam, Anthony Lu, Yura Perov, Taylor Campbell, and Vlad Firoiu for work on Venture, and Feras Saad, Marco Cusumano Towner, and Anthony Lu for work on case studies. This research was supported by the DARPA PPAML and SD2 programs (constracts FA8750-14-2-0004 and FA8750-17-C-0239), IARPA (contract 2015-15061000003), the Office of Naval Research (contract N000141310333), the Army Research Office (agreement W911NF-13-1- 0212), gifts from Analog Devices and Google, grants from the MIT Media Lab/Harvard Berkman Center Ethics & Governance of AI Fund and the MIT CSAIL Systems that Learn Initiative, and a gift from the Aphorism Foundation.

References

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

[2] Bob Carpenter, Andrew Gelman, Matt Hoffman, Daniel Lee, Ben Goodrich, Michael Betancourt, Michael A. Brubaker, Jiqiang Guo, Peter Li, and Allen Riddell. 2016. Stan: A probabilistic programming language. *Journal of Statistical Software* 20 (2016), 1–37.

[3] P Robert Christian and George Casella. 1999. Monte Carlo statistical methods. (1999).

[4] Gregory F Cooper. 1990. The computational complexity of probabilistic inference using Bayesian belief networks. *Artificial intelligence* 42, 2-3 (1990), 393–405.

[5] Marco. Cusumano-Towner. 2016. Venture implementation of the CrossCat method for Automatic Model Discovery for Multivariate Data Tables. https://github.com/probcomp/class-9s915-f2016-assets/blob/master/10-10-problem-set-2/ps2-problem3-soln.ipynb. (2016).

[6] Marco F Cusumano-Towner, Alexey Radul, David Wingate, and Vikash K Mansinghka. 2017. Probabilistic programs for inferring the goals of autonomous agents. *arXiv preprint arXiv:1704.04977* (2017).

[7] DARPA. 2017. Probabilistic Programming for Advancing Machine Learning (PPAML). (2017).

[8] Pierre Del Moral, Arnaud Doucet, and Ajay Jasra. 2006. Sequential monte carlo samplers. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)* 68, 3 (2006), 411–436.

[9] Arnaud Doucet, Nando De Freitas, and Neil Gordon. 2001. An introduction to sequential Monte Carlo methods. In *Sequential Monte Carlo methods in practice*. Springer, 3–14.

[10] David A Forsyth and Jean Ponce. 2002. *Computer vision: a modern approach*. Prentice Hall Professional Technical Reference.

[11] Hong Ge, Kai Xu, and Zoubin Ghahramani. 2018. Turing: Composable inference for probabilistic programming. In *International Conference on Artificial Intelligence and Statistics*. 1682–1690.

[12] Andrew Gelman, John B Carlin, Hal S Stern, David B Dunson, Aki Vehtari, and Donald B Rubin. 2014. *Bayesian data analysis*. Vol. 2. CRC press Boca Raton, FL.

[13] Noah Goodman, Vikash Mansinghka, Daniel M Roy, Keith Bonawitz, and Joshua B Tenenbaum. 2008. Church: a language for generative models. In *Proceedings of the 25th International Conference on Uncertainty in Artificial Intelligence*.

[14] Noah D. Goodman and Andreas Stuhlmueller. 2014. The Design and Implementation of Probabilistic Programming Languages. http://dippl.org. (2014). Accessed: 2017-11-15.

[15] Andrew D. Gordon, Thore Graepel, Nicolas Rolland, Claudio Russo, Johannes Borgstrom, and John Guiver. 2014. Tabular: a schema-driven probabilistic programming language. In *ACM SIGPLAN Notices*, Vol. 49. ACM, 321–334.

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

[17] Shivam Handa, Vikash Mansinghka, and Martin Rinard. 2018. Probabilistic Programming with Programmable Inference - Tech Report. https://people.csail.mit.edu/rinard/paper/pldi18.techreport.pdf. (2018). Accessed: 2018-05-03.

[18] Daniel Huang, Jean-Baptiste Tristan, and Greg Morrisett. 2017. Compiling Markov chain Monte Carlo algorithms for probabilistic modeling. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 111–125.

[19] Frank R Kschischang, Brendan J Frey, and H-A Loeliger. 2001. Factor graphs and the sum-product algorithm. *IEEE Transactions on information theory* 47, 2 (2001), 498–519.

[20] Uber AI Labs. 2017. Pyro, a deep probabilistic programming Language. (2017). https://eng.uber.com/pyro/

[21] Jun S Liu. 2008. *Monte Carlo strategies in scientific computing*. Springer Science & Business Media.

[22] James Robert Lloyd, David K Duvenaud, Roger B Grosse, Joshua B Tenenbaum, and Zoubin Ghahramani. 2014. Automatic Construction and Natural-Language Description of Nonparametric Regression Models.. In *AAAI*. 1242–1250.

[23] David JC MacKay. 2003. *Information theory, inference and learning algorithms*. Cambridge university press.

[24] Vikash Mansinghka, Daniel Selsam, and Yura Perov. 2014. Venture: a higher-order probabilistic programming platform with programmable inference. *arXiv preprint arXiv:1404.0099* (2014).

[25] Vikash Mansinghka, Patrick Shafto, Eric Jonas, Cap Petschulat, Max Gasner, and Joshua B Tenenbaum. 2016. Crosscat: A fully bayesian nonparametric method for analyzing heterogeneous, high dimensional data. *The Journal of Machine Learning Research* 17, 1 (2016), 4760–4808.

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

[27] Brian Milch, Bhaskara Marthi, Stuart Russell, David Sontag, Daniel L. Ong, and Andrey Kolobov. 2007. BLOG: Probabilistic models with unknown objects. *Statistical relational learning* (2007), 373.

[28] Thomas P Minka. 2001. Expectation propagation for approximate Bayesian inference. In *Proceedings of the Seventeenth conference on Uncertainty in artificial intelligence*. Morgan Kaufmann Publishers Inc., 362–369.

[29] Kevin P Murphy. 2012. *Machine learning: a probabilistic perspective*. MIT press.

[30] Lawrence M Murray. 2013. Bayesian state-space modelling on high-performance hardware using LibBi. *arXiv preprint arXiv:1306.3277* (2013).

[31] Alexey Radul, Anthony Lu, and Vikash. Mansinghka. 2015. Venture solution to DARPA PPAML Challenge Problem 1 on Simultaneous Localization and Mapping. https://github.com/probcomp/ppaml-cps/tree/master/cp1. (2015).

[32] Rajesh Ranganath, Dustin Tran, Jaan Altosaar, and David Blei. 2016. Operator variational inference. In *Advances in Neural Information Processing Systems*. 496–504.

[33] Carl Edward Rasmussen and Christopher KI Williams. 2006. Gaussian processes for machine learning. *The MIT Press, Cambridge, MA, USA* 38 (2006), 715–719.

[34] Christian Robert and George Casella. 2010. *Introducing Monte Carlo Methods with R*. Springer Science & Business Media.

[35] Stuart J. Russell and Peter Norvig. 2003. *Artificial Intelligence: A Modern Approach* (2 ed.). Pearson Education.

[36] Ardavan Saeedi, Vlad Firoiu, and Vikash Mansinghka. 2015. Automatic Inference for Inverting Software Simulators via Probabilistic Programming. *arXiv preprint arXiv:1506.00308* (2015).

[37] John Salvatier, Thomas V Wiecki, and Christopher Fonnesbeck. 2016. Probabilistic programming in Python using PyMC3. *PeerJ Computer Science* 2 (2016), e55.

[38] Ulrich Schaechtle, Feras Saad, Alexey Radul, and Vikash Mansinghka. 2016. Time Series Structure Discovery via Probabilistic Program Synthesis. *arXiv preprint arXiv:1611.07051* (2016).

[39] Ulrich Schaechtle, Feras Saad, Alexey Radul, and Vikash Mansinghka. 2016. Time Series Structure Discovery via Probabilistic Program Synthesis. *arXiv preprint arXiv:1611.07051* (2016).

[40] Ulrich Schaechtle, Ben Zinberg, Alexey Radul, Kostas Stathis, and Vikash K Mansinghka. 2015. Probabilistic Programming with Gaussian Process Memoization. *arXiv preprint arXiv:1512.05665* (2015).

[41] Sam Staton, Hongseok Yang, Frank Wood, Chris Heunen, and Ohad Kammar. 2016. Semantics for probabilistic programming: higher-order functions, continuous distributions, and soft constraints. In *Proceedings of the 31st Annual ACM/IEEE Symposium on Logic in Computer Science*.

ACM, 525–534.

[42] Sebastian Thrun, Wolfram Burgard, and Dieter Fox. 2005. *Probabilistic robotics*. MIT press.

[43] David Tolpin, Jan-Willem van de Meent, and Frank Wood. 2015. Probabilistic programming in Anglican. In *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*. Springer, 308–311.

[44] Dustin Tran, Matthew D. Hoffman, Rif A. Saurous, Eugene Brevdo, Kevin Murphy, and David M. Blei. 2017. Deep Probabilistic Programming. In *Proceedings of the 2017 nternational Conference on Learning Representations*.

[45] Jean-Baptiste Tristan, Daniel Huang, Joseph Tassarotti, Adam C. Pocock, Stephen Green, and Guy L. Steele. 2014. Augur: Data-parallel probabilistic modeling. In *Advances in Neural Information Processing Systems*. 2600–2608.

[46] Martin J Wainwright, Michael I Jordan, et al. 2008. Graphical models, exponential families, and variational inference. *Foundations and Trends® in Machine Learning* 1, 1–2 (2008), 1–305.