

# Metaprob: a simple, extensible language for probabilistic programming and meta-programming

**Alexey Radul**

*Brain and Cognitive Science Laboratory  
Massachusetts Institute of Technology  
Cambridge, MA 02139, USA*

AXCH@MIT.EDU

**Vikash K. Mansinghka**

*Brain and Cognitive Science Laboratory  
Massachusetts Institute of Technology  
Cambridge, MA 02139, USA*

VKM@MIT.EDU

**Editor:** Vikash Mansinghka and Daniel Roy

## Abstract

Probabilistic programming is based on two ideas: (i) representing probabilistic models as programs and (ii) representing inference and learning operations as meta-programs. Most existing probabilistic languages provide a small set of built-in meta-programs; adding new inference and learning techniques requires extending the language implementation. Metaprob is a probabilistic meta-programming language that aims to remove this limitation. In Metaprob, partial execution traces of probabilistic programs are first-class objects. The Metaprob interpreter can execute probabilistic programs against partial traces, treating traces as a source of Pearl-style interventions on the behavior of a probabilistic program. Metaprob (meta-)programs can also trace the behavior of other probabilistic programs, modify these traces, and use interventions based on these modified traces to implement Monte Carlo inference strategies. This chapter shows how to use Metaprob's language constructs to implement two distinctive features of the Church probabilistic programming language—namely, stochastic memoization and Metropolis-Hastings transitions on execution traces—as ordinary user-space meta-programs.

## 1. Introduction

Probabilistic programming is based on two ideas: (i) probabilistic models can be represented as executable computer programs and (ii) inference and learning algorithms can be represented as meta-programs that take these programs as inputs and/or produce them as outputs. These ideas are the basis of a rapidly growing collection of languages and implementations. Each language provides some “built-in” probability distributions, plus language constructs for combining them to larger programs that represent meaningful probabilistic models. Each language also provides a small set of “built-in” inference algorithms that can infer likely outcomes from these probability distributions given data.

Unfortunately, existing languages do not provide language constructs for adding new probability distributions or inference algorithms. If the built-in set is inadequate for some problem, the probabilistic programmer needs to either choose a different language or learn how to extend the language implementation. Most existing languages also do not provide

constructs for combining built-in general-purpose algorithms with user-specified algorithms that are customized for a given application. These are fundamental limitations. In practice, they restrict the use of probabilistic programming to applications where a small set of probability distributions and generic inference algorithms are known to be adequate ahead of time. They also greatly limit the extent to which probabilistic programmers can take advantage of advances in algorithms for approximate inference.

Metaprob is a simple, extensible language for probabilistic programming and meta-programming. Metaprob is simple enough that an implementation of the language fits on a single page, requiring under 100 lines of Metaprob code. Also, in Metaprob, new probability distributions and inference algorithms can be written as ordinary user-space programs. Metaprob is extensible and expressive enough that distinctive features of other probabilistic languages, such as Metropolis-Hastings inference, sequential Monte Carlo inference, and modeling with Dirichlet processes, can each be written in under one page of Metaprob code. Inference in Metaprob is not limited to Monte Carlo sampling. For example, this chapter includes a Metaprob meta-program that calculates probabilities by enumerating the executions of another Metaprob program.

It is helpful to view Metaprob in terms of an analogy to Lisp. Lisp is well known both for its unusual simplicity and for its support for meta-programming. Lisp source code follows a small set of rules, and is easy to represent using lists, the central data structure in the language. The process of Lisp program execution also follows a small set of rules that closely mirror the structure of Lisp source code. Thus Lisp interpreters can be written as short Lisp programs. Executable Lisp programs can also be represented in the language, as “procedures” that take a specified list of formal parameters. These procedures are ordinary data Lisp objects that can be produced and executed by other Lisp programs. This combination of simplicity and expressiveness helped make Lisp a widely used language for research in artificial intelligence and in programming language design and implementation. Lisp also provides a simple model of computation that has been useful for teaching.

The design of Metaprob aims to preserve the simplicity and expressiveness of Lisp, while adding the minimal additional features needed for research in probabilistic artificial intelligence and in probabilistic programming. Metaprob also aims to serve as a simple model of probabilistic computation that can be used to teach the key concepts to a broad audience. This model of computation differs from the Lisp model in three fundamental ways:

1. The built-in Metaprob interpreter allows the execution of a Metaprob program to be *intervened on*, i.e., forced to take on specific values at chosen points in a program’s execution. Metaprob provides language constructs for naming points in a program’s execution and a data structure called a *trace* for storing mappings between execution points and values. Metaprob’s syntax is designed to make it easy to use these capabilities.
2. Probabilistic programs in Metaprob constitute extensible packages of executable code, source code, meta-data, and meta-programs. These packages can be re-opened by other Metaprob (meta)programs, for example to retrieve the source code of a program or to retrieve a program that specifies the output probability distribution of some other program.

3. Typical Metaprob programs rely on a standard library<sup>1</sup> of “inference meta-programs”, not just a built-in interpreter or compiler. These inference meta-programs take a probabilistic program and also a “target” trace as input, and generate executions of the program that are compatible with the “target” trace.

These differences constitute the core contributions of Metaprob’s design. This chapter introduces the language features that underlie these differences and shows that they are sufficient to implement a broad class of techniques for probabilistic modeling and inference. Applications are drawn from hierarchical Bayesian modeling, Bayesian networks, and non-parametric Bayesian statistics. To the best of our knowledge, Metaprob is the first probabilistic language with an explicit meta-circular implementation, and the only probabilistic language in which the distinctive modeling and inference features of expressive languages such as Church can be implemented as short (meta)programs. We also believe that Metaprob is the only probabilistic language with support for interventions and for adding new probability distributions within the language.

This chapter is organized as follows. The remainder of this section gives a tutorial introduction to Metaprob. The next introduces the core language. The following two sections describe meta-programs for probabilistic modeling and inference. The final section discusses the relationship between Metaprob and other probabilistic languages, and outlines important directions for future research. It is also helpful to navigate the chapter in light of the main Metaprob example programs that it presents. These programs solve the following problems:

1. Inferring if a coin is tricky or fair via a Monte Carlo inference in a hierarchical Bayesian model.
  2. Comparing inference given observations to inference given interventions on a discrete-variable Bayesian network, using both approximate and exact inference algorithms.
  3. Implementing an interpreter for Metaprob that supports overriding the execution of a program via interventions.
  4. Implementing a tracing interpreter for Metaprob that records the stochastic choices made by a probabilistic program during its execution.
  5. Adding the Gaussian probability distribution, packaging code for a sampler with code for the log output probability density of the Gaussian.
  6. Performing inference via rejection sampling and sampling-importance-resampling, i.e. via sampling, weighting, and resampling traces based on their compatibility with a trace representing the constraints on the execution of a program.
  7. Implementing an approximate inference meta-program that changes a single random choice at the time. This program reimplements the single-site Metropolis-Hastings
- 
1. In principle, Metaprob is sufficiently expressive to implement meta-programs for a broad class of Monte Carlo, variational, message-passing, gradient-based, and “deep” inference algorithms. However, at present, the standard library only includes a few simple examples of Monte Carlo and enumeration-based inference techniques.

inference algorithm that is the basis of probabilistic programming languages such as Church and WebPPL.

8. Implementing an inference algorithm that enumerates all possible executions of probabilistic programs with finite support and deterministic control flow.
9. Adding the Chinese restaurant process, and adding a memoizer to the language, both of which use traces to store their internal state.
10. Putting these pieces together to build a Chinese restaurant process mixture model and using it to infer clusters from numerical data.

## 2. Tutorial Examples

We begin our presentation of Metaprob with two examples that we hope will be familiar to our readers: a simple hierarchical Bayesian model for inferring whether a given coin is fair by observing several flips, and the classic Bayesian network about a home security alarm that can be triggered by a burglar or an earthquake. The goal of this section is to give a basic feel for Metaprob. Detailed discussion of the parts of which Metaprob is made will follow in the sequel.

### 2.1 Trick Coin—Hierarchical Bayes

A classic introductory model of the class called hierarchical Bayes covers the following situation:

A magician produces a coin from the ear of an audience member. They display both sides, and it looks like a perfectly normal product of the mint. And yet, as the magician starts flipping it, the coin keeps landing on “heads”. Perhaps there is some trick and it’s actually weighted? How soon and how strongly do you start smelling a rat, depending on the flip results you observe?

People’s intuitions about this situation tend to

1. Give the benefit of the doubt that the coin may be fair when observing short sequences of all heads (say, 1-3 flips), but
2. Lean towards a weighted coin if they observe long (say, 7-15) sequences of all heads or all tails, and
3. Still lean weighted even if there is an occasional “wrong” flip (e.g., 14 heads and one tail), but
4. Assume a long sequence likely came from a fair coin if the number of heads and tails is roughly equal (unless there’s some other striking pattern, but cross-flip dependencies are beyond the scope of this example).

To formalize this as a probability problem, we need to supply two additional pieces of information: the prior (before seeing any flips) probability that the coin is fair (vs weighted)

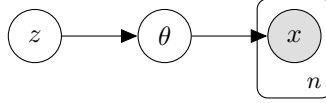
and a model for how *a priori* plausible various weights are if we assume the model is, indeed, weighted. With that filled in, a complete model for this problem could look like this:

$$\begin{aligned} z &\sim \text{Bernoulli}(0.1) \\ \theta|z &\sim \begin{cases} \mathcal{U}(0, 1) & z = 1 \\ 0.5 & \text{otherwise} \end{cases} \\ x_i|\theta &\sim \text{Bernoulli}(\theta) \text{ for } 1 \leq i \leq n, \end{aligned}$$

defining the joint probability distribution<sup>2</sup>

$$p(z, \theta, \mathbf{x})d\theta = \begin{cases} 0.1 \theta^k \theta^{n-k} d\theta & z = 1, 0 \leq \theta \leq 1, k = \sum x_i \\ 0.9 \left(\frac{1}{2}\right)^n & z = 0, \theta = 0.5 \\ 0 & \text{otherwise.} \end{cases}$$

In plate notation, that model (on  $n$  observations) renders as



The quantity of interest in the scenario is the conditional distribution  $p(z|\mathbf{x})$  of whether the coin is weighted given some particular flip sequence  $\mathbf{x}$ . Just to fully spell it out, the product and marginalization rules give

$$\begin{aligned} p(z|\mathbf{x}) &= \int p(z, \theta|\mathbf{x})d\theta = \frac{1}{p(\mathbf{x})} \int p(z, \theta, \mathbf{x})d\theta, \\ p(\mathbf{x}) &= \sum_z \int p(z, \theta, \mathbf{x})d\theta. \end{aligned}$$

Figure 1 shows a Metaprob transcript defining and playing with this model for this situation. There you can see the major parts of probabilistic programming in Metaprob:

- Defining a model as a program that samples from it (`flip_coins`, A);
- Tracing its execution (B) and examining some latent state of interest (`tricky`, C);
- Enforcing some behavior, such as observations (five “heads”, D);
- Defining (E) and running (F) an inference meta-program that operates on execution traces; and
- Examining inferred latent state (G).

2. A notational note: We choose to write density functions in a differential notation because we feel reasoning locally (that is, at infinitesimal points) is more intuitive than talking about measures (and sets of positive size) explicitly. However, we do not adopt pure density function notation for a model such as this, because the base measure would be non-trivial and non-intuitive: Lebesgue measure on  $\theta$  if  $z = 1$ , but counting measure on  $\theta$  otherwise. Hence the explicit differentials.

The presence of a differential such as  $d\theta$  should be read as “a small Lebesgue-ball around  $\theta$ ”. The way to read this equation as an equation about measures is “The  $p$ -size of a small ball about the point  $(z, \theta, \mathbf{x})$  is either  $0.1 \theta^k \theta^{n-k}$  times the Lebesgue-size of the  $\theta$  component thereof, or just  $0.9 \left(\frac{1}{2}\right)^n$ , depending on the conditions given.

```

// (A) Define a function whose traces will serve as the model
flip_coins = (n) ~> {
  root_addr = &this;
  tricky = flip(0.1);
  weight = if (tricky) {uniform(0, 1);} else {0.5;};
  map((i) ~> with_address /$root_addr/datum/$i/: flip(weight),
    range(n));
};

// (B) Run it once
a_trace = {{ }};
trace_choices(
  program      = flip_coins,
  inputs       = [5],
  intervention_trace = {{ }},
  output_trace = a_trace);

// (C) Have a look at (a sample of) whether the coin is tricky
print(*a_trace[/1/tricky/flip/]);
False

// (D) Force the coin to come up heads every time
a_trace[/datum/0/flip/] := true;
a_trace[/datum/1/flip/] := true;
a_trace[/datum/2/flip/] := true;
a_trace[/datum/3/flip/] := true;
a_trace[/datum/4/flip/] := true;

// (E) Define a Metropolis-based inference strategy
approximate_inference_update = () ~> {
  single_site_metropolis_hastings_step(
    program = flip_coins,
    inputs  = [5],
    trace   = a_trace,
    constraint_addresses = set_difference(
      addresses_of(a_trace),
      [/1/tricky/flip/, /2/weight/then/0/uniform/]));
};

// (F) Run it a bit
repeat(times = 20, program = approximate_inference_update);

// (G) Look at (a sample of) the (inferred) trickiness
print(*a_trace[/1/tricky/flip/]);
False

```

Figure 1: A Metaprob transcript, modeling  $n$  flips of a potentially biased coin. The prior is that the coin has an 0.1 probability of being biased, in which case it has an unknown uniformly distributed weight; otherwise is fair. Each output is a sample from the distribution on values of the `tricky` variable at that point; they will vary with the initial entropy given to the pseudo-random number generator.

The state space of this model is computationally represented by *traces* of the execution of the `flip_coins` program. Figure 2 shows two representative examples of such traces when two flips of the coin are involved. Given the way `flip_coins` is coded, the  $z$  variable is always stored at the *address* `/1/tricky/flip/`. If  $z = 1$ , the  $\theta$  variable is stored at the address `/2/weight/then/0/beta/` (if  $z = 0$ , then  $\theta = 0.5$  is understood implicitly). The  $x_i$  are stored at the addresses `/datum/i/flip/`.

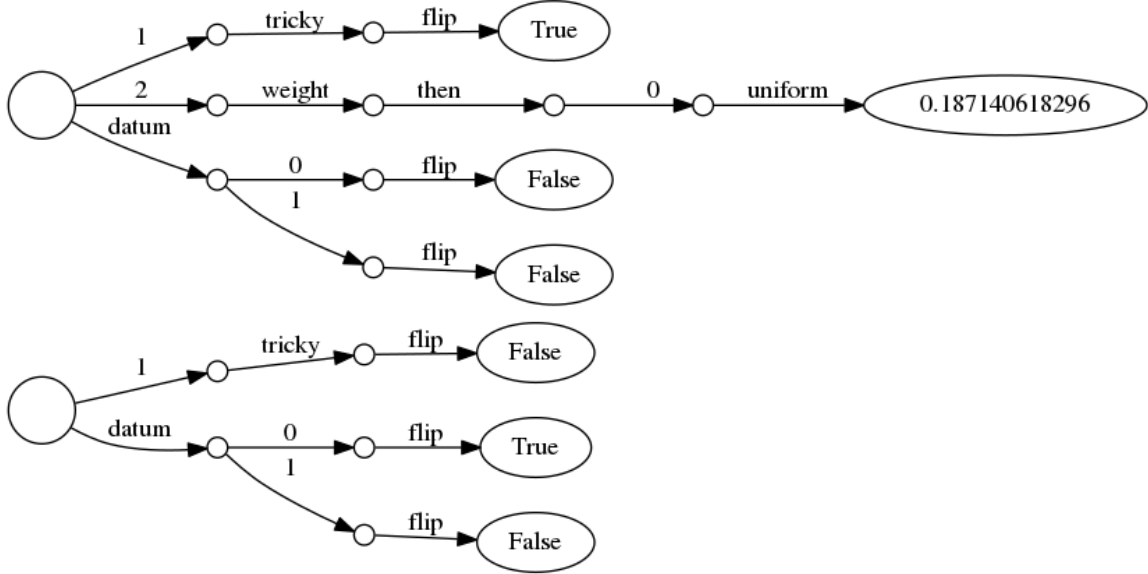


Figure 2: Two traces of executing the `flip_coins` program from Figure 1 for two flips. The example in the top pane takes the “tricky coin” control flow path; note the coin weight drawn from the Beta distribution. The example in the bottom pane takes the “fair coin” control flow path. The coin weight (0.5) is not explicitly recorded there because it is deterministic in the source code. The large node on the left is the *root* of the trace. We can name any node (or subtrace) by listing the edge labels that lead to it from the root. This is an *address*, which we write slash-bounded, by analogy with URLs. For example, the Boolean indicating whether the coin is tricky lives at the address `/1/tricky/flip/`.

The inference invoked in blocks (E) and (F) of Figure 1 operates by stochastically modifying a complete trace of `flip_coins` in a pattern that is arranged to converge, after many repetitions, to sampling that trace from the distribution  $p(z, \theta | \mathbf{x})$ , from which a sample from  $p(z | \mathbf{x})$  can be extracted by reading the trace, as in line (G). Or, which is the same thing in technical jargon, running a Markov chain on the space of execution histories of `flip_coins` whose stationary distribution is  $p(z, \theta | \mathbf{x})$ .

## 2.2 Burglary or Earthquake—Bayesian Network

The earthquake-burglary scenario is used to teach basics of probabilistic inference, and in particular the formalism of Bayesian networks. The scenario goes like this:

The protagonist lives in California and has a burglar alarm. On any given day, there is a chance of one’s house being the victim of a burglary, which, in turn, has

some chance of triggering the alarm. There is also a chance of an earthquake, which may (erroneously) trigger the alarm as well. But, if one is away from home, one will not perceive the alarm directly; rather, one’s neighbors John or Mary may or may not call—but they can also do that independently.

To complete this scenario to a probabilistic model over five Boolean variables  $e, b, a, j, m$  it suffices to give the conditional probability tables, for instance something like this:

$p(e)$	$p(b)$	$e$	$b$	$p(a e, b)$	$a$	$p(j a)$	$a$	$p(m a)$
0.1	0.1	True	True	0.9	True	0.8	True	0.9
		True	False	0.2	False	0.1	False	0.4
		False	True	0.85				
		False	False	0.05				

This is now a complete model, which can be used to answer various questions like “If John called, how likely is it that there was a burglary (or an earthquake)?” or “If I directly know that the alarm went off, does learning whether John called give me any information about whether Mary did?”

This model can be represented in Metaprob as the function that samples possible world states. Figure 3 shows code for that function, and then Figure 4 shows the prior distribution on situations that it induces. In particular, with these parameters, about 40% of the time nothing happens.

More than just probability models, Bayesian networks have also been proposed as models of causality. The intuition is that if one learns, in the probabilistic model, that the alarm went off, one suspects much more strongly (though, since the probabilities are neither 0 nor 1, without certainty) that there was, indeed, a burglary, and also that one’s neighbors will call. If, however, one goes home early and purposely sets the alarm off oneself, there is a difference: the probability of concerned calls from the neighbors goes up the same way, but the probability that there had been a burglary does not. Even though the alarm is ringing.

Interventions on probabilistic models represented as programs have a natural computational representation: An intervention just sets the variable being intervened on to the demanded value at the time that variable is defined. The intervention has no effect on the variable’s predecessors (causes), but the intervened value is used when sampling the successors (consequences). As a graph, intervention cuts all the parent edges of the node intervened upon, but leaves the child edges in place. Figure 5 shows Metaprob code for drawing samples from a version of the earthquake network where we intervene to set the `alarm` variable to `true`, as well as a rendering of the network and a histogram of the resulting distribution. Contrast this with Figure 6, which is the same network but observing that the `alarm` variable was `true`, without modifying its causes. Both constraints and interventions are specified as traces, which map execution addresses to the constraining (respectively, intervening) values.



```

earthquake_bayesian_network = () -> {
  earthquake = flip(0.1);
  burglary   = flip(0.1);
  p_alarm =
    if (burglary && earthquake) 0.9
    else if (burglary) 0.85
    else if (earthquake) 0.2
    else 0.05;
  alarm      = flip(p_alarm);
  p_john_call = if (alarm) 0.8 else 0.1;
  john_call  = flip(p_john_call);
  p_mary_call = if (alarm) 0.9 else 0.4;
  mary_call  = flip(p_mary_call);
  "ok";
};

query = (state) -> {
  earthquake = *state[/0/earthquake/flip/];
  burglary   = *state[/1/burglary/flip/];
  alarm      = *state[/3/alarm/flip/];
  john_call  = *state[/5/john_call/flip/];
  mary_call  = *state[/7/mary_call/flip/];
  (earthquake, burglary, alarm, john_call, mary_call);
};

```

Figure 3: Definition of the classic earthquake-burglary Bayes net in Metaprob, including a function for collecting the variables of interest from a trace of its execution.

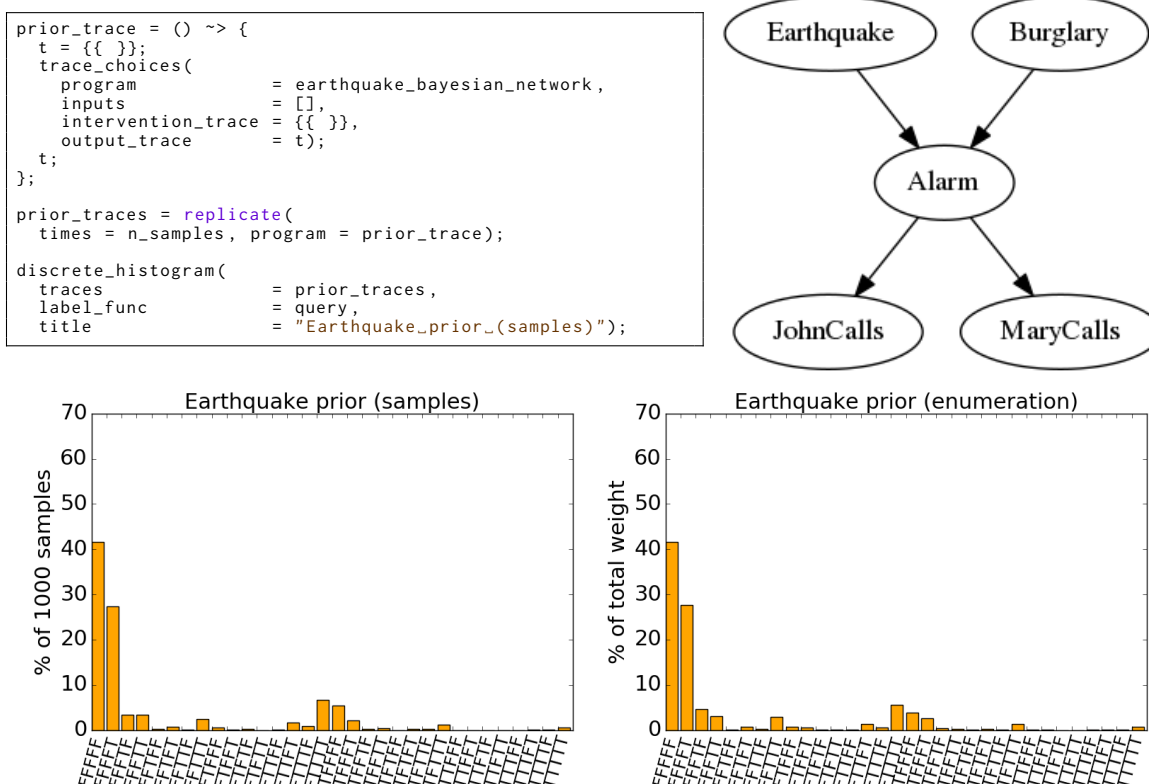


Figure 4: The joint prior distribution induced by the earthquake-burglary Bayes net of Figure 3. The top row shows code for sampling from the prior distribution on traces of this Bayes net (top left) and a graphical representation of the network (top right). The bottom row shows, from left to right, the empirical distribution of sampling from it and the exact distribution computed by enumerating over possible states.

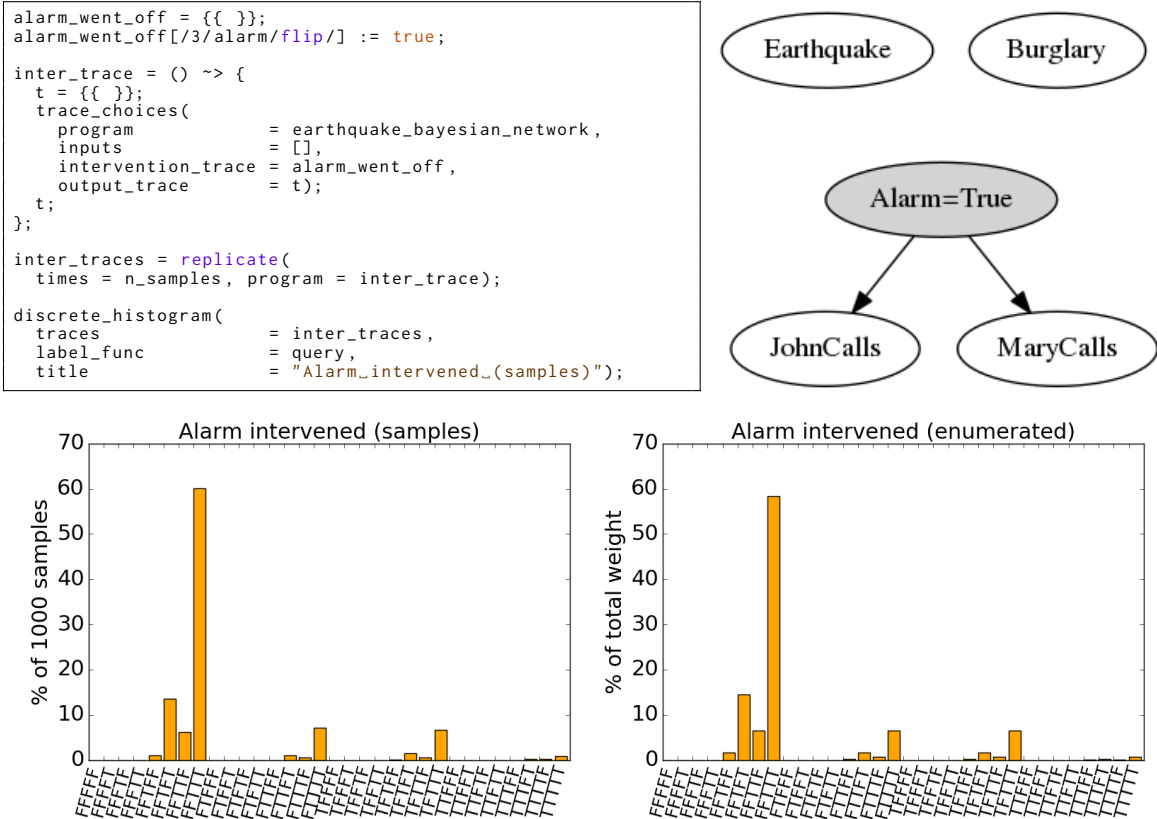


Figure 5: The joint distribution induced by intervening on the earthquake-burglary Bayes net of Figure 3 to set the `alarm` variable to `true`. The top row shows code for sampling from the distribution on traces of this Bayes net (top left) and a graphical representation of the network induced by the intervention (top right). The bottom row shows, from left to right, the empirical distribution of sampling from it and the exact distribution computed by enumerating over possible states.

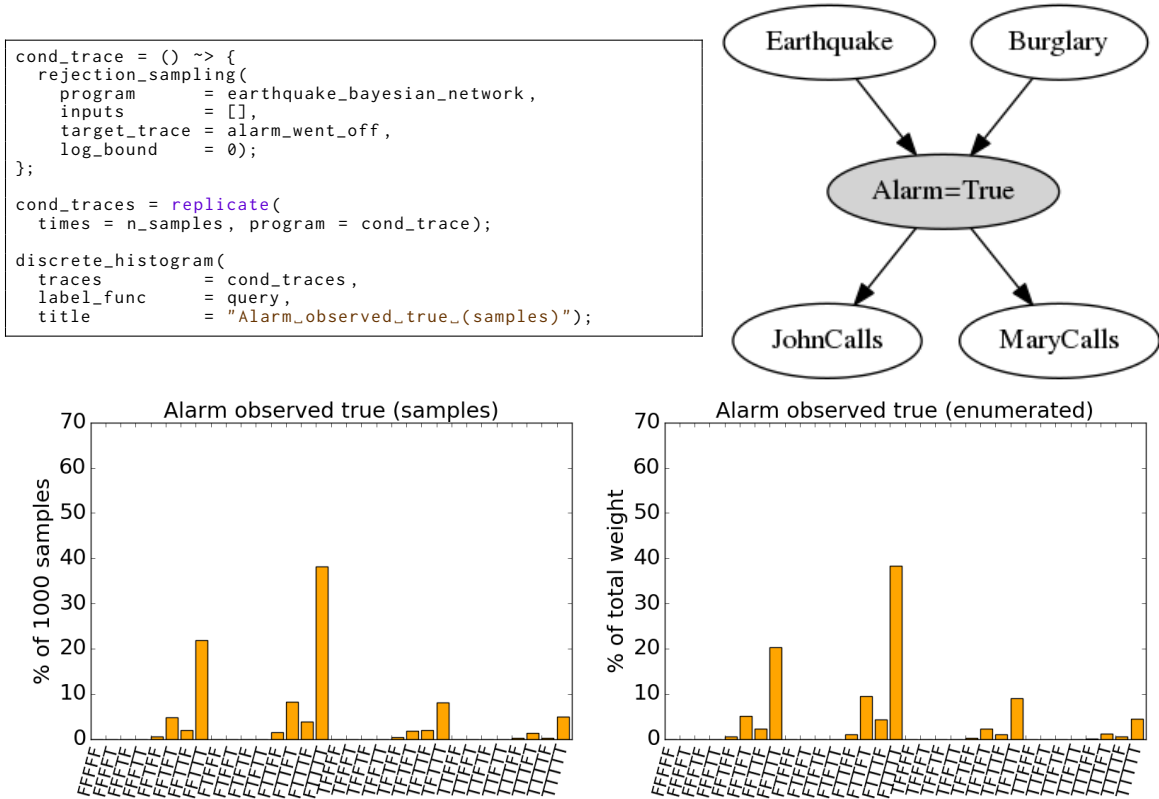


Figure 6: The joint distribution induced by observing the `alarm` variable in the earthquake-burglary Bayes net of Figure 3 being `true`. The top row shows code for sampling from the distribution on traces of this Bayes net (top left) and a graphical representation of the network induced by the observation (top right). The bottom row shows, from left to right, the empirical distribution of sampling from it and the exact distribution computed by enumerating over possible states.

### 3. Metaprob fundamentals

We now delve into the building blocks of Metaprob in some depth.

#### 3.1 Traces

A *trace* is a prefix-tree-structured map from *addresses* to values. Each Metaprob address is a (possibly empty) sequence of *keys*. A prefix tree is an association data structure optimized for the situation where the addresses being looked up are expected to have large common prefixes: instead of a flat table mapping each address to its value independently, the root for a prefix tree is a node that maps the first key of each address to the sub-prefix-tree that contains all the entries for all the addresses that had that key first. In comparison with, say, hash tables, prefix trees offer fast selection of subtrees by prefix. Additionally, if the alphabet of possible keys is small and known in advance, a prefix tree implementation can also be optimized to be more compact than the corresponding hash table, and perform insertions and lookups faster (as it is not necessary to compute a hash function over the entire address, if a small prefix narrows the options down to just one candidate).

Metaprob traces make the further choice that intermediate prefix tree nodes (not just leaves) can (but need not) hold a value. See Figure 7 for a tutorial on the syntax and effects of the trace operations Metaprob supports.

The trace is the fundamental data structure in Metaprob. It can be used to represent

1. Conventional records, as a depth-1 trace whose keys are the record fields;
2. Program source code, mapping the recursive expression structure into subtrace structure; and
3. Program executions, using a similar mapping.

Figures 8 and 9 illustrate the representation of Metaprob source code as Metaprob traces, repeating the example `flip_coins` program from Figure 1. We discuss the use of traces to record execution histories of Metaprob programs in the next sections.


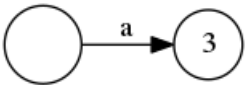
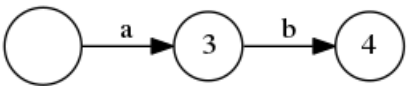
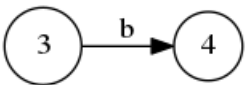
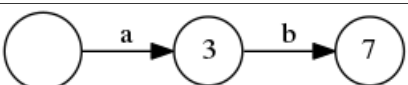
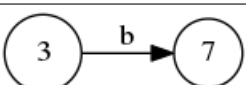
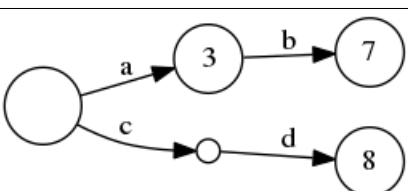
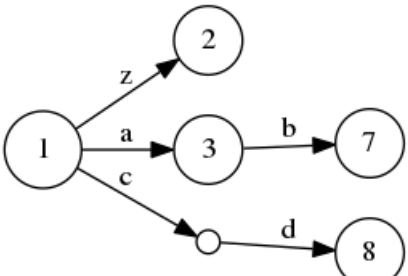
Metaprob syntax	Semantic effect	Result
<code>trace = {{ }};</code>	Literal empty trace	
<code>trace[/a/] := 3;</code>	Assign 3 at the key a	
<code>trace[/a/b/] := 4;</code>	Assign 4 at the two-key address /a/b/	
<code>subtrace = trace[/a/];</code>	Obtain the subtrace at the key a	
<code>*subtrace;</code>	Fetch the value at the root node (of subtrace)	3
<code>trace[/a/b/] := 7;</code>	Repeated assignments overwrite	
<code>subtrace;</code>	The <b>subtrace</b> is an alias, so sees the modification	
<code>trace[/c/d/] := 8;</code>	Traces have unlimited fanout, and nodes may lack values	
<code>trace has_value;</code>	Postfix operator checking for a value at the root node	False
<code>addresses_of(trace);</code>	List of addresses that have values	/a/, /a/b/, /c/d/
<code>{{ 1, z = 2, a = **subtrace, */c/d/ = 8}};</code>	Trace literal syntax can include a root value, assignment by key, and splicing of subtraces (with <b>**</b> ) or addresses (with <b>*</b> )	

Figure 7: Basic operations on Metaprob traces. This table can be read down the left column as a transcript of a Metaprob session, showing the effects of the trace operations.

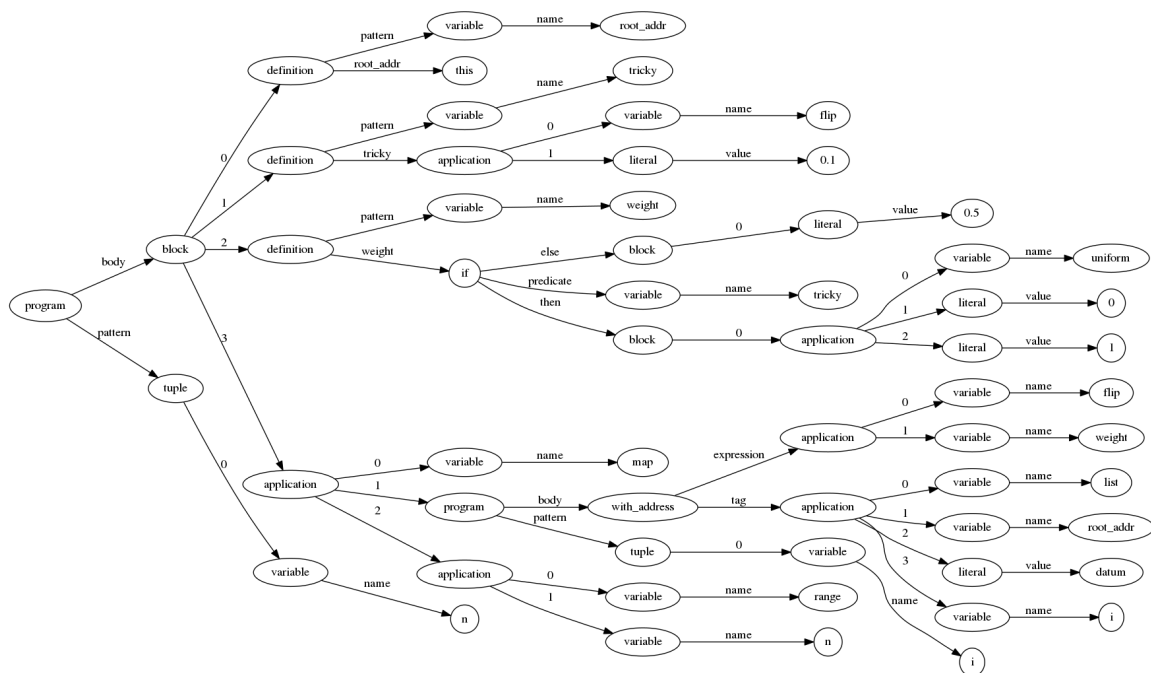


Figure 8: The source code of the `flip_coins` program from Figure 1, represented as a trace.

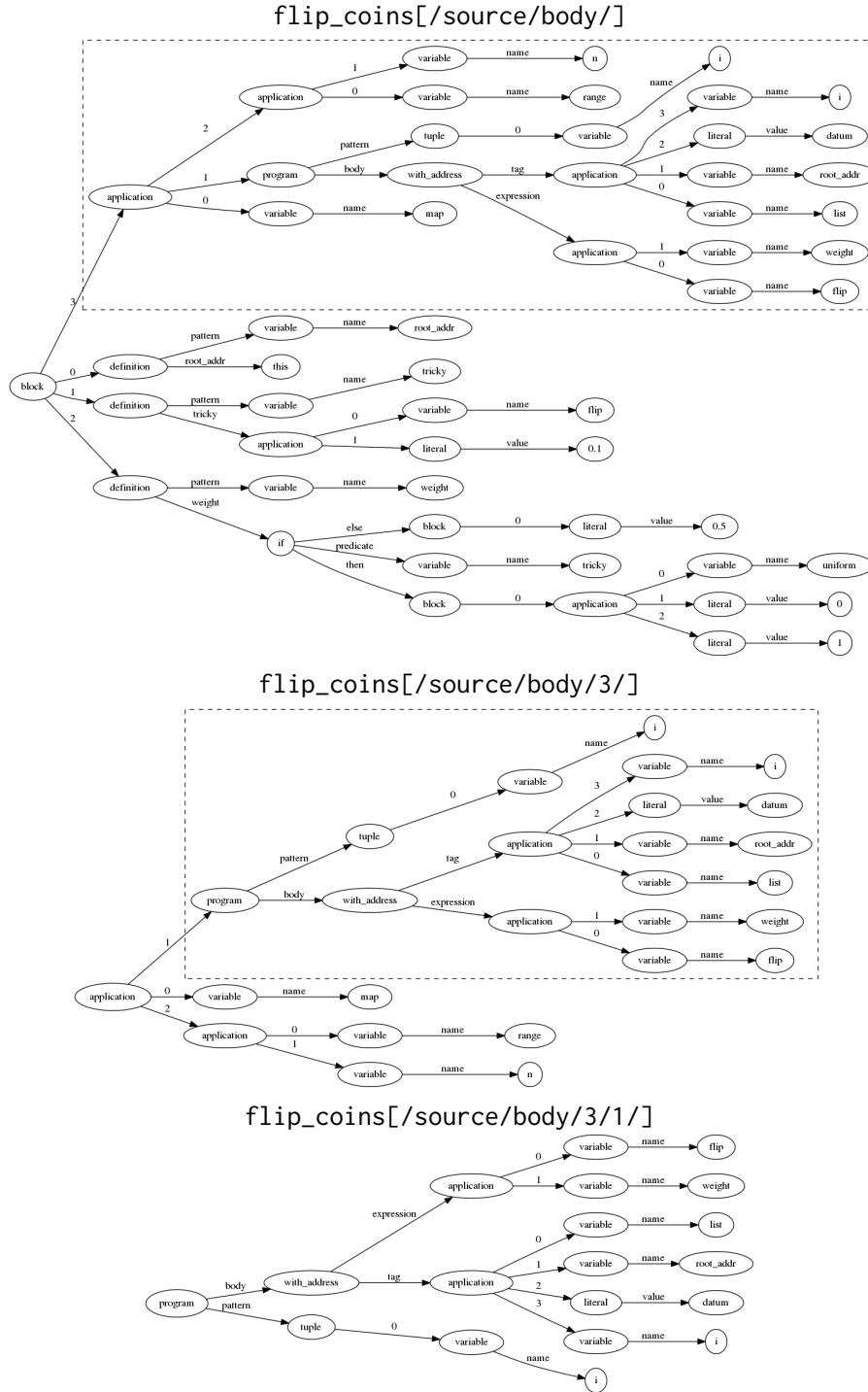


Figure 9: An illustration of trace prefix selection, using the source code from Figure 8 as a starting point. Each dashed box highlights which subtrace is shown in the subsequent pane.



### 3.2 A metacircular interpreter

We now have the machinery to define the semantics of Metaprob constructs as a metacircular interpreter written in Metaprob itself. Figures 10 and 11 give two mutually recursive Metaprob functions that define how to handle each Metaprob syntactic form. This is written in the style of an eval-apply interpreter <sup>?</sup>, if you are familiar with that. Figure 12 shows the representation of Metaprob lexical environments as traces.

```
interpret = (program, inputs, intervention_trace) ->
  if (trace_has_key(program, "executable")) {
    interpret_prim(*program[/executable/], inputs, intervention_trace);
  } else if (trace_has_key(program, "custom_interpreter")) {
    i_inputs = [inputs, intervention_trace];
    interpret(
      program=*program[/custom_interpreter/],
      inputs=i_inputs,
      intervention_trace={ { } });
  } else if (trace_has_key(program, "source")) {
    new_env = make_env(parent=*program[/environment/]);
    match_bind(program[/source/pattern/], inputs, new_env);
    interpret_eval(program[/source/body/], new_env, intervention_trace);
  } else { pprint(program); error("Not_a_prob_prog"); }
```

Figure 10: A user-space meta-program for simulating Metaprob programs subject to interventions: probabilistic program invocation.

An interpreter such as this is the basic metaprogram that constitutes a programming language. One of the interesting properties of Metaprob is that the space of potentially useful such metaprograms is open, and indeed we will present several more in the sequel. This one in particular defines the effect of interventions.

```

interpret_eval = (exp, env, intervention_trace) -> {
  walk = (exp, addr) -> {
    v = if (*exp == "application") {
      n = length(trace_subkeys(exp));
      values = map((i) -> walk(exp[/$i/], addr + /$i/, range(n));
      oper = first(values);
      name = *oper[/name/];
      interpret(
        program=oper,
        inputs=rest(values),
        intervention_trace=intervention_trace[addr + /$name/];
      )
    } else if (*exp == "variable") {
      env_lookup(env, *exp[/name/]);
    } else if (*exp == "literal") {
      *exp[/value/];
    } else if (*exp == "program") {
      {{ "prob_prog", name = exp, source = **exp, environment = env }};
    } else if (*exp == "if") {
      pred = walk(exp[/predicate/], addr + /predicate/);
      if (pred) {
        walk(exp[/then/], addr + /then/);
      } else {
        walk(exp[/else/], addr + /else/);
      }
    } else if (*exp == "block") {
      n = length(trace_subkeys(exp));
      values = map((i) -> walk(exp[/$i/], addr + /$i/, range(n));
      if (length(values) > 0)
        last(values)
      else {{ }};
    } else if (*exp == "tuple") {
      n = length(trace_subkeys(exp));
      values = map((i) -> walk(exp[/$i/], addr + /$i/, range(n));
      list_to_array(values);
    } else if (*exp == "definition") {
      subaddr = name_for_definiens(exp[/pattern/]);
      val = walk(exp[subaddr], addr + subaddr);
      match_bind(exp[/pattern/], val, env);
    } else if (*exp == "this") {
      capture_tag_address(intervention_trace, {{ }}, {{ }});
    } else if (*exp == "with_address") {
      tag_addr = walk(exp[/tag/], addr + /tag/);
      (new_intervene, _, _) = resolve_tag_address(tag_addr);
      interpret_eval(exp[/expression/], env, new_intervene);
    } else {
      pprint(exp);
      error("Not_a_code_expression");
    }
  };
  if (intervention_trace[addr] has_value) {
    *intervention_trace[addr];
  } else {
    v;
  };
  walk(exp, /);
};

```

Figure 11: A user-space meta-program for simulating Metaprob programs subject to interventions: source interpretation.

```

make_env = (parent) -> {{ "environment", parent = parent }};

match_bind = (pat, val, env) -> {
  if (*pat == "variable") {
    name = *pat[/name/];
    if (name != "_") {
      env[/variables/$name/] := val;
    } else "ok";
  } else if (*pat == "tuple") {
    vals = collection_to_array(val);
    n = length(pat);
    for_each(range(n), (k) -> {
      match_bind(pat[/$k/], *vals[/$k/], env);
    });
  } else { pprint(pat); error("Invalid_binding_pattern."); };
};

env_lookup = (env, name) -> {
  if (trace_has_key(env[/variables/], name)) {
    *env[/variables/$name/];
  } else if (trace_has_key(env, "parent")) {
    env_lookup(*env[/parent/], name);
  } else { error("Unbound_variable_" + name); };
};

```

Figure 12: Implementation of lexical environments as trace data structures: construction, variable binding, and variable lookup.

### 3.3 A tracing metacircular interpreter

The fast subtrace selection makes prefix trees a good choice for representing recursively structured information such as program source code or program execution histories. This is the most interesting use of traces in Metaprob: tracing program executions.

Within a given procedure body, address keys represent walking down the syntax of the procedure: integers for successive statements; **predicate**, **then**, and **else** for the predicate, true, and false branches of an **if** statement, respectively; etc. Tracing the application of a procedure is recorded by adding the code body of that procedure as a key and recurring into its body. Thus any given address represents a stack trace, uniquely identifying a point in the execution history of a program.

```

trace_choices = (program, inputs, intervention_trace, output_trace) ->
  if (trace_has_key(program, "custom_choice_tracer")) {
    tc_inputs = [inputs, intervention_trace, output_trace];
    interpret(
      program=*program[/custom_choice_tracer/],
      inputs=tc_inputs,
      intervention_trace={ } );
  } else if (trace_has_key(program, "source")) {
    new_exp = program[/source/body/];
    new_env = make_env(parent=*program[/environment/]);
    match_bind(program[/source/pattern/], inputs, new_env);
    tc_eval(new_exp, new_env, intervention_trace, output_trace);
  } else if (trace_has_key(program, "executable")) {
    // Default choice tracing protocol
    val = interpret_prim(*program[/executable/], inputs, intervention_trace);
    output_trace := val;
    val;
  } else { pprint(program); error("Not_a_prob_prog"); };

```

Figure 13: A user-space meta-program for simulating Metaprob programs subject to interventions and tracing the choices they make: probabilistic program invocation.

Figures 13 and 14 give Metaprob source for a Metaprob interpreter that supports interventions and recording program executions into an `output_trace`. This gives the standard addressing scheme for what addresses name what points in the execution of a (traced) Metaprob program.

That addressing scheme can be a bit cumbersome to use. For this reason, Metaprob supports the `with_address` construct (and the complementary `&this` construct). Those two together provide an escape hatch by which the user can define their own addresses for parts of the program that they expect, when writing it, to access. We used this in the `flip_coins` example to simplify the addresses of the individual data points. For comparison, Figure 15 shows what a trace of the same run of that program would look like if we didn't have the `with_address` form there.

```

tc_eval = (exp, env, intervention_trace, output_trace) -> {
  walk = (exp, addr) -> {
    v = if (*exp == "application") {
      n = length(trace_subkeys(exp));
      values = map((i) -> walk(exp[/$/], addr + /$/), range(n));
      oper = first(values);
      name = *oper[/name/];
      trace_choices(
        program=oper,
        inputs=rest(values),
        intervention_trace=intervention_trace[addr + /$name/],
        output_trace=output_trace[addr + /$name/]);
    } else if (*exp == "variable") {
      env_lookup(env, *exp[/name/]);
    } else if (*exp == "literal") {
      *exp[/value/];
    } else if (*exp == "program") {
      {{ "prob_prog", name = exp, source = **exp, environment = env }};
    } else if (*exp == "if") {
      pred = walk(exp[/predicate/], addr + /predicate/);
      if (pred) {
        walk(exp[/then/], addr + /then/);
      } else {
        walk(exp[/else/], addr + /else/);
      }
    } else if (*exp == "block") {
      n = length(trace_subkeys(exp));
      values = map((i) -> walk(exp[/$/], addr + /$/), range(n));
      if (length(values) > 0)
        last(values)
      else {{ }};
    } else if (*exp == "tuple") {
      n = length(trace_subkeys(exp));
      values = map((i) -> walk(exp[/$/], addr + /$/), range(n));
      list_to_array(values);
    } else if (*exp == "definition") {
      subaddr = name_for_definiens(exp[/pattern/]);
      val = walk(exp[subaddr], addr + subaddr);
      match_bind(exp[/pattern/], val, env);
    } else if (*exp == "this") {
      capture_tag_address(intervention_trace, {{ }}, output_trace);
    } else if (*exp == "with_address") {
      tag_addr = walk(exp[/tag/], addr + /tag/);
      (new_intervene, _, new_output) = resolve_tag_address(tag_addr);
      tc_eval(exp[/expression/], env, new_intervene, new_output);
    } else {
      pprint(exp);
      error("Not_a_code_expression");
    }
  };
  if (intervention_trace[addr] has_value) {
    *intervention_trace[addr];
  } else {
    v;
  };
};
walk(exp, /);
};

```

Figure 14: A user-space meta-program for simulating Metaprob programs subject to interventions and tracing the choices they make: source interpretation.

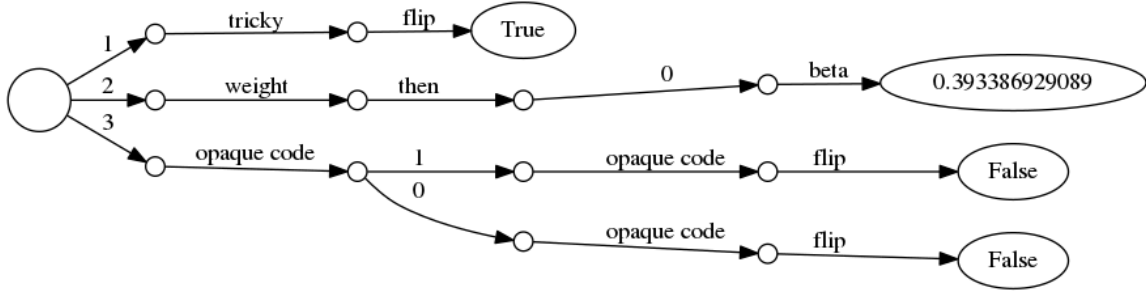


Figure 15: The `with_address` form allows a user to change the addressing scheme used to trace program executions. For an example of its effect, here’s what the trace from Figure 2 would look like if the `with_address` were ignored. The leftmost “opaque code” label refers to the call to `map`, which actually uses `with_address` internally to address calls to the procedure being mapped in a more intuitive way. The two rightmost “opaque code” labels refer to applications of the anonymous procedure whose body consists of `flip(weight)`.

### 3.4 Probabilistic Semantics

**Notation** In equations, we will denote traces by lower-case Greek letters  $\tau$ ,  $\phi$ ,  $\rho$ ,  $\xi$ , and addresses by lower-case Roman letters  $i$ ,  $j$ , to connote indexing, and also  $s$  for “site”. We say that an address  $i$  *occurs in* a trace  $\tau$  if  $\tau$  has a value for it, and we write the value  $\tau_i$ . By abuse of notation, we denote the set of addresses that occur in a trace by the same letter as the trace itself.

Given a program  $\mathcal{P}$ , inputs  $\mathbf{in}$ , and a trace  $\tau$ , we call an execution of  $\mathcal{P}$  on  $\mathbf{in}$  *consistent with*  $\tau$  if, for every address  $i$  the execution encountered, if  $i$  also occurs in  $\tau$ , then the value at that point was  $\tau_i$ . Note that different executions of the same  $\mathcal{P}$  on the same  $\mathbf{in}$  may encounter different addresses, because some stochastic choice in  $\mathcal{P}$  may influence downstream control flow. Both the execution and the trace are free to contain other addresses. We use the same language for traces: a trace  $\tau$  is *consistent with* another trace  $\rho$  if they agree on all addresses they share:  $\forall_{i \in \tau \cap \rho} \tau_i = \rho_i$ .

We call a trace  $\tau$  *complete* for  $\mathcal{P}$  on  $\mathbf{in}$  if there is a unique execution of  $\mathcal{P}$  on  $\mathbf{in}$  that is consistent with  $\tau$ . For a trace to be complete, it suffices for it to record a value for every random choice made during the corresponding execution.

We call a trace  $\tau$  *supported* by  $\mathcal{P}$  on  $\mathbf{in}$  if every execution of  $\mathcal{P}$  on  $\mathbf{in}$  that is consistent with  $\tau$  encounters every address in  $\tau$ . Note that a given execution may be consistent with more than one complete, supported trace, because traces are free to redundantly record deterministic intermediate values. When we speak of  $\tau$  as a trace *of* an execution of a program  $\mathcal{P}$  on inputs  $\mathbf{in}$ , we mean that  $\tau$  is consistent with and supported by that execution, but not necessarily complete.

To define the probability of a given execution history or recorded trace, we need notation for the local probability distribution that obtains at a given address. Since addresses are unique, the functional form of the relevant probability density is uniquely determined. The distribution may, however, be parameterized by previously computed values. Thus, for a complete trace  $\tau$  of an execution of  $\mathcal{P}$  on  $\mathbf{in}$ , we write  $p_i^{\mathcal{P}(\mathbf{in})}(\tau_i | \tau_i^{\mathbf{args}})$  for the probability distribution on values  $\tau_i$  that occur at address  $i$ .  $\tau_i^{\mathbf{args}}$  is taken to represent the direct parents of the computation at address  $i$ . We write  $p^{\mathcal{P}(\mathbf{in})}$  to emphasize the dependence on the program; a trace does not have a probabilistic semantics if taken alone.

Writing  $\tau$  for the `output_trace`, the probability density on traces written by `trace_choices` on program  $\mathcal{P}$  with inputs  $\mathbf{in}$  and intervention trace  $\phi$  is written  $p^{\mathcal{P}(\mathbf{in})}(\tau | do(\phi))$  and given by

$$p^{\mathcal{P}(\mathbf{in})}(\tau | do(\phi)) = \prod_{i \in \tau - \phi} p_i^{\mathcal{P}(\mathbf{in})}(\tau_i | \tau_i^{\mathbf{args}}) \prod_{i \in \tau \cap \phi} \mathbb{1}\{\tau_i = \phi_i\}. \quad (1)$$

[This ignores any records that may have been present in  $\tau$  before it was passed to `trace_choices`. Those at addresses encountered during the execution are overwritten; others are left be.] Note that, as  $\tau$  is complete, the value that `trace_choices` returns is deterministic conditioned on  $\tau$ .

The probability distribution on values returned by `interpret` is the same as for `trace_choices` on the same inputs, marginalizing over possible `output_traces`.

### 3.5 Probabilistic Programs

A Metaprob probabilistic program (what is called a “function” or “procedure” in other programming languages) is directly represented by a trace which has some or all of the keys listed in Table 1. Of particular note are the `/executable/`, which corresponds to the code body of a procedure that would be considered “primitive” in another programming language; and the `/source/` (and `/environment/`), which comprise the information necessary to evaluate the program as a Metaprob compound.

These two representations are generally considered mutually exclusive in other programming languages, as the usual idea is that the only thing that can be done with a program is to execute it. Metaprob, however, admits an open set of metaprograms, which may use a `/executable/` or the `/source/` in various ways, even if both are available. Such metaprograms are also free to define their own annotation types and protocols. Indeed, in this chapter we use the other annotations listed in Table 1 for various purposes. Beyond the content of the chapter, this annotation mechanism could also be used to implement, for example, automatic differentiation.

It is worth explicitly describing the contracts of the annotations that are themselves probabilistic programs. For the sake of pedagogy, let’s start with `/log_output_probability_density/`. Consider a probabilistic program  $\mathcal{P}$  whose behavior (absent constraints or interventions) is mathematically characterized by a probability distribution  $p$ . To wit,

$$\Pr(\mathcal{P}(\mathbf{in}) = \mathbf{v}) = p(\mathbf{v}|\mathbf{in}).$$

Then, if we have access to a Metaprob program that computes  $p$ , we can annotate  $\mathcal{P}$  with that program at the `/log_output_probability_density/` address, and it will henceforth behave as an opaque primitive to the standard interpreters. Figure 16 shows code implementing the Gaussian distribution this way (via the Box-Muller sampling transform). In particular, a call to such an annotated  $\mathcal{P}$  will register as one address, and the annotation program will be used to compute  $p$  whenever it appears in a term such as (3). The type signature of the `/log_output_probability_density/` program annotating a program  $\mathcal{P} : i_1, i_2, \dots, i_n \mapsto o$  must be  $o, i_1, i_2, \dots, i_n \mapsto \mathbb{R}$ , in log space.

The other program-shaped annotations we use in this chapter are the four custom interpreter overrides. They all serve a fundamentally similar purpose. The idea is that these annotations permit in-language definitions of “primitive” probabilistic programs that have the same internal flexibility as possessed by a compound procedure interpreted by `interpret` or `trace_choices`, above, or `propose` or `propose_and_trace_choices`, below.

For a program  $\mathcal{P}$  with type signature  $i_1, i_2, \dots, i_n \mapsto o$ , any `/custom_interpreter/` annotation is expected to have the type signature  $[i_1, i_2, \dots, i_n], \tau \mapsto o$ . Here square brackets denote the arguments coming in as an explicit list, and  $\tau$  is a trace. The custom interpreter is to sample from the probability distribution that  $\mathcal{P}$  (as annotated) represents, appropriately respecting any interventions in  $\tau$ . Taken alone, this is not actually much of a constraint at all. One of the things this can be used for is probabilistic programs that explicitly establish an internal addressing scheme (by reading relevant locations from  $\tau$ ) that is inconsistent with any possible Metaprob source code.

The `/custom_choice_tracer/` annotation is to `trace_choices` as the `/custom_interpreter/` annotation is to `interpret`. To wit, the type signature of such annotation is expected to be



<code>prog[/name/]</code>	A name for this program. This is used to demarcate applications of this program in the default addressing scheme
<code>prog[/executable/]</code>	Opaque code (in the host language) that can be just run
<code>prog[/source/]</code>	Metaprob source code (represented as a trace)
<code>prog[/environment/]</code>	Lexical environment in which the <code>prog[/source/]</code> can be interpreted
<code>prog[/log_output_probability_density/]</code>	A probabilistic program that computes the density of this program
<code>prog[/custom_interpreter/]</code>	A probabilistic program that explicitly implements interventions on this program (see <code>interpret</code> )
<code>prog[/custom_choice_tracer/]</code>	A probabilistic program that explicitly implements interventions on and choice tracing of this program (see <code>trace_choices</code> )
<code>prog[/custom_proposer/]</code>	A probabilistic program that explicitly implements interventions on and constraint of this program (see <code>propose</code> )
<code>prog[/custom_choice_tracing_proposer/]</code>	A probabilistic program that explicitly implements interventions on, constraints of, and choice tracing of this program (see <code>propose_and_trace_choices</code> )
<code>prog[/support/]</code>	A list of the values this program can return (used to enumerate possibilities in Figure 29)

Table 1: Annotations on Metaprob programs that are used in this chapter.

$[i_1, i_2, \dots, i_n], \tau_1, \tau_2 \mapsto o$ , and the behavior is to sample from the represented distribution, respecting any interventions from  $\tau_1$ , and writing any records to  $\tau_2$ .

In our thinking, we have generally assumed that any such annotation will define a single, complete set  $S$  of intervenable and recordable addresses, such that (i) any intervention present in  $\tau_1 \cap S$  will be recorded in  $\tau_2$ , and (ii) replaying a record in  $\tau_2$  as an intervention (with the same input arguments) will reproduce the same behavior as that which generated  $\tau_2$ . Whether violating this assumption can cause any harm, or be of any use, remains an open question.

Likewise, the `/custom_choice_tracing_proposer/` annotation is a userland override for `propose_propose_and_trace_choices`. The type signature of a `/custom_choice_tracing_proposer/` annotation is expected to be  $[i_1, i_2, \dots, i_n], \tau_1, \tau_2, \tau_3 \mapsto (o, \mathbb{R})$ . Now  $\tau_1$  is still an intervention trace, but  $\tau_2$  is the constraint trace, and the output records trace is  $\tau_3$ . A `/custom_choice_tracing_proposer/` has more freedom than the `propose_and_trace_choices`

metaprogram itself uses, in that it is free to read the content of  $\tau_2$  in order to choose its sampling distribution. Specifically, the weight the annotation returns is expected to obey the same probabilistic construct (4), but is free to implement whatever proposal distribution  $q$  it sees fit, including potentially arbitrary dependencies on the content of  $\tau_2$ .

Finally, `/custom_proposer/` is a simplification that is not called upon to record an output trace. The type signature expected of it is  $[i_1, i_2, \dots, i_n], \tau_1, \tau_2 \mapsto (o, \mathbb{R})$ .

We generally expect the four custom interpretation annotations to be consistent with each other. Specifically, if both a `/custom_proposer/` and a `/custom_choice_tracing_proposer/` are present, we expect the former to have the same effect as invoking the latter and forgetting the output trace, and etc. The status of this assumption is also open: we are not aware of any explicit dependencies on it in Metaprob, nor are we aware of any positive reason to behave otherwise.

```
std_normal = () ~> {
  u1 = uniform(0, 1);
  u2 = uniform(0, 1);
  sqrt(-2 × log(u1)) × cos(2 × 3.14159265 × u2);
};

std_normal_density = (x) -> {
  -0.5 × log(2 × 3.14159265) - 0.5 × x × x;
};

gaussian = (mu, sigma) ~> {
  mu + std_normal() × sigma;
};

gaussian[/log_output_probability_density/] := (x, mu, sigma) ~> {
  std_normal_density((x - mu) ÷ sigma) - log(sigma);
};
```

Figure 16: The Gaussian distribution, sampled using the Box-Muller transform, including the density annotation, as a user-space program.

Metaprob probabilistic programs are accessible to the language as such traces, and can be directly constructed and inspected. This gives Metaprob capabilities that are, as far as we know, unique among extant probabilistic programming systems. For example, Figure 16 shows how one can implement the Gaussian distribution in Metaprob by explicitly writing a sampler (using the Box-Muller transform from uniformly distributed samples, in this case) and a density function, and attaching the latter to the former as an annotation. Figure 17 shows setting up an inference problem using this new component; and we will cover performing inference with it in Section 4.

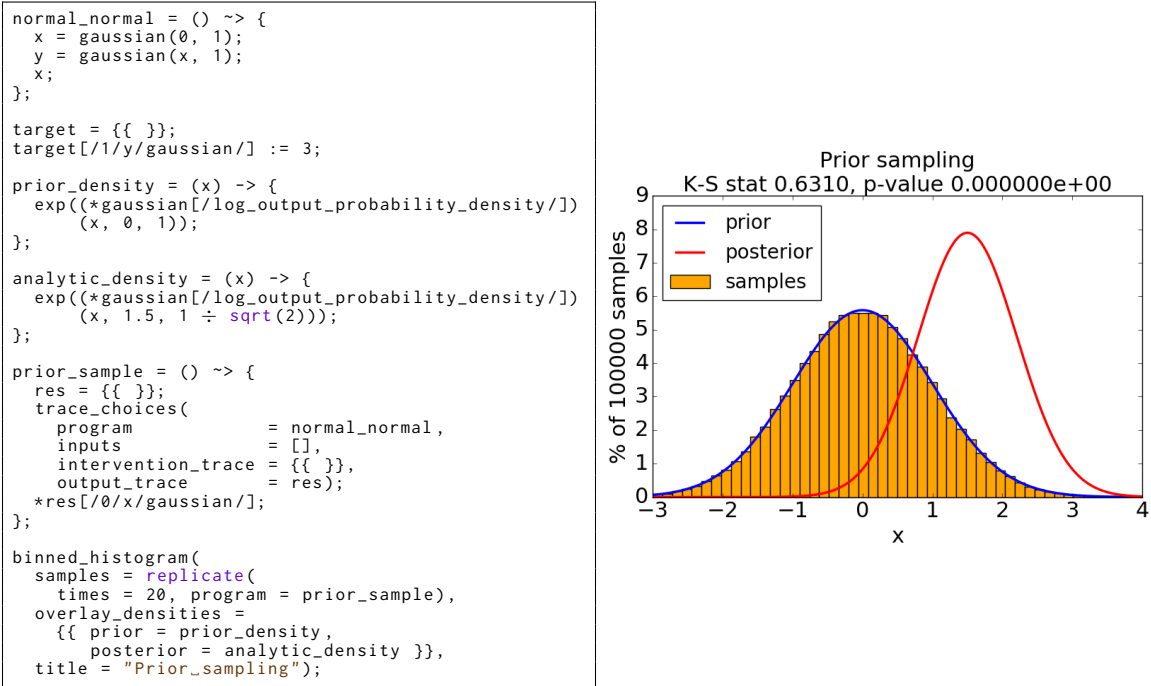


Figure 17: A model using the user-space Gaussian distribution of Figure 16. The left panel shows the model definition, as well as the analytic posterior density, and code for plotting a histogram of the prior. The right panel shows the samples thus obtained (compared to the analytic prior and posterior densities).

## 4. Metaprograms for inference

The capability that makes Metaprob interesting is support for writing in user space generic metaprograms for various forms of probabilistic inference over models defined by Metaprob programs. In this section, we discuss two standard building blocks (`propose` and `propose_and_trace_choices`) and several specific inference tactics that can be written with them.

### 4.1 Metacircular importance sampling proposer

```

propose = (program, inputs, intervention_trace, target_trace) ->
  if (trace_has_key(program, "custom_proposer")) {
    propose_inputs = [inputs, intervention_trace, target_trace];
    interpret(
      program=*program[/custom_proposer/],
      inputs=propose_inputs,
      intervention_trace={ }
    );
  } else if (trace_has_key(program, "source")) {
    new_exp = program[/source/body/];
    new_env = make_env(parent=*program[/environment/]);
    match_bind(program[/source/pattern/], inputs, new_env);
    propose_eval(new_exp, new_env, intervention_trace, target_trace);
  } else if (trace_has_key(program, "executable")) {
    // Default proposal protocol
    val = interpret_prim(*program[/executable/], inputs, intervention_trace);
    (val, 0);
  } else { pprint(program); error("Not_a_prob_prog"); }

```

Figure 18: A user-space meta-program for proposing Metaprob programs subject to interventions and constraints: probabilistic program invocation.

Figures 18 and 19 give a Metaprob interpreter that implements a canonical probabilistic conditioning mechanism. Figures 20 and 21 give its choice-tracing variant.

The probabilistic semantics for these two programs are as follows. The probability density that `propose_and_trace_choices` will write a given `output_trace`  $\rho$  given program  $\mathcal{P}$ , inputs `in`, a intervention trace  $\phi$ , and a target trace  $\tau$  is written  $q^{\mathcal{P}(\text{in})}(\rho|do(\phi), \tau)$  and given by

$$q^{\mathcal{P}(\text{in})}(\rho|do(\phi), \tau) = \prod_{i \in \rho - \phi - \tau} p_i^{\mathcal{P}(\text{in})}(\rho_i | \rho_i^{\text{args}}) \prod_{i \in \rho \cap \phi} \mathbb{1}\{\rho_i = \phi_i\} \prod_{i \in \rho \cap \tau} \mathbb{1}\{\rho_i = \tau_i\}. \quad (2)$$

Jointly with this, `propose_and_trace_choices` returns a weight given by

$$\text{weight} = \sum_{i \in \rho \cap \tau} \log p_i^{\mathcal{P}(\text{in})}(\rho_i | \rho_i^{\text{args}}). \quad (3)$$

This is designed to ensure that `propose_and_trace_choices` forms a valid importance sampling proposal for conditioning the unconstrained distribution of Equation 1 on consistency with  $\tau$ . Indeed, for  $\rho$  consistent with  $\phi$  and  $\tau$ ,

$$\begin{aligned}
\frac{p^{\mathcal{P}(\text{in})}(\rho|do(\phi))}{q^{\mathcal{P}(\text{in})}(\rho|do(\phi), \tau)} &= \frac{\prod_{i \in \rho - \phi} p_i^{\mathcal{P}(\text{in})}(\rho_i | \rho_i^{\text{args}}) \prod_{i \in \rho \cap \phi} \mathbb{1}\{\rho_i = \phi_i\}}{\prod_{i \in \rho - \phi - \tau} p_i^{\mathcal{P}(\text{in})}(\rho_i | \rho_i^{\text{args}}) \prod_{i \in \rho \cap \phi} \mathbb{1}\{\rho_i = \phi_i\} \prod_{i \in \rho \cap \tau} \mathbb{1}\{\rho_i = \tau_i\}} \\
&= \prod_{i \in \rho \cap \tau} p_i^{\mathcal{P}(\text{in})}(\rho_i | \rho_i^{\text{args}}) \\
&= \exp(\text{weight}).
\end{aligned} \quad (4)$$

```

propose_eval = (exp, env, intervention_trace, target_trace) -> {
  walk = (exp, addr) -> {
    (v, score) =
      if (*exp == "application") {
        n = length(trace_subkeys(exp));
        subscore = {{ 0 }};
        values = map((i) -> {
          (v, s) = walk(exp[/i/], addr + /i/);
          subscore := *subscore + s;
          v;
        }, range(n));
        oper = first(values);
        name = *oper[/name/];
        (val, score) = propose(
          program=oper,
          inputs=rest(values),
          intervention_trace=intervention_trace[addr + /$name/],
          target_trace=target_trace[addr + /$name/]);
      } else if (*exp == "variable") {
        (env_lookup(env, *exp[/name/]), 0);
      } else if (*exp == "literal") {
        (*exp[/value/], 0);
      } else if (*exp == "program") {
        ({{ "prob_prog", name = exp, source = **exp, environment = env }}, 0);
      } else if (*exp == "if") {
        (pred, p_score) = walk(exp[/predicate/], addr + /predicate/);
        if (pred) {
          (val, score) = walk(exp[/then/], addr + /then/);
          (val, p_score + score);
        } else {
          (val, score) = walk(exp[/else/], addr + /else/);
          (val, p_score + score);
        };
      } else if (*exp == "block") {
        n = length(trace_subkeys(exp));
        subscore = {{ 0 }};
        values = map((i) -> {
          (v, s) = walk(exp[/i/], addr + /i/);
          subscore := *subscore + s;
          v;
        }, range(n));
        if (length(values) > 0) {
          (last(values), *subscore);
        } else {
          ({{ }}, *subscore);
        };
      } else if (*exp == "tuple") {
        n = length(trace_subkeys(exp));
        subscore = {{ 0 }};
        values = map((i) -> {
          (v, s) = walk(exp[/i/], addr + /i/);
          subscore := *subscore + s;
          v;
        }, range(n));
        (list_to_array(values), *subscore);
      } else if (*exp == "definition") {
        subaddr = name_for_definiens(exp[/pattern/]);
        (val, score) = walk(exp[subaddr], addr + subaddr);
        (match_bind(exp[/pattern/], val, env), score);
      } else if (*exp == "this") {
        (capture_tag_address(intervention_trace, target_trace, {{ }}), 0);
      } else if (*exp == "with_address") {
        (tag_addr, tag_score) = walk(exp[/tag/], addr + /tag/);
        (new_intervene, new_target, _) = resolve_tag_address(tag_addr);
        (val, score) = propose_eval(exp[/expression/], env, new_intervene, new_target);
        (val, tag_score + score);
      } else {
        pprint(exp);
        error("Not_a_code_expression");
      };
      if (intervention_trace[addr] has_value) {
        (*intervention_trace[addr], score);
      } else {
        (v, score);
      };
    };
    walk(exp, /);
  };
};

```

Figure 19: A user-space meta-program for proposing Metaprob programs subject to interventions and constraints: source interpretation.

```

propose_and_trace_choices =
(program, inputs, intervention_trace, target_trace, output_trace) ->
  if (trace_has_key(program, "custom_choice_tracing_proposer")) {
    ptc_inputs = [inputs, intervention_trace, target_trace, output_trace];
    interpret(
      program=*program[/custom_choice_tracing_proposer/],
      inputs=ptc_inputs,
      intervention_trace={ { } });
  } else if (trace_has_key(program, "source")) {
    new_env = make_env(parent=*program[/environment/]);
    match_bind(program[/source/pattern/], inputs, new_env);
    ptc_eval(program[/source/body/], new_env,
      intervention_trace, target_trace, output_trace);
  } else if (trace_has_key(program, "executable")) {
    // Default proposal+tracing protocol
    val = interpret_prim(*program[/executable/], inputs, intervention_trace);
    output_trace := val;
    (val, 0);
  } else { pprint(program); error("Not_a_prob_prog"); };

```

Figure 20: A user-space meta-program for proposing Metaprob programs subject to interventions and constraints, and tracing the choices they make: probabilistic program invocation.

The `propose` metaprogram has the same probabilistic behavior as `propose_and_trace_choices`, it just doesn't record an output trace (so the probability distribution on values and weights is the appropriate marginal).

```

ptc_eval = (exp, env, intervention_trace, target_trace, output_trace) -> {
  walk = (exp, addr) -> {
    (v, score) =
      if (*exp == "application") {
        n = length(trace_subkeys(exp));
        subscore = {{ 0 }};
        values = map((i) -> {
          (v, s) = walk(exp[/i/], addr + /i/);
          subscore := *subscore + s;
          v;
        }, range(n));
        oper = first(values);
        name = *oper[/name/];
        (val, score) = propose_and_trace_choices(
          program=oper,
          inputs=rest(values),
          intervention_trace=intervention_trace[addr + /$name/],
          target_trace=target_trace[addr + /$name/],
          output_trace=output_trace[addr + /$name/]);
        (val, *subscore + score);
      } else if (*exp == "variable") {
        (env_lookup(env, *exp[/name/]), 0);
      } else if (*exp == "literal") {
        (*exp[/value/], 0);
      } else if (*exp == "program") {
        ({{ "prob_prog", name = exp, source = **exp, environment = env }}, 0);
      } else if (*exp == "if") {
        (pred, p_score) = walk(exp[/predicate/], addr + /predicate/);
        if (pred) {
          (val, score) = walk(exp[/then/], addr + /then/);
          (val, p_score + score);
        } else {
          (val, score) = walk(exp[/else/], addr + /else/);
          (val, p_score + score);
        };
      } else if (*exp == "block") {
        n = length(trace_subkeys(exp));
        subscore = {{ 0 }};
        values = map((i) -> {
          (v, s) = walk(exp[/i/], addr + /i/);
          subscore := *subscore + s;
          v;
        }, range(n));
        if (length(values) > 0) {
          (last(values), *subscore);
        } else {
          ({{ }}, *subscore);
        };
      } else if (*exp == "tuple") {
        n = length(trace_subkeys(exp));
        subscore = {{ 0 }};
        values = map((i) -> {
          (v, s) = walk(exp[/i/], addr + /i/);
          subscore := *subscore + s;
          v;
        }, range(n));
        (list_to_array(values), *subscore);
      } else if (*exp == "definition") {
        subaddr = name_for_definiens(exp[/pattern/]);
        (val, score) = walk(exp[subaddr], addr + subaddr);
        (match_bind(exp[/pattern/], val, env), score);
      } else if (*exp == "this") {
        (capture_tag_address(intervention_trace, target_trace, output_trace), 0);
      } else if (*exp == "with_address") {
        (tag_addr, tag_score) = walk(exp[/tag/], addr + /tag/);
        (new_intervene, new_target, new_output) = resolve_tag_address(tag_addr);
        (val, score) = ptc_eval(exp[/expression/], env, new_intervene, new_target, new_output);
        (val, tag_score + score);
      } else {
        pprint(exp);
        error("Not_a_code_expression");
      };
      if (intervention_trace[addr] has_value) {
        (*intervention_trace[addr], score);
      } else {
        (v, score);
      };
    };
    walk(exp, /);
  };
};

```

Figure 21: A user-space meta-program for proposing Metaprob programs subject to interventions and constraints, and tracing the choices they make: source interpretation.

## 4.2 Exact inference by trial and rejection

```

rejection_sampling = (program, inputs, target, log_bound) ~> {
  candidate_trace = {{ }};
  (_, score) = propose_and_trace_choices(
    program          = program,
    inputs           = inputs,
    intervention_trace = {{ }},
    target_trace      = target,
    output_trace      = candidate_trace);
  if (log(uniform(0, 1)) < score - log_bound) {
    candidate_trace;
  } else {
    rejection_sampling(program, inputs, target, log_bound);
  };
};

```

Figure 22: A meta-program for drawing samples from the exact conditional distribution on traces using rejection sampling.

Figure 22 gives a generic implementation of the rejection sampling algorithm for drawing traces of a given program that are consistent with a given target trace. Intuitively, rejection sampling is a probabilistic form of “generate and test”—one makes up an example from a so-called *proposal distribution*, evaluates how good it is, and randomly decides whether to keep it or try again based on a criterion designed to ensure that the distribution on accepted samples is the desired one.

For those in the know, the  $q$  distribution of Equation 3 serves as the proposal distribution. The soundness condition is that the given `log_bound` must be no less than the maximum possible `weight`. By Equation 4, the `weight` is exactly the density ratio needed for the rejection algorithm.

Figure 23 shows how to use the `rejection_sampling` inference tactic to draw samples from the posterior of a model (in this case, the normal-normal model defined in Figure 17). Observe that the Gaussian distribution that appears in the model was defined in the Metaprob user space (Figure 16); yet it participates fully in this inference scheme.

## 4.3 Approximate inference by sampling, weighting, and resampling complete traces

Metaprob gives the user the tools to employ many different inference tactics besides rejection sampling. They offer a range of problem-specific tradeoffs between computational cost and accuracy of the resulting solutions. Figure 24 shows an instance of another widely used and generic inference scheme called importance sampling with resampling.

Intuitively, the scheme consists of drawing some pre-selected number of candidates, which are traditionally called *particles*, evaluating their fit with the desired distribution, and randomly selecting one to keep in proportion to those weights. The advantage over rejection



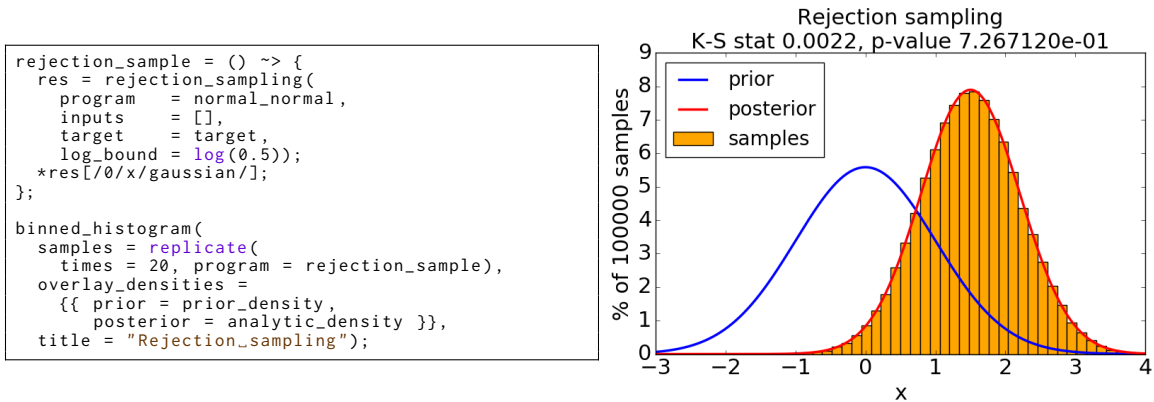


Figure 23: Rejection sampling in the model from Figure 17, using the user-space Gaussian distribution of Figure 16. The left panel shows code for plotting a histogram of the posterior by rejection sampling. The right panel the samples thus obtained (compared to the analytic prior and posterior densities).

sampling is that the computational cost is bounded and controllable; but the disadvantage is that the resulting distribution is only an approximation to the target. As the number of particles drawn increases, the approximation moves from the proposal distribution toward the target distribution, as illustrated in the bottom pane of Figure 24.

```

importance_sample = () ~> {
  trace_scores = {{ }};
  traces = {{ }};
  for_each(range(p), (i) ~> {
    candidate_trace = {{ }};
    (_, score) = propose_and_trace_choices(
      program      = normal_normal,
      inputs       = [],
      intervention_trace = {{ }},
      target_trace  = target,
      output_trace  = candidate_trace);
    trace_scores[/ $i/] := score;
    traces[/ $i/] := candidate_trace;
  });
  result = sample_discrete_random_variate(
    outputs      = traces,
    log_weights  = trace_scores);
  *result[/ 0/x/gaussian/];
};

binned_histogram(
  samples = replicate(times = 20, program = importance_sample),
  overlay_densities =
    {{ prior = prior_density, posterior = analytic_density }},
  title = "Importance_sampling_(" + string(p) + "_particles)");

```

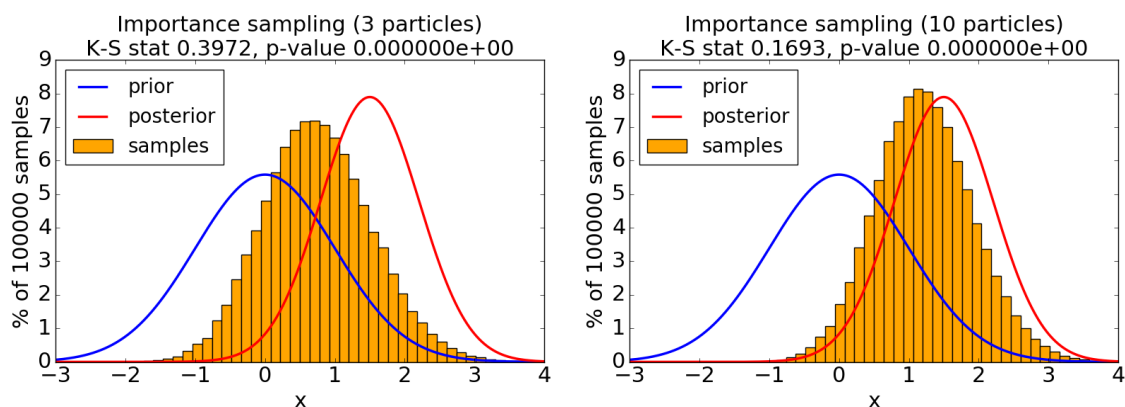


Figure 24: Importance sampling inference (with  $p$  particles) over the user-space Gaussian, in the model of Figure 17. The top panel is a transcript of invoking importance sampling on this problem, and the bottom panel shows histograms of the samples thus obtained.

#### 4.4 A metaprogram for approximate inference by changing one random choice at a time

```

1 single_site_metropolis_hastings_step =
2 (program, inputs, trace, constraint_addresses) ~>
3 {
4   // Choose a target site to propose uniformly
5   choice_addresses = addresses_of(trace);
6   candidates       = set_difference(choice_addresses,
7                                     constraint_addresses);
8   target_address   = uniform_sample(candidates);
9
10  // Compute a proposal trace
11  initial_value     = *trace[target_address];
12  initial_num_choices = length(candidates);
13  del trace[target_address];
14  new_trace = {{ }};
15  (_, forward_score) = propose_and_trace_choices(
16    program           = program,
17    inputs            = inputs,
18    intervention_trace = {{ }},
19    target_trace      = trace,
20    output_trace      = new_trace);
21  new_value          = *new_trace[target_address];
22
23  // The proposal is to move from trace to new_trace;
24  // Now compute the acceptance ratio.
25  new_choice_addresses = addresses_of(new_trace);
26  new_candidates       = set_difference(new_choice_addresses,
27                                        constraint_addresses);
28  new_num_choices      = length(new_candidates);
29  restoring_trace = {(*$target_address = initial_value)};
30  for_each( set_difference(choice_addresses,
31                          new_choice_addresses),
32    (addr) -> { restoring_trace[addr] := *trace[addr]; });
33  del new_trace[target_address];
34  (_, reverse_score) = propose(
35    program           = program,
36    inputs            = inputs,
37    intervention_trace = restoring_trace,
38    target_trace      = new_trace);
39  new_trace[target_address] := new_value;
40  log_p_accept = forward_score - reverse_score
41    - log(initial_num_choices) + log(new_num_choices);
42
43  if (log(uniform(0, 1)) < log_p_accept) { // Accept
44    for_each( set_difference(choice_addresses,
45                          new_choice_addresses),
46      (addr) -> { del trace[addr]; });
47    for_each( new_choice_addresses,
48      (addr) -> { trace[addr] := *new_trace[addr]; });
49  } else { // Reject
50    trace[target_address] := initial_value;
51  };
52 };

```

Figure 25: A user-space meta-program for performing a single step of “lightweight” Metropolis-Hastings inference. The program uses standard meta-programs for proposing program traces subject to constraints to create a proposal trace and calculate the acceptance ratio.

Another common but more complex strategy for inference can be thought of by analogy with incremental-improvement searches for constraint satisfaction problems. Namely, one starts with some candidate trace, and randomly incrementally modifies one location at a time with a mechanism constructed to lead, after a large number of modifications, to a sample from the desired distribution.

Figure 25 shows a Metaprob implementation for a generic metaprogram that takes one such step on any extant trace. In technical jargon, this might be called a single-site

Metropolis-Hastings transition operator (with local resimulation proposals). To argue that it works as advertised, we need to speak more formally:

Holding fixed the **program**  $\mathcal{P}$ , inputs **in**, and constraint addresses  $C$ , the metaprogram in Figure 25 implements a transition operator  $\mathcal{T}^{\mathcal{P}(\text{in}),C}$  that stochastically mutates its input trace.

Claim: The stationary distribution of the Markov chain induced by iterating  $\mathcal{T}^{\mathcal{P}(\text{in}),C}$  is the conditional distribution  $p^{\mathcal{P}(\text{in})}(\tau|\tau_C)$  where  $\tau_C$  is the set of values present in  $\tau$  at the **constraint\_addresses**.

Proof:  $\mathcal{T}^{\mathcal{P}(\text{in}),C}$  is a Metropolis-Hastings style transition operator, so it suffices to prove that the computed **log\_p\_accept** is the logarithm of the M-H acceptance ratio  $\alpha_{\mathcal{T}^{\mathcal{P}(\text{in}),C}}$ . Following the notation from ?, let us call the input **trace**  $\rho$  and the proposed **new\_trace**  $\xi$ . Let us also call the selected **target\_address**  $s$ . The target probability  $p^{\mathcal{P}(\text{in})}(\rho|do(\phi), \rho_C)$  is given, up to an unknown normalization constant  $Z_{\rho_C}$ , by

$$p^{\mathcal{P}(\text{in})}(\rho|do(\phi), \rho_C) = \frac{1}{Z_{\rho_C}} \prod_{i \in \rho - \phi} p_i^{\mathcal{P}(\text{in})}(\rho_i|\rho_i^{\text{args}}) \prod_{i \in \rho \cap \phi} \mathbb{1}\{\rho_i = \phi_i\}.$$

The proposal probability  $q(\xi|\rho)$  is determined by the uniform choice of  $s$  and the behavior of the **propose\_and\_trace\_choices** call that computes  $\xi$ . The latter constructs  $\xi$  by running **program** on **inputs**, except that for each choice  $i$  that occurs in  $\rho$ , except  $s$ , it forces the value  $\xi_i$  to agree with  $\rho_i$  and accumulates  $\log(p_i^{\mathcal{P}(\text{in})}(\xi_i|\xi_i^{\text{args}}))$  in the **new\_score**. Thus, the only stochastic behavior in this call is at  $s$  or at random choices in  $\xi$  that do not occur in  $\rho$  (which are possible because a different value for  $s$  may affect the control flow of the **program**). From this discussion we have

$$\begin{aligned} \xi_i &= \rho_i \text{ for } i \in \xi \cap \rho - \{s\}, \\ q(\xi|\rho) &= \frac{1}{N_\rho} p_s^{\mathcal{P}(\text{in})}(\xi_s|\xi_s^{\text{args}}) \prod_{i \in \xi - \rho} p_i^{\mathcal{P}(\text{in})}(\xi_i|\xi_i^{\text{args}}), \\ \text{new\_score} &= \sum_{i \in \xi \cap \rho - \{s\}} \log p_i^{\mathcal{P}(\text{in})}(\xi_i|\xi_i^{\text{args}}), \end{aligned}$$

where  $N_\rho$  is the number of addresses in  $\rho$  that are not constrained. Therefore,

$$\begin{aligned} \frac{p^{\mathcal{P}(\text{in})}(\xi)}{q^{\mathcal{P}(\text{in})}(\xi|\rho)} &= \frac{N_\rho \prod_{i \in \xi} p_i^{\mathcal{P}(\text{in})}(\xi_i|\xi_i^{\text{args}})}{p_s^{\mathcal{P}(\text{in})}(\xi_s|\xi_s^{\text{args}}) \prod_{i \in \xi - \rho} p_i^{\mathcal{P}(\text{in})}(\xi_i|\xi_i^{\text{args}})} \\ &= N_\rho \prod_{i \in \xi \cap \rho - \{s\}} p_i^{\mathcal{P}(\text{in})}(\xi_i|\xi_i^{\text{args}}) \\ &= \text{new\_num\_choices} * \exp(\text{new\_score}). \end{aligned}$$

Let us now examine the behavior of the **propose** call that computes **initial\_score**. Lemma: This call to **propose** exactly recapitulates the control flow and values of  $\rho$ . Proof: Proceed by induction on the execution history. Initially, the claim is true vacuously. Consider a choice  $i$  that this **propose** encounters, assuming previous choices are a replay of  $\rho$ . Because the previous values are the same, the control flow hasn't changed before reaching  $i$ .

Therefore,  $i$  is in the `choice_addresses`. If  $i = s$  or  $i$  is not in the `new_choice_addresses`, then  $i$  occurs in the `intervention_trace`, and the value is forced to  $\rho_i$ . Otherwise,  $i$  is in  $\xi \cap \rho - \{s\}$ , so (a) it occurs in `new_trace` and the value is set to  $\xi_i$ ; and (b)  $\xi_i = \rho_i$ .

In light of the above, the `propose` call accumulates density terms for all  $i \in \rho \cap \xi - \{s\}$ , and the parent values for those terms are drawn from  $\rho$ . Thus we have proven

$$\text{initial\_score} = \sum_{i \in \rho \cap \xi - \{s\}} \log p_i^{\mathcal{P}(\text{in})}(\rho_i | \rho_i^{\text{args}}).$$

Ergo,

$$\begin{aligned} \frac{p^{\mathcal{P}(\text{in})}(\rho)}{q^{\mathcal{P}(\text{in})}(\rho | \xi)} &= \frac{N_\xi \prod_{i \in \rho} p_i^{\mathcal{P}(\text{in})}(\rho_i | \rho_i^{\text{args}})}{p_s^{\mathcal{P}(\text{in})}(\rho_s | \rho_s^{\text{args}}) \prod_{i \in \rho - \xi} p_i^{\mathcal{P}(\text{in})}(\rho_i | \rho_i^{\text{args}})} \\ &= N_\xi \prod_{i \in \rho \cap \xi - \{s\}} p_i^{\mathcal{P}(\text{in})}(\rho_i | \rho_i^{\text{args}}) \\ &= \text{initial\_num\_choices} * \exp(\text{initial\_score}), \end{aligned}$$

and therefore

$$\begin{aligned} \alpha_{\mathcal{T}^{\mathcal{P}(\text{in}), C}} &= \frac{p^{\mathcal{P}(\text{in})}(\xi) q^{\mathcal{P}(\text{in})}(\rho | \xi)}{p^{\mathcal{P}(\text{in})}(\rho) q^{\mathcal{P}(\text{in})}(\xi | \rho)} \\ &= \left( \frac{p^{\mathcal{P}(\text{in})}(\xi)}{q^{\mathcal{P}(\text{in})}(\xi | \rho)} \right) \cdot \left( \frac{q^{\mathcal{P}(\text{in})}(\rho | \xi)}{p^{\mathcal{P}(\text{in})}(\rho)} \right) \\ &= \frac{\text{new\_num\_choices} * \exp(\text{new\_score})}{\text{initial\_num\_choices} * \exp(\text{initial\_score})} \\ &= \exp(\log\_p\_accept), \end{aligned}$$

as desired.

#### 4.4.1 HOW SINGLE-SITE METROPOLIS-HASTINGS OPERATES

In the interest of giving some intuition for how this metaprogram operates in practice, Figures 26 and 27 show two representative instrumented executions on traces of the `flip_coins` model program from Figure 1.

#### 4.4.2 RUNNING GAUSSIAN EXAMPLE

Figure 28 exemplifies another situation where Metropolis-Hastings inference can be invoked (again successfully using the in-language definition of the Gaussian distribution), as well as an indication of its approximateness and effectiveness.

program (on entry)	flip_coins
inputs (on entry)	[2]
trace (on entry)	
constraint_addresses (on entry)	/datum/0/flip/, /datum/1/flip/
choice_addresses (line 5)	/1/tricky/flip/, /2/weight/then/0/uniform/, /datum/0/flip/, /datum/1/flip/
candidates (line 6)	/1/tricky/flip/, /2/weight/then/0/uniform/
target_address (line 8)	/1/tricky/flip/
initial_value (line 11)	True
initial_num_choices (line 12)	2
new_value (line 21)	False
new_trace (at line 21)	
forward_score (line 15)	-1.38629436112
new_choice_addresses (line 25)	/1/tricky/flip/, /datum/0/flip/, /datum/1/flip/
new_candidates (line 26)	/1/tricky/flip/
new_num_choices (line 28)	1
restoring_trace (at line 32)	
reverse_score (line 34)	-0.351655574377
Proposal summary	Change /1/tricky/flip/ from True to False
Forward weight $\frac{p(False)}{q(False True)}$	$\frac{1}{2} * \exp(-1.38629436112) = 0.125$
Reverse weight $\frac{p(True)}{q(True False)}$	$\frac{1}{1} * \exp(-0.351655574377) = 0.70352239139$
Acceptance probability (line 40)	$\exp(-1.7277859673) = 0.17767735829$

Figure 26: Execution log of one step of `single_site_metropolis_hastings_step` on the trick coin example. Refer to Figure 25 for the line numbers.

program (on entry)	flip_coins
inputs (on entry)	[2]
trace (on entry)	
constraint_addresses (on entry)	/datum/0/flip/, /datum/1/flip/
choice_addresses (line 5)	/1/tricky/flip/, /datum/0/flip/, /datum/1/flip/
candidates (line 6)	/1/tricky/flip/
target_address (line 8)	/1/tricky/flip/
initial_value (line 11)	False
initial_num_choices (line 12)	1
new_value (line 21)	True
new_trace (at line 21)	
forward_score (line 15)	-1.45265413562
new_choice_addresses (line 25)	/1/tricky/flip/, /2/weight/then/0/uniform/, /datum/0/flip/, /datum/1/flip/
new_candidates (line 26)	/1/tricky/flip/, /2/weight/then/0/uniform/
new_num_choices (line 28)	2
restoring_trace (at line 32)	
reverse_score (line 34)	-1.38629436112
Proposal summary	Change /1/tricky/flip/ from False to True
Forward weight $\frac{p(Trace)}{q(Trace)}$	$\frac{1}{1} * \exp(-1.45265413562) = 0.233948532213$
Reverse weight $\frac{p(Trace)}{q(Trace)}$	$\frac{1}{2} * \exp(-1.38629436112) = 0.125$
Acceptance probability (line 40)	$\exp(0.626787406057) = 1.87158825771$

Figure 27: Execution log of another step of `single_site_metropolis_hastings_step` on the trick coin example. Refer to Figure 25 for the line numbers.

```

mcmc_sample = () ~> {
  result = {{ }};
  constraints = addresses_of(target);
  trace_choices(
    program          = normal_normal,
    inputs           = args,
    intervention_trace = target,
    output_trace      = result);
  repeat(times = s, program = () ~> {
    single_site_metropolis_hastings_step(
      program = normal_normal,
      inputs  = args,
      trace   = result,
      constraint_addresses = constraints);
  });
  *result[/0/x/gaussian/];
};

binned_histogram(
  samples = replicate(times = 20, program = mcmc_sample),
  overlay_densities =
    {{ prior = prior_density, posterior = analytic_density }},
  title = "MCMC_" + string(s) + "_resimulation_steps");

```

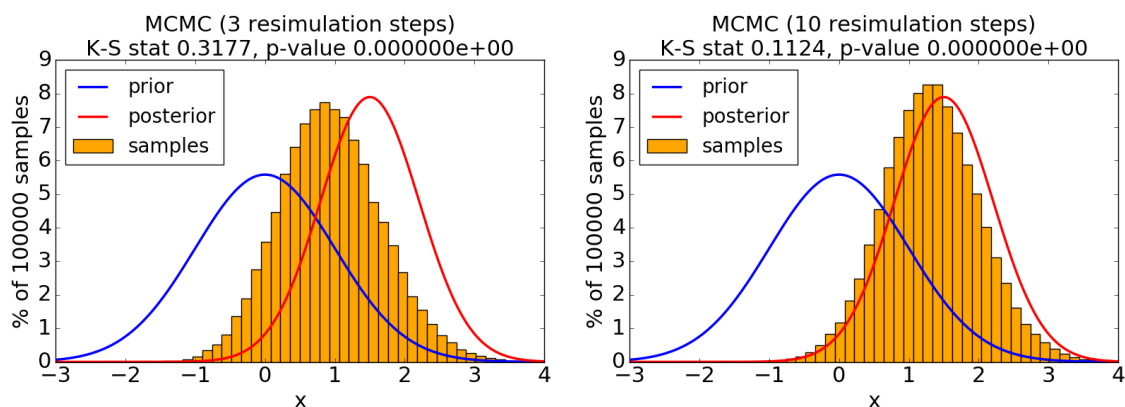


Figure 28: Markov-chain Monte Carlo inference over the user-space Gaussian, in the model of Figure 17. The top panel is a transcript of invoking MCMC (with a prior resimulation proposal), and the bottom panel shows histograms of the samples thus obtained.



## 4.5 Enumerating all possible traces for programs with finite support

```

joint_enumerate = (sites) ->
  if (is_pair(sites)) {
    others = joint_enumerate(rest(sites));
    oper_name = last(first(sites));
    // This only works if the enumeeree only applies
    // top-level prob progs
    oper = toplevel_lookup(oper_name);
    value_candidates = *oper[/support/];
    trace_lists = map((value) -> {
      map((t) -> {
        t1 = trace_copy(t);
        t1[first(sites)] := value;
        t1; }, others);
      }, value_candidates);
    concat(trace_lists);
  } else {
    pair({{ }}, {{ }});
  };

enumerate_executions =
(program, inputs, intervention_trace, target_trace) -> {
  one_run = {{ }};
  trace_choices(
    program      = program,
    inputs       = inputs,
    intervention_trace = intervention_trace,
    output_trace = one_run);
  all_sites = addresses_of(one_run);
  free_sites =
    set_difference(
      set_difference(all_sites, addresses_of(intervention_trace)),
      addresses_of(target_trace));
  candidates = joint_enumerate(free_sites);
  map((candidate) -> {
    trace_update(candidate, target_trace);
    t = {{ }};
    (_, score) = propose_and_trace_choices(
      program      = program,
      inputs       = inputs,
      intervention_trace = intervention_trace,
      target_trace  = candidate,
      output_trace  = t);
    (t, score);
  }, candidates);
};

```

Figure 29: A meta-program for computing an exact distribution, represented by a materialized list of traces with accompanying weights.

Yet a third inferential that can be implemented in user-space in Metaprob is exhaustive enumeration of all the possibilities. Such enumeration is of course impossible for models that use primitives with unstructured infinite support, and quite subtle to implement with good coverage even if all primitives are assumed to have finite support. However, as shown in Figure 29, a basic version leaning on several simplifying assumptions can be quite compact.

This `enumerate_executions` program was used for the enumeration results shown in Figures 4, 5, and 6.

## 5. Metaprograms for modelling

### 5.1 Mem

```

mem = (f) -> {
  root = &this;
  cache = {{ }};
  (arg) ~> {
    addr = /$arg/;
    if (cache[addr] has_value)
      *cache[addr]
    else {
      val = with_address /$root/cache/ + addr: f(arg);
      cache[addr] := val;
      val;
    };
  };
};

```

Figure 30: The `mem` construct as a user-space program. Note the use of `with_address` to control the addresses at which invocations of the memoized procedure are traced, and to avoid tracing the state maintenance code inside the implementation of `mem` itself.

## 5.2 The Chinese Restaurant Process

```

new_crp_state = () -> {
  {{ tables = **{{ }},
    next_table = 1,
    num_customers = 0
  }};
};

crp_sample = (state, alpha) ~> {
  old_tables = trace_subkeys(state[/table_counts/]);
  old_counts = map((k) -> *state[/table_counts/$k/], old_tables);
  weights = pair(alpha, old_counts);
  candidates = pair(*state[/next_table/], old_tables);
  categorical(normalize(weights), candidates);
};

crp_log_density = (state, alpha, new_table) -> {
  prob_num = if (state[/table_counts/$new_table/] has_value) {
    log(*state[/table_counts/$new_table/]);
  } else if (new_table == *state[/next_table/]) {
    log(alpha);
  } else {
    -1÷0; // Impossible
  };
  prob_num - log(alpha + *state[/num_customers/]);
};

crp_incorporate = (state, new_table) -> {
  if (new_table >= *state[/next_table/]) {
    state[/next_table/] := new_table + 1;
  } else "ok";
  if (state[/table_counts/$new_table/] has_value) {
    state[/table_counts/$new_table/] :=
      *state[/table_counts/$new_table/] + 1;
  } else {
    state[/table_counts/$new_table/] := 1;
  };
  state[/num_customers/] := *state[/num_customers/] + 1;
};

```

Figure 31: The computational content of the Chinese Restaurant Process: how to create a summary of a fresh empty sequence, how to draw one more sample for a sequence summarized by the `state`, how to evaluate the log density of such a draw, and how to update the summary to account for a new draw. This is packaged into a user-space definition of a CRP modeling construct in Figure 32.

```

make_chinese_restaurant_sampler = (alpha) -> {
  state = new_crp_state();
  tracing_proposer = (args, intervention, target, output) ~> {
    if (target has_value) {
      new_table = *target;
      prob = crp_log_density(state, alpha, new_table);
      crp_incorporate(state, new_table);
      output := new_table;
      (new_table, prob);
    } else {
      if (intervention has_value) {
        crp_incorporate(state, *intervention);
        (*intervention, 0.0);
      } else {
        ans = crp_sample(state, alpha);
        crp_incorporate(state, ans);
        output := ans;
        (ans, 0.0);
      };
    };
  };
};

{{ "prob_prog",
  name = "crp",
  custom_interpreter = ((args, intervene) ~> {
    (v, _) = tracing_proposer(args, intervene, {{ }}, {{ }});
    v;
  }),
  custom_choice_tracer = ((args, intervene, output) ~> {
    (v, _) = tracing_proposer(args, intervene, {{ }}, output);
    v;
  }),
  custom_proposer = ((args, intervene, target) ~> {
    tracing_proposer(args, intervene, target, {{ }});
  }),
  custom_choice_tracing_proposer = tracing_proposer
}};
};

```

Figure 32: Packaging the computational content of the CRP (from Figure 31) into a user-space modeling construct. Each instance of a Chinese Restaurant Process is represented by a prob prog that can be invoked to draw successive assignments. This prob prog's behavior is given by the `tracing_proposer`.

### 5.3 Dirichlet Process Mixture Example

```

generate_from_dirichlet_process_mixture =
(num_datapoints) ~> {
  root_addr      = &this;

  alpha          ~ gamma(1.0, 1.0);
  sample_assignment = make_chinese_restaurant_sampler(alpha);
  get_cluster     = mem( (i) ~> { sample_assignment(); } );

  a              ~ inverse_gamma(3.0, 10.0);
  b              ~ uniform(0.0, 10.0);
  mu             ~ uniform(-150.0, 150.0);
  V              ~ inverse_gamma(2, 100);

  get_mu         = mem( (cluster) ~> {
    normal(mu, V × get_sigma(cluster));
  } );
  get_sigma      = mem( (cluster) ~> {
    sqrt(inverse_gamma(a, b));
  } );

  get_datapoint  = mem( (i) ~> {
    cluster = get_cluster(i);

    with_address /$root_addr/datum/$i/:
    normal( get_mu(cluster),
            get_sigma(cluster) );
  } );
  map(get_datapoint, range(num_datapoints));
};

```

Figure 33: Metaprob source code for a program that generates data from a Dirichlet process mixture of Gaussians with randomly chosen hyper-parameters. This program uses user-space implementations of `make_chinese_restaurant_sampler` and `mem`. It also uses `with_address` to ensure that the data generated by this sampler is easy to find, in traces of this program; see Figure 35 for code that depends on these address names.

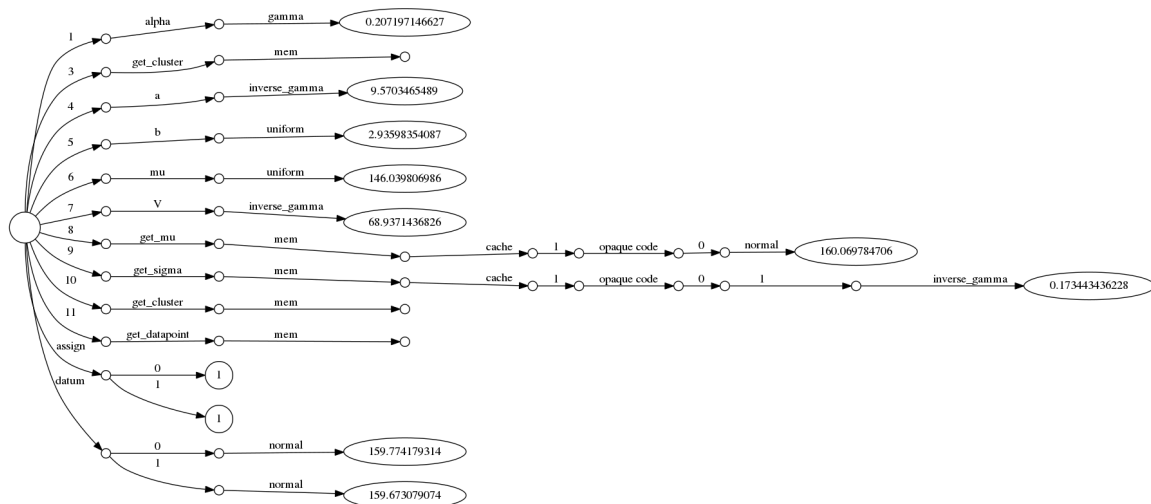


Figure 34: Trace from a small run of the model program of Figure 33

```

target_trace = {{ }};
data = [100, 101, -99, -100, 99, -101, 100.5, 99.5, -100.5, -99.5];
for_each(range(length(data)), (i) -> {
  target_trace[/datum/$i/normal/] := *data[/$i/];
});
data_addresses = addresses_of(target_trace);

markov_chain_state = {{ }};

trace_choices(
  program          = generate_from_dirichlet_process_mixture,
  inputs           = [length(data)],
  // ensures the initial state has the data
  intervention_trace = target_trace,
  output_trace     = markov_chain_state
);

approximate_inference_update = () ~> {
  single_site_metropolis_hastings_step(
    program = generate_from_dirichlet_process_mixture,
    inputs  = [length(data)],
    trace   = markov_chain_state,
    constraint_addresses = data_addresses
  );
};

repeat(
  times=1000,
  program = approximate_inference_update
);

all_data =
  interpret(
    program = generate_from_dirichlet_process_mixture,
    inputs  = [length(data) + 100], // 100 predictive samples
    intervention_trace = markov_chain_state );
predictive_data = all_data[length(data):end];

```

Figure 35: Metaprob source code for a meta-program that adds an observed dataset, does 1000 steps of lightweight Metropolis-Hastings inference on executions of the program from Figure 33, and then generates 100 samples from the predictive distribution. This meta-program traces the program to obtain an initial state, runs 1000 transitions, then re-executes the original program treating this final trace as source of interventions, to generate samples from the posterior predictive.



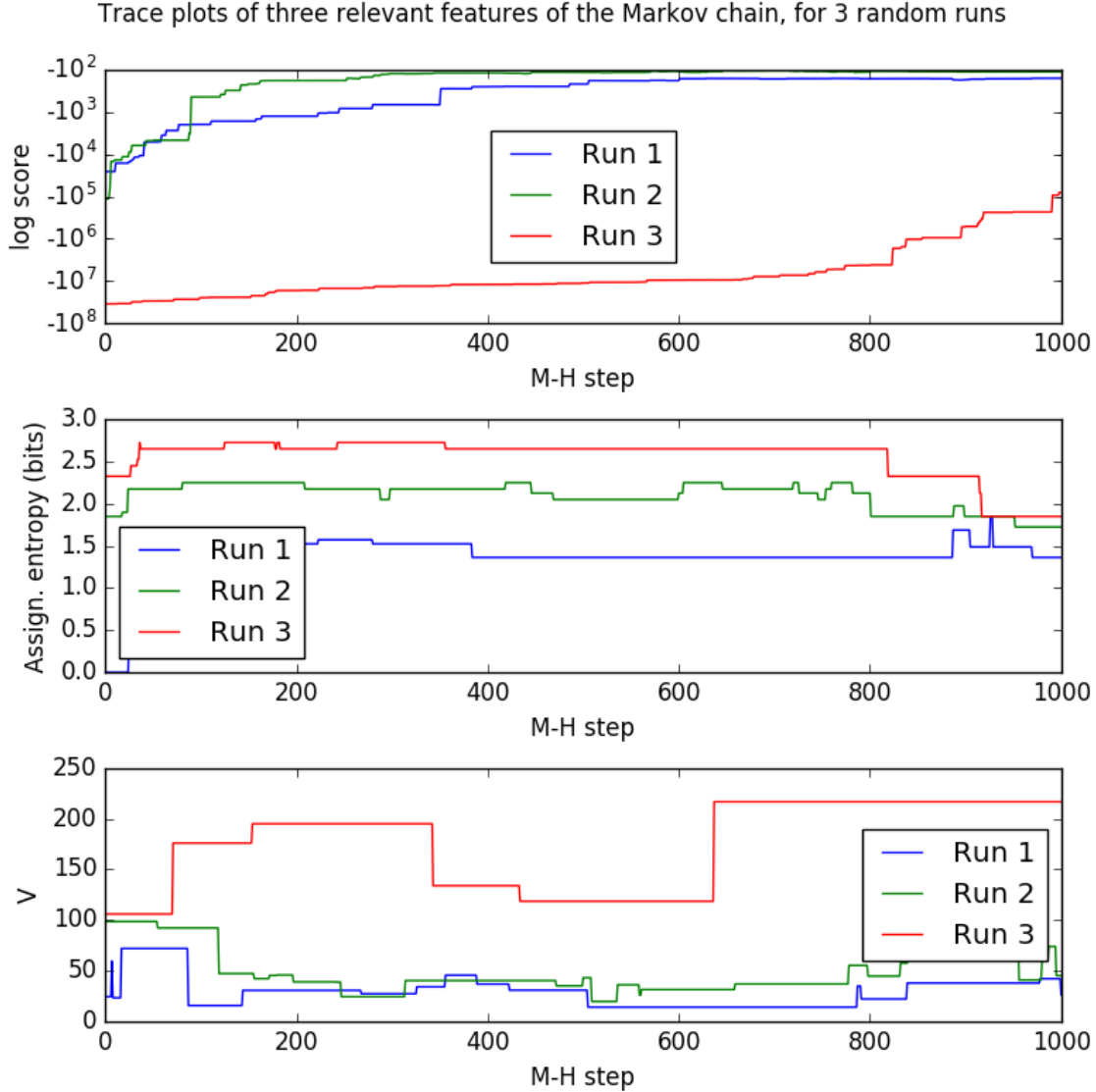


Figure 36: Progress of inference in the Dirichlet process mixture of Gaussians model with hyperparameter inference from Figure 33. The top panel shows the evolution of the log probability (density) of the current state. Note that the logarithm is itself displayed on a logarithmic scale. The center panel shows the Shannon entropy of the cluster assignment in bits: 0 corresponds to all points in one cluster, 1 to the points split evenly among two clusters, etc. The bottom panel shows the values taken by the  $V$  hyperparameter, which controls the cluster spread to cluster width ratio.

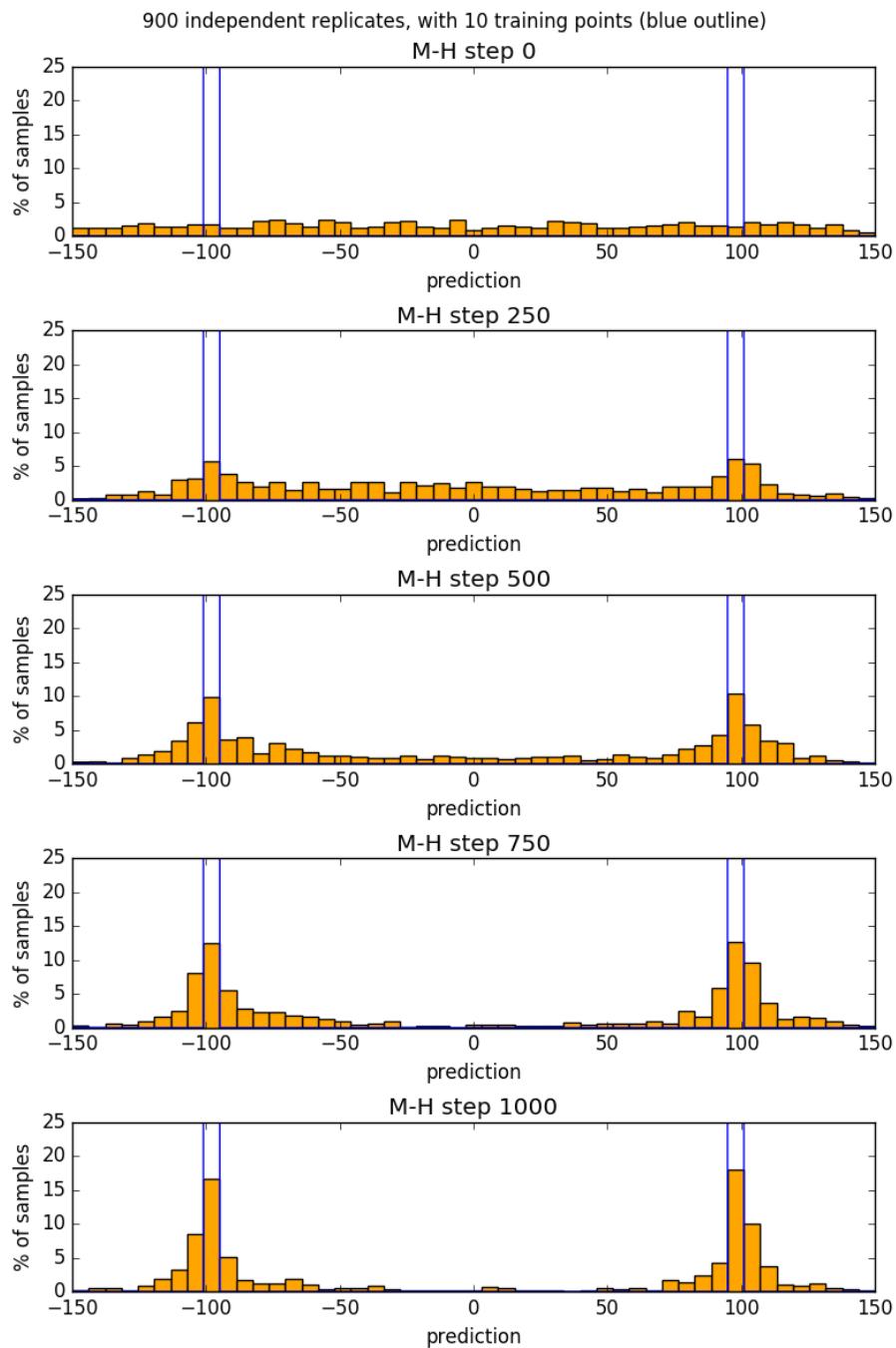


Figure 37: Progress of inference in the Dirichlet process mixture of Gaussians model with hyperparameter inference from Figure 33. Showing several iterates of the Markov chain defined by repeated application of resimulation M-H. The orange bars represent the observed data.

## 6. Discussion

### 6.1 Comparison to Traditional Programming

Metaprob surface syntax broadly follows in the block-structured tradition, whose modern adherents include C++, Java, and JavaScript, among many others. Its deterministic behavior is more directly inspired by the Lisp school, and specifically Scheme: Metaprob is eagerly evaluated, dynamically typed, memory-managed, and function- (rather than object-) oriented. Metaprob probabilistic programs are first-class objects in Metaprob.

The major difference in Metaprob is that program behavior is amenable to tracing. There is a well-defined (and user-adjustable, via `with_address`) subcomputation addressing scheme that gives a name to every point in the execution of any Metaprob program of interest. The random choices made during such execution can be captured into a trace, reifying them for manipulation by metaprograms.

Those metaprograms are what give Metaprob its name. In this chapter, we have studied the use of traced programs to define probabilistic models (in effect, injecting every computable probability question into the question “What plausible executions of this program likely produce this result?”), and several different metaprograms for solving (approximately and/or asymptotically) that question by operating on such reifications of program behavior.

The mechanism of “intervention” provides an inverse, a way to reflect a trace containing an execution record into a consistent program behavior. In the event the reflected record is partial, the behavior will complete it.

The other major feature of Metaprob is that Metaprob programs may carry annotations describing their probabilistic behavior (for example, a probability density function reporting the likelihood of a given output conditioned on given inputs). One may think of such a density as an unusually detailed type or contract: it describes not only which results are possible, but also how likely or unlikely they are. Densities make much finer distinctions than is traditional for a type system, but conversely there are no efficient and complete algorithms to infer the density associated with a composition from the densities associated with the parts—instead, a rich space of possible metaprograms that will be better or worse at evaluating or sampling from the distributions implied by one or another kind of problem.

Likewise, densities may themselves be more complex than (at least non-dependent) types. They are all at the very least numerical expressions, and are generally encoded as programs themselves. Why do program annotations as rich (and expensive to interpret) as densities arise in the probability space? One can view a stochastic process  $\mathcal{P}$  as a deterministic process that consumes an extra argument (a seed for a pseudo-random number generator). On one hand, executing  $\mathcal{P}$  once (that is, at one value of the seed) does not provide complete information about its behavior (across different seeds). On the other hand, we expect (or engineer)  $\mathcal{P}$  to exhibit some uniformity in its behavior as the extra argument varies. Thus it can be helpful to encode some knowledge about that variation, even if the encoding takes the form of a program (for computing the density function). One hopes that this describer program is more efficient or less stochastic than recovering the information in it by invoking  $\mathcal{P}$  a large number of times.

### 6.1.1 REFLECTION

A major strength of the Lisp family of programming languages is the simplicity with which program source code can be reified into Lisp data structures, and then reflected (via `eval`) back into running behavior. Metaprob repeats such reification, but directly on the behavior of the program, rather than its representation as source code.

As a source of capabilities for reflectively analyzing the detailed behavior of programs, inspecting program source code has proven too brittle: there are both too many ways to code superficially-different procedures whose behavior is substantially the same, and too many procedures with near-identical code but drastically different behavior. Translucent procedures <http://hdl.handle.net/1721.1/7053> are one approach to mitigating the first problem, by exposing the syntactic structure of a procedure only up to a structural equivalence.

Metaprob addresses the second problem by reifying program behavior directly. It suffers from the first, however—the standard addressing scheme is strongly dependent on the details of a program’s source code, and could perhaps benefit from more abstraction. Perhaps something along the lines of translucent procedures, applied consistently to traces of program behavior and addresses therein, could make reflective Metaprob programs easier to maintain.

## 6.2 Computational Efficiency

As presently implemented, the `propose` operations in Metaprob consist of rerunning the whole program being proposed, modulating its behavior with the interventions and constraints defined by the arguments to `propose`. In the case of a local inference program such as `single_site_metropolis_hastings_step`, this is asymptotically inefficient, since most of the behavior is not supposed to be changed by the proposal.

In the specific case of `single_site_metropolis_hastings_step`, consider some random choice  $p_i$  whose arguments and output are unaffected by the proposal to the target address  $s$ . Then both the `forward_score` and the `reverse_score` contain the same term  $p_i^{\mathcal{P}(\text{in})}(\xi_i | \xi_i^{\text{args}}) = p_i^{\mathcal{P}(\text{in})}(\rho_i | \rho_i^{\text{args}})$ , which therefore cancels from the final acceptance ratio. It is therefore an opportunity for savings to avoid computing it in the first place. In fact, to the extent that the proposal to  $s$  is “local”, many, or perhaps even almost all, choices  $p_i$  in the program admit such savings, in which case avoiding even traversing them can be a significant opportunity as well.

Indeed, a practitioner hand-coding a transition operator such as would be induced by `single_site_metropolis_hastings_step` generally would (perhaps implicitly) carry out a program analysis to determine the addresses that may be affected by a proposal to  $s$ , and arrange to avoid spending computation on others. VentureScript ? is an example of a probabilistic programming language whose implementation does perform a (conservative) such analysis, at runtime.

One can imagine writing a Metaprob implementation to do likewise. In addition to recording the values that occurred at various addresses, it would also be necessary to construct a data structure representing information about the dependence of one address’s value on those of others. Such a structure could then (i) be analyzed to determine which addresses to examine in an inference procedure such as `single_site_metropolis_hastings_step`, and

(ii) used to incrementalize re-execution of the traced program so as to avoid unnecessary work.

Note that there is a difference here from incremental computation [?](#), since change propagation can be “absorbed” by stochastic choices. To wit, if a stochastic choice  $p_i$  can emit the same value it did last time with positive probability, it would be valid to stop propagating, and include that probability in the acceptance ratio (this is what `single_site_metropolis_hastings_step` does).

Note that it may not always be profitable to stop propagating change at the first opportunity. A candidate stopping choice  $p_i$  may be more strongly coupled to its immediate predecessors than some downstream choice  $p_j$ . In this case, propagating the change through  $p_i$  and absorbing at  $p_j$  instead may lead to faster overall convergence, due to a higher acceptance rate. Indeed,  $p_i$  being deterministic is the limit of infinitely strong coupling.

This degree of freedom suggests that a metaprogram like `single_site_metropolis_hastings_step` may be made more flexible by accepting some controls on which choices to propose to together, or otherwise determine where to propagate and where to stop. That, in turn, suggests the possibility of a programmable mini-language for subproblem selection, either operating on the existing addressing structures in Metaprob or possibly adding some additional decoration layer(s) to describe regions of program behavior more concisely.

### 6.2.1 COMPILATION

Metaprob is of course also amenable to being compiled. The last half-century’s worth of progress on compilers would certainly improve upon the definitional interpreters presented here. That said, Metaprob offers several unique challenges and opportunities for compiling.

- Challenge: Metaprob depends on being able to reflect on the behavior of Metaprob (sub-)programs. As currently constituted, this reflection depends on a standard addressing scheme derived from the source code of Metaprob programs. Any performance-enhancing source code transformations would need to either conserve the addressing (perhaps by maintaining an explicit map) or be paired with corresponding transformations of the addresses (when those are accessible at compile time).
- Opportunity: We expect a substantial class of Metaprob programs to have the same structure as the examples in this chapter, namely some inference metaprogram applied repeatedly to executions of some program under study. In this setting, interpretations of or proposals with the model program serve as the inner loop of the inference metaprogram. To the extent that this structure may be detectable (or user-specified), the looping in the metaprogram may provide a computational budget to optimize the model program very aggressively.
- Challenge: On the other hand, Metaprob differs from most other probabilistic programming systems in that it supports a Turing-complete diversity of possible metaprograms, making it that much more difficult to know in advance what any given such metaprogram may seek to do.

## Appendix A. Surface syntax

metaprob(top)	::= statements(ss).
statements(one)	::= statement(s) T_SEMI.
statements(many)	::= statements(ss) statement(s) T_SEMI.
statement(assign)	::= L_NAME(n) T_EQDEF expression(e).
statement(letvalues)	::= T_LROUND arraybody(pattern) T_RROUND T_EQDEF expression(e).
statement(tr_assign)	::= boolean_or(1) T_COLON T_EQDEF expression(r).
statement(none)	::= expression(e).
expression(top)	::= arrow(e).
arrow(tuple)	::= T_LROUND arraybody(pat) T_RROUND T_RARR expression(body).
arrow(none)	::= with_address(e).
with_address(form)	::= K_WITH_ADDRESS boolean_or(tag) T_COLON with_address(e).
with_address(none)	::= boolean_or(e).
boolean_or(or)	::= boolean_or(1) K_OR T_OR(op) boolean_and(r).
boolean_or(none)	::= boolean_and(e).
boolean_and(and)	::= boolean_and(1) K_AND T_AND(op) equality(r).
boolean_and(none)	::= equality(e).
equality(eq)	::= equality(1) K_EQ T_EQ(op) comparison(r).
equality(neq)	::= equality(1) K_NEQ T_NEQ(op) comparison(r).
equality(none)	::= comparison(e).
comparison(lt)	::= comparison(1) K_LT T_LT(op) additive(r).
comparison(le)	::= comparison(1) K_LE T_LE(op) additive(r).
comparison(ge)	::= comparison(1) K_GE T_GE(op) additive(r).
comparison(gt)	::= comparison(1) K_GT T_GT(op) additive(r).
comparison(none)	::= additive(e).
additive(add)	::= additive(1) K_ADD T_ADD(op) multiplicative(r).
additive(sub)	::= additive(1) K_SUB T_SUB(op) multiplicative(r).
additive(none)	::= multiplicative(e).
multiplicative(mul)	::= multiplicative(1) K_MUL T_MUL(op) unary(r).
multiplicative(div)	::= multiplicative(1) K_DIV T_DIV(op) unary(r).
multiplicative(none)	::= unary(e).
unary(pos)	::= T_ADD(op) unary(e).
unary(neg)	::= T_SUB(op) unary(e).
unary(none)	::= exponential(e).
exponential(pow)	::= accessing(1) K_POW T_POW(op) exponential(r).
exponential(none)	::= accessing(e).
accessing(get)	::= T_STAR testing(e).
accessing(del)	::= K_DEL testing(e).
accessing(none)	::= testing(e).

Figure 38: The formal grammar of Metaprob (continued on next page).

```

testing(one)           ::= applicative(e) K_HAS_VALUE.
testing(none)          ::= applicative(e).

applicative(app)       ::= applicative(fn) T_LROUND arraybody(args) T_RROUND.
applicative(lookup)    ::= applicative(a) T_LSQUARE expression(index)
                        T_RSQUARE.
applicative(splice_end) ::= applicative(a) T_LSQUARE expression(index) T_COLON K_END
                        T_RSQUARE.
applicative(none)      ::= primary(e).

primary(paren)         ::= T_LROUND arraybody(items) T_RROUND.
primary(brace)         ::= T_LCURLY statements(ss) T_RCURLY.
primary(if)            ::= K_IF T_LROUND expression(p) T_RROUND
                        expression(c) K_ELSE primary(a).
primary(array)         ::= T_LSQUARE arraybody(a) T_RSQUARE.
primary(none)          ::= trace(e).

trace(one)             ::= T_LTRACE expression(e) T_RTRACE.
trace(filled)          ::= T_LTRACE expression(e) T_COMMA entrylist(es) T_RTRACE.
trace(unfilled)        ::= T_LTRACE entrylist(es) T_RTRACE.
trace(none)            ::= addr(e).

entrylist(none)        ::= .
entrylist(some)        ::= entries(es).
entries(one)           ::= spl_addr(a) T_EQDEF spl_expression(e).
entries(many)          ::= entries(es) T_COMMA spl_addr(a) T_EQDEF spl_expression(e).

spl_expression(none)   ::= applicative(e).
spl_expression(splice) ::= T_POW applicative(e).

spl_addr(none)         ::= addr(e).
spl_addr(splice)       ::= T_STAR addr(e).

addr(no)               ::= name(e).
addr(yes)              ::= T_SLASH addr_entries(ks).

addr_entries(none)     ::= .
addr_entries(many)     ::= name(n) T_SLASH addr_entries(a).

name(unquote)          ::= T_LDOLLAR primary(e).
name(symbol)           ::= L_NAME(s).
name(literal)          ::= literal(l).

literal(true)          ::= T_TRUE.
literal(false)         ::= T_FALSE.
literal(integer)        ::= L_INTEGER(v).
literal(real)           ::= L_REAL(v).
literal(string)         ::= L_STRING(v).
literal(this)           ::= K_THIS.

arraybody(none)        ::= .
arraybody(some)        ::= arrayelts(es).
arraybody(somecomma)   ::= arrayelts(es) T_COMMA.
arrayelts(one)         ::= expression(e).
arrayelts(tagged)      ::= L_NAME T_EQDEF expression(e).
arrayelts(many)        ::= arrayelts(es) T_COMMA expression(e).
arrayelts(many_tagged) ::= arrayelts(es) T_COMMA L_NAME T_EQDEF expression(e).
    
```

Figure 39: The formal grammar of Metaprob (continued from previous page).

Lexical category		Grammar role
Identifiers are alphanumeric with underscores; no leading digits		L_NAME
" delimits literal strings, with conventional escape sequences		L_STRING
Digits with no radix point are literal decimal integers		L_INTEGER
Digits with a radix point are literal decimal floating-point numbers		L_REAL
// begins a line comment		ignored
/* and */ enclose multi-line comments		ignored

Lexical token	Grammar terminal
(	T_LROUND
)	T_RROUND
,	T_COMMA
:	T_COLON
;	T_SEMI
[	T_LSQUARE
]	T_RSQUARE
{	T_LCURLY
}	T_RCURLY
\$	T_LDOLLAR
=	T_EQDEF
~	T_EQDEF
->	T_RARR
~>	T_RARR
	T_OR
&&	T_AND
==	T_EQ
!=	T_NEQ
<=	T_LE
>=	T_GE
<	T_LT
>	T_GT
+	T_ADD
-	T_SUB
/	T_SLASH
÷	T_DIV
×	T_MUL
*	T_STAR
**	T_POW
:	T_COLON
{{	T_LTRACE
}}	T_RTRACE

Keyword	Grammar terminal
add	K_ADD
and	K_AND
div	K_DIV
del	K_DEL
else	K_ELSE
end	K_END
eq	K_EQ
ge	K_GE
gt	K_GT
has_value	K_HAS_VALUE
if	K_IF
le	K_LE
lt	K_LT
mul	K_MUL
neq	K_NEQ
or	K_OR
pow	K_POW
sub	K_SUB
true	T_TRUE
false	T_FALSE
&this	K_THIS
with_address	K_WITH_ADDRESS

Figure 40: Lexical syntax of Metaprob. Whitespace and comments separate tokens, but serve no other function in the grammar



Syntax	Example	Effect
Pattern-matching bindings	<code>(v, score) &lt;- propose(...);</code>	<code>propose</code> is expected to return an ordered collection with two elements; they are extracted and bound to the variables <code>v</code> and <code>score</code> .
Empty bindings	<code>(v, _) &lt;- ...;</code>	A single underscore in binding position means “do not bind a value”. This is mostly useful for extracting some components in a pattern match without binding names to others.
Grouping parens	<code>(3 + 4) × 7</code>	Paired parentheses without intervening commas serve as precedence grouping (except when they occur as part of some other syntax).
Explicit tuples	<code>(3, 4, 5)</code>	A paren-enclosed comma-separated list is syntactic sugar for tuple construction.
Explicit tuples	<code>[3, 4, 5]</code>	Square brackets serve the same purpose, with the difference that they are not used for grouping, so <code>[1]</code> is a one-element tuple.
Unquote	<code>/datum/\$i/</code>	The dollar sign (\$) is used for interpolation into address literals. Specifically, the expression immediately following the \$ (which is often a variable reference but may be a paren-grouped compound expression) is evaluated and its value inserted as a key in that spot in the address.

Figure 41: Semantics of a few additional syntax elements not explicitly discussed in the main text.