Metaprob: a minimal probabilistic programming language with first-class inference metaprogramming

Anonymous Author(s)

Abstract

Probabilistic modeling and inference are central to modern data science, robotics, and artificial intelligence. Unfortunately, most probabilistic programming langauges do not allow developers to customize inference to exploit the structure of the problem at hand. This leads to suboptimal performance. This paper introduces Metaprob, a minimal probabilistic programming language with novel language constructs and abstractions for customizing inference. It uses multiple examples to illustrate the expressive power of these abstractions, highlighting novel inference metaprogramming capabilities. It reports performance measurements from a prototype implementation in Clojure, establishing that these abstractions enable developers to improve performance by customizing inference. This paper also presents a precise formal characterization of the key concepts in Metaprob along with propositions that establish key properties. Finally, it illustrates the application of Metaprob to real-world data science problems such as modeling and searching multivariate data from the U.S. Census.

1 Introduction

Probabilistic modeling and inference are central to modern statistics, computer vision, machine learning, robotics, and artificial intelligence [7, 9, 19, 24, 26, 28]. Because of the intellectual and social importance of these applications, there is a widespread need for languages that formalize the core technical concepts in modeling and inference, and provide developers with automation for time-consuming and/or error prone aspects of application development. To meet this need, probabilistic programming languages have been emerging [3, 11–14, 18, 20, 23, 29, 31], developed by researchers in the statistics, cognitive science, machine learning, and artificial intelligence communities. These languages provide language constructs for specifying probabilistic models and for performing inference given data.

Unfortunately, early probabilistic programming languages such as Church [11] and BLOG [23] provided little to no ability to customize inference to exploit the structure of the problem at hand. Users specified models using "generative" code that makes a stochastic choice for each latent variable in the associated model. Users then relied on built-in, black-box mechanisms for inference given data, that could not be customized without extending the language implementation. As a result, probabilistic programming implementations of inference in these languages were often suboptimal, in terms of accuracy and efficiency, as compared to what could be achieved by developing algorithms without probabilistic programming's abstractions.

Conference'19, July 2019, Washington, DC, USA
2018. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM. . . $15.00

To address these limitations, the concept of inference metaprogramming [22] has been introduced, to give developers the flexibility to customize inference and thereby improve performance and accuracy. Example languages that support inference metaprogramming in some form include Venture[20] and Turing[8]. The associated language constructs are for "metaprogramming" in that they operate on data structures representing the runtime behavior of generative code representing a probabilistic model. These existing inference metaprogramming languages have fundamental limitations. For example, both Venture and Turing provide primitive inference tactics that cannot be implemented using the built-in inference metaprogramming constructs. Control flow for inference metaprograms is provided via restricted DSLs that lack the full flexibility of their host langauges. It is also infeasible to add new primitives — let alone new domain-specific languages — that rely on arbitrary custom inference algorithms.

Key challenges. The limitations of existing language-level support for inference metaprogramming raise the following technical challenges: (i) Is it possible to develop simple abstractions that are sufficient for custom inference metaprogramming? (ii) If so, can these abstractions support both general-purpose inference algorithms, new primitives with custom inference, and new DSLs with custom inference engines? Can they (iii) be implemented with a modest amount of code, yet (iv) deliver performance gains in practice?

This work. This paper introduces programming abstractions for first-class inference metaprogramming. Specific abstractions include:

1. *Generative procedures* that define probabilistic models over the space of possible executions of the procedure. Generative procedures are not unique to Metaprob, though this paper gives a precise formal characterization of this concept.
2. *Trace* data structures that represent complete and partial executions of generative procedures.
3. *Partial traces that represent observations and interventions.* Interventions override the runtime behavior of

a generative procedure, at arbitrary points in the procedure's execution. Observations represent data that needs to be accounted for via an inference algorithm.

4. *Inference procedures* that encapsulate probabilistic models along with an associated – and potentially approximate – implementation of inference. Inference procedures accept partial traces representing the observations and interventions to take into account during inference. Inference procedures return a value, along with an execution trace that represents how the value was produced, and a score indicating how accurately the observations were accounted for.
5. *A canonical metacircular inference interpreter* that can promote any probabilistic program into a valid inference procedure. See sections 3-5 for a formal model and the supplemental material for a full implementation.
6. *A language construct for overriding the tracing behavior of the canonical interpreter*, enabling Metaprob programs to produce more compact, readable traces that are easier to use for inference metaprogramming and that hide information about their implementations.

1.1 Contributions

This paper makes the following contributions:
New abstractions and language constructs. These include the concept of an *inference procedure*; the `inf` construct for creating inference procedures; the canonical metacircular inference interpreter; and a construct for overriding the default tracing behavior of the canonical interpreter.
Performance measurements. This paper reports performance measurements that assess the runtime overhead of Metaprob's implementation and show that customizing inference can improve performance and accuracy.
Formalism and propositions. This paper presents a formal semantics of Metaprob that precisely characterizes key concepts. It presents propositions that establish key properties, such as the correspondence between observations as specified via the canonical metacircular inference interpreter with the semantics of conditioning in probability theory.

These contributions are illustrated using several examples, including Metaprob programs that create new primitives with custom inference algorithms; new general-purpose inference algorithms that apply to broad classes of models; new custom inference algorithms for complex user-specified programs; and a custom domain-specific probabilistic language with a custom inference engine, applied to model multivariate data from the US Census. This paper also shows that inference algorithms in Metaprob can be used to perform probabilistic reasoning about the behavior of other inference algorithms.

2 Examples

We begin with two examples to introduce the key ideas behind Metaprob's design. The first uses Metaprob to perform a curve-fitting task that would work similarly in most existing probabilistic programming languages. The second showcases Metaprob's custom inference abstractions, demonstrating both the ability to define new multivariate random primitives in user-space, and to customize inference for particular models to achieve significant performance gains.

2.1 Bayesian model selection and parameter estimation

Our first example (Figures 1 and 2) introduces the key ideas behind probabilistic modeling and inference in Metaprob. The example uses probabilistic programming to infer the degrees and coefficients of polynomials that might explain a set of observed points (x, y) on the plane.

Code for defining the model. The first step in probabilistic programming is to define a proabilistic model using executable code. Figure 1a shows generative code for doing so. It uses the keyword `gen` to implement a *generative procedure* `curve-model`. Conceptually, this procedure simulates fake data: given a list of x coordinates, `curve-model` applies a randomly sampled polynomial to each, then adds a small amount of Gaussian noise. Note that developers writing generative code in Metaprob have full access to standard functional programming constructs in defining models: `generate-curve`, for example, represents a probability distribution over polynomials (themselves represented as Metaprob generative functions).

Code for generating random data and a random execution constrained to match that data. This model can be run via an ordinary interpreter in which `gen` is treated as a synonym for `lambda`. Figure 1b shows code for invoking `curve-model` on a list of x coordinates, producing a list of simulated y coordinates. This model can also be run using the canonical metacircular inference interpreter `infer-apply`. This produces not just a return value but also an *execution trace* (Figure 1c) and a *score*. Conceptually, the code representing the probabilistic model induces a probability distribution on execution traces. The space of possible execution traces constitutes the domain of the probabilistic model. The execution trace records all random choices made during the model's execution—in this case, we can see that the degree was chosen to be 2, and the coefficients to be 0.32 and 0.56, resulting in the curve $f(x) = 0.56x + 0.32$. We have provided to `infer-apply` an *observation trace* (highlighted in green in Figure 1c), constraining the model to produce the same five y values produced by our earlier, "generative-mode" invocation. The sampled curve f is unlikely to have produced these particular values without our meddling: this mismatch is quantified by the *score*, here −10.55, which is a log likelihood

```
(define generate-curve
  (gen []
    (define degree (uniform-sample [1 2 3 4]))
    (define coeffs (replicate degree (gen [] (gaussian 0 1))))
    (gen [x]
      (sum
        (map
          (gen [n] (* (nth coeffs n) (pow x n))
          (range degree))))))

(define add-noise-to-curve
  (gen [curve]
    (gen [x] (gaussian (curve x) 0.1))))

(define curve-model
  (gen [xs]
    (map (add-noise-to-curve (generate-curve)) xs)))
```

(a) A probabilistic model of polynomial curves.

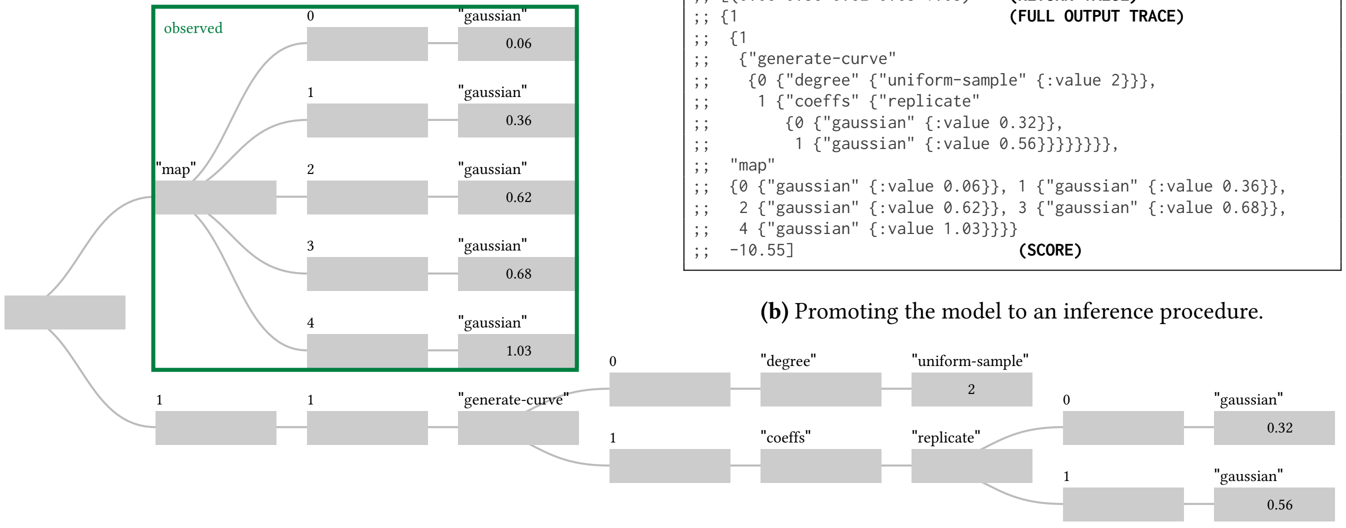
```
;; Running the model as a generative function
(define xs '(-0.5 -0.3 0.1 0.2 0.5))
(define ys (curve-model xs))
;; => (0.06 0.36 0.62 0.68 1.03)

;; Constructing a partial execution trace to encode
;; our observations.
(define observations
  {"map"
   {0 {"gaussian" {:value 0.06}}, 1 {"gaussian" {:value 0.36}},
    2 {"gaussian" {:value 0.62}}, 3 {"gaussian" {:value 0.68}},
    4 {"gaussian" {:value 1.03}}}})

;; Promoting the model to an inference procedure and invoking it,
;; to get value, trace, & score constrained by an observation trace.
(infer-apply
  :procedure curve-model,
  :inputs [xs],
  :observation-trace observations)

;; =>
;; [(0.06 0.36 0.62 0.68 1.03)    (RETURN VALUE)
;; {1                             (FULL OUTPUT TRACE)
;;  {1
;;   {"generate-curve"
;;    {0 {"degree" {"uniform-sample" {:value 2}}},
;;     1 {"coeffs" {"replicate"
;;        {0 {"gaussian" {:value 0.32}},
;;         1 {"gaussian" {:value 0.56}}}}}}}}},
;;  "map"
;;  {0 {"gaussian" {:value 0.06}}, 1 {"gaussian" {:value 0.36}},
;;   2 {"gaussian" {:value 0.62}}, 3 {"gaussian" {:value 0.68}},
;;   4 {"gaussian" {:value 1.03}}}}
;;  -10.55]                        (SCORE)
```

(b) Promoting the model to an inference procedure.



(c) Graphical representation of the *trace* of **`curve-model`**, which records the values of all random choices made during execution.

Figure 1. Tutorial example. We define a model (a) that randomly samples constant, linear, quadratic, or cubic curves. The model can be invoked as a generative procedure or promoted to an inference procedure (b). As a generative procedure, the model samples a random curve f, and returns noisy observations of the values $f(x)$ at each input in **`xs`**. As an inference procedure, it is possible to constrain the model's execution with observation or intervention traces, and to retrieve not just a return value but also an execution trace and a score. The execution trace (c) records every random choice made during the model's execution, and is guaranteed to be consistent with observations and interventions.

of the observation trace given all the values previously sampled during the model's execution. Note that the subtrace highlighted in green corresponds to the trace of the observed data, that in this case was generated by executing the generative code representing the probabilistic model using an ordinary interpreter.

Inference code. A key idea in probabilistic modeling is that models can not just be run, but also queried, to answer questions of the form "How must the model probably have executed, in order to produce a certain result?" Most languages support a number of general-purpose inference algorithms for answering these queries. In Metaprob, such algorithms are implemented in user-space, as inference metaprograms. Developers can use the canonical metacircular inference engine **`infer-apply`** as a building block to implement standard general-purpose inference algorithms as higher-order functions. Figure 2a shows the implementation of the library function **`importance-resample`**, which first calls **`infer-apply`** many times to generate candidate traces ("particles") consistent with observations. It then samples one at random, with probabilities proportional to each trace's likelihood (the

```
(define importance-resample
  (gen [& {:keys [procedure inputs intervention-trace
                  observation-trace n-particles]}]
    (define weighted-traces
      (replicate n-particles
        (gen []
          (define [retval trace importance-weight]
            (infer-apply
              :procedure procedure
              :inputs inputs
              :intervention-trace intervention-trace
              :observation-trace observation-trace))
          [trace importance-weight])))
    (nth (map first weighted-traces)
      (log-categorical (map second weighted-traces)))))
```

(a) Sampling/importance resampling as a short library function.

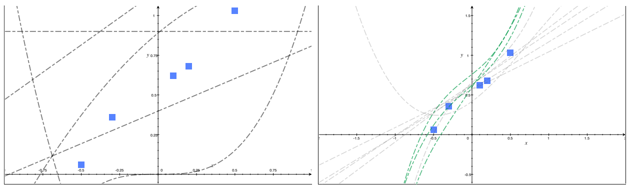
```
;; Sample an execution trace from the approximate posterior,
;; given xs and ys.
(importance-resample
  :procedure curve-model,
  :inputs [xs],
  :observation-trace observations,
  :n-particles 100)
```

(b) Inferring the curve given data.

```
;; Sample an execution trace from the approximate posterior,
;; given xs and ys, but intervene to set degree to 4.
(importance-resample
  :procedure curve-model
  :inputs [xs]
  :intervention-trace
    (trace-with-values-at
      '(1 1 "generate-curve" 0 "degree" "uniform-sample") 4)
  :observation-trace observations
  :n-particles 100)
```

(c) Using an intervention trace to fit a cubic to the data.

Figure 2. On top of **`infer-apply`**, we are able to imlpement algorithms like sampling/importance resampling as short, higher-order functions (a). Invoking this algorithm with an observation trace (b) gives approximate posterior samples of curves, given the data. We can also intervene on the model to ensure that the curves fit are cubic (c).



(a) Samples of curves from the prior: most do not fit the data. **(b)** Curves sampled with observations (gray) and interventions (green)

Figure 3. Importance sampling (e) produces curves that fit the data better than curves from the prior (d).

exponential of the score). This technique, called *sampling/importance resampling*, is known to approximate the true posterior distribution arbitrarily well with a high enough number of particles [1, 6].

Curves sampled directly from the model, without any inference to incorporate the data, are unlikely to fit the data (Figure 2d). Using **`importance-resample`** to condition the model on the observed data yields curves that fit better (Figure 2e, gray curves). Metaprob also supports Pearl-style interventions on program executions, in which the behavior of the generative code is overridden at arbitrary points. Figure 2c shows an example of this, in which the model is constrained to always produce cubic polynomials. The green curves in 2e show the results of this intervention, in which every sampled curve is evidently cubic.

Table 1. Comparison of four inference algorithms for Circus Brothers. Shows time per sample required before samples reliably cross some KL divergence threshold.

	KL<2	KL<0.5	KL<0.1
Importance sampling	42ms	> 270ms	> 270ms
Single-site MH	41ms	250ms	> 270ms
Gaussian drift MH	22ms	50ms	142ms
Exact inference	1ms	1ms	1ms

2.2 Primitives and composite models with optimized inference metaprograms.

This tutorial example shows that custom inference metaprograms can improve performance over generic inference algorithms. It also illustrates (i) how to add new primitive probability distributions to the language, equipped with their own optimized inference metaprograms; (ii) how to use those constructs to write composite models; and (iii) how to optimize inference for a composite model by writing a custom inference metaprogram that does not rely on the general-purpose inference library. Note that (i) and (iii) cannot be done via ordinary user-space code in other probabilistic programming languages, including Venture.

Code for defining the model. Figure 4a shows the generative code for defining the model. Conceptually, the model represents a scenario in which there are two brothers in a circus act masquerading as one very tall man, whose height is returned by the procedure. One natural inference task in this model is to observe the height of the two brothers together, and to infer their individual heights.

Code for defining a new inference procedure. Figure 4b shows the definition of an inference procedure that implements a bivariate Gaussian distribution. The model definition is straightforward: itshows two variables, **`x1`** and **`x2`**, where **`x1`** is drawn from a Gaussian distribution with the given input parameters, and **`x2`** is drawn from a conditional Gaussian distribution (implementation not shown) such that the joint distribution matches the input parameters. Note that **`with-explicit-tracer`** is used to ensure that the return traces do not leak information about the implementation

```
(define circus-brothers (gen [] (define [h1 h2]
  (bivariate-gaussian :means [70 70] :covariance [[9 5] [5 9]]))
  (gaussian (+ h1 h2) 3))) (define observation {1 {"gaussian" {:value
        155}}})
```

(a) Generative model and observation trace

```
(define bivariate-gaussian
  (inf
    ;; MODEL:
    (gen [params]
      (with-explicit-tracer t
        (define [[mu1 _] [[s11 _] [_ _]]] params)
        (define x1 (t "x1" gaussian mu1 (sqrt s11)))
        (define x2 (t "x2" conditional-gaussian x1 params))))
    ;; CUSTOM INFERENCE PROCEDURE:
    (gen [params traces]
      (define [[mu1 mu2] [[s11 s12] [s21 s22]]] params)
      (define flipped-params [[mu2 mu1] [[s22 s12] [s21 s11]]])
      (define x1
        (cond
          (contains? traces "x1") (specified-value traces "x1")
          (observed? traces "x2")
          (conditional-gaussian
            (observed-value traces "x2")
            flipped-params)
          true (gaussian mu1 (sqrt s11))))
      (define x2
        (if (contains? traces "x2")
          (specified-value traces "x2")
          (conditional-gaussian x1 params)))
      (define score
        (bivariate-gaussian-log-density ;; accounts for nil args
          params
          (observed-value traces "x1")
          (observed-value traces "x2")))
      [[x1 x2] (trace-with-values-at "x1" x1, "x2" x2) score])))
```

(b) User-space implementation of **bivariate-gaussian**

```
(define circus-exact
  (inf
    circus-brothers ;; MODEL
    (gen [_ traces] ;; CUSTOM INFERENCE PROCEDURE
      (define [h-mean h-var h12-covar]
        (if (observed? traces __total_addr)
          [(+ 17.02 (* 0.378 (observed-value traces __total_addr)))
           3.7 -0.2966]
          [70 9 5]))
      (define bivariate-gaussian-params
        [:means [h-mean h-mean]
         :covariance [[h-var h12-covar] [h12-covar h-var]]])
      (define h12-traces (subtraces traces __h12_addr))
      (define [[h1 h2] h12-output-trace _]
        (infer-apply
          :procedure bivariate-gaussian
          :inputs bivariate-gaussian-params
          :observation-trace (get h12-traces :observation-trace)
          :intervention-trace (get h12-traces :intervention-trace)))
      (define total-height
        (if (contains? traces __total_addr)
          (specified-value traces __total_addr)
          (gaussian (+ h1 h2) 3)))
      [[h1 h2]
       (trace-set-subtrace
         {1 {"gaussian" {:value total-height}}}
         __h12_addr h12-output-trace)
         (circus-exact-marginal traces)])))
```

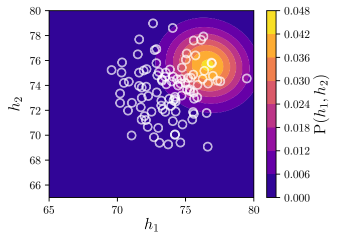
(c) Version of **circus-brothers** supporting exact inference

```
(define circus-custom-step
  (gen [_ _ prev-trace _]
    (define [h1 h2] (circus-extract-heights prev-trace))
    (define [h1' h2']
      (bivariate-gaussian :means [h1 h2] :covariance [[1 0] [0 1]]))
    (define [_ _ old-score]
      (infer-apply
        :procedure circus-brothers,
        :observation-trace prev-trace))
    (define [_ proposed-trace new-score]
      (infer-apply :procedure circus-brothers,
        :observation-trace (trace-set-values-at prev-trace
          __h1_addr h1', __h2_addr h2')))
    (if (flip (exp (- new-score old-score)))
      proposed-trace prev-trace)))
```

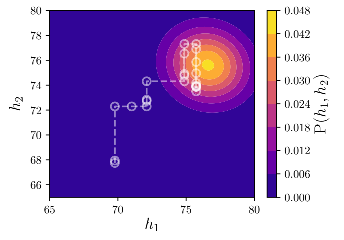
(d) Gaussian drift MH proposal

```
(importance-resample
  :procedure circus-brothers
  :observation-trace observation
  :n-particles 10)
```

```
(mh :procedure circus-brothers
  :transition single-site-step
  :observation-trace observation
  :iterations 100)
```



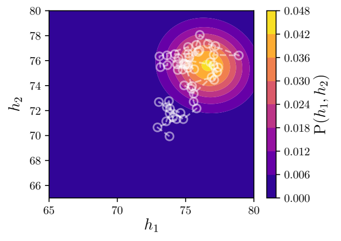
(e) Importance sampling



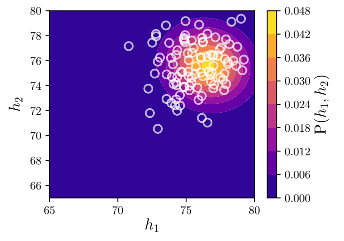
(f) Single-site MH

```
(mh :procedure circus-brothers
  :transition circus-custom-step
  :observation-trace observation
  :iterations 100)
```

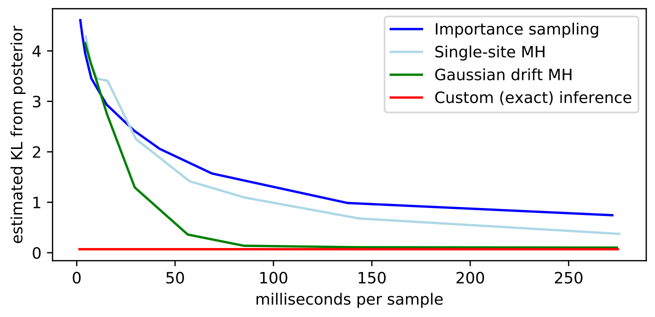
```
(infer-apply
  :procedure circus-exact
  :observation-trace observation)
```



(g) Gaussian-drift MH



(h) Exact inference



(i) Time vs. accuracy plot

Figure 4. One model (a), four inference strategies (e, f, g, h). We imagine two brothers in a circus act masquerading as one very tall man, and wish to infer their individual heights. The bivariate Gaussian (b) is written to support exact posterior inference, enabling the implementation of an exact inference algorithm for **circus-brothers** (c). Approximate inference can also be customized, for instance with a custom MH proposal (d).

of the **model**, and instead return traces with a simple, intuitive interface. The custom inference implementation for the bivariate Gaussian is also shown. This procedure has to correctly generate samples and score them, for any combination of input traces in which **x1** and/or **x2** could be observed, or intervened on.

Code for exact inference in circus brothers. Figure 4c shows the definition of an exact inference procedure for the model from Figure 4a. This procedure works as follows. First, it checks if the total height was specified in the constraint trace, and if so, it calculates the parameters of a bivariate Gaussian distribution representing the exact posterior over **h1** and **h2**. It then delegates to the inference procedure for the bivariate Gaussian defined in Figure 4b. Finally, returns the heights, a trace, and score.

Code for a custom inference metaprogram based on Metropolis-Hastings inference with a custom proposal. Figure 4d shows how to implement a Metropolis-Hastings proposal that applies a Gaussian perturbation to both **h1** and **h2**. This custom inference algorithm can account for the posterior interactions between **h1** and **h2**, unlike the generic algorithms.

Performance results. Performance results for inference are shown in Figure 4e (importance sampling with resampling from Figure 2a), Figure 4f (generic single-site Metropolis-Hastings from Figure 9), Figure 4g (custom Gaussian drift Metropolis-Hastings inference from Figure 4d), and Figure 4h (exact inference from Figure 4c).

Exact inference, representing the highest degree of customization, is the most efficient. The Gaussian drift Metropolis-Hastings algorithm is the next most efficient. The single-site Metropolis-Hastings inference algorithm is the next most efficient, and importance sampling — the only inference algorithm provided directly by the canonical metacircular inference interpreter — is the least efficient algorithm. Table 1 shows another view on this same performance data, illustrating the fundamental tradeoff between generality of inference metaprogram and performance.

3 Case studies

We now present five case studies designed to further illustrate Metaprob features.

Table 2. Overhead of trace-based inference: times for one sample from a mixture model.

	Sampling	Tracing	Scoring
Metaprob (default)	0.03ms	1.75ms	1.75ms
Metaprob (custom)	0.01ms	0.19ms	0.25ms
Anglican	0.02ms	N/A	0.02ms

Table 3. Comparison of importance sampling in Anglican and Metaprob, and exact inference in Metaprob. Shows time per sample required before samples reliably cross some KL divergence threshold in the multi-mixture model.

	KL < 0.1	KL < 0.01
Anglican (importance sampling)	0.42ms	1.63ms
Metaprob (importance sampling)	27.03ms	78.01ms
Metaprob (exact inference)	0.48ms	0.48ms

3.1 Implementing a domain-specific modeling language with applications to data science

This case study shows that Metaprob can be used to implement domain-specific probabilistic languages suitable for modeling real-world data, equipped with custom inference algorithms that are more efficient than general-purpose implementations of widely-used Monte Carlo algorithms.

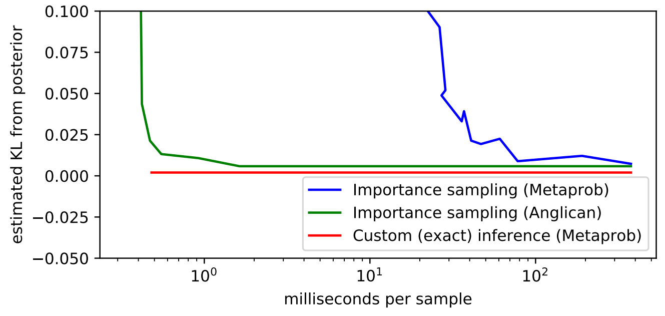
The code shown in Figure 5a defines a domain-specific language for factorial mixture models. In this language, which applies to multivariate tabular data, the variables (columns) are partitioned into subsets (**varset**s in the code), each of which is modeled independently of the others. Each **varset** is modeled by its own mixture model that posits a latent clustering to explain variation in the observed values. This model class corresponds closely to the output of CrossCat [21], a non-parametric Bayesian technique for modeling multivariate data, and to the probabilistic models that are automatically synthesized from data via the BayesDB platform. Importantly, the function **multi-mixture**, which serves as an entry-point into the DSL, automatically *synthesizes* a fast, exact custom inference procedure for any model written in the DSL. (See supplemental material for the full source code.)

This model class is expressive, and has been applied to multiple real-world data science problems. For example, Figure 5b shows code for a model of a subset of US Census data, including variables related to public health. Figure 5d qualitatively illustrates the accuracy of the probabilistic programs by showing a high degree of overlap between the real data and synthetic data produced by repeatedly re-executing the program from Figure 5b. Figure 5d shows comparisons of the performance and accuracy of different inference metaprograms for performing inference, to generate samples of one variable given observed values of another.

The fast synthesized inference algorithm can be used to solve interesting problems like *similarity search*, following Saad et al. [27]. In our model, for example, searching for counties similar to a hypothetical county with high exercise opportunities and health insurance rates yields urban counties, whereas querying for moderate obesity and lower exercise opportunities yields more rural counties (Figure 5f).
Peformance measurements. Performance measurements on this case study show that custom inference metaprograms achieves greater performance than generic inference. Similar measurements show that Metaprob has significant some

```
(define varset
  (gen [dists [cluster-probs cluster-params]]
    [dists cluster-probs cluster-params]))
(define clusters
  (gen [& args] [(take-nth 2 args) (take-nth 2 (rest args))]))
(define multi-mixture
  (gen [& varsets]
    (inf
      ;; MODEL:
      (gen []
        (apply concat
          (map
            (gen [[dists probs params]]
              (define param-set (nth params (categorical probs)))
              (map (gen [v] (apply (get dists v) (get param-set v)))
                (keys varset)))
            varsets)))
      ;; CUSTOM INFERENCE PROCEDURE (see supplement):
      (multi-mixture-inference-procedure-for varsets))))
```

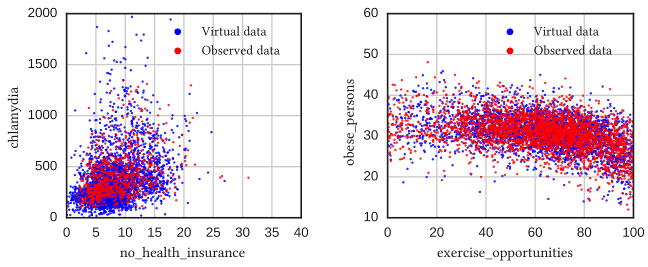
(a) Defining a DSL for mixture models.



(b) Comparison of time/accuracy tradeoffs in importance sampling vs the synthesized exact inference procedure. The task is to sample from the posterior distribution on chlamydia rates in a county where 20% do not have health insurance.

```
(define generate-county-data
  (multi-mixture
    (varset {"no_health_insurance" gaussian, "chlamydia" gaussian}
      (clusters
        0.46 {"no_health_insurance" [6.2 2.5],
              "chlamydia" [201.4 70.1]}
        0.26 {"no_health_insurance" [7.6 2.4],
              "chlamydia" [366.3 111.0]}
        0.14 {"no_health_insurance" [13.0 3.8],
              "chlamydia" [589.7 222.9]}
        0.07 {"no_health_insurance" [14.0 2.2],
              "chlamydia" [363.5 101.2]}
        0.03 {"no_health_insurance" [9.7 2.4],
              "chlamydia" [1024.7 256.7]}
        0.02 {"no_health_insurance" [5.1 1.3],
              "chlamydia" [644.7 125.1]}
        0.02 {"no_health_insurance" [10.6 2.3],
              "chlamydia" [1369.9 629.4]}))
    (varset {"obese_persons" gaussian,
        "exercise_opportunities" gaussian,
        "opioid_deaths" gaussian}
      (clusters
        0.51 {"obese_persons" [31.8 2.8],
              "exercise_opportunities" [58.1 16.2],
              "opioid-deaths" [1184.2 232.6]}
        ;; and six more clusters elided
        ))))
```

(c) A multi-mixture model learned from county-level census data.



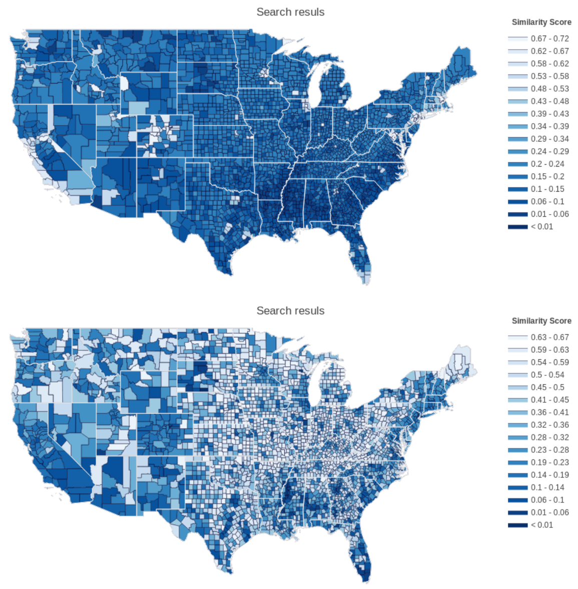
(d) Comparing relationships between variables in the original census data and in our model's simulated counties. Models in this DSL are capable of capturing the distribution of the data.

```
(define similarity-scores
  (gen [mixture-model query-item dataset]
    ;; Implementation relies on multi-mixture's
    ;; custom inference procedure to compute
    ;; similarity scores based on the probability
    ;; of identical cluster assignments in the
    ;; posterior distribution over traces.
    ...))

(similarity-scores
  {'exercise_opportunities': 90,
   'no_health_insurance': 10}
  all-counties)

(similarity-scores
  {'exercise_opportunities': 40,
  'obese_persons': 30}
  all-counties)
```

(e) Mixture models with fast custom inference enable many other inference tasks. It is possible, for instance, to implement a function **`similarity-scores`**, which scores each item in a real-world dataset by similarity to a query item. A query for counties similar to a hypothetical county with high exercise opportunities and insurance rates yields mostly urban results, whereas one with moderate obesity and lower exercise opportunities yields more rural results (5f).



(f) Results from the two queries shown in Figure 5e.

Figure 5. Metaprob supports the implementation of small modeling DSLs via inference procedure *constructors*.

runtime overhead as compared to Anglican, another probabilistic programming language that is embedded in Clojure. See Table 2 and Table 3 for numerical data. Note that Anglican's efficieny comes at the loss of flexibility: Anglican does not support custom inference metaprogramming. A custom exact inference engine written in Metaprob thus outperforms Anglican on this problem.

3.2 Customizing the trace of a generative procedure.

In formulating inference metaprograms and queries, it is often necessary to think explicitly about the trace structure of a model. Metaprob provides a modeling construct, **`with-explicit-tracer`**, which enables users to shape the traces of their models so that they are easier to work with.

As an example of this, consider the higher-order function **`map`**, which applies a function to each element of a list. If we write **`map`** naively (Figure 6a), a trace of an execution of the function (e.g., **`(map flip '(0.1 0.5 0.9))`**) will be long and unweildy, reflecting the recursive structure of the computation (Figure 6c).

Instead, we can use **`with-explicit-tracer`** to write a better version. An expression of the form **`(with-explicit-tracer t body)`** binds the current *tracing context* to the variable **`t`** for a scope local to **`body`**. The tracing context behaves as a procedure: the expression **`(t addr f x1 x2 . . . xn)`** applies **`f`** to its inputs **`x1, . . . , xn`**, but *traces* **`f`** at the provided address, relative to the place **`t`** was bound. Within the body of a **`with-explicit-tracer`** expression, nothing is traced unless explicitly done so by invoking **`t`**.

The **`map`** implementation in Figure 6b uses this construct to trace the invocations of its argument **`f`** at simple integer addresses (as shown in Figure 6c).

3.3 Causal reasoning with interventions

As an example of the difference between interventions and observations, consider a simplified model of athletic success (Figure 7). Under the model, athletes have a randomly sampled "skill" score between 0 and 1. Athletes with higher skill levels have an easier time getting sponsorship contracts, and athletes with sponsorship contracts have an easier time getting rich.

Figure 8 shows the invocation and results of inference in the model. If we observe that an athlete is wealthy, this increases the chances he has skill—and indeed, importance sampling reveals that we might expect a wealthy athlete to have a skill level of around 0.67, much higher than the average (0.5). But intervention changes that story. If our company decides to give an athlete a sponsorship contract for reasons *unrelated* to his skill (that is, if they *intervene* in the model), we can no longer draw any conclusions about his skill simply by observing he is wealthy. This is reflected in the inference results in Figure 8: with intervention, observing that an athlete is wealthy leaves the expected skill level unchanged, at about 0.5.

```
(define map
  (gen [f l]
    (if (empty? l)
        '()
        (cons (f (first l)) (map f (rest l))))))
```

(a) A naive implementation of **`map`**.

```
(define map
  (gen [f l]
    (with-explicit-tracer t
      (define helper
        (gen [i l]
          (if (empty? l) '()
              (cons (t i f (first l))
                    (helper (+ i 1) (rest l))))))
      (helper 0 l))))
```

(b) An implementation of **`map`** with custom tracing.

```
(infer-apply
  :procedure map
  :inputs [flip '(0.1 0.5 0.9)])

;; =>                         (naive implementation)
;; {"else"
;;  {1
;;   {"f" {:value false}}
;;   2 {"map"
;;      {"else"
;;       {1 {"f" {:value false}}
;;        2 {"map" {"else" {1 {"f" {:value true}}}}}}}}}
;;
;; =>                         (better implementation)
;; {0 {:value false}
;;  1 {:value false}
;;  2 {:value true}}
```

(c) Traces of naive and better implementations of **`map`**.

Figure 6. The **`with-explicit-tracer`** construct allows users to customize the traces of their models so that it is easier to write inference metaprograms that manipulate those traces.

```
(define athlete-model
  (gen []
    (define skill (uniform 0 1))
    (define has-sponsorship-contract?
        (flip (pow skill 8)))
    (define is-wealthy?
      (flip
        (if has-sponsorship-contract?
          0.8
          0.1)))
    [skill has-sponsorship-contract? is-wealthy?]))
```

Figure 7. A toy model of success in athletic careers.

3.4 General-purpose inference metaprograms

This case study shows how to use user-space inference metaprogramming to implement a general-purpose inference metaprogram: single-site Metropolis-Hastings inference. That such algorithms can be implemented as short higher-order functions (importance sampling from Figure 2 is another example) is significant, not only because they are widely used, but also because they were the basis of the infernence engines built in to the original implementation of the Church probabilistic programming langauge. Metaprob's support for first-class

```
;; Define relevant addresses from the model
(define [__skill_addr __contract_addr __wealthy_addr] ...)

;; Create observation and intervention traces
(define is-wealthy-trace
  (trace-with-values-at __wealthy_addr true))
(define has-contract-trace
  (trace-with-values-at __contract_addr true))

;; Sample execution traces likely resulting
;; in wealth=true.
(define observed-wealthy-samples
  (replicate 100
    (gen []
      (importance-resample
        :procedure athlete-model
        :observation-trace is-wealthy-trace
        :n-particles 100))))

;; The mean skill of these samples is 0.67:
;; higher than your average athlete.
(mean (map (gen [t] (trace-value t __skill_addr))
           observed-wealthy-samples))
;; => 0.67

;; Sample execution traces likely resulting in
;; wealth=true, assuming we've intervened to give
;; the athlete a contract.
(define observed-wealthy-with-intervention
  (replicate 100
    (gen []
      (importance-resample
        :procedure athlete-model
        :observation-trace  is-wealthy-trace
        :intervention-trace has-contract-trace
        :n-particles 100))))

;; The mean skill of these samples is 0.51:
;; about average. The causal connection between
;; wealth and skill has been severed, so we can
;; no longer use wealth to reason about skill.
(mean (map (gen [t] (trace-value t __skill_addr))
           observed-wealthy-samples-with-intervention))
;; => 0.51
```

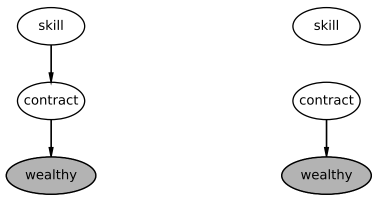


Figure 8. Intervening on the toy model severs causal connections.

inference metaprogramming makes it feasible to implement both these strategies as short user-space programs; this provides evidence for the expressiveness of Metaprob.

Single-site MH is the transition operator for a Markov chain Monte Carlo algorithm that infers probable traces of an input procedure, given a partial trace specifying a set of observations. It can be used as a building block for MCMC or SMC algorithms. (The supplement shows the implementation of a **mh**, procedure which repeatedly applies a transition operator to perform inference.)

The **single-site-mh-step** procedure works as follows. First, it chooses an address **resim-addr** in the input trace to modify, and makes a new trace, labeled **resimulate-constraints**,

```
(define single-site-mh-step
  (gen [model inputs t constraint-addrs]
    ;; choose an address to modify, uniformly at random
    (define addrs (addresses-of t))
    (define candidates (set-difference addrs constraint-addrs))
    (define initial-num-choices (count candidates))
    (define resim-addr (uniform-sample candidates))

    ;; generate a proposal trace
    (define initial-value (trace-value t resim-addr))
    (define resim-constraints (trace-clear-value t resim-addr))
    (define [_ proposed forward-score]
      (infer-apply
        :procedure model,
        :inputs inputs,
        :observation-trace resim-constraints))
    (define resimulated-value
      (trace-value proposed resim-addr))

    ;; the proposal is to move from t to proposed.
    ;; now calculate the Metropolis-Hastings acceptance ratio
    (define new-addrs (addresses-of proposed))
    (define new-candidates
      (set-difference new-addrs constraint-addrs))
    (define new-num-choices (count new-candidates))

    ;; make a trace that can be used to restore the original trace
    (define restoring-trace
      (trace-partial-copy t
        (cons resim-addr (set-difference addrs new-addrs))))

    ;; remove the new value
    (define reverse-proposal-constraints
      (trace-clear-value proposed resim-addr))

    (define [_ _ reverse-score]
      (infer-apply
        :procedure model,
        :inputs   inputs,
        :intervention-trace restoring-trace,
        :observation-trace reverse-proposal-constraints))

    (define log-acceptance-probability
      (- (+ forward-score (log new-num-choices))
         (+ reverse-score (log initial-num-choices))))

    (if (flip (exp log-acceptance-probability))
      proposed
      t)))
```

Figure 9. Metaprogram for implementing a single step of a single-site Metropolis-Hastings algorithm for approximate inference. This metaprogram can be used to supply the **transition** kernel for the **markov-chain-inference** procedure specified earlier.

containing a copy of the input minus the value at that address. It then generates a new candidate trace, by invoking the canonical metacircular inference procedure, treating **resimulate-constraints** as the observations. The result is a new trace that has a fresh value at **resim-addr**. The next step is to calculate the probability that repeating the proposal can reverse the change to the trace. This is done via another call to **infer-apply**. Finally, a coin is flipped to decide whether to accept the new trace or return the old trace, according to the Metropolis-Hastings acceptance criterion.

3.5 Using inference to reason probabilistically about inference algorithm behavior.

Debugging inference algorithms is notoriously difficult, because they are nondeterministic and approximate. In Metaprob, these algorithms are indistinguishable from generative models, and as such, can be reasoned about using inference. In Figure 10, we use importance sampling to better understand the behavior of single-site MH. We invoke **infer-apply** with **single-site-mh-step** as the probabilistic model. (Since **single-site-mh-step** itself calls **infer-apply**, we rely here on the fact that it is a metacircular interpreter, and can itself be traced and scored.) We generate likely accepted "next steps" for the MCMC algorithm to take, starting from two different points. In the plot, each starting point is shown as a green dot; the likely next steps are shown in white. (This kind of "reflective" inference has also recently been introduced in the machine learning literature [5, 15, 16].)

```
;; Construct an initial trace
(define initial-trace
  (trace-with-values-at
    __h1_addr 72
    __h2_addr 80
    __total_adr 155))

;; Define constraint addrs
(define constraint-addrs (list __total_addr))

;; Trace a single MH step, so that we can pull out the
;; relevant addresses from the trace.
(define [_ mh-tr _]
  (infer-apply
    :procedure single-site-mh-step
    :inputs [circus-brothers [] initial-trace constraint-addrs]))
(define [__choice_addr __resampled_addr __accept_addr]
  (addresses-of mh-tr))

;; Condition on acceptance of our proposal and sample
;; 20 proposed "height 1" values from the approximate posterior.
(define posterior-samples
  (replicate 20
    (gen []
      (trace-value
        (importance-resample
          :procedure single-site-mh-step,
          :inputs [circus-brothers [] initial-trace
                             constraint-addrs],
          :intervention-trace
          (trace-with-values-at __choice_addr __h1_addr),
          :observation-trace
          (trace-with-values-at __accept_addr true),
          :n-particles 10)
      __resampled_addr))))
```

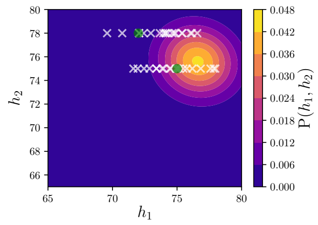


Figure 10. Inference metaprograms are generative procedures, so they can be used as models, and queried (via other inference metaprograms) as a means of debugging.

4 Generative Procedures

Metaprob's abstractions of traces, generative procedures, and inference procedures provide an effective language for inference metaprogramming. We next give these abstractions a semantics through a simple formal model of Metaprob that we then relate to standard concepts in probability theory. Specifically, our development in these next two sections contributes a semantics of generative procedures and inference procedures, with the final result being the semantics of `infer-apply`, Metaprob's canonical metacircular interpeter.

4.1 Syntax

Figure 11 presents the syntax of *generative procedures*, written $P(A)$ for a generative procedure that returns a value of type A. A Sample k s node randomly draws a sample from the probability distribution s, and associates the trace key k with the result of that draw. A ret e node directly returns a value e, and a sequencing node $x \leftarrow p_1$; $p_2(x)$ runs the probabilistic program p_1 to return a value x, then runs the probabilistic program $p_2(x)$. For the purposes of producing a small development, we defer the addition of a construct mapping to Metaprob's `inf` construct until Section 5.3.

4.2 Semantics

A generative procedure induces a probability distribution on its return values.

Measure Theory Preliminaries. We assume that the functional language we extend admits only terminating programs (Metaprob does allow general recursion, but we consider only the fragment with well-founded recursion here). Accordingly, it is possible to consistently interpret the functional programs here as measurable maps, where types correspond to measurable spaces. Where higher-order functions may be involved (such as **add-noise-to-curve** in Figure 1a), we can extend this interpretation accordingly, following Heunen et al. [17].Given a type A, we write $\mathcal{R}(A)$ for the space of probability distributions on A. We use $\delta : A \to \mathcal{R}(A)$ for the Dirac delta distribution. Given a measure μ on A and a non-negative real c, we write $c \times \mu$ for the measure μ scaled by c. Given a function f from A to measures on B, we write the integral $\int f(x)\, d\mu(x)$ (where the variable mention x after the $d\mu$ binds x inside the integrand) describes a measure ν on B satisfying $\nu(U) = \int f(x)(U)\, d\mu(x)$. We say the real-valued function f is the *Radon-Nikodym derivative* of ν with respect to μ if $\nu = \int f(x) \times \delta(x)\, d\mu(x)$, and we write $\frac{d\nu}{d\mu} = f$. A distribution $\mu : \mathcal{R}(A \times \mathbb{R}^+)$ where $\mathbb{R}^+ = \{x : \mathbb{R} \mid 0 \leq x\}$, may yield an importance sampler

$$\mathrm{imp}(\mu) \triangleq \frac{1}{\int w\, d\mu(x, w)} \times \int w \times \delta(x) d\mu(x, w),$$

when the integral in the quotient is finite.

Primitive Samplers. A primitive sampler s on the type A is represented by a probability distribution s.sample : $\mathcal{R}(A)$ and a score function s.score : $A \longrightarrow \mathbb{R}$ that indicates the likelihood of a given value. They are related by a base measure A.base_measure that is only a property of the type of the value returned according to $\frac{d(s.\text{sample})}{d(A.\text{base\_measure})} = s.\text{score}$. Note that we use (for convenience) likelihoods rather than *log*-likelihoods. A generative procedure induces a probability distribution on its return values as shown by $[\![\cdot]\!]_g$ in Figure 12.

4.2.1 Traces

Generative procedures are also characterized by the distribution it induces on the trace of its random choices, which matters for inference. Every random sample Sample k s is tagged with a trace key k. Note that in this formal model, we assume that trace keys are specified explicitly, whereas in Metaprob they are automatically generated. The tracing program transformation, shown in Figure 13, modifies a generative procedure to return a trace that records the values of each such random sample as a map indexed by trace keys. For simplicity, we flatten the tree-like structure of traces as defined previously: a *trace*, indicated by type Trace, is just a finite map from keys K to values (of any type). As long as trace keys do not collide, a trace records all the random decisions made in the run of a probabilistic program. Metaprob's automatic addressing ensures that trace keys do not collide.

5 Inference

We next give a semantics to inference, which includes inference procedures – which are generative procedures that have been extended to support observations and intervention (via Metaprob's **inf**) – and Metaprob's metacircular interpreter, **infer-apply**. We produce the final result, the semantics of **infer-apply**, by first giving a semantics to observations and then extending that semantics with interventions.

5.1 Conditioning with partial traces

Given a full trace, we can remove some elements, resulting in what we call a *partial trace.*

Observational interpreter. We can run an *observational interpreter* inf_obs, defined in Figure 14, on a partial trace t to generate a generative procedure for an importance sampler of the conditional distribution on the full trace given t.

Definition 5.1. [4] Given a distribution $\mu : \mathcal{R}(A)$ and a random variable $T : A \longrightarrow B$ to condition on, $\nu : B \longrightarrow \mathcal{R}(A)$ is the *conditional probability* (or *disintegration*) of μ with respect to T, written $\nu(t) = \mu(\cdot \mid T = t)$, if $\mu = \int \nu(T(x))\, d\mu(x)$ and for almost all x drawn from μ, for almost all y drawn from $\nu(T(x))$, we have $T(y) = T(x)$.

The observational interpreter generates conditional distributions whenever we condition on a subset S of trace keys such that every $k \in S$ almost always appears in $[\![\text{tracing}(p)]\!]_g$.

Proposition 5.2 (Importance sampling of conditionals)**.** *Let $p : P(A)$ be a generative procedure that almost never has a trace collision, and let S be a subset of trace keys that almost always appear. Then the importance sampler given by $\lambda t.$ inf_obs$_t$ (p) gives the conditional probability of $[\![\text{tracing}(p)]\!]_g$ with respect to* filter_keys$_S$ ∘ snd *, i.e.,*

$$[\![\text{inf\_obs}_t\ (p)]\!]_{\text{imp}} = [\![\text{tracing}(p)]\!]_g(\cdot \mid \text{filter\_keys}_S \circ \text{snd} = t),$$

where $[\![\mu]\!]_{\text{imp}} \triangleq \text{imp}([\![\mu]\!]_g)$.

5.2 Interventions

Given a partial trace for a program, we formalize *intervention* as a program transformation where we replace random sampling points with constants given by their values in the partial trace, as shown in Figure 15. The metacircular interpreter infer (which is synonymous with Metaprob's **infer-apply**), shown in Figure 16, combines observation and intervention in a single pass. It expects a partial trace passed in where each value is tagged to indicate whether observation or intervention should be performed. Conceptually, it can be understood as intervention followed by observation:

Proposition 5.3. *Decomposing a trace t into its interventions t_i and observations t_o, we have*

$$[\![\text{infer}_t\ (p)]\!]_g = [\![(x, \tau, w) \longleftarrow \text{inf\_obs}_{t_o}\ (\text{intervene}_{t_i}\ (p))\ ;\ \text{ret}\ (x, t_i \cup \tau, w)]\!]_g$$

5.3 Custom Inference Procedures

We next allow any generative procedure $p : P(A)$ to be annotated with a program of the type f : Trace $\longrightarrow P(A\times\text{Trace}\times\mathbb{R})$ with the syntax Inf p f (synonymous with Metaprob's **inf**) , such that the infer function in Figure 16 runs infer$_t$(Inf p f) = $f(t)$ rather than recursing on p. Such a function f is considered *valid* when for any trace t, for almost all (x, t', w) drawn from $[\![f(t)]\!]_g$,

$$w = \frac{d([\![\text{tracing}(\text{intervene}_t(p))]\!]_g)}{d(\text{fst}_*[\![f(t)]\!]_g)}(x, t')$$

(where the intervene$_t$ takes only the interventions from t).

Claim 5.4. *If $p : P(A)$ is a generative procedure such that all its component* Infs *are valid, then for all traces t,*

$$[\![\text{infer}_t(p)]\!]_{\text{imp}} = [\![\text{infer}_t(\tilde{p})]\!]_{\text{imp}}$$

where $\tilde{p}$ is p where all parts of p of the form Inf p' f *are replaced with just p', i.e., all* Infs *are replaced by their specifications.*

Note that the condition $[\![f(t)]\!]_{\text{imp}} = [\![\text{infer}_t(p)]\!]_{\text{imp}}$ on individual Infs would be insufficient for Claim 5.4 to hold. For instance, even though **bivariate-gaussian** (Figure 4b) yields an exact posterior, it does not simply always return the score 0, which would guarantee the above property but not the stronger notion of validity. The stronger notion is compositional, enabling Claim 5.4 to hold. Claim 5.4 demonstrates that the infer abstraction and custom **inf**s together support composition of custom inference within a model.

$$
\begin{aligned}
P &\rightarrow && \text{Sample } K\ S \mid \text{ret } E \mid x \leftarrow P\ ;\ P \\
E &: && A \\
K &: && \text{key} \\
S &: && \begin{cases} \text{sample} : \mathcal{R}(A) \\ \text{score} : A \rightarrow \mathbb{R}^+ \end{cases}
\end{aligned}
$$

Figure 11. Syntax of generative procedures (not including Infs, to be added in 5.3.

$$
\begin{aligned}
[\![\cdot]\!]_g &: P(A) \rightarrow \mathcal{R}(A) \\
[\![\text{Sample } k\ s]\!]_g &\triangleq s.\text{sample} \\
[\![\text{ret } e]\!]_g &\triangleq \delta(e) \\
[\![x \leftarrow p_1\ ;\ p_2]\!]_g &\triangleq \int [\![p_2(x)]\!]_g \, d[\![p_1]\!]_g(x)
\end{aligned}
$$

Figure 12. A generative procedure induces a probability distribution on its return values.

$$
\begin{aligned}
\text{tracing} &: P(A) \rightarrow P(A \times \text{Trace}) \\
\text{tracing}\,(\text{Sample } k\ s) &\triangleq x \leftarrow \text{Sample } k\ s\ ;\ \text{ret } (x, \{k \mapsto x\}) \\
\text{tracing}\,(\text{ret } e) &\triangleq \text{ret } (e, \{\}) \\
\text{tracing}\,(x \leftarrow p_1\ ;\ p_2) &\triangleq (x, t_1) \leftarrow \text{tracing}(p_1) \\
&\quad ;\ (y, t_2) \leftarrow \text{tracing}(p_2(x)) \\
&\quad ;\ \text{ret } (y, t_1 \cup t_2)
\end{aligned}
$$

Figure 13. A program transformation to trace random choices.

$$
\begin{aligned}
\text{inf\_obs}_t &: P(A) \rightarrow P(A \times \text{Trace} \times \mathbb{R}^+) \\
\text{inf\_obs}_t\,(\text{Sample } k\ s) &\triangleq \\
(y, s) &\leftarrow \begin{cases} \text{ret } (x, s.\text{score}(x)) & t(k) = x \\ x \leftarrow \text{Sample } k\ s\ ;\ \text{ret}(x, 1) & k \notin \text{dom}(t) \end{cases} \\
&\quad ;\ \text{ret } (y, \{k \mapsto y\}, s) \\
\text{inf\_obs}_t\,(\text{ret } e) &\triangleq \text{ret } (e, \{\}, 1) \\
\text{inf\_obs}_t\,(x \leftarrow p_1\ ;\ p_2) &\triangleq (x, t_1, w_1) \leftarrow \text{inf\_obs}_t\,(p_1) \\
&\quad ;\ (y, t_2, w_2) \leftarrow \text{inf\_obs}_t\,(p_2(x)) \\
&\quad ;\ \text{ret } (y, t_1 \cup t_2, w_1 w_2)
\end{aligned}
$$

Figure 14. Observation traces as weighted samplers.

$$
\begin{aligned}
\text{intervene}_t &: P(A) \rightarrow P(A) \\
\text{intervene}_t\,(\text{Sample } k\ s) &\triangleq \begin{cases} \text{ret } x & t(k) = x \\ \text{Sample } k\ s & k \notin \text{dom}(t) \end{cases} \\
\text{intervene}_t\,(\text{ret } e) &\triangleq \text{ret } e \\
\text{intervene}_t\,(x \leftarrow p_1\ ;\ p_2) &\triangleq \\
&\quad x \leftarrow \text{intervene}_t\,(p_1)\ ;\ \text{intervene}_t\,(p_2(x))
\end{aligned}
$$

Figure 15. Intervention as a program transformation.

$$
\begin{aligned}
\text{infer}_t &: P(A) \rightarrow P(A \times \text{Trace} \times \mathbb{R}^+) \\
\text{infer}_t\,(\text{Sample } k\ s) &\triangleq \\
(y, s) &\leftarrow \begin{cases} \text{ret } (x, s.\text{score}(x)) & t(k) = \text{Observe } x \\ \text{ret } (x, 1) & t(k) = \text{Intervene } x \\ x \leftarrow \text{Sample } k\ s\ ;\ \text{ret}(x, 1) & k \notin \text{dom}(t) \end{cases} \\
&\quad ;\ \text{ret } (y, \{k \mapsto y\}, s) \\
\text{infer}_t\,(\text{ret } e) &\triangleq \text{ret } (e, \{\}, 1) \\
\text{infer}_t\,(x \leftarrow p_1\ ;\ p_2) &\triangleq (x, t_1, w_1) \leftarrow \text{infer}_t\,(p_1) \\
&\quad ;\ (y, t_2, w_2) \leftarrow \text{infer}_t\,(p_2(x))\, v \\
&\quad ;\ \text{ret } (y, t_1 \cup t_2, w_1 w_2)
\end{aligned}
$$

Figure 16. The metacircular interpreter simultaneously performs observation and intervention, yielding a weighted sampler of a return value and trace.

6 Related Work

We discuss related work in two fields.

Probabilistic programming. Researchers and practitioners have introduced many probabilistic programming languages [2, 10, 11, 20, 23, 30, 32]. Unlike Metaprob, most of these languages have little to no support for customizing inference. New primitives and inference algorithms can be added via user-space code, without modifying the language implementation. This is a fundamental advance over languages such as Church [11], which required users to modify the language implementation to add new primitives and inference strategies. This is also a fundamental advance over languages with support for inference metaprogramming, such as Venture[20] and Turing[8]. For example, most inference tactics in Venture cannot be implemented in Venture, though they can be implemented in Metaprob.

Probabilistic inference. Several lines of research in probabilistic inference have introduced the idea of treating inference algorithms as probabilistic models, and reasoning about the random choices made by an inference algorithm over the course of execution [5, 15, 16]. Metaprob is well suited to expressing these techniques, as inference algorithms are represented as ordinary generative code. Metaprob programs can also be contrasted with Bayesian networks[25]. Like Bayesian networks, Metaprob programs have well-defined notions of observation and intervention, but unlike Bayesian networks, all valid Metaprob inference algorithms can be described as Metaprob programs.

7 Conclusion

This paper introduces new language constructs for first-class inference metaprogramming, illustrated their usefulness on a real-world case study with data from the U.S. Census, and presented a formal semantics. It also presents performance measurements showing that support for custom inference metaprogramming can improve performance and accuracy.

References

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

[2] Eli Bingham, Jonathan P Chen, Martin Jankowiak, Fritz Obermeyer, Neeraj Pradhan, Theofanis Karaletsos, Rohit Singh, Paul Szerlip, Paul Horsfall, and Noah D Goodman. 2018. Pyro: Deep Universal Probabilistic Programming. *arXiv preprint arXiv:1810.09538* (2018).

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

[4] Joseph T Chang and David Pollard. 1997. Conditioning as disintegration. *Statistica Neerlandica* 51, 3 (1997), 287–317.

[5] Marco Cusumano-Towner and Vikash K Mansinghka. 2017. AIDE: An algorithm for measuring the accuracy of probabilistic inference algorithms. In *Advances in Neural Information Processing Systems*. 3000–3010.

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

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

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

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

[10] Andrew Gelman, Daniel Lee, and Jiqiang Guo. 2015. Stan: A probabilistic programming language for Bayesian inference and optimization. *Journal of Educational and Behavioral Statistics* 40, 5 (2015), 530–543.

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

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

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

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

[15] Roger B Grosse, Siddharth Ancha, and Daniel M Roy. 2016. Measuring the reliability of MCMC inference with bidirectional Monte Carlo. In *Advances in Neural Information Processing Systems*. 2451–2459.

[16] Roger B Grosse, Zoubin Ghahramani, and Ryan P Adams. 2015. Sandwiching the marginal likelihood using bidirectional Monte Carlo. *arXiv preprint arXiv:1511.02543* (2015).

[17] Chris Heunen, Ohad Kammar, Sam Staton, and Hongseok Yang. 2017. A convenient category for higher-order probability theory. In *Logic in Computer Science (LICS), 2017 32nd Annual ACM/IEEE Symposium on*. IEEE, 1–12.

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

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

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

[21] Vikash Mansinghka, Patrick Shafto, Eric Jonas, Cap Petschulat, Max Gasner, and Joshua B Tenenbaum. 2016. Crosscat: A fully bayesian nonparametric method for analyzing heterogeneous, high dimensional data. *arXiv preprint arXiv:1512.01272* (2016).

[22] Vikash K Mansinghka, Ulrich Schaechtle, Shivam Handa, Alexey Radul, Yutian Chen, and Martin Rinard. 2018. Probabilistic programming with programmable inference. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 603–616.

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

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

[25] Judea Pearl. 1988. Probabilistic inference in intelligent systems.

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

[27] Feras Saad, Leonardo Casarsa, and Vikash Mansinghka. 2017. Probabilistic Search for Structured Data via Probabilistic Programming and Nonparametric Bayes. *arXiv preprint arXiv:1704.01087* (2017).

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

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

[30] Dustin Tran, Matthew D Hoffman, Rif A Saurous, Eugene Brevdo, Kevin Murphy, and David M Blei. 2017. Deep probabilistic programming. *arXiv preprint arXiv:1701.03757* (2017).

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

[32] Frank Wood, Jan Willem Meent, and Vikash Mansinghka. 2014. A new approach to probabilistic programming inference. In *Artificial Intelligence and Statistics*. 1024–1032.