Effect Handlers for Programmable Inference

Minh Nguyen
min.nguyen@bristol.ac.uk
University of Bristol
Bristol, UK

Roly Perera
roly.perera@bristol.ac.uk
University of Bristol
Bristol, UK

Meng Wang
meng.wang@bristol.ac.uk
University of Bristol
Bristol, UK

Steven Ramsay
steven.ramsay@bristol.ac.uk
University of Bristol
Bristol, UK

Abstract

Inference algorithms for probabilistic programming are complex imperative programs with many moving parts. Efficient inference often requires customising an algorithm to a particular probabilistic model or problem, sometimes called *inference programming*. Most inference frameworks are implemented in languages that lack a disciplined approach to side effects, which can result in monolithic implementations where the structure of the algorithms is obscured and inference programming is hard. Functional programming with typed effects offers a more structured and modular foundation for programmable inference, with monad transformers being the primary structuring mechanism explored to date.

This paper presents an alternative approach to inference programming based on algebraic effects. Using effect signatures to specify the key operations of the algorithms, and effect handlers to modularly interpret those operations for specific variants, we develop two abstract algorithms, or *inference patterns*, representing two important classes of inference: Metropolis-Hastings and particle filtering. We show how our approach reveals the algorithms' high-level structure, and makes it easy to tailor and recombine their parts into new variants. We implement the two inference patterns as a Haskell library, and discuss the pros and cons of algebraic effects *vis-à-vis* monad transformers as a structuring mechanism for modular imperative algorithm design.

CCS Concepts: • **Software and its engineering** → **Functional languages**; **Software libraries and repositories**; • **Mathematics of computing** → **Probabilistic algorithms**.

Keywords: probabilistic programming, algebraic effects, functional programming, modularity

1 Introduction

Probabilistic programming languages allow modellers to use programs to formulate inference problems over models. For example in ProbFX [Nguyen et al. 2022], a probabilistic language embedded in Haskell, a linear regression model relating input x and output y linearly can be expressed as:

```
linRegr  :: Double → Double → Model (Double, Double)
linRegr  x y = do
    m ← call (Sample (Normal 0 3))
    c ← call (Sample (Normal 0 2))
    call  (Observe (Normal (m * x + c) 1) y)
    pure  (m, c)
```

The two Sample operations specify the distributions that the line's slope m and intercept c are sampled from, representing our *prior* beliefs about m and c before accounting for any data, denoted $\mathbb{P}(m, c)$. Given an observed output y for some fixed input x, the operation Observe represents a conditioning side-effect, conditioning the model against the *likelihood* of y having been generated (in this case) from the normal distribution with mean $m * x + c$ and standard deviation of 1, denoted $\mathbb{P}(y \mid m, c; x)$. The variables x and y here are observable, whereas m and c that relate them are latent.

Inference over such a model is then the process of revising our estimation of its latent variables on the basis of the observed data, obtaining a *posterior* distribution. For the linear regression example, the Bayesian update rule yields the following equation for the posterior $\mathbb{P}(m, c \mid y; x)$:

$$\underbrace{\mathbb{P}(m, c \mid y; x)}_{\text{posterior}} = \frac{\overbrace{\mathbb{P}(y \mid m, c; x)}^{\text{likelihood}} \cdot \overbrace{\mathbb{P}(m, c)}^{\text{prior}}}{\underbrace{\mathbb{P}(y; x)}_{\text{evidence}}}$$

Unfortunately, extracting an exact form for the posterior is rarely simple. Although the Sample and Observe operations in linRegr determine the prior and likelihood respectively, computing the *evidence* $\mathbb{P}(y; x)$ that forms the denominator often involves complex, high-dimensional integration [Ackerman et al. 2011], and probabilistic languages in practice hence use approximation algorithms such as Monte Carlo methods [Andrieu et al. 2003] or variational inference [Fox and Roberts 2012]. Most techniques involve treating the model *generatively* — as something from which samples can be drawn — and then iteratively constraining the behaviour of the model so that, over time, those samples eventually conform to the observations.

When using the model generatively in this way, inference algorithms need to provide their own semantics for sampling and observing. For example, Metropolis-Hastings algorithms [Beichl and Sullivan 2000] execute the target model under specific *proposals*, that fix the stochastic choices made by the model on a given run. By selectively accepting or rejecting proposals, the algorithm controls how samples are generated, and guarantees that as more samples are produced, the distribution of values eventually converges on the desired posterior. Pseudocode for a generic Metropolis-Hastings iteration is shown here for linear regression:

```
do  (m′, c′) ← propose (m, c)
    ρ′       ← exec (linRegr x y) (m′, c′)
    b        ← accept ρ′ ρ
    pure  (if  b  then (m′, c′) else (m, c))
```

First new values m' and c' are proposed for the slope and intercept, given the values from the previous iteration, m and c. The function exec then executes the linear regression model with a custom semantics for sampling and observing, ensuring that m' and c' are used for the corresponding Sample operations, and conditioning with observations x and y. The resulting likelihood ρ' is compared with ρ from the previous iteration to determine whether to accept the new proposal or keep the current one. Running this procedure for many iterations will generate a sequence of samples m and c that approximate the posterior distribution $\mathbb{P}(m, c \mid y; x)$.

We think of Metropolis-Hastings, as sketched here, as an inference *pattern* rather than an inference *algorithm*: there are many algorithmic variants with this particular structure, differing only in how they implement propose, exec, and accept. Indeed, most algorithms come in similar families of variants, with abstract operations and skeletal behaviour shared by the variants, as well as their own bespoke execution semantics for models. Particle filters [Djuric et al. 2003], for example, also called sequential Monte Carlo methods, rely on being able to *partially* execute collections of models called *particles* from observation point to observation point; at each observation, particles are randomly filtered, or *resampled*, to retain only those likely to have come from the
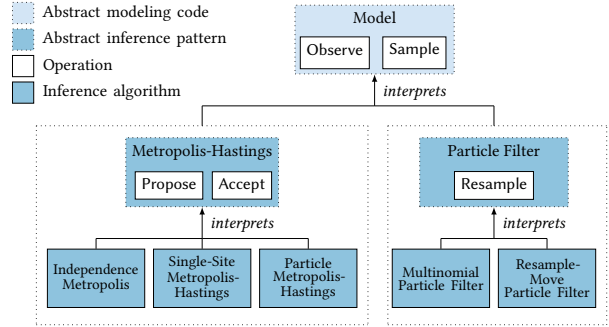

Figure 1. Inference patterns presented in this paper

target posterior. Different instances of the Particle Filter pattern vary in how the resampling operation works, and how particles are executed between observation points; different choices yield different well-known algorithms.

The task of implementing these algorithmic variants falls not just to library designers; model authors also often need to be versed in the intricacies of inference to achieve acceptable performance. Programming new inference algorithms out of reusable parts of existing ones is sometimes called *inference programming* [Mansinghka et al. 2014]. Existing approaches include: Venture [Mansinghka et al. 2018], a Lisp-based language using metaprogramming techniques; Monad-Bayes [Ścibior et al. 2018], which uses monad transformers to implement a modular library for inference programming in Haskell; and Gen [Cusumano-Towner et al. 2019], a inference programming framework in Julia which relies on a fixed black-box interface for executing models generatively.

In this paper, we present an approach to programmable inference based on algebraic effects. We use effect signatures to specify the key operations of various classes of abstract inference algorithms, and effect handlers to specialise those algorithms into concrete variants adapted to specific problems. We use the approach to develop two abstract algorithms, or *inference patterns*, representing two important classes of inference, and implement them in Haskell. Our specific contributions are as follows:

- §2 informally introduces the idea of an inference pattern.
- §3 presents the *Metropolis-Hastings* inference pattern, with Independence Metropolis and Single-Site Metropolis-Hastings as illustrative instances.
- §4 presents the *Particle Filter* inference pattern, also known as sequential Monte Carlo, with Multinomial Particle Filter and Resample-Move Particle Filter as instances. We also derive Particle Metropolis-Hastings, an instance of Metropolis-Hastings which uses Particle Filter.
- §5 shows that the performance of our approach is competitive with state-of-the-art systems for programmable inference based on other techniques.

- §6 contrasts our approach to untyped approaches such as Gen and Venture, and MonadBayes, the main existing framework based on typed effects.

The algorithms we discuss are well known; what we bring to the picture is the novel modular architecture, outlined in Fig. 1, which reveals the high-level structure of the algorithms and makes it easy to tailor and recombine their parts into new variants. The *model*, provided by the user, expresses an abstract inference problem in terms of Sample and Observe operations. *Inference patterns*, provided by library designers, assign specific semantics to those operations, and provide skeletal procedures for iteratively executing a model under those semantics. These procedures are in turn expressed in terms of their own abstract operations, which can also be assigned a semantics to obtain a concrete algorithm capable of generating samples from the model's posterior.

This design makes it easy to define new algorithmic variants out of existing ones. For example, we can easily build a particle filter algorithm which uses another well-known inference algorithm, Metropolis-Hastings, as an internal component; equally easily, we can derive a version of Metropolis-Hastings that uses particle filter. Moreover each of these complex scenarios arise in real-world solutions.

We build on two pieces of prior work: the *extensible freer monad* (§ 1.1), which adds an extensible effect system to Haskell, and ProbFX (§ 1.2), an embedding of probabilistic models in Haskell based on this approach.

1.1 Background: An Embedding of Extensible Effects

Effect systems model effects as coroutine-like interactions between side-effecting expressions that request *operations* to be performed, and special contexts, called *handlers*, that assign meaning to those operations [Bauer and Pretnar 2015]. An operation may provide a continuation, allowing the handler to return control to the requesting expression. Effect systems offer a flexible alternative to monad transformers [Liang et al. 1995] for adding complex imperative features to functional languages, making them an appealing tool for structuring inference algorithms. But the only precedent we know of is by Ścibior and Kammar [2015] on basic rejection sampling.

The *extensible freer monad* [Kiselyov and Ishii 2015] is an embedding of a typed effect system into Haskell, exploiting Haskell's rich support for embedded languages. The basic idea is to represent an effectful computation using the recursive datatype Comp es a at the top of Fig. 2. A term of type Comp es a represents a computation that produces a value of type a, whilst possibly performing any of the computational effects specified by the *effect signature* es, a type-level list of type constructors. Leaf nodes Val x contain pure values x of type a. Operation nodes Op op k contain *operations* op of the abstract datatype EffectSum es b, representing the invocation of an operation of type e b for some effect type constructor e in es, where b is the (existentially quantified) return type of the operation; the argument k is a continuation of type

```
data Comp (es :: [Type → Type]) (a :: Type) where
  Val :: a → Comp es a
  Op  :: EffectSum es b → (b → Comp es a) → Comp es a

instance Monad (Comp es) where
  (>>=)    :: Comp es a → (a → Comp es b) → Comp es b
  Val x    >>= f = f x
  Op op k  >>= f = Op op (k >=> f)

class e ∈ es where
  inj :: e a → EffectSum es a
  prj :: EffectSum es a → Maybe (e a)

call :: e ∈ es ⇒ e a → Comp es a
call op = Op (inj op) Val
```

Figure 2. Extensible freer monad embedding

b → Comp es a that takes the result of the operation and constructs the remainder of the computation.

As one might surmise, Comp es is a monad, allowing effectful code to piggyback on Haskell's **do** notation for sequential chaining of monadic computations. The bind operator (>>=) can be viewed as taking a computation tree of type Comp es a and extending it at its leaves with a computation generated by f :: a → Comp es b. In the Val x case, a new computation f x is returned; otherwise for Op op k, the rest of the computation k is composed with f using Kleisli composition (>=>). Values of type Comp es a are thus uninterpreted "computation trees" comprised of pure values and operation calls chosen from es.

EffectSum es is key to the extensibility of the approach, representing an "open" (extensible) sum of effect type constructors; a concrete value of type EffectSum es a is an operation of type e a for exactly one effect type constructor e contained in es. The implementation of EffectSum is hidden; the type class ∈ provides methods for safely injecting and projecting effectful operations of type e a into and out of EffectSum es a, with the constraint e ∈ es asserting that e is a member of es. The helper call makes it easy to write imperative code, as we saw in the linRegr example, injecting the supplied operation into EffectSum es a and supplying the leaf continuation Val.

1.1.1 Interpreting Effectful Computations. Fig. 2 provided the machinery required to construct effectful computations; Fig. 3 shows the machinery required to execute them. Executing an effectful computation means providing a "semantics" for each of its effects, in the form of an interpreter called an *effect handler*. A handler for effect type e has the type Handler e es a b; it assigns partial meaning to a computation tree by interpreting all operations of type e, *discharging* e from the front of the effect signature, and transforming the result type from a to b. Effect handlers are thus modular building blocks which compose to constitute full interpretations of programs.

The helpers handle and handleWith make it easy to implement handlers; handleWith is used for handlers that also

```
type Handler e es a b = Comp (e : es) a → Comp es b

handle
  :: (a → Comp es b)
  → (forall c. e c → (c → Comp es b) → Comp es b)
  → Handler e es a b
handle hval hop = handleWith () (const hval) (const hop′)
    where hop′ op k = hop op (k ())

handleWith
  :: s
  → (s → a → Comp es b)
  → (forall c. s → e c → (s → c → Comp es b) → Comp es b)
  → Handler e es a b
handleWith s hval _      (Val x)   = hval s x
handleWith s hval hop    (Op op k) = case decomp op of
    Left    op_e  → hop s op_e k′
    Right   op_es → Op op_es (k′ s)
    where k′ s′ = handleWith s′ hval hop ∘ k

decomp :: EffectSum (e : es) a → Either (e a) (EffectSum es a)
```

Figure 3. Effect handlers and handle/handleWith helpers

thread a state of type s, whereas handle sets s to be the trivial unit type. Both take two higher-order arguments: hval, which says how to interpret pure values, and hop, which says how to interpret operations of effect type e. In the Val x case, where the computation contains no operations, we simply apply hval to the return value (and state), yielding a computation from which e has been discharged. In the Op op k case, where op has type EffectSum (e : es) a, the auxiliary function decomp determines whether op belongs to the leftmost effect e, and can thus be handled by hop, or whether it belongs to an effect in es, in which case we can simply reconstruct the operation at the narrower type. In either case we recurse (by extending the continuation) to ensure that the rest of the computation is handled similarly.

1.2 Effects for Probabilistic Models

Nguyen et al. [2022] use the extensible freer monad representation from §1.1 to define an embedding of probabilistic models. *Models*, in Fig. 4, are simply computations with access to two specific effects, Sample and Observe, each with one operation: Sample d samples from probability distribution d, and Observe d y conditions d on an observed value y before returning that same value y. These operations characterise the minimal interface assumed by most inference methods, and for simplicity here, we assume they are the *only* model effects required, along with IO for random number generation.

Both Sample and Observe are constrained by the type class Dist d a, specifying that the type d represents a primitive distribution generating values of type a, with the functional dependency d → a indicating that d fully determines a. Instances of Dist d a must implement two functions: (i) draw,

```
type Model a = Comp [Observe, Sample, IO] a

data Sample a where
  Sample :: Dist d a ⇒ d → Sample a

data Observe a where
  Observe :: Dist d a ⇒ d → a → Observe a

class Dist d a | d → a where
  draw     :: d → Double → a
  logProb  :: d → a → LogP

type LogP = Double
```

Figure 4. Models as computations that sample and observe

which takes a distribution d and random point r from the unit interval $[0, 1]$, and draws a sample by inverting the cumulative distribution function of d at r; and (ii) logProb, which computes the log probability of d generating a particular value. (The synonym LogP is helpful for distinguishing log probabilities from other values of type Double.) For example, the Bernoulli distribution over Booleans, with probability p of generating True, and $1 - p$ for False, can be implemented as:

```
data Bernoulli  = Bernoulli { p :: Double }
instance Distribution Bernoulli Bool where
  draw    (Bernoulli p) r = r ≤ p
  logProb (Bernoulli p) b = if b then log p else log (1 – p)
```

This states that drawing True corresponds to drawing a random value $r \leq p$ uniformly from $[0, 1]$, with the log probabilities $\log p$ and $\log (1 - p)$ of drawing True and False respectively.

1.2.1 Interpreting Probabilistic Models.
Interpreting a model means providing a semantics for Sample and Observe. The most basic interpretation of a model, as a generative process with no inference, is usually called *simulating* (or *sampling from*) the model, and can be defined as the composition of the handlers shown in Fig. 5. The handler defaultObserve (trivially) interprets Observe d y operations to return the observed value y, via the continuation. The handler defaultSample interprets Sample d operations, as long as IO is also present in the effect signature, by first drawing a random value r uniformly from the interval $[0, 1]$ using the IO function random, and then generating a sample from d using draw. Lastly, the function runIO discharges the final IO effect by simply extracting and sequencing the IO actions, running the computation as a top-level Haskell program.

2 Inference Patterns
Our approach to programmable inference builds on the general embedding of extensible effects from §1.1 and probabilistic models from §1.2. Our key insight is that algebraic effects seem to be a natural fit for two kinds of extensibility central to programmable inference. First, representing models as (reinterpretable) effectful computations allows them to be assigned semantics tailored to specific algorithms. For

```
simulate  ::  Model a → IO a
simulate = runIO ∘ defaultSample ∘ defaultObserve

defaultObserve  ::  Handler Observe es a a
defaultObserve = handle Val hop where
  hop (Observe d y) k = k y

defaultSample  ::  IO ∈ es ⇒ Handler Sample es a a
defaultSample = handle Val hop where
  hop (Sample d) k = do r ← call random; k (draw d r)

runIO  ::  Comp [IO] a → IO a
runIO (Val x)   = pure x
runIO (Op op k) = fromJust ( prj op) >>= runIO ∘ k

random :: IO Double
```

Figure 5. Effect handlers for model simulation

example, we can instrument models to produce the traces needed for Metropolis-Hastings (§3), or arrange for models to execute stepwise rather than to completion for particle filters (§4). Second, we can take a similar view of the algorithms themselves. By representing the key actions of each broad approach to inference as reinterpretable "inference operations" — for example resampling, in the case of particle filters — we can turn them into extension points that can be given different meanings by different members of the same broad family of algorithms. Deriving a concrete inference algorithm is then a matter of supplying appropriate interpreters for the model and for the inference operations themselves. Moreover these extension points advertise to non-experts the key steps in the algorithms.

As well as offering a modular and programmable approach to algorithm design, this perspective also provides a useful conceptual framework for understanding inference. For example, Metropolis-Hastings and particle filters might look quite different algorithmically, but our approach provides a uniform way of looking at them: each can be understood as an abstract algorithm, parameterised by a model interpreter, and expressed using abstract operations whose interpretation is deferred to concrete implementations. This informal organisational structure we call an *inference pattern*, and is shown on the left-hand side of Fig. 6; a library designer developing their own abstract inference algorithms using our approach would most likely follow this high-level template. We now flesh out the idea of an inference pattern a little before turning to the patterns we developed for this paper.

Inference patterns. The core of an inference pattern (see Fig. 6, left) is an abstract algorithm expressing an inference procedure. Taking inspiration from the parallelism literature [Darlington et al. 1995], we call this an *inference skeleton*. Inference skeletons depend on algebraic effects in two essential ways. First, each skeleton is parameterised by a *model interpreter*, giving concrete algorithms control over model

execution; second, the skeleton is expressed in terms of abstract *inference operations* unique to the pattern, which act as additional extension points where concrete algorithms can plug in specific behaviour.

The model interpreter has a *model interpreter type*, whose exact form depends on the pattern, but is roughly:

```
type ModelExec a b = Model a → IO b
```

and is used by the skeleton to fully interpret the model into an IO action on each iteration. Having the inference skeleton execute the model all the way to an IO action allows the model and inference algorithm to have distinct effect signatures. Assuming inference operations with concrete type InfEffect, a skeleton will have a type resembling:

```
infSkeleton :: (InfEffect ∈ fs, IO ∈ fs)
    ⇒ ModelExec a b → Model a → Comp fs b
```

where fs contains only the effects specific to the algorithm. If instead the skeleton were to incorporate the effects of the model into its own computation, and the interpretation of the model deferred until the handling of the inference operations, fs would need to include model operations like Observe and Sample, and the resulting computation trees would be much larger. Keeping the effect signatures distinct makes for a more modular and efficient design.

Pattern instances. A pattern instance (Fig. 6, right) provides a *concrete algorithm*. It instantiates an inference skeleton with a specific model interpreter, determining the specific model execution semantics to be used, and then composes the result with an *inference handler* providing a specific interpretation of the inference operations. Pattern instances may also have auxiliary definitions.

We use this informal template to present our two inference patterns: Metropolis-Hastings (§3) and Particle Filter (§4), along with concrete instances illustrating the compositionality and programmability of the approach. These are available in an open source Haskell library. [1]

3 Inference Pattern: Metropolis-Hastings

Metropolis-Hastings algorithms repeatedly draw samples from a chosen "proposal" distribution. How these samples are generated is controlled by an accept/reject scheme, determining whether to accept a new proposal and thus move to a new configuration, or to reject it and remain in the current configuration. Under certain standard assumptions, then, these samples yield a Markov chain that converges to the target posterior. (Here we only consider the case where the proposal distribution is the actual model we are performing inference over.)

The key operations of the algorithm are proposing and accepting/rejecting proposals. To expose them as extension

---

[1] github.com/min-nguyen/prob-fx-2

INFERENCE PATTERN	PATTERN INSTANCE
Inference skeleton	*Concrete algorithm*
Abstract inference algorithm. Given a model and a model interpreter, yields a computation expressed in terms of inference operations.	Instantiates the inference skeleton with a model interpreter, and post-composes with an inference handler.
Inference operations	*Inference handler*
Operations specific to inference pattern, e.g. proposal or resampling.	Assigns a semantics to each inference operation.
Model interpreter type	*Model interpreter*
Pattern-specific type of model interpreters, which assign meaning to Sample and Observe and execute a model to an IO action.	Interprets a model with semantics specific to a concrete algorithm.

Figure 6. Inference patterns (left) and pattern instances (right)

points, we represent them by the inference effect Propose w in Fig. 7. The parameter w is a particular representation of probability, or *weight*; the datatype Trace represents proposals. A trace fixes a subset of the stochastic choices made by a model, which is key to how the algorithm controls where samples are drawn from.

The inference skeleton mh n τ_0 executes n abstract iterations of Metropolis-Hastings, iterating mhStep to generate a Markov chain of length n, from a (typically empty) starting trace τ_0. The head of the Markov chain $(x, (w, \tau))$ represents the current configuration; x is the sample last drawn from the model, τ is the trace for that model run, and w is an associated weight of type w, representing the probability density at τ. First, mhStep calls Propose τ to generate a new proposal $\tau^\dagger$ derived from τ. Then, the model interpreter exec is used to run the Model (Fig. 4), using the information in $\tau^\dagger$ to fix stochastic choices, and resulting in a new trace τ' and associated weight w'. The new trace contains at least as much information as $\tau^\dagger$, but additionally stores any choices not determined by $\tau^\dagger$. The result of exec is an IO computation, which is inserted into the computation tree using call. Finally, mhStep calls Accept to determine whether the new configuration is by some (unspecified) measure "better" than the current one, returning it if so, and otherwise retaining the current.

To fix stochastic choices, a trace must associate to each Sample operation enough information to determinise that sample. This can be achieved in various ways, but here we assume that Sample nodes are identified by *addresses α* [Tolpin et al. 2016] of abstract type Addr, either generated behind the scenes or manually assigned by the user; a trace is then a map from addresses to random values $r \in [0, 1]$ providing the source of randomness for drawing the sample associated with a given address. The Sample handler reuseTrace τ is used for executing a model under a trace τ: it generates the draw using the stored random value for α if there is one, and otherwise generates a fresh value r which is recorded in an updated trace. Since draw is pure, executing a model under a fixed (and sufficiently large) trace is deterministic,

---

INFERENCE PATTERN: METROPOLIS-HASTINGS

Inference skeleton
```
mh :: (Propose w ∈ fs, IO ∈ fs)
   ⇒ Int → Trace → ModelExec w a → Model a
   → Comp fs [(a, (w, Trace ))]
mh n τ₀ exec model = do
  let mhStep i chain
      | i < n = do
          let (x, (w, τ)) = head chain
          τ†              ← call (Propose τ)
          (x', (w', τ')) ← call (exec τ† model)
          nodeᵢ₊₁         ← call (Accept (x, (w, τ)) (x', (w', τ')))
          mhStep (i + 1) (nodeᵢ₊₁ : chain)
      | otherwise = pure chain
  node₀ ← call (exec τ₀ model)        -- initialise first node
  mhStep 0 [node₀]
```

Inference operations
```
data Propose w a where
  Propose :: Trace → Propose w Trace
  Accept  :: (a, (w, Trace)) → (a, (w, Trace))
          → Propose w (a, (w, Trace))
```

Model interpreter type
```
type ModelExec w a = Trace → Model a → IO (a, (w, Trace))
```

Auxiliary definitions
```
type Trace = Map Addr Double
reuseTrace :: IO ∈ es ⇒ Trace → Handler Sample es a (a, Trace)
reuseTrace τ₀ = handleWith τ₀ (λτ' x → Val (x, τ')) hop where
  hop τ (Sample d α) k = do r ← call random
                            let (r', τ') = findOrInsert α r τ
                            k τ' (draw d r')
```

Figure 7. Inference Pattern: Metropolis-Hastings

allowing the generative behaviour of the model to be controlled by providing it a specific trace. The reuseTrace handler is thus a reusable "inference component" which can be used by concrete instances of Metropolis-Hastings, of which we now present two examples: Independence Metropolis (§3.1) and Single-Site Metropolis-Hastings (§3.2).

PATTERN INSTANCE: INDEPENDENCE METROPOLIS

Concrete algorithm

im :: Int → Model a → IO [(a, (LogP, Trace))]
im n = runIO ∘ handlePropose$_{im}$ ∘ mh n empty execModel$_{im}$

Inference handler

handlePropose$_{im}$ :: IO ∈ fs ⇒ Handler (Propose LogP) fs a a
handlePropose$_{im}$ = handle Val hop **where**
  hop (Propose τ) k = **do** τ' ← mapM (const (call random)) τ
                              k τ'
  hop (Accept r@(_, (w, _)) r'@(_, (w', _))) k
               = **do let** ratio = w' – w
                     u ← call random
                     k (**if** exp ratio ≥ u **then** r' **else** r)

Model interpreter

execModel$_{im}$ :: ModelExec LogP a
execModel$_{im}$ τ = rassoc ∘ runIO ∘ reuseTrace τ ∘ likelihood

---

Auxiliary definitions

likelihood :: Handler Observe es a (a, LogP)
likelihood = handleWith 0 (λw x → Val (x, w)) hop **where**
    hop w (Observe d y α) k = k (w + logProb d y) y

rassoc = fmap ($\lambda((x, w), \tau) → (x, (w, \tau))$)

Figure 8. Pattern Instance: Independence Metropolis

3.1 Pattern Instance: Independence Metropolis

Fig. 8 defines a simple Metropolis-Hastings variant called Independence Metropolis, where each iteration proposes an entirely new set of samples, and determines whether the proposal is accepted by comparing its likelihood with the previous iteration. This specialises the weight type w in Propose w and ModelExec w a to the type LogP for log likelihoods.

The handler handlePropose$_{im}$ interprets Propose by mapping new random values over the entire trace. (One can equivalently return the empty trace, but our particular approach becomes useful for Particle Metropolis-Hastings in §4.3.) To interpret Accept, we compute the likelihood ratio between the current and previous iteration, and accept only if greater than a random point in the interval [0, 1].

For model execution, likelihood handles Observe by summing the log likelihood w over all observations with 0 as the starting value. The full Independence Metropolis algorithm is then derivable by providing mh with a number of iterations n, the empty map as the initial trace, and the model interpreter, before post-composing with handlePropose$_{im}$ and runIO to yield a Markov chain of n proposals for a given model.

3.2 Pattern Instance: Single-Site Metropolis-Hastings

The rate of accepted proposals in Independence Metropolis suffers as more variables are sampled from: because each proposal generates an entirely new trace, achieving a high likelihood means sampling an entire set of likely proposals. Fig. 9 defines Single-Site Metropolis-Hastings [Wingate et al.

PATTERN INSTANCE: SINGLE-SITE METROPOLIS-HASTINGS

Concrete algorithm

ssmh :: Int → Trace → Model a → IO [(a, (LPTrace, Trace))]
ssmh n τ = runIO ∘ handlePropose$_{ssmh}$ ∘ mh n τ execModel$_{ssmh}$

Inference handler

handlePropose$_{ssmh}$ :: IO ∈ fs ⇒ Handler (Propose LPTrace) fs a a
handlePropose$_{ssmh}$ = handleWith α_0 (const Val) hop **where**
  hop _ (Propose τ) k
    = **do** α ← call (randomFrom (keys τ))
            r ← call random
            k α (insert α r τ)
  hop α (Accept (x, (w, τ)) (x', (w', τ'))) k
    = **do** **let** ratio = (sum ∘ elems ∘ delete α)
                      (intersectionWith (–) w' w)
                  + log (size τ) - log (size τ')
            u ← call random
            k α (**if** exp ratio ≥ u **then** (x', (w', intersection τ' w'))
                            else (x, (w, τ)))

Model interpreter

execModel$_{ssmh}$ :: ModelExec LPTrace a
execModel$_{ssmh}$ τ
  = rassoc ∘ runIO ∘ reuseTrace τ ∘ defaultObserve ∘ traceLP

Auxiliary definitions

type LPTrace = Map Addr LogP

traceLP :: (Observe ∈ es, Sample ∈ es)
           ⇒ Comp es a → Comp es (a, LPTrace)
traceLP = loop empty **where**
  loop w (Val x) = Val (x, w)
  loop w (Op op k)
    | Just (Observe d y α) ← prj op
      = Op op (λx → loop (insert α (logProb d x) w) (k x))
    | Just (Sample d α)   ← prj op
      = Op op (λx → loop (insert α (logProb d x) w) (k x))
    | **otherwise** = Op op (loop w ∘ k)

randomFrom :: [a] → IO a

Figure 9. Pattern Instance: Single-Site Metropolis-Hastings

2011], which uses an alternative semantics for model execution and inference: proposing just one sample per iteration, and otherwise reusing samples from the previous iteration. The acceptance/rejection scheme is also slightly different, comparing individual probabilities of Sample and Observe operations with respect to the proposed sample. This specialises the weight type w of Propose w to LPTrace, mapping addresses to their log probabilities.

The handler handlePropose$_{ssmh}$ threads an address α, identifying the sample currently being proposed. (The initial value of this argument is unused, so we supply an arbitrary value α_0.) For Propose, we use a helper randomFrom to select a new address α uniformly from the keys of trace τ, and then return τ updated with a new random value for α. For Accept, the

acceptance ratio between w' and w is computed for corresponding addresses by intersectionWith (−), using delete α to exclude the current proposal site, and also accounting for the ratio of sizes between the two traces. [2] If the new trace τ' is accepted then intersection τ' w' clears all unused samples from it, given that w' will only ever store addresses relevant to the model's execution, as described next.

The semantics for model execution differs only slightly from Independence Metropolis. Instead of summing the log probabilities of Observe operations, we record the log probabilities of all Observe and Sample operations encountered into a fresh map w of type LPTrace, via the handler traceLP shown in Fig. 9. This deviates from the normal handler pattern, matching on the result of prj op (§1) to intercept operations of different effect types but leaving them unhandled. Here we simply modify the continuation k to store the log probability of the operation's result. The case of prj returning Nothing follows the same pattern as decomp returning Left in Fig. 3.

All conditioning side-effects are in fact taken care of by traceLP, so the residual Observe operations are handled by defaultObserve to simply return the observed values, and the Sample operations by reuseTrace as before; the interpreted model has type IO (a, (LPTrace, Trace)), containing the final log probability and execution traces. We now have the parts to derive Single-Site Metropolis-Hastings from the mh pattern.

4 Inference Pattern: Particle Filter

Particle filters [Doucet et al. 2009] generate samples from the posterior by considering *partial* model executions. The idea is to spawn multiple instances of the model called *particles*, and then repeatedly switch between (i) running the various particles in parallel up to their next observation, and (ii) subjecting them to a *resampling* process [Hol et al. 2006]. Resampling is a stochastic strategy for filtering out particles whose observations are deemed unlikely to have come from the posterior, i.e. are weighted lower than other particles. Ideally, after many resampling steps, only particles that closely approximate the posterior will remain.

A particle filter configuration is a list of (particle, weight) pairs of type (Model a, w). The key operation is resampling, which transforms a configuration by discarding some particles and duplicating others, but usually keeping the number of particles constant; we expose this as an extension point via the inference effect type Resample w in Fig. 10. The model interpreter type ModelStep w a for particle filter is distinctive because it characterises *particle steppers*, which partially execute particles: a particle stepper resumes a suspended particle with weight w, executes it by some unspecified amount, and returns an updated particle and weight.

---

[2] By using intersectionWith (-), we assume that each execution of the model encounters the same (addresses of) Observe operations, which is a common assumption in probabilistic programming languages.

INFERENCE PATTERN: PARTICLE FILTER

Inference skeleton
```
pfilter :: (Resample w ∈ fs, IO ∈ fs)
    ⇒ Int → w → ModelStep w a → Model a → Comp fs [(a, w)]
pfilter n w₀ step model = pfStep (replicate n (model, w₀))
  where pfStep pws = do
          pws' ← call (mapM step pws)
          case done pws' of
            Just  rs  → Val rs
            Nothing → call (Resample pws') >>= pfStep
```

Inference operations
```
data Resample w a where
  Resample :: [(Model a, w)] → Resample w [(Model a, w)]
```

Model interpreter type
```
type ModelStep w a = (Model a, w) → IO (Model a, w)
```

Auxiliary definitions
```
done :: [(Model a, w)] → Maybe [(a, w)]
done ((Val x, w) : pws) = done pws >>= Just ∘ ((x, w) :)
done (_ : _)            = Nothing
done []                 = Just []

advance :: LogP
          → Handler Observe es a (Comp (Observe : es) a, LogP)
advance w (Val x)  = Val (Val x, w)
advance w (Op op k) = case decomp op of
  Left  (Observe d y _) → Val (k y, w + logProb d y)
  Right opₑₛ             → Op opₑₛ (advance w ∘ k)
```

Figure 10. Inference Pattern: Particle Filter

The inference skeleton pfilter n w₀ describes a generic particle filter, recursively running a set of n particles with a starting weight of w₀ until termination using pfStep, at each iteration using the particle stepper step to obtain a new configuration pws'. The function done examines the new configuration to determine whether all particles have terminated, in which case the return values and final weights of the particles rs are returned, or whether some particles are still executing, in which case the algorithm calls Resample on the configuration and continues with the filtered result.

The handler advance is a reusable inference component for implementing particle steppers. Given an initial weight w, it advances a particle to the next Observe, returning the rest of the computation k y *unhandled*, along with the accumulated weight at that point. Matching on Val instead means the particle has terminated, and so is returned alongside its final weight. Notice that advance is not implemented in terms of handle; this is because handle produces a "deep" handler [Hillerström and Lindley 2018] which discharges the handled effect from the effect signature, and so does not support the shallow (partial) handling needed for suspensions.

We now present two instances of Particle Filter: Multinomial Particle Filter (§4.1) and Resample-Move Particle Filter

PATTERN INSTANCE: MULTINOMIAL PARTICLE FILTER

Concrete algorithm

```
mulpfilter :: Int → Model a → IO [(a, LogP)]
mulpfilter n
  = runIO ∘ handleResample_mul ∘ pfilter n 0 stepModel_mul
```

Inference handler

```
handleResample_mul :: IO ∈ fs ⇒ Handler (Resample LogP) fs a a
handleResample_mul = handle Val hop where
  hop (Resample pws) k = do
    let (ps, ws)      = unzip pws
        (ws_norm, w̄s̄) = (normalise ws, logMeanExp ws)
    idxs ← call (replicateM (length ps) (categorical ws_norm))
    k (map ((, w̄s̄) ∘ ps!!) idxs)
```

Model interpreter

```
stepModel_mul :: ModelStep LogP a
stepModel_mul (p, w) = (runIO ∘ defaultSample ∘ advance w) p
```

Auxiliary definitions

```
normalise  :: [LogP] → [LogP]
logMeanExp :: [LogP] → LogP
categorical :: [LogP] → IO Int
```

Figure 11. Pattern Instance: Multinomial Particle Filter

(§4.2), the latter constructed using Metropolis-Hastings. We also present Particle Metropolis-Hastings (§4.3), an instance of Metropolis-Hastings constructed using Particle Filter.

4.1 Pattern Instance: Multinomial Particle Filter

Many basic variants of particle filters can be implemented by recording just the log probabilities of particles, specialising w in Resample w and ModelStep w to LogP. A popular example is a particle filter that uses a "multinomial resampling" algorithm, defined in Fig. 11. To interpret Resample pws, containing n particles ps and their weights ws, we use the categorical distribution to draw n integers from the range $0 \ldots n - 1$ with log probabilities corresponding to the normalised weights ws_{norm}. These integers indicate the positions of particles to continue executing with, which are extracted by indexing with (!!), and then uniformly paired with the log mean of the weights, $\overline{ws}$; it is expected for particles with higher weight to be selected more than once, and unlikely ones pruned.

For model execution, Observe is handled by advance, and Sample simply by defaultSample for drawing random values. Then we can derive mulpfilter by using pfilter n 0 handleParticle to construct an abstract particle filter of n particles with starting weight 0, and composing with (runIO ∘ handleResample_mul) to specialise to a multinomial particle filter that generates n samples from the posterior and their final weights.

4.2 Pattern Instance: Resample-Move Particle Filter

Complex inference problems often require the programmer to combine different top-level inference procedures, each

PATTERN INSTANCE: RESAMPLE-MOVE PARTICLE FILTER

Concrete algorithm

```
rmpf :: Int → Int → Model a → IO [(a, PState)]
rmpf n m model = (runIO ∘ handleResample_rmpf m model
                  ∘ pfilter n (0, empty) stepModel_rmpf) model
```

Inference handler

```
handleResample_rmpf :: IO ∈ fs
  ⇒ Int → Model a → Handler (Resample PState) fs a a
handleResample_rmpf m model = handleWith 0 (const Val) hop
  where
  hop t (Resample pwτs) k = do
    let (ws, τs)       = (unzip ∘ map snd) pwτs
        (ws_norm, w̄s̄) = (normalise ws, logMeanExp ws)
    idxs ← call (replicateM (length τs) (categorical ws_norm))
    let τs_res  = map (τs !!) idxs
        model_t = suspendAfter t model
    k (t + 1) =<< forM τs_res (λτ → do
         (p_mov, (_, τ_mov)):_ ← call (ssmh m τ model_t)
         pure (p_mov, (w̄s̄, τ_mov)))
```

Model interpreter

```
stepModel_rmpf :: ModelStep PState a
stepModel_rmpf (p, (w, τ))
  = (rassoc ∘ runIO ∘ reuseTrace τ ∘ advance w) p
```

Auxiliary definitions

```
type PState = (LogP, Trace)

suspendAfter :: Observe ∈ es
  ⇒ Int → Comp es a → Comp es (Comp es a)
suspendAfter _ (Val x)    = Val (Val x)
suspendAfter t (Op op k) = case prj op of
  Just (Observe d y _) → if t ≤ 0 then Val (k y)
                                  else Op op (suspendAfter (t - 1) ∘ k)
  Nothing              → Op op (suspendAfter t ∘ k)
```

Figure 12. Pattern Instance: Resample-Move Particle Filter

addressing a different sub-problem. For example, the resampling step in particle filtering can result in many particles becoming the same, limiting the range of values sampled from the posterior, a problem called *particle degeneracy*. One solution is to use Metropolis-Hastings proposals to "move around" the sampled values of each particle after resampling, an approach called the Resample-Move Particle Filter [Gilks and Berzuini 2001]. This kind of wholesale algorithm reuse is also supported in our framework, and we show this now by deriving Resample-Move Particle Filter (Fig. 12) as an instance of Particle Filter, by providing a Resample handler which calls Single-Site Metropolis-Hastings (§3.2).

To specialise pfilter to use Metropolis-Hastings, we set the weight parameter w to PState, now also storing the particle's execution trace to allow for proposals. To know how far to

---

PATTERN INSTANCE: PARTICLE METROPOLIS-HASTINGS

Concrete algorithm

```
pmh :: Int → Int → [Addr] → Model a → IO [(a, (LogP, Trace))]
pmh m n θ model = do
    (_, τ) ← (runIO ∘ reuseTrace empty ∘ defaultObserve) model
    let τθ = filterKey (`elem` θ) τ
    (runIO ∘ handleProposeim ∘ mh m τθ (execModelpmh n)) model
```

Model interpreter

```
execModelpmh :: Int → ModelExec LogP a
execModelpmh n τθ model = do
    let stepModelpmh :: ModelStep LogP a
        stepModelpmh (p, w) = (fmap fst ∘ runIO
                                        ∘ reuseTrace τθ ∘ advance w) p
    (xs, ws) ← (fmap unzip ∘ runIO ∘ handleResamplemul
                    ∘ pfilter n 0 stepModelpmh) model
    let (wsnorm, w̄s) = (normalise ws, logMeanExp ws)
    idx ← categorical wsnorm
    pure (xs !! idx, (w̄s, τθ))
```

Figure 13. Pattern Instance: Particle Metropolis-Hastings

execute a particle under a given proposal, the Resample handler increments a state variable t at each Resample, tracking the number of observations encountered so far in the model.

To handle Resample, we unzip the particle states into their weights ws and traces $τs$, using ws to carry out multinomial resampling as in Fig. 11 but for resampling a selection of traces $τs_{res}$. The helper suspendAfter then produces a copy $model_t$ of the model suspended after observation t, which will let us instantiate new particles that resume at that point. We execute $model_t$ under each resampled trace in $τs_{res}$ for a series of ssmh updates; the most recent update is taken, from which the final moved particle p_{mov} and its trace are used.

The model interpreter is simply a particle stepper which uses reuseTrace instead of defaultSample to record/reuse the particle's trace. The concrete algorithm rmpf n m can then be assembled from these parts, yielding a multinomial particle filter of n particles, where each resampling step is followed by m Single-Site Metropolis-Hastings updates to each particle.

4.3 Pattern Instance: Particle Metropolis-Hastings

We now revisit the Metropolis-Hastings inference pattern from §3, and show that our framework makes it equally easy to reuse a particle filter inside Metropolis-Hastings. In §3, we only considered algorithms where the proposed traces $τ$ fixed the values of *all* latent variables, fully determinising the model. But often we only care about proposing a *subset* of the trace, $τθ$ for some variables of interest $θ$, allowing the other latent variables to be freshly sampled. It then becomes possible to use a particle filter to run each proposal for many different simulations, averaging over the particles to compute the likelihood used to accept or reject the proposal. This is known as Particle Metropolis-Hastings [Dahlin

et al. 2015] and is used to reduce the variance of likelihood estimates of proposals. Fig. 13 derives a version of this from the Metropolis-Hastings pattern, providing a ModelExec that also calls a multinomial particle filter (§4.1), and reusing the Propose handler from Independence Metropolis (§3.1).

The model interpreter takes a number of particles n and trace $τθ$ providing values for the latent variables of interest, i.e. addresses $θ$. It begins by defining an internal particle stepper which executes a particle to the next observation as usual, but handles Sample with reuseTrace $τθ$ so that each particle uses fixed values for the latent variables in $θ$, using fmap fst to ignore the updated trace. The particle stepper is then used to instantiate a particle filter otherwise identical to the multinomial one, producing a list of particle outputs xs and weights ws. To conform to the ModelExec type for Metropolis-Hastings, the model interpreter must return a model result plus a weight and trace; for the model result we draw an element of xs with probability proportional to the weights, and for the weight we use the log mean of ws. For the trace, we return $τθ$ rather than the possibly extended trace returned by reuseTrace, to avoid fixing stochastic choices other than those in $τθ$ (when handling Propose in §3.1).

The algorithm pmh m n $θ$ then describes m Independence Metropolis proposals for addresses $θ$, but where each proposal is weighted by simulating the model as n particles. The first two lines initialise $τθ$, using reuseTrace empty to populate an empty trace, and then filtering to the addresses in $θ$.

5 Performance Evaluation

Before considering how well our approach achieves the goals set out in §1, we consider how practical it is for actually running inference. This section shows that our implementation is capable of competing with real-world probabilistic programming systems, suggesting that the choice of algebraic effects as a foundation does not imply a compromise on performance. We compare with two state-of-the-art systems designed with programmable inference as an explicit goal: MonadBayes [3] [Ścibior et al. 2018], a Haskell library that uses a monad transformer effect system, and Gen [4] [Cusumano-Towner et al. 2019], an embedded language in Julia.

We compared the mean execution times of four algorithms: Single-Site Metropolis-Hastings (SSMH), Multinomial Particle Filter (MPF), Particle Metropolis-Hastings (PMH), and Resample-Move Particle Filter (RMPF). Each algorithm is applied across three types of model: linear regression, hidden Markov model, and Latent Dirichlet allocation. These experiments were carried out on an Intel Core i7-9700 CPU with 16GB RAM.

On average, we outperform either one or both of the other systems across all algorithms, sometimes asymptotically or by several orders of magnitude. When varying the number of iterations performed or particles used by each algorithm

---
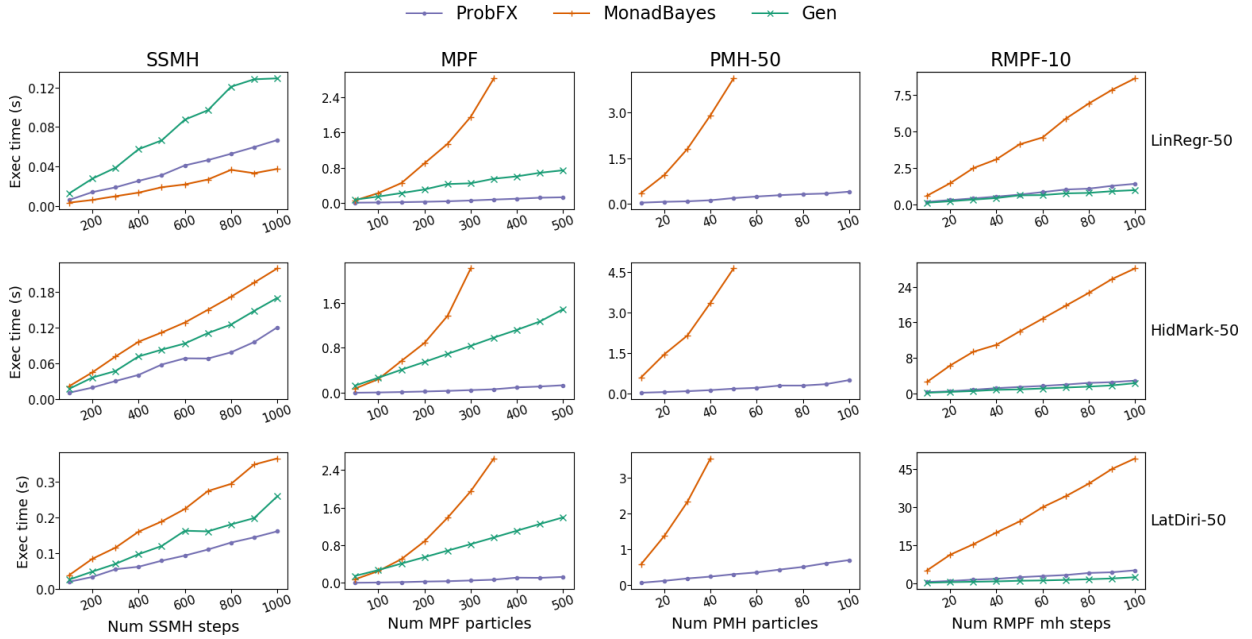
(a) Execution times of inference algorithms (top) with varying number of algorithm iterations or particles. The right-hand axis fixes the number of observations. PMH-50 indicates 50 MH updates that vary in the number of particles, and RMPF-10 indicates 10 particles that vary in the number of MH updates.
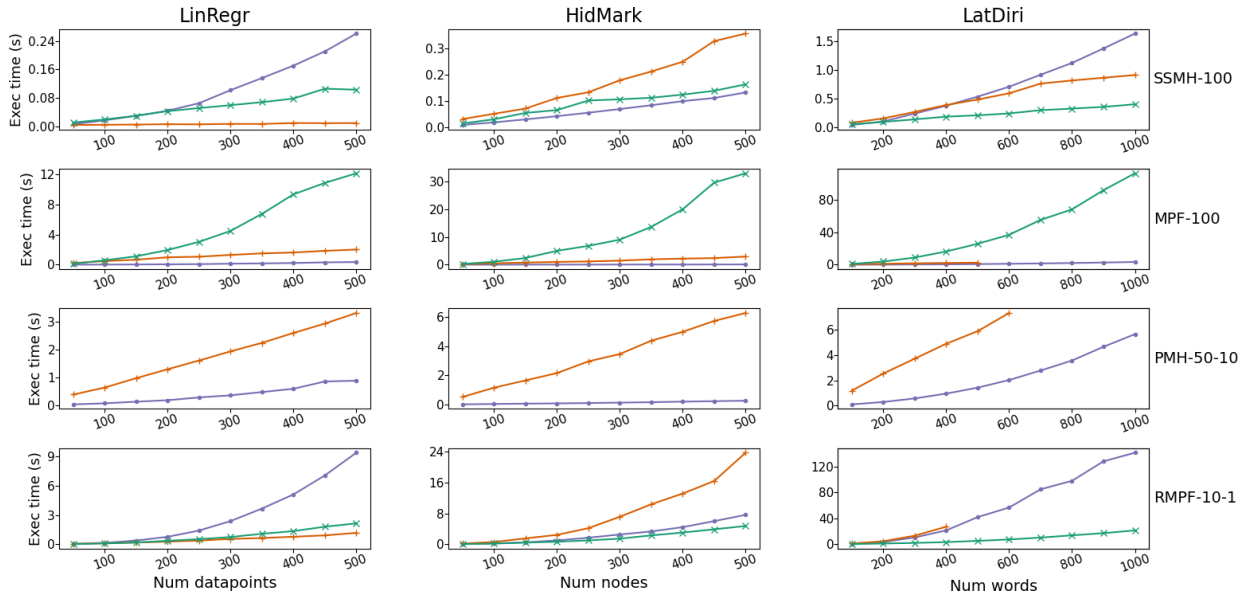
(b) Execution times of inference algorithms (right) with varying number of observations. The right-hand axis fixes the number of algorithm iterations or particles. PMH-50-10 indicates 50 MH updates that use 10 particles; RMPF-10-1 indicates 10 particles that use 1 MH update.

Figure 14. Performance comparison of our system, ProbFX, with MonadBayes and Gen in terms of mean execution times. The number of executions per mean is left to the control of the benchmarking suites, Criterion (Haskell) and BenchmarkTools.jl (Julia). Truncated line plots indicate an algorithm being killed early by the host machine for certain benchmark parameters. Missing line plots indicate an algorithm not being readily implemented in the system.

in Fig. 14a, our performance scales linearly across all models. Our performance remains linear when varying the number of observations provided to models in Fig. 14b, except for RMPF where, like MonadBayes and Gen, we scale quadratically.

Against MonadBayes, for SSMH we are on average 15x slower for linear regression, and 1.8x faster for other models. The former result is likely because of the specific linear regression model used, which varies only in the number of observe operations, and in contrast to our implementation, their version of SSMH does not store log weights for individual observations, but instead simply sums over them. For MPF, PMH, and RMPF, we average faster by 27x, 16x, and 4.9x across all models. When increasing the number of particles in MPF and PMH, the runtime of MonadBayes scales quadratically, and the process is killed when more than a moderate number of particles are used. We suspect this is due to their use of the ListT monad transformer to represent collections of particles, which in our experience scales poorly as the size of the transformer stack grows.

Comparing with Gen, we are roughly 1.1x and 72x faster for SSMH and MPF, the latter arising mainly because Gen's MPF implementation scales quadratically with the number of model observations, and for RMPF, we are on average 2.9x slower. We do not compare PMH since it is not directly provided in Gen, and so leave this to future work.

6 Qualitative Comparison

§1 identified two key forms of extensibility central to programmable inference, of which we have seen several examples in the preceding sections:

1. **Reinterpretable models.** Different algorithms require custom semantics for how models sample and observe, as well as fine-grained control over model execution in order to implement essential behaviours like suspended particles and tracing. "Programmability" here means being able to easily customise how models execute in order to derive or adapt inference algorithms.

2. **Modular, reusable algorithms.** Different algorithms from the same broad family implement key behaviours like resampling or proposing differently. "Programmability" here means being able to plug alternative behaviours into an existing algorithm without reimplementing it from scratch, but also being able to define new abstract algorithms that are easily pluggable in this way.

Given that inference programming is often undertaken by domain experts, for whom the activity may primarily be a means to an end, programmability matters. Here we look at how programmability is achieved in existing systems, briefly considering dynamically typed settings in §6.1, and then turning in more detail to MonadBayes in §6.2, the main existing system based on typed effects.

6.1 Dynamically typed approaches

Most programmable inference systems to date have been implemented in dynamic languages. We consider Venture, Gen, Pyro, and Edward; other mainstream systems like Anglican [Tolpin et al. 2016] and Turing [Ge et al. 2018] were not designed with inference programming in mind.

Reinterpretable models. In Venture [Mansinghka et al. 2014], modelling and inference instructions are interleaved, with the inference code affecting the semantics of preceding modelling code; this is flexible but lacks a clear delination between model and inference. Gen (in Julia) provides a black-box interface for interacting with models, exposing capabilities such as simulating and tracing, but the operations are non-programmable (have fixed meanings). Pyro [Bingham et al. 2019] and Edward [Moore and Gorinova 2018] (in Python) are more flexible, relying on a stack of programmable coroutines that are sequentially invoked by sample and observe calls; this has some flavour of algebraic effects, allowing bespoke semantics for sample and observe, albeit without a type discipline for tracking effects and associating them to handlers, and requiring global state to maintain the coroutine stack.

Control over model execution is realised in different ways. For particle stepping, Gen requires the programmer to manage this themselves, parameterising their model on the number of steps to be executed. In Pyro, the programmer must implement a method step for any model they want to execute in this way. Other dynamic languages rely on continuation-passing-style transformations [Goodman 2014; Tolpin et al. 2016]. Algebraic effects seem to offer a clear advantage here, providing handlers with access to the continuation and making idioms like stepwise execution easy to implement in inference code, rather than requiring any changes to models.

Modular, reusable algorithms. Venture offers a range of high-level inference procedures as reusable primitives, but new inference primitives must be written in Venture's DSL, which cannot reuse external inference code. In Gen, Pyro, and Edward, the inference libraries are implemented using regular host-language functions. While technically reusable, the lack of an effect discipline means these functions tend to mix arbitrary computation with model interactions, rather than being organised explicitly around the key operations of the algorithm, making them challenging to reuse in new contexts.

6.2 MonadBayes

MonadBayes is a Haskell library for typed programmable inference based on the Monad Transformer Library (MTL). MTL is an imperative programming framework that allows the programmer to stack monads, producing a combined effect consisting of "layers" of elementary monadic effects called *monad transformers* [Kiselyov et al. 2013]. A given set of monads may be layered in different ways; moreover

Sampling and observing as type class methods
```
class  Monad m ⇒ MSamp m where
  rand  :: m Double
class  Monad m ⇒ MCond m where
  score  :: LogP → m ()
```

Monad for weighting a model
```
data Weighted m a = W (StateT LogP m a)
instance MSamp m ⇒ MSamp (Weighted m) where
  rand    = lift ∘ rand
instance Monad m ⇒ MCond (Weighted m) where
  score  w = W (modify (+ w))
```

Monad for tracing a model
```
data Traced m a = Tr (Weighted (FreeT SamF m) a) (m (Trace' a))
instance MSamp m ⇒ MSamp (Traced m) where
  rand    = Tr rand (fmap singleton rand)
instance MCond m ⇒ MCond (Traced m) where
  score  w = Tr (score  w) (score  w >> pure (score  w))
```

(a) MonadBayes

Sampling and observing as data constructors
```
data Sample a where
  Sample :: Dist d a ⇒ d → Sample a
data Observe a where
  Observe :: Dist d a ⇒ d → a → Observe a
```

Handler for weighting a model
```
likelihood  :: Handler Observe es a (a, LogP)
likelihood  = handleWith 0 (λw x → Val (x, w))
  (λw (Observe d y) k → k (w + logProb d y)  y)
```

Handler for tracing a model
```
reuseTrace  :: Trace → Handler Sample es a (a, Trace)
reuseTrace  τ₀ = handleWith τ₀ (λτ x → Val (x, τ))
  (λτ (Sample d α) k → do r ← call random
                          let  (r', τ') = findOrInsert  α r τ
                          k  τ' (draw d r'))
```

(b) Our library

Figure 15. Support for reinterpretable models

layers can be abstract, with their operations defined by a type class. To invoke an operation of a specific abstract monad m from the stack, the user (or library) must define how each monad transformer above m *relays* that operation call further down the stack. A program written in MTL, whose type is an abstract stack of monad transformers, determines its semantics by instantiating to a particular concrete stack.

Reinterpretable models. In MonadBayes, reinterpretable models are provided by MTL's support for abstract monad stacks. The constrained type (MSamp m, MCond m) ⇒ m a represents a model, where the type constructor m is an abstract stack of monad transformers, each providing semantics for sampling (rand) and observing (score) by implementing the type classes MSamp and MCond in Fig. 15a. Following the usual MTL pattern, each concrete monad must either give a concrete behaviour for rand and/or score, or relay that operation to a monad further down the stack. For example, the Weighted m monad is for weighting a model m; it updates a stored weight when observing with score, but simply delegates any calls to rand to its contained monad m, using lift. The analogue of MSamp and MCond in our library are the concrete datatypes Sample and Observe in Fig. 15b, whose operations are also abstract (now as data constructors), but with semantics given by effect handlers rather than class instances; the counterpart to the Weighted m monad is the likelihood handler which interprets Observe to accumulate a weight. The analogue of relaying comes "for free" in the algebraic effects implementation, via handleWith (§1).

While monad transformers are both compositional and type-safe, the network of relaying that arises in MonadBayes is non-trivial. More than one concrete monad in the stack may provide sample and observe behaviours (such as Traced

in Fig. 15a, which recursively applies rand and score to its components); others may opt *not* to relay. As relaying is carried out implicitly, via type class resolution, the eventual runtime behaviour of a model may not be obvious. With algebraic effects, the correspondence between operations and their semantics is usually more evident, in the form of handlers, such as the reuseTrace and likelihood handlers in Fig. 15b which provide semantics for Sample and Observe.

For control over model execution e.g. for particle stepping, MonadBayes requires the programmer to use specific control effects, namely the free monad transformer FreeT and the Coroutine monad. Although model authors are oblivious to this particular detail, inference code can still require a significant amount of plumbing which can obscure the key operations of the algorithm. Algebraic effects instead provide access to the continuation in each handler, allowing the advertised effect signature to remain domain-specific.

Modular, reusable algorithms. The reusable building blocks in MonadBayes are datatypes that implement the type classes MSamp and MCond from Fig. 15a, such as Weighted and Traced. Inference algorithms are functions that instantiate a model's type from an abstract stack to a specific sequence of these datatypes. To illustrate, the (simplified) type of rmpf in Fig. 16a, read inside-out, instantiates the supplied model to "list of weighted, traced executions". This expresses Resample-Move Particle Filter as a computation that nests Metropolis-Hastings (using Traced) inside a particle filter (using ListT for particles). Conversely, the type of pmh suggests that Particle Metropolis-Hastings uses a particle filter inside Metropolis-Hastings. Thus the construction of inference algorithms out of reusable parts is expressed primarily at the type level: by selecting combinations of datatypes, one

rmpf :: Traced (Weighted (ListT IO)) a → ...

pmh :: Weighted (ListT (Traced IO)) a → ...

(a) MonadBayes

rmpf = handleResample$_{rmpf}$ ∘ pfilter stepModel$_{rmpf}$ **where**
    stepModel$_{rmpf}$ = reuseTrace ∘ advance

pmh = handlePropose$_{im}$ ∘ mh execModel$_{pmh}$ **where**
    execModel$_{pmh}$ = handleResample$_{mul}$ ∘ pfilter stepModel$_{pmh}$

(b) Our library

Figure 16. Support for inference as modular building blocks

determines the specific sampling and conditioning effects that occur at run-time, and the order in which they interact.

Algebraic effects are similar in a way: the programmer also selects an ordering of abstract operations when instantiating the effect signature es in Comp es a. However, the operations' semantics are not determined by the effect types themselves, but are given separately by effect handlers. For instance, the algorithm rmpf in Fig. 16b is implemented by choosing a composition of handlers stepModel$_{rmpf}$ for executing the model, plus a handler handleResample$_{rmpf}$ for the inference effect. Here, constructing inference algorithms out of reusable parts is expressed mainly at the value level, via effect handler composition.

While each of the concrete monads in MonadBayes is by itself intuitive, for sophisticated algorithms like rmpf the transformer stacks can become unwieldy. To extend an algorithm with a new monad, perhaps with its own type class operations, requires each existing monad in the stack to provide a corresponding instance, and the new monad in turn to implement each supported operation in the stack. Thus programmability comes with a certain cost in terms of the amount of boilerplate required. With algebraic effects, support for new semantics is often more lightweight, requiring only a new handler to define the relevant operations. For example by swapping out the Resample handler in multinomial particle filter (§ 4.1), we were able to derive several other variants not discussed in the paper such as *residual* and *systematic* particle filter [Doucet et al. 2009], and also compose these parts to form other algorithms like Resample-Move Particle Metropolis-Hastings [Chopin et al. 2013].

7 Conclusion and Future Work

Typed functional languages like Haskell offer a type-safe and compositional foundation for inference programming. However, the intersection of these paradigms can involve a steep learning curve for individuals not already well versed in both. This paper presented a technique based on algebraic effects and operations for explicating the core structure of inference algorithms, and effect handlers as an intuitive and modular interface for programming them. We used this technique to implement some off-the-shelf algorithms in a modular way.

One area of future work is to explore the existing Haskell support for automatic differentiation [Kmett et al. 2021; van den Berg et al. 2022] and its interplay with effect handlers, which would enable modern inference techniques like HMC [Chen et al. 2014] and variational autoencoders [Kingma and Welling 2013] that require differentiable models. Another is to formalise some properties of our library. For example, MonadBayes has modular proofs that ensure each of its monad transformers correctly produces an "unbiased sampler" for inference [Ścibior et al. 2017]; it may be possible to transfer the semantics of monad transformers to an algebraic effect setting, perhaps using work by Schrijvers et al. [2019] that specifies when one is expressable in terms of the other. Finally, we are interested in how effect handlers compare to "unembedding" [Matsuda et al. 2023] as a technique for embedding abstract probabilistic programs; this may allow regular Haskell-bound variables to be assigned the various non-standard semantics that come with probabilistic languages, e.g. of random variables, or optimisable variables that make use of differentiation.

Acknowledgements

References

Nathanael L Ackerman, Cameron E Freer, and Daniel M Roy. 2011. Noncomputable conditional distributions. In *2011 IEEE 26th Annual Symposium on Logic in Computer Science*. IEEE, 107–116. https://doi.org/10.1109/LICS.2011.49

Christophe Andrieu, Nando De Freitas, Arnaud Doucet, and Michael I Jordan. 2003. An introduction to MCMC for machine learning. *Machine learning* 50, 1 (2003), 5–43. https://doi.org/10.1023/A:1020281327116

Andrej Bauer and Matija Pretnar. 2015. Programming with algebraic effects and handlers. *Journal of logical and algebraic methods in programming* 84, 1 (2015), 108–123. https://doi.org/10.1016/j.jlamp.2014.02.001

Isabel Beichl and Francis Sullivan. 2000. The metropolis algorithm. *Computing in Science & Engineering* 2, 1 (2000), 65–69. https://doi.org/10.1109/5992.814660

Eli Bingham, Jonathan P Chen, Martin Jankowiak, Fritz Obermeyer, Neeraj Pradhan, Theofanis Karaletsos, Rohit Singh, Paul Szerlip, Paul Horsfall, and Noah D Goodman. 2019. Pyro: Deep universal probabilistic programming. *The Journal of Machine Learning Research* 20, 1 (2019), 973–978. https://doi.org/10.5555/3322706.3322734

Tianqi Chen, Emily Fox, and Carlos Guestrin. 2014. Stochastic gradient hamiltonian monte carlo. In *International conference on machine learning*. PMLR, 1683–1691. https://doi.org/10.48550/arXiv.1402.4102

Nicolas Chopin, Pierre E Jacob, and Omiros Papaspiliopoulos. 2013. SMC2: an efficient algorithm for sequential analysis of state space models. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)* 75, 3 (2013), 397–426. https://doi.org/10.1111/j.1467-9868.2012.01046.x

Marco F Cusumano-Towner, Feras A Saad, Alexander K Lew, and Vikash K Mansinghka. 2019. Gen: a general-purpose probabilistic programming system with programmable inference. In *Proceedings of the 40th acm sigplan conference on programming language design and implementation*. 221–236. https://doi.org/10.1145/3314221.3314642

Johan Dahlin, Fredrik Lindsten, and Thomas B Schön. 2015. Particle Metropolis–Hastings using gradient and Hessian information. *Statistics and computing* 25, 1 (2015), 81–92. https://doi.org/10.1007/s11222-014-9510-0

John Darlington, Yi-ke Guo, Hing Wing To, and Jin Yang. 1995. Parallel Skeletons for Structured Composition. *SIGPLAN Not.* 30, 8 (aug 1995), 19–28. https://doi.org/10.1145/209937.209940

P.M. Djuric, J.H. Kotecha, Jianqui Zhang, Yufei Huang, T. Ghirmai, M.F. Bugallo, and J. Miguez. 2003. Particle filtering. *IEEE Signal Processing Magazine* 20, 5 (2003), 19–38. https://doi.org/10.1109/MSP.2003.1236770

Arnaud Doucet, Adam M Johansen, et al. 2009. A tutorial on particle filtering and smoothing: Fifteen years later. *Handbook of nonlinear filtering* 12, 656-704 (2009), 3.

Charles W Fox and Stephen J Roberts. 2012. A tutorial on variational Bayesian inference. *Artificial intelligence review* 38, 2 (2012), 85–95. https://doi.org/10.1007/s10462-011-9236-8

Hong Ge, Kai Xu, and Zoubin Ghahramani. 2018. Turing: a language for flexible probabilistic inference. In *International Conference on Artificial Intelligence and Statistics, AISTATS 2018, 9-11 April 2018, Playa Blanca, Lanzarote, Canary Islands, Spain.* 1682–1690. https://doi.org/10.17863/CAM.42246

Walter R Gilks and Carlo Berzuini. 2001. Following a moving target—Monte Carlo inference for dynamic Bayesian models. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)* 63, 1 (2001), 127–146. https://doi.org/10.1111/1467-9868.00280

Noah D Goodman. 2014. The Design and Implementation of Probabilistic Programming Languages. http://dippl.org.

Daniel Hillerström and Sam Lindley. 2018. Shallow effect handlers. In *Asian Symposium on Programming Languages and Systems.* Springer, 415–435. https://doi.org/10.1007/978-3-030-02768-1_22

Jeroen D Hol, Thomas B Schon, and Fredrik Gustafsson. 2006. On resampling algorithms for particle filters. In *2006 IEEE nonlinear statistical signal processing workshop.* IEEE, 79–82. https://doi.org/10.1109/NSSPW.2006.4378824

Diederik P Kingma and Max Welling. 2013. Auto-encoding variational bayes. *arXiv preprint arXiv:1312.6114* (2013). https://doi.org/10.48550/arXiv.1312.6114

Oleg Kiselyov and Hiromi Ishii. 2015. Freer Monads, More Extensible Effects. In *Proceedings of the 2015 ACM SIGPLAN Symposium on Haskell* (Vancouver, BC, Canada) *(Haskell '15).* Association for Computing Machinery, New York, NY, USA, 94–105. https://doi.org/10.1145/2804302.2804319

Oleg Kiselyov, Amr Sabry, and Cameron Swords. 2013. Extensible effects: an alternative to monad transformers. *ACM SIGPLAN Notices* 48, 12 (2013), 59–70. https://doi.org/10.1145/2578854.2503791

Edward Kmett, Barak Pearlmutter, and Jeffrey Mark Siskind. 2010-2021. ad: Automatic Differentiation. Haskell package at https://hackage.haskell.org/package/ad.

Sheng Liang, Paul Hudak, and Mark Jones. 1995. Monad Transformers and Modular Interpreters. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Francisco, California, USA) *(POPL '95).* Association for Computing Machinery, New York, NY, USA, 333–343. https://doi.org/10.1145/199448.199528

Vikash Mansinghka, Daniel Selsam, and Yura Perov. 2014. Venture: a higher-order probabilistic programming platform with programmable inference. *arXiv preprint* (2014). https://doi.org/10.48550/arXiv.1404.0099

Vikash K. Mansinghka, Ulrich Schaechtle, Shivam Handa, Alexey Radul, Yutian Chen, and Martin Rinard. 2018. Probabilistic Programming with Programmable Inference. *SIGPLAN Not.* 53, 4 (jun 2018), 603–616. https://doi.org/10.1145/3296979.3192409

Kazutaka Matsuda, Samantha Frolich, Meng Wang, and Nicolas Wu. 2023. Embedding by Unembedding. *Proceedings of the ACM on Programming Languages* 7, ICFP, Article 189 (aug 2023). https://doi.org/10.1145/3607830

Dave Moore and Maria I Gorinova. 2018. Effect handling for composable program transformations in Edward2. *arXiv preprint arXiv:1811.06150* (2018). https://doi.org/10.48550/arXiv.1811.06150

Minh Nguyen, Roly Perera, Meng Wang, and Nicolas Wu. 2022. Modular Probabilistic Models via Algebraic Effects. *Proceedings of the ACM on Programming Languages* 6, ICFP, Article 104 (aug 2022). https://doi.org/10.1145/3547635

Tom Schrijvers, Maciej Piróg, Nicolas Wu, and Mauro Jaskelioff. 2019. Monad transformers and modular algebraic effects: what binds them together. In *Proceedings of the 12th ACM SIGPLAN International Symposium on Haskell.* 98–113. https://doi.org/10.1145/3331545.3342595

Adam Ścibior and Ohad Kammar. 2015. Effects in Bayesian inference. In *Workshop on Higher-Order Programming with Effects (HOPE).* https://www.cs.ubc.ca/~ascibior/assets/pdf/hope.pdf

Adam Ścibior, Ohad Kammar, and Zoubin Ghahramani. 2018. Functional Programming for Modular Bayesian Inference. *Proceedings of the ACM on Programming Languages* 2, ICFP, Article 83 (2018), 29 pages. https://doi.org/10.1145/3236778

Adam Ścibior, Ohad Kammar, Matthijs Vákár, Sam Staton, Hongseok Yang, Yufei Cai, Klaus Ostermann, Sean K. Moss, Chris Heunen, and Zoubin Ghahramani. 2017. Denotational Validation of Higher-Order Bayesian Inference. *Proc. ACM Program. Lang.* 2, POPL, Article 60 (dec 2017), 29 pages. https://doi.org/10.1145/3158148

David Tolpin, Jan-Willem van de Meent, Hongseok Yang, and Frank Wood. 2016. Design and implementation of probabilistic programming language anglican. In *Proceedings of the 28th Symposium on the Implementation and Application of Functional programming Languages.* 1–12. https://doi.org/10.1145/3064899.3064910

Birthe van den Berg, Tom Schrijvers, James McKinna, and Alexander Vandenbroucke. 2022. Forward-or Reverse-Mode Automatic Differentiation: What's the Difference? *Available at SSRN 4358090* (2022). https://doi.org/10.48550/arXiv.2212.11088

David Wingate, Andreas Stuhlmüller, and Noah Goodman. 2011. Lightweight implementations of probabilistic programming languages via transformational compilation. In *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics.* JMLR Workshop and Conference Proceedings, 770–778.