Effects and Effect Handlers for Probabilistic Programming

Minh Nguyen

A thesis submitted to the University of Bristol in accordance with the requirements for award of the degree of PhD in the Faculty of Engineering.

School of Computer Science
September 26, 2023

Word count: 30657

Abstract

Probabilistic programming languages allow programmers to construct statistical models, representing random variables they know and those they wish to learn. Using the same language, the programmer can then simulate data from the model, or apply an inference algorithm to learn the relationships between the model's variables. Although used widely, existing probabilistic languages do not fully support modular and type-safe programming, which has specific impacts on end-users. When modelling, models are either not readily composable, or are restricted to a specific instance of simulation or inference, thus limiting their reusability. Most inference frameworks are then designed without a disciplined approach to side-effects, which can result in monolithic implementations where the structure of the inference algorithms is obscured and programming (customising) them is hard.

This thesis describes a novel approach for designing modular and type-safe probabilistic programming languages, based on *algebraic effects and effect handlers* – a typed functional programming technique for structuring effects. The approach is demonstrated in Haskell as a host language. Part I develops a language for probabilistic models that are modular, first-class, and reusable for both simulation and inference; it shows how these features enable new highly expressive treatment of models, such as composition and higher-orderness. Part II then develops a framework for inference programming that is modular and type-driven, where specific algorithms can be modularly derived from abstract classes of inference algorithms; it illustrates how the approach reveals the algorithms' high-level structure, and makes it possible to tailor and recombine their parts into new variants.

Acknowledgements

To my supervisors, Meng Wang and Roly Perera, I could not have asked for a better pair of mentors. Meng, I would be near lost without your guidance; thank you for driving me to be the independent researcher I am today, for pushing me beyond my comfort zone, and for never failing to believe in me when I have often struggled to do so myself. There are few better feelings in the world than knowing that you always have my back, and that you are there to lead me out of my own chaos. Roly, thank you for being my role model in how I care about and conduct research – for keeping me in love with my job when it has been hard, for being my voice of reason, and for making me strive to be proud of the work I do. No one has had a bigger impact on how I work, and it is nothing less than an absolute joy to be able to learn from you and do research with you.

To my master's supervisor Nicolas Wu, the human catalyst for ideas, your love for functional programming is contagious and I am so glad you are the one I caught it from. I am incredibly fond of all the work and time we have spent together, and I am envious of anyone who gets to feed off of your excitement. There are a number of people I am indebted to for investing their time in me and expecting nothing in return: Ohad Kammar for patiently teaching me and looking after me during my internship at Edinburgh; Oleg Kiselyov for giving me an important leg-up on my first paper; and James Geddes and Radka Jersakova at the Alan Turing Institute for always welcoming me and sparking lots of fun discussion.

I am so blessed to be part of and grow up with the family that is the Bristol Programming Languages research group; my next-door neighbour Eddie, my PhD sister Sam, Joe, Hanliang, Jess, Charlie, Steven, Cristina, Alex, François, and others – thank you. The silly, wonderful, and inspiring environment that each of you bring to my everyday work life is invaluable.

To my best friends Mihajlo and Alessio, and my best brother Quang, I love you all beyond words.

Author's Declaration

I declare that the work in this thesis was carried out in accordance with the requirements of the University's Regulations and Code of Practice for Research Degree Programmes and that it has not been submitted for any other academic award. Except where indicated by specific reference in the text, the work is the candidate's own work. Work done in collaboration with, or with the assistance of, others, is indicated as such. Any views expressed in the thesis are those of the author.

Minh Nguyen

September 26, 2023

Contents

CHAPTER 1

Introduction

The Bayesian approach to data analysis offers a powerful way to make inferences about missing data, by using random (i.e. probabilistic) variables to represent various uncertain aspects of the problem. The approach can be idealised into two steps:

1. Modelling: Specifying a model as a *joint probability distribution* over all observable and unobservable variables in a problem.
2. Inference: Conditioning the model on some observed data, producing a *posterior probability distribution* over the unobservable variables.

In modern practice, applying these concepts to solve real-world problems means expressing them as programs, to be executed by machines. *Probabilistic programming languages (PPLs)* make this transition from paper to program more ergonomic. The philosophy of probabilistic programming is to decouple modelling from inference, providing users with bespoke programming language constructs for specifying models, and a framework for automating inference over them.

The principles of modular and structured programming in the design of PPLs have a direct impact on end-users, determining how economical it is to build and maintain complex models, and how straightforward it is to extend inference algorithms in a reliable fashion. But existing PPLs generally struggle in these aspects in one way or another, as introduced in the next two sections. This thesis then proposes that both areas can be improved, substantially, using the same typed functional programming abstraction of *algebraic effects and effect handlers*.

1.1 Bayesian modelling in practice

A probabilistic model captures a real-world phenomenon as a set of relationships between two kinds of random variables: *latent* variables, whose values we cannot directly observe, and *observable* variables, whose values we can. By integrating such notions into general-purpose languages, PPLs allow programmers to build and execute probabilistic models. For example, consider a simple linear regression model that assumes a linear relationship between input variable x and output variable y; this can be represented using the standard mathematical notation shown on the left below. Using the language presented in this thesis, the right-hand side shows how one could express the same model as a functional program in Haskell.

$$\begin{aligned} m &\sim \mathsf{Normal}(0,3) \\ c &\sim \mathsf{Normal}(0,2) \\ y &\sim \mathsf{Normal}(m * x + c,\ 1) \end{aligned}$$

```
linRegr  x = do
  m  ← normal 0 3 #m
  c  ← normal 0 2 #c
  y  ← normal (m ∗ x + c) 1 #y
  return  y
```

Both representations take a non-random input x and specify the distributions that generate the line's slope m and intercept c; the output y is then generated from the normal distribution using mean $m * x + c$ and standard deviation 1. Here the random variable y is *observable*, whereas the model parameters m and c that relate it to x are *latent*. In the program representation, each primitive distribution is associated with a corresponding "conditionable variable" indicated by the # syntax; this is an optional argument, and its purpose will become clear shortly.

Given a probabilistic model, the programmer or data scientist will typically want to use it in at least two different ways. *Simulation* involves providing fixed values for the latent parameters to generate the resulting model outputs. Conversely, *inference* generally entails providing observed values for the model outputs, in attempt to learn the latent parameters.

What then distinguishes a probabilistic language from a general-purpose one are two effectful operations: Sample, namely drawing a value from a probability distribution, and Observe, which is to incorporate an observation about external data by conditioning a distribution against it [Gordon et al. 2014]. Given these two notions, one should then be able to *simulate* outputs from a model, and *infer* the parameters of a model.

For example, we might simulate from linRegr in our language as follows:

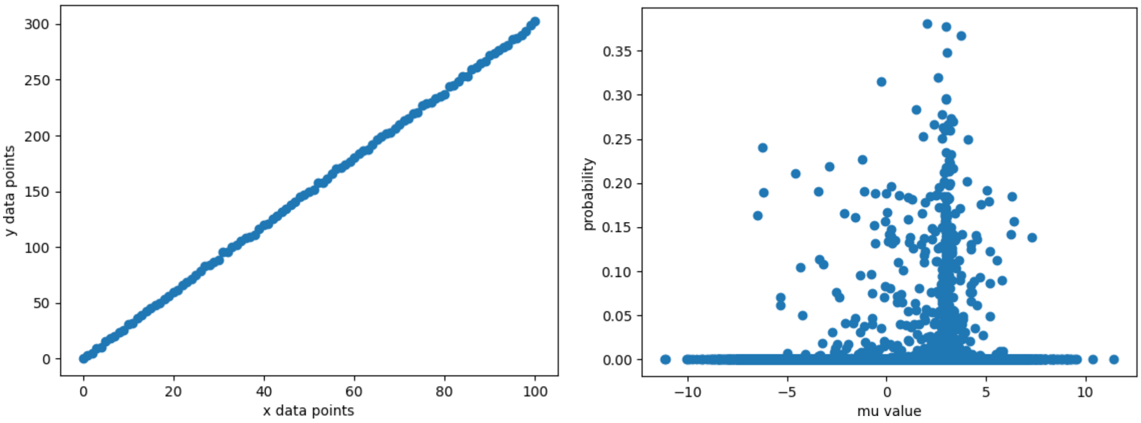
```
let  xs   = [ 0 .. 100 ]
     env  = (#m := [3]) • (#c := [0]) • (#y := []) • ENil
in   map (λx → simulateWith env (linRegr x)) xs
```

First we declare a list of model inputs xs from 0 to 100. Then we define a "model environment" env which assigns values 3 and 0 to conditionable variables #m and #c. This expresses our intention to *observe* parameters m and c – that is, to provide external data 3 as the value of m whilst conditioning on the likelihood that $m = 3$, and similarly for c. On the other hand no values are specified for #y in env, expressing our intent to *sample* the model output y – that is, to draw a value from its probability distribution. We then use our library function simulateWith to simulate a single output from the model for each data point in xs under the specified environment, producing the result visualised in Fig. 1.1a.

Alternatively, we can perform inference on linRegr, for example using the Likelihood Weighting algorithm [Meent et al. 2018], as follows:

```
let  xs    = [ 0 .. 100 ]
     envs  = [(#m := []) • (#c := []) • (#y := [3 ∗ x]) • ENil | x ← xs]
in   zipWith  (λx env → lwWith 200 env (linRegr x)) xs envs
```

Here we define a list of environments envs, which for each model input x, contains an environment that assigns the value $3 * x$ to #y but nothing to #m and #c. This expresses our intention to *observe* y but *sample* m and c. We then use library function lwWith to perform 200 iterations of Likelihood Weighting



(a) Simulation

(b) Inference: Likelihood Weighting

Figure 1.1: Visualising Linear Regression

for each pair of model input and environment, producing a trace of weighted parameters m and c whose distributions express the most likely parameter values to give rise to y. Fig. 1.1b visualises the likelihoods of samples for m, where values around $m = 3$ clearly accumulate higher probabilities.

We refer to a model that can be used for both simulation and inference — where random variables can be switched between Sample and Observe modes without altering the model itself — as a *multimodal model*. While multimodal models have a clear benefit, letting the same model be interacted with for a variety of applications, few existing PPLs support them. Most frameworks, such as MonadBayes [Ścibior, Kammar, and Ghahramani 2018] and Anglican [Tolpin et al. 2016], instead require programmers to express models in terms of explicit Sample and Observe operations, which considerably limits their reusability. If the user wishes to interact with the "same" model in a new way, they have little choice but to reimplement it with a different configuration of Sample and Observe operations.

Indeed, the number of possible model interpretations extends far beyond the two general scenarios of simulation and inference, potentially including *any* combination of Sample and Observe operations that can be instantiated for a model's random variables. Depending on available data and uncertainty about the model, it is common to explore the model's output space by providing only some of its latent parameters (and randomly sampling the rest) [Kline and Tamer 2016], or, to alternate between which variables are being conditioned on [Moon 1996]. Ideally, all of these possible scenarios would be expressible with a single multimodal model definition, avoiding the need to define and separately maintain a different version of the model for each use-case.

While some PPLs *do* support multimodal models, it is usually difficult or impossible to reuse existing models when creating new ones. Special-purpose PPLs like Stan [Carpenter et al. 2017] and WinBUGS [Lunn et al. 2000] provide a bespoke language construct for models with its own distinctive semantics, but as well as lacking high-level programming features beyond those essential to model specification, model definitions are unable to reuse other model definitions. PPLs like Turing [Ge et al. 2018] and Gen [Cusumano-Towner et al. 2019] instead extend general-purpose languages, implementing multimodal models as macros that are compiled into functions; although they provide some support for combining models, models are not first-class values, and so their ability to be composed or manipulated by host language features is limited. These modularity limitations are especially significant for hierarchical modelling, where the goal is to explicitly define a composite model with independently defined sub-models [Gelman and Hill 2006].

1.2 Bayesian inference in practice

Bayesian inference is a recipe for learning from data. Consider the linear regression model with latent variables m and c; given our *prior* beliefs $\mathbb{P}(m, c)$ about their values, Bayesian inference provides a way to calculate the *posterior* distribution, $\mathbb{P}(m, c \mid y; x)$, quantifying how our beliefs about the latent variables should change after some observations for y (given x).

However, the posterior of a model rarely has an analytic solution [Ackerman et al. 2011], and so PPLs rely on approximation methods to perform inference. Because no one method provides an optimal (or even useful) solution to every inference problem, there are many approaches, divided broadly into *Monte Carlo* methods like Likelihood Weighting above, which use random sampling to numerically approximate the posterior, and *parameter estimation* techniques, which construct analytic approximations. We refer the interested reader to Gelman, Carlin, et al. [1995] and Zhang et al. [2018] for more background.

Most inference 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. Each run of the model thus represents only an *approximant* of the ideal semantic interpretation of the model (as a system which generates samples from its posterior); moreover, a run may execute only partially, or be instrumented to produce additional information required by the algorithm, such as parameter gradients or traces recording stochastic choices for latent variables. So inference algorithms rely essentially on being able to execute models under custom semantics tailored to the specific needs of the algorithm.

Each algorithm also comes in many variants optimised for specific situations, and the task of implementing such variants falls not just to library designers. To achieve acceptable performance, model authors often need to be versed in the intricacies of inference as well. For example, the efficiency of Monto Carlo methods such as particle filtering is highly dependent on the choice of *proposal* distributions encoding knowledge of the inference problem, often provided by the user [Snyder et al. 2015]. This need for both library designers and end users to adapt existing solutions to specific settings has led to interest in language and implementation techniques that make it easy to assemble new inference algorithms out of reusable parts.

Customising and repurposing inference algorithms in this way is sometimes called *programmable inference* [Mansinghka, Selsam, et al. 2014]. These techniques have been explored with some success in systems like Venture [Mansinghka, Schaechtle, et al. 2018], Pyro [Bingham et al. 2019], and Gen [Cusumano-Towner et al. 2019], but without using a disciplined approach to side-effects. Inference algorithms, being complex imperative programs, can then result having monolithic implementations, where the structure of the algorithms is obscured and inference programming is hard.

Perhaps counter-intuitively, the heavy reliance of inference on side-effects makes programmable inference an ideal target for typed functional programming. Modern functional languages provide powerful compositional type disciplines for managing effects; the key example to date is the *monad transformer* approach [S. Liang et al. 1995] which was put to work in MonadBayes [Ścibior, Kammar, and Ghahramani 2018], demonstrating the approach's compositionality on a variety of common inference algorithms. However, monad transformers can be rather non-trivial to work with, and for many statisticians further complicate the task of inference programming.

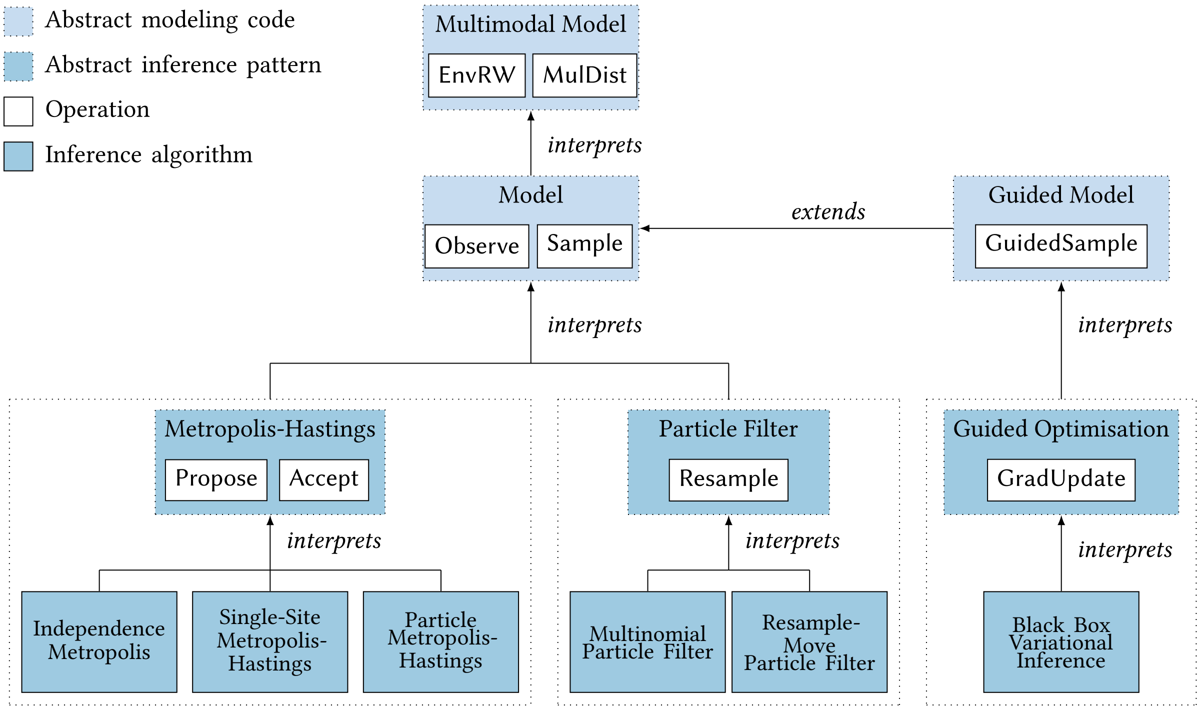


Figure 1.2: Overall architecture of the PPL developed in this thesis

1.3 Thesis outline and contributions

This thesis describes a novel approach for implementing PPLs based on *algebraic effects and effect handlers* — a typed functional programming technique for structuring effects — to address the challenges of typed, modular modelling and inference set out above. The approach is used to develop a language in Haskell called *ProbFX*; its implementation is visualised in Fig. 1.2 and freely available online. [1]

This work is presented in two main parts, focusing on the challenges of modelling from Section 1.1, and of inference from Section 1.2. We discuss any evaluation and related work at the end of each chapter, rather than at the end of the thesis. The following is a summary of the chapters:

Language Overview and Background

- Chapter 2 gives an overview of our language. We explain how the task of Bayesian data analysis is carried out in PPLs, and use this to motivate our approach towards modelling and inference. We then introduce the technique of algebraic effects that our approach is based on.

Part I: Modelling

- Chapter 3 describes a new representation of probabilistic models based on algebraic effects, and a modular type-based mechanism called *model environments* for conditioning models against observed data. Using these techniques, we develop the first PPL to support models that are both first-class and multimodal (and type-safe). This design is visualised in the top half of Fig. 1.2.

  We demonstrate how our approach enables highly modular treatment of models via a realistic case

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

study with real-world applications: the spread of disease during an epidemic. We then empirically evaluate our supported modelling features against a range of modern PPLs.

This chapter is based on Minh Nguyen, Roly Perera, Meng Wang, and Nicolas Wu (2022). "Modular probabilistic models via algebraic effects". In: *Proceedings of the ACM on Programming Languages* 6.ICFP, pp. 381–410.

- Chapter 4 presents an idealised minimal calculus for a language supporting type-safe and first-class multimodal models. We formalise the key abstractions needed for multimodality as primitives in the calculus, and integrate them with a native type-and-effect system based on algebraic effects. We then present a small-step operational semantics for the language and some corresponding properties.

  This chapter contains unpublished work.

Part II: Inference

- Chapter 5 describes a new approach for programmable inference based on algebraic effects, called *inference patterns*, which are abstract classes of inference algorithms from which specific algorithms can be modularly derived. We consider three important classes of algorithms: Metropolis-Hastings, Particle Filtering, and Guided Optimisation, illustrating how the approach reveals the algorithms' high-level structure, and makes it possible to tailor and recombine their parts into new variants. This design is visualised in the bottom half of Fig. 1.2.

  We show that our approach is highly competitive, performance-wise, with state-of-the-art systems for programmable inference based on other techniques. We then evaluate our approach from a usability and modularity perspective, contrasting with untyped systems such as Gen and Pyro, and MonadBayes, the main existing system based on typed effects.

  This chapter is based on Minh Nguyen, Roly Perera, Meng Wang, and Steven Ramsay (2023). "Effect handlers for programmable inference". In: *Proceedings of the 16th ACM SIGPLAN International Symposium on Haskell*. Haskell '23. Seattle, WA, USA: Association for Computing Machinery, pp. 44–58. ISBN: 9798400702983.

Conclusion

- Chapter 6 summarises the main achievements in the thesis and discusses future work.

CHAPTER 2

Language Overview and Background

This chapter gives an overview of our language, ProbFX, from the end-user perspective. We explain how the task of Bayesian statistics is carried out using probabilistic programming, and use this to motivate our approach to modelling in Section 2.1 and inference in Section 2.2. Then, Section 2.3 introduces the idea of algebraic effects that our approach is based on, implementing the specific effect framework used in the rest of the thesis.

2.1 Probabilistic modelling

A probabilistic model describes how a set of random variables are distributed relative to some fixed input. If the model does not condition against any external data, the distribution it describes is the so-called *joint probability distribution*, giving the probabilities of all possible values that its random variables can assume. For example, the linear regression model from the introduction describes the distribution $\mathbb{P}(y, m, c; x)$ – the joint distribution over random variables y, m, and c, given fixed input x as a non-random parameter:

```
linRegr x = do
  m ← normal 0 3 #m              ⎫
  c ← normal 0 2 #c              ⎬ joint ℙ(y, m, c; x)
  y ← normal (m ∗ x + c) 1 #y    ⎭
  return y
```

In real-world applications, we typically have known values for only a subset of these random variables, and are interested in how the other variables are distributed with respect to those known values. Consider providing known data $\hat{y}$ for random variable y in linear regression; we say that we *condition* the model on the *observation* that $y = \hat{y}$. By using the well-known chain rule for two random variables:

$$\mathbb{P}(Y, X) \;=\; \mathbb{P}(Y \mid X) \cdot \mathbb{P}(X) \qquad \text{(chain rule)}$$

we can derive the resulting distribution as the product $\mathbb{P}(y = \hat{y} \mid m, c; x) \cdot \mathbb{P}(m, c)$. [1] The first component is called a *conditional distribution*, and describes the probability that $y = \hat{y}$ given different possible values for m and c; the second describes how those different possible values for m and c are distributed. Providing observed data to a probabilistic model can hence be seen as *specialising* its joint distribution to some product of conditional and associated distribution that is favourable to modelling.

[1] An equivalent derivation is $\mathbb{P}(m, c \mid y = \hat{y}; x) \cdot \mathbb{P}(y = \hat{y})$, due to the two variables in the chain rule being arbitrary.

2.1.1 Multimodal models

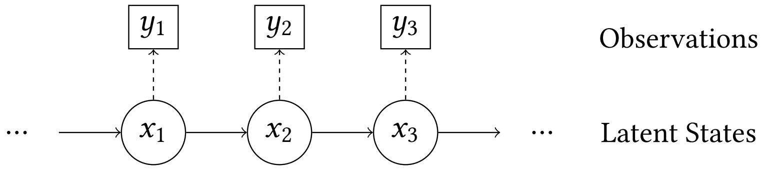
The chain rule is a powerful tool that allows us to describe a jointly occurring set of events in terms of the variables we will provide data for, and then compute other variables of interest with respect to this; and there are of course as many ways to decompose a joint distribution as there are combinations of variables that can be conditioned on. Since statisticians often have a clear understanding of the variables they wish to learn and those they wish to condition against, models are in practice often specialised to specific conditional distributions (through the chain rule) and expressed as low-level algorithms that explicitly perform sampling and conditioning, such as in Polson et al. [2013]; Ding et al. [2019].

Most PPLs, such as WebPPL [Goodman and Stuhlmüller 2014] and Anglican [Tolpin et al. 2016], are then designed to support the direct translation of these low-level model specifications from paper to program via the operations Sample and Observe. These languages are useful for creating model instances tailored to specific situations, but the resulting models are not easy to experiment with. Tasks which should be straightforward, such as exploring random variable behaviours by isolating which ones are sampled from [Idreos et al. 2015] or selectively optimising model parameters [Yekutieli 2012], require alternative specialisations to be created by hand.

With *multimodal* PPLs, the programmer specifies a single model which can be used to generate multiple specialisations, representing specific conditional distributions. Such languages require a mechanism for specifying observed data to random variables, determining whether they are to be sampled or observed. For example, Turing [Ge et al. 2018] lets users choose whether to provide observed values as arguments when invoking a model, with omitting an argument defaulting to sampling; in Pyro [Bingham et al. 2019], users specify mappings between random variables and observed data via *Python context managers* that later constrain the values of runtime sampling operations. However, these solutions are dynamically typed with no guarantee that the named variables exist or are provided values of the correct type.

2.1.1.1 Multimodal models via model environments

ProbFX supports multimodal models through a novel notion of *model environment*, which we explain in the context of a Hidden Markov Model (HMM) [Rabiner and Juang 1986]. The idea of a HMM is that we have a series of latent states x_i which are related in some way to observations y_i.



The HMM is then defined by two sub-models: a transition model (⟶) that determines how latent states x_i are transitioned between, and an observation model (⇡) that determines how x_i is projected to an observation y_i. The objective is to learn about x_i given y_i.

A simple HMM expressed in typical statistical pseudocode is shown in Fig. 2.1a; we describe its corresponding implemention in ProbFX in Fig. 2.1b:

```
hmm($n, x_0$)
  $\Delta_x \sim$ Uniform(0, 1)           } Model parameters
  $\Delta_y \sim$ Uniform(0, 1)
  for $i = 1...n$:
    $\delta x_i \sim$ Bernoulli($\Delta_x$)      } Transition model
    $x_i = x_{i-1} + \delta x_i$
    $y_i \sim$ Binomial($x_i, \Delta_y$)    } Observation model
  return $x_n$
```

(a) Statistical pseudocode

```
hmm :: (Conditionables env ["y_i"] Int
       , Conditionables env ["Δx", "Δy"] Double)
     ⇒ Int → Int → MulModel env es Int
hmm n x_0 = do
  Δx ← uniform 0 1 #Δx
  Δy ← uniform 0 1 #Δy
  let loop i x_{i-1} | i < n = do
                           δx_i ← bernoulli' Δx
                           let x_i = x_{i-1} + δx_i
                           y_i  ← binomial x_i Δy #y_i
                           loop (i + 1) x_i
                     | otherwise = return x_{i-1}
  loop 0 x_0
```

(b) ProbFX implementation

Figure 2.1: Hidden Markov Model

The type of hmm says it is a function that takes two Ints as input and returns a MulModel env es Int, where env is the model environment, es is the effects which the model can invoke (as in Section 2.3), and Int is the type of values the model generates. The constraint Conditionables states that #y_i :: Int, #Δ_x :: Double, and #Δ_y :: Double, are *conditionable variables* in the model environment which may be provided observed values.

The function hmm takes the HMM length n and initial latent state x_0 as inputs, and specifies the transition and observation parameters Δ_x and Δ_y to be distributed uniformly. It then iterates over the n nodes, applying the transition and observation models at each step. The transition model computes latent state x_i from state x_{i-1} by adding a value δx_i generated from a Bernoulli distribution bernoulli′ Δ_x. The observation model generates observation y_i from x_i via the distribution binomial x_i Δ_y. [2] The final latent state is returned at the end.

The hash syntax, for example in #y_i, constructs the unique inhabitant of the type-level string "y_i", and is how the programmer associates variables in the Conditionables env constraint with specific primitive distributions. They do this to indicate that they may later be interested in providing observed values for these variables to condition on. When they execute a model, they must provide a concrete environment of type env, and the presence or absence of observed values in that environment will determine whether the distribution tagged with #y_i is to be interpreted as Observe or Sample. The distribution bernoulli′ Δ_x has no conditionable variable, indicating that it is not possible to condition on δx_i; these primed variants of distributions are always interpreted as Sample, and are a convenience for when the programmer deems it unlikely for certain variables to ever be given observations.

A model can then be interpreted as any of its conditioned forms by specifying an appropriate model environment. In our example, the HMM in its unspecialised form represents the joint distribution over its latent states x_i, observations y_i, and parameters Δ_x, Δ_y (given fixed input x_0 as the first latent state):

$$\mathbb{P}(x_1 ... x_n,\ y_1 ... y_n,\ \Delta_x,\ \Delta_y;\ x_0)$$

We can then simulate the HMM (with length $n = 10$ and initial state $x_0 = 0$) by providing values for

[2] Binding to the local name y_i is technically redundant here since y_i is never used, but emphasises the connection to the $\sim$ notation used by statisticians.

$\#\Delta_x$ and $\#\Delta_y$ in an environment env, and calling the library function simulateWith:

```
let  $x_0$ = 0; $n$ = 10;
     env = ($\#\Delta_x$ := [0.5]) • ($\#\Delta_y$ := [0.8]) • ($\#y_i$ := []) • nil
in   simulateWith env (hmm $n$ $x_0$)
```

This indicates that we want to Observe 0.5 and 0.8 for Δ_x and Δ_y, and Sample for each occurrence of y_i because we provided no values for $\#y_i$. There are multiple occurrences of y_i at runtime, one for each $i \in \{1...n\}$, thanks to the iterative structure of the HMM. By the chain rule, the probability density expressed by instantiating the model with this environment is:

$$\mathbb{P}(\Delta_x = 0.5,\ \Delta_y = 0.8 \mid x_1...x_{10},\ y_1...y_{10};\ x_0 = 0) \cdot \mathbb{P}(x_1...x_{10},\ y_1...y_{10})$$

In the case of inference, on the other hand, we provide an observation for each y_i and try to learn Δ_x and Δ_x. This is why, as the reader may already have noticed, a model environment provides a *list* of values for each conditionable variable, allowing for the situation where the conditionable variable has multiple dynamic occurrences. In this case we provide 10 observations, one for each $i \in \{1...n\}$, before calling the library function lwWith for Likelihood Weighting inference:

```
let  $x_0$ = 0; $n$ = 10;
     env = ($\#\Delta_x$ := []) • ($\#\Delta_y$ := []) • ($\#y_i$ := [0, 1, 1, 3, 4, 5, 5, 5, 6, 5]) • nil
in   lwWith 100 env (hmm $n$ $x_0$)
```

At runtime, the values associated with $\#y_i$ in the model environment are used to condition against the occurrences of $\#y_i$ that arise during execution, in the order in which they arise. By the chain rule, the probability density expressed is:

$$\mathbb{P}(y_1 = 0 \ldots y_{10} = 5 \mid x_1...x_{10},\ \Delta_x,\ \Delta_y;\ x_0 = 0) \cdot \mathbb{P}(x_1...x_{10},\ \Delta_x,\ \Delta_y)$$

Although the type system ensures that model environments map conditionable variables to values of an appropriate type, it does not constrain the number of values that are provided. Should observed values run out for a particular variable, any remaining runtime occurrences of the variable will default to Sample; conversely, any surplus of values is ignored. Alternative designs are discussed in Section 6.1.

2.1.1.2 First-class models

Fig. 2.1a used a single procedure, written in statistical pseudocode, to express a Hidden Markov Model. Such notations are understood by most mathematicians and are widely used in statistical journals. Even when the model is complex, a monolithic style of presentation prevails, where models are defined from scratch each time rather than built out of reusable components. The design of PPLs such as Stan [Carpenter et al. 2017], PyMC3 [Salvatier et al. 2016] and Bugs [Lunn et al. 2000] reflect these non-modular conventions.

Programmers, on the other hand, recognise the importance of modularity to maintainability and reusability: they expect to be able to decompose models into meaningful parts. Fig. 2.2a shows how the programmer may imagine the same HMM as a composition of parts; because ProbFX embeds models into a functional language, it can then support this treatment of models as first-class functions in Fig. 2.2b.

In Fig. 2.2b we define the transition and observation models separately as transModel and obsModel. These are composed by hmmNode to define the behaviour of a single node, which is in turn used by hmm

```
transModel($\Delta_x, x_{i-1}$)
  $\delta x_i$ ~  Bernoulli($\Delta_x$)
  return  $x_{i-1} + \delta x_i$

obsModel($\Delta_y, x_i$)
  $y_i$  ~  Binomial($x_i$, $\Delta_y$)
  return  $y_i$

hmmNode($\Delta_x, \Delta_y, x_{i-1}$)
  $x_i$  ~  transModel($\Delta_x, x_{i-1}$)
  obsModel($\Delta_y, x_i$)
  return  $x_i$

hmm($n, x_0$)
  $\Delta_x$ ~  Uniform(0,  1)
  $\Delta_y$ ~  Uniform(0,  1)
  for  $i$ = 1...$n$;
    $x_i$ ~  hmmNode($\Delta_x, \Delta_y, x_{i-1}$)
  return  $x_n$
```

(a) Statistical pseudocode

```
transModel  ::  Double ⟶ Int ⟶ MulModel env es Int
transModel $\Delta_x$ $x_{i-1}$ = do
  $\delta x_i$ ⟵ bernoulli′ $\Delta_x$
  return  $x_{i-1} + \delta x_i$

obsModel  ::  (Conditionables  env  ["$y_i$"]  Int )
              ⇒ Double ⟶ Int ⟶ MulModel env es Int
obsModel $\Delta_y$ $x_i$ = do
  $y_i$ ⟵ binomial $x_i$ $\Delta_y$ #$y_i$
  return  $y_i$

hmmNode :: (Conditionables env  ["$y_i$"]  Int )
              ⇒ Double ⟶ Double ⟶ Int ⟶ MulModel env es Int
hmmNode $\Delta_x$ $\Delta_y$ $x_{i-1}$ = do
  $x_i$ ⟵ transModel $\Delta_x$ $x_{i-1}$
  obsModel $\Delta_y$ $x_i$
  return  $x_i$

hmm :: (Conditionables  env  ["$y_i$"]  Int
       , Conditionables  env  ["$\Delta_x$","$\Delta_y$"]  Double)
     ⇒ Int ⟶ Int ⟶ MulModel env es Int
hmm $n$ $x_0$ = do
  $\Delta_x$ ⟵ uniform 0 1 #$\Delta_x$
  $\Delta_y$ ⟵ uniform 0 1 #$\Delta_y$
  foldl  (>=>) return  (replicate  $n$  (hmmNode $\Delta_x$ $\Delta_y$)) $x_0$
```

(b) ProbFX implementation

Figure 2.2: A modular Hidden Markov Model

to create a chain of nodes of length n, using replicate and a fold of Kleisli composition (>=>) to propagate each node's output to the next one in the chain.

```
(>=>) ::  (a  ⟶ MulModel env es b) ⟶ (b ⟶ MulModel env es c) ⟶ (a ⟶ MulModel env es c)
```

The conditionable variables of hmm are now inherited from its sub-models: transModel has none and so lacks a Conditionables constraint, whereas obsModel declares #y_i as its only conditionable variable.

As well as improving reusability, compositionality also allows programmers to organise models around the structure of the problem domain. For example, the functions in Fig. 2.2b correspond in a straightforward way to the abstract components of a HMM: transModel makes it obvious that latent state x_i depends only on the previous state x_{i-1} (and Δ_x), and obsModel that observation y_i depends only on the state x_i that produced it (and Δ_y).

The implementation of multimodal models in ProbFX is developed in Part I, alongside a real-world case study demonstrating the approach's support for modular and reusable modelling.

2.2 Probabilistic inference

The start of Section 2.1 considered linear regression, linRegr, as a multimodal model of type MulModel, abstract in whether its random variables were sampled or observed. By providing observed data $y = \hat{y}$, we formulated an *inference problem* over that multimodal model, that is, a concrete probabilistic model that samples and observes; this would be akin to linRegr′ of type Model below, also written in ProbFX:

```
linRegr′ :: Double → Double → Model Double
linRegr′ x ŷ = do
  m  ← call (Sample (Normal 0 3))                  }  prior ℙ(m, c)
  c  ← call (Sample (Normal 0 2))                  }
  y  ← call (Observe (Normal (m ∗ x + c) 1) ŷ)     }  likelihood ℙ(y = ŷ | m, c; x)
  return y
```

The Sample operations represent our *prior* beliefs about latent variables m and c before accounting for any data. The operation Observe represents the *likelihood* of observable variable y indeed being $\hat{y}$, given y is normally distributed with mean $m * x + c$ and standard deviation 1.

Bayesian inference is then to try and solve this inference problem by using the Bayesian update rule, which revises the prior beliefs on the basis of observations to obtain a *posterior* distribution:

$$\underbrace{\mathbb{P}(X \mid Y)}_{\text{posterior}} = \frac{\overbrace{\mathbb{P}(Y \mid X)}^{\text{likelihood}} \cdot \overbrace{\mathbb{P}(X)}^{\text{prior}}}{\underbrace{\mathbb{P}(Y)}_{\text{evidence}}} \qquad \text{(Bayesian update rule)}$$

For linRegr′, this derives an equation for the posterior $\mathbb{P}(m, c \mid y = \hat{y}; x)$. 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* that forms the denominator, in this case $\mathbb{P}(y = \hat{y}; x)$, 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]. The general pattern is to interpret the model as a generative process, from which samples can be drawn, then iteratively constrain the model so that eventually the samples 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] are Monte Carlo methods that 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 $(m, c)$
    $\rho'$       ← exec (linRegr $x$ $\hat{y}$) $(m', c')$
    b        ← accept $\rho'$ $\rho$
    return (if b then $(m', c')$ else $(m, c)$)
```

Figure 2.3: Pseudocode: generic Metropolis-Hastings step on a linear regression model

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 the observation $\hat{y}$ for input x. 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 $\mathbb{P}(m, c \mid y = \hat{y}; x)$.

2.2.1 Inference patterns and pattern instances

We find it useful to think of Metropolis-Hastings, as sketched in Fig. 2.3, as an inference *pattern* rather than a specific 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 [3] [Djuric et al. 2003], for example, 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 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.

Achieving the kind of modular design suggested by this perspective, however, is not straightforward. The goal of "programmable inference" remains a challenging one. Interesting examples to date include Venture [Mansinghka, Schaechtle, et al. 2018], which uses metaprogramming techniques to enable inference programming in a Lisp-based language; MonadBayes [Ścibior, Kammar, and Ghahramani 2018], which uses monad transformers to implement a modular library for inference programming in Haskell; and Gen [Cusumano-Towner et al. 2019], a framework for inference programming in Julia which relies on a fixed black box interface for executing models generatively.

ProbFX offers programmable inference via the approach of *inference patterns* as described above, representing broad families of inference algorithms. The framework we present to the user follows the diagram in Fig. 1.2. The Model, provided by the user, expresses an inference problem in terms of abstract Sample and Observe operations. The inference pattern, provided by the library designer, defines a skeletal procedure for iteratively executing a Model under a specific semantics, similar to Fig. 2.3. These procedures in turn have their own abstract operations, like Propose and Accept. By assigning these operations a semantics, the user can then derive a concrete algorithm capable of generating samples from the model's posterior; we call these *pattern instances*.

The framework of inference patterns in ProbFX is developed in Part II, showing how the approach reveals the high-level structure of well-known algorithms, and makes it possible to derive new algorithm variants out of existing ones.

[3] also called Sequential Monte Carlo methods

2.3 Algebraic effects and effect handlers

Algebraic effects and effect handlers is a typed functional programming technique for managing the side-effects of a computation. In this setting, effects are modelled as coroutine-like interactions between: (i) side-effecting expressions that request (abstract) *operations* to be performed, and (ii) interpreters, called *effect handlers*, that assign meaning to those operations [Pretnar 2015]. An operation may also provide a continuation, allowing the handler to return control to the requesting expression. Algebraic effects thus lets us orthogonally specify the syntax and semantics of operations in programs, which will become fundamental in our approach to modular probabilistic programming.

The specific effect framework used in this thesis is based on Kiselyov and Ishii [2015]'s *extensible freer monad*: an embedding of a type-and-effect system into Haskell, exploiting Haskell's rich support for embedded languages. We explain this next, and discuss alternative representations in related work.

2.3.1 Implementing effectful computations

The *extensible freer monad* [Kiselyov and Ishii 2015] represents an effectful computation using the recursive datatype Comp es a at the top of Fig. 2.4. A term of type Comp es a is 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 b → Comp es a that takes the result of the operation and constructs the remainder of the computation. Values of type Comp es a thus represent uninterpreted *computation trees* comprised of pure values and operation calls chosen from es.

The type Comp es is a monad, allowing effectful code to piggyback on Haskell's **do** notation for sequential chaining of monadic computations (used by the linRegr′ model in Section 2.2). 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 (>=>) for composing monadic functions.

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. Its implementation is kept hidden; instead, the type class e ∈ es is used to assert that e is a member of es, and provides methods for safely injecting (inj) and projecting (prj) an operation of type e a into and out of EffectSum es a. Then, the helper function call makes it easy to call an operation (also used in linRegr′, Section 2.2). It injects the supplied operation of type e a into EffectSum es a, and then lifts the result to Comp es a supplied with the trivial leaf continuation Val.

2.3.2 Example: defining and using effects

Using this embedding, effects are represented by *type constructors* of kind Type → Type, whose type argument represents a return value; their operations are then *data constructors* which specify this type argument. For instance, the Reader s effect below is for reading a state of type s, and its single

Computations

```
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
  return :: a → Comp es a
  return x = Val x

  (>>=) :: Comp es a → (a → Comp es b) → Comp es b
  Val x     >>= f = f x
  Op op k >>= f = Op op (k >=> f)                 equivalently Op op (λx → k x >>= f)

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

Operations

```
data EffectSum (es :: [Type → Type]) (a :: Type) where ...

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

Figure 2.4: The Comp type for effectful computations, based on the *extensible freer monad*

operation Read of type Reader s s suggests that the operation returns the current state. The Printer effect also has one operation, Print, which takes a string argument and returns the unit value. The program readAndPrintTwice then uses the infrastructure from Fig. 2.4 to express a computation with these effects; it constrains the computation's effect signature with Reader Int ∈ es and Printer ∈ es, allowing the operation Read to be called for retrieving the integer state, and Print to be called with the string form of that integer.

```
data Reader s a where                 readAndPrintTwice :: (Reader Int ∈ es, Printer ∈ es) ⇒ Comp es ()
  Read :: Reader s s                  readAndPrintTwice = do
                                        i ← call Read
data Printer a where                    call (Print (show i))
  Print :: String → Printer ()          j ← call Read
                                        call (Print (show j))
```

2.3.3 Implementing effect handlers

Fig. 2.4 provided the machinery required to construct effectful computations; Fig. 2.5 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 thread a state of type s, whereas handle sets s to be the trivial unit type to be ignored.

```
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 () hval′ hop′
  where
  hval′ () x    = hval x
  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 2.5: The Handler type for effect handlers, and helpers for implementing handlers

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 can 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. [4]

Lastly, there are two special cases of handling below. The first is when the effect signature is empty; in this case, there are no operations left in the computation tree, and so runPure runs the pure computation to extract its return value. The second case, when the final effect is a monad m, is useful for executing probabilistic computations; here, runImpure produces an impure computation in the monadic context m, by simply extracting and sequencing the m operations.

```
runPure ∷ Comp [] a → a
runPure (Val x)   = x
runPure (Op op k) = error "impossible"
```

```
runImpure ∷ Monad m ⇒ Comp [m] a → m a
runImpure (Val x)   = return x
runImpure (Op op k) = fromJust (prj op) >>= (runImpure ∘ k)
```

[4] A handler that recurses like this is called a *deep* handler, and can be considered the default mechanism of effect handlers. The alternative is a *shallow* handler [Kammar et al. 2013; Hillerström and Lindley 2018].

2.3.4 Example: defining and using effect handlers

A key appeal of effect handlers is the ease at which new semantics can be defined for the same effect, and the semantics for each effect then composed to form new interpretations of entire programs. Some examples are given below for the effects Reader s and Printer from earlier.

For the Reader s effect, immutRead uses the handle helper to define an interpretation where the supplied state s_0 that is read from is immutable; its component hop interprets Read by returning s_0, and Val is used to simply return the computation's result upon termination. The handler incrRead instead uses handleWith to increment the integer state at each read request, and const Val (where const x y = x) to discard the final state and return only the computation's result.

```
immutRead :: s ⟶ Handler (Reader s) es a a
immutRead s0 = handle Val hop
  where
  hop Read k = k s0
```

```
incrRead :: Int ⟶ Handler (Reader Int) es a a
incrRead s0 = handleWith s0 (const Val) hop
  where
  hop s Read k = k (s + 1) s
```

For the Printer effect, purePrint provides a pure interpretation that appends the printed strings, returning the final string upon termination; impurePrint instead prints each string to the terminal by calling the IO.print function from Haskell's prelude, inserting this as an operation in the computation tree as long as IO is present in the effect signature.

```
purePrint :: Handler Printer es a (a, String)
purePrint = handleWith "" (λs x ⟶ Val (x, s)) hop
  where
  hop str (Print str') k = k (str ++ str') ()
```

```
impurePrint :: IO ∈ es ⟹ Handler Printer es a a
impurePrint = handle Val hop
  where
  hop (Print str') k = do call (IO.print str)
                          k ()
```

Composing these handlers to assemble full interpretations of programs can then be done in a variety of ways, taking the program readAndPrintTwice from earlier as an example:

```
ghci> (runPure ∘ purePrint ∘ immutRead 0) readAndPrintTwice        :: ((), String)
((), "00")

ghci> (runPure ∘ purePrint ∘ incrRead 0) readAndPrintTwice         :: ((), String)
((), "01")

ghci> (runImpure ∘ impurePrint ∘ immutRead 0) readAndPrintTwice    :: IO ()
0
0

ghci> (runImpure ∘ impurePrint ∘ incrRead 0) readAndPrintTwice     :: IO ()
0
1
```

2.4 Related work

The theory of algebraic effects and operations was pioneered by Plotkin and Power [2003] and effect handlers later added by Plotkin and Pretnar [2009], giving rise to a general framework for structuring effects in programs, and many specific implementations. We provide a brief overview of some variants.

Libraries for algebraic effects As described originally by Plotkin and Pretnar [2013], the signature of an effect's operations forms a free algebra, which in turn gives rise to a *free monad*; this provides a natural way for representing computation trees and the handlers that give semantics to those trees [Swierstra 2008]. Thus, many library implementations of effects are historically based on free monads, using languages with built-in monadic support. For example in Haskell, Kammar et al. [2013] develop an efficient library in Template Haskell that optimises the free monad to a continuation monad; Wu and Schrijvers [2015] parameterise the free monad by a coproduct of effects, and embed effect handlers as type classes; Kiselyov, Sabry, et al. [2013] combine the free monad with a type-class-only approach for extensible effect signatures, this later leading to the extensible freer monad [Kiselyov and Ishii 2015] that our implementation of Comp es a is based on. Other examples of host languages used include Idris [Brady 2021], whose dependent type system enables effects to be reasoned about in accordance to some user-provided specification [Brady 2013], and Purescript [Freeman 2017], which facilitates a row polymorphic approach to embedding effect signatures.

Languages for algebraic effects Programming languages designed with native support for algebraic effects need not use monads, instead treating effects and handlers as built-in primitives. Eff [Bauer and Pretnar 2015], which is an ML-style language, was the first full-fledged language built around algebraic effects, implementing its own effect subtyping system and providing first-class support for effect handlers and algebraic effects. Rather than featuring an explicit free monad construction, Eff distinguishes between the syntax of effectful computations and of pure values, and associates operations with a delimited continuation that lets them be handled. Frank [Lindley, McBride, et al. 2017], a strict functional language with a type-and-effect system, replaces functions entirely with handlers which may interpret effects by pattern matching on computation trees. Koka [Leijen 2017] and Links [Hillerström and Lindley 2016] implement an effect system based on row polymorphism, allowing effects to be extensibly modelled and managed at the type-level. To then implement handlers, which requires capturing the continuation of operations, Koka compiles programs into continuation-passing-style, and Links uses its support for manipulating first-class continuations.

Generalising the implementation in this thesis While the language developed in this thesis is implemented in Haskell, the key ideas of the overall approach can be captured in other host languages. In particular, any typed functional language that supports *polymorphic rows*, like PureScript [Freeman 2017] and OCaml [Leroy et al. 2020], or with a type system powerful enough to express something similar, like Haskell, should be capable of meeting the main requirements: (i) *polymorphic sums* for expressing *effect signatures*, and (ii) *polymorphic records* for expressing *model environments*. This was demonstrated using Idris2 [Brady 2021] as a host language instead. [5] Similarly, the choice of the *extensible freer monad* [Kiselyov and Ishii 2015] as a specific representation of algebraic effects is inessential, and any other embedding of algebraic effects should work. This was shown using *fused effects* [Wu and Schrijvers 2015] as an alternative effects framework in Haskell. [6]

[5] https://github.com/idris-bayes/prob-fx

[6] https://github.com/Functional-Labelling-Lab/fused-probfx

Part I

Effects and Effect Handlers for Probabilistic Modelling

CHAPTER 3

A Language for Multimodal Models

This chapter describes our approach for implementing the multimodal models of Section 2.1. Our insight is that algebraic effects, in combination with ideas from row polymorphism [Leijen 2005], offers a modular and type-safe solution for this. First, we can represent multimodal models as effectful computations that contain primitive distributions as abstract operations; this will let us defer interpreting each distribution as either Sample or Observe until we know which random variables we want to condition on. Second, we can represent a collection of random variables as a (polymorphic) record, associating the names of random variables with their observed values to be conditioned on; this is what we earlier called (Section 2.1) a *model environment*.

Our embedding of multimodal models is developed in Section 3.1. We then implement model environments in Section 3.2, and effect handlers for interpreting models under these environments in Section 3.3.

3.1 Embedding of multimodal models

We define a multimodal model (top, Fig. 3.1) to be an effectful computation of type Comp es a where es includes at least two specific effects: MulDist and EnvRW env. The MulDist effect allows the model to make use of primitive distributions, such as the normal and uniform distribution, in a multimodal fashion. The EnvRW env effect allows the model to read and update the values of conditionable variables in a model environment of type env. While these two effects suffice for model specification, others may be useful for model execution, and using the ∈ constraint lets the model remain polymorphic in any such additional effects.

3.1.1 Effect: multimodal distributions

The core computational effect of a multimodal model is the MulDist effect type (middle, Fig. 3.1), allowing models to be formulated in terms of primitive probability distributions that are multimodal. Each operation of MulDist would, in concept, correspond to a different primitive distribution.

For the purposes of extensibility, we instead define it with a single operation, MulDist, which can be used to represent many possible distributions. Its first argument of type d is constrained by the type class Dist d a, specifying that d is a primitive distribution that generates values of type a, with the functional dependency d → a indicating that d fully determines a. The second argument of type Maybe a is key to

Multimodal models

```
newtype MulModel env es a
  = MulModel { runMulModel :: (MulDist ∈ es, EnvRW env ∈ es) ⇒ Comp es a }

instance Monad (MulModel env es) where
  return x              = MulModel (return x)
  MulModel mx >>= f     = MulModel (mx >>= (λx → runMulModel (f x)))
```

Effect: multimodal distributions

```
data MulDist a where
  MulDist :: Dist d a ⇒ d → Maybe a → MulDist a

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

type LogP = Double
```

Effect: model environment reading/writing

```
data EnvRW env a where
  EnvRead  :: Conditionable env x a ⇒ CondVar x → EnvRW env (Maybe a)
  EnvWrite :: Conditionable env x a ⇒ CondVar x → a → EnvRW env ()
```

Figure 3.1: Multimodal models

supporting multimodality: it allows every primitive distribution to, at runtime, be interpretable as either Sample or Observe, depending on the availability of an observed value to condition on. This detail is hidden from the user, but is used internally to determine how operation calls are to be interpreted, as we discuss later (Section 3.3.2).

Instances of Dist d a must implement two functions: (i) draw, which takes a distribution d and random point r chosen 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. Log probabilities are used rather than probabilities as addition is generally more efficient to compute than multiplication, and the product of probabilities can be recovered by exponentiating the sum of their log probabilities.)

For example, below implements the Bernoulli distribution over Booleans, with probability p of generating True, and 1 - p for False:

```
data Bernoulli = Bernoulli { p :: Double }

instance Dist Bernoulli Bool where
  draw    (Bernoulli p) r = r ≤ p
  logProb (Bernoulli p) b = if b then log p else log (1 − p)
```

Figure 3.2: Bernoulli distribution + type class instance for Dist

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

3.1.2 Effect: model environment reading

In order to use model environments to condition on variables, we require models to support the effect EnvRW env (bottom, Fig. 3.1) for reading and updating observed values from environment env.

The types Conditionable env x a and CondVar x in the operations' type signatures are defined later (during model environments, Section 3.2), but we give an intuition for what they mean. The constraint Conditionable env x a says that env contains a conditionable variable x whose observed values are of type a. The datatype CondVar x lets the user represent x using the special # syntax, such as #*m* in Section 1.1.

The first operation EnvRead takes a conditionable variable that can be looked up in the model environment, and returns a Maybe-type value, indicating the presence or absence of an observed value to condition on. The second operation, EnvWrite, is used to write to an output environment that is *separate* from what EnvRead reads from; this output environment instead records all the sampled and observed values (of conditionable variables) that arise during model execution.

3.1.3 User-interface for writing multimodal models

The user is not expected to invoke the effects MulDist and EnvRW directly. Rather, for each primitive distribution, we provide a *smart constructor* [Swierstra 2008] that manages the requests to read from conditionable variables before calling the distribution. For example, consider the Bernoulli distribution from earlier:

```
data Bernoulli = Bernoulli { p :: Double }

bernoulli  :: Conditionable env x Bool ⇒ Double → CondVar x → MulModel env es Bool
bernoulli p x = MulModel (do maybev ← call (EnvRead x)
                             v      ← call (MulDist (Bernoulli p) maybev)
                             call (EnvWrite x v)
                             return v)

bernoulli′ :: Double → MulModel env es Bool
bernoulli′ p = MulModel (call (MulDist (Bernoulli p) Nothing))
```

Figure 3.3: Bernoulli distribution + smart constructors

The smart constructor bernoulli takes, in addition to the usual distribution parameters, a conditionable variable *x* :: CondVar x. The role of a smart constructor is to call an EnvRead operation for *x*, retrieving its possible observed value of type Maybe a from the input model environment, which is then used to call the MulDist operation; EnvWrite then writes the distribution's result to an output model environment.

```
coinFlip = do
  p ← uniform 0 1 #p
  y ← bernoulli p #y
  return y
```

(a) User code, with smart constructors

```
coinFlip = do
  maybep ← call (EnvRead #p)
  p      ← call (MulDist (Uniform 0 1) maybep)
  call (EnvWrite #p p)

  maybey ← call (EnvRead #y)
  y      ← call (MulDist (Bernoulli p) maybey)
  call (EnvWrite #y y)
  return y
```

(b) Without smart constructors

Figure 3.4: Desugaring of a multimodal model

Model environments

```
data Env (env :: [Assign Symbol Type]) where
  ENil   :: Env []
  ECons  :: [a] → Env env → Env ((x := a) : env)

data Assign x a = x := a

(•) :: Assign (CondVar x) [a] → Env env → Env ((x := a) : env)
(x := as) • env = ECons as env
```

Conditionable variables

```
data CondVar (x :: Symbol) where
  CondVar :: KnownSymbol x ⇒ CondVar x

instance (KnownSymbol x, x ∼ x′) ⇒ IsLabel x (CondVar x′) where ...
```

Constraining model environments

```
class Conditionable env x a
  get :: CondVar x → Env env → [a]
  set :: CondVar x → [a] → Env env → Env env

type family Conditionables env xs a :: Constraint where
  Conditionables env (x : xs) a = (Conditionable env x a, Conditionables env xs a)
  Conditionables env [] v = ()
```

Figure 3.5: Model environments and conditionable variables

A primed variant (bernoulli′) of each smart constructor is also provided that uses Nothing as the observed value, for the common case when a distribution does not need to be conditioned on.

To illustrate how smart constructors work, consider the simple model in Fig. 3.4 which generates a bias p from a uniform distribution and uses it to parameterise a Bernoulli distribution, determining whether the outcome of a coin flip y is more likely to be heads (True) or tails (False). Fig. 3.4a shows how the user might write the program (omitting the type signature); Fig. 3.4b shows the equivalent program written without smart constructors.

3.2 Model environments

Model environments are a type-safe interface for assigning observed values to the conditionable variables of a model. For the sake of compositionality, models should only need to mention the conditionable variables they make use of, and be polymorphic in the rest. Moreover, a given conditionable variable may bind *multiple* successive values at runtime – one for each time the variable is evaluated (as seen in Section 2.1.1.1). These two design constraints suggest a representation of model environments as extensible records, where the fields are conditionable variable names and the values are *lists* of observed values. We encode these ideas in Fig. 3.5.

Model environments are represented by the datatype Env env. Its type parameter env describes the type of the environment as a type-level list of pairs Assign x a, with the constructor (:=) associating type-level strings x of kind Symbol with value types a of kind Type; this tracks the variable names in the environment and their corresponding types. The constructor ENil is the empty environment, and the constructor ECons takes a list of values of type a and an environment of type env and prepends a new entry for x, producing an environment of type (x := a) : env.

The conditionable variable names in Env, being type-level strings of kind Symbol, have no value representation; to then use them as record fields at the value-level, we give the singleton datatype CondVar x, acting as a container for Symbols by storing them as a phantom parameter x. String values can be neatly promoted to such containers by deriving an instance of the IsLabel class from Haskell's OverloadedLabels language extension; this enables values of type CondVar to be created using the # syntax, so that for example, the value #foo has the type CondVar "foo".

Model environments are then constructed using the infix cons-like operator (•), which makes use of the (:=) constructor at the value-level. For instance:

```
example_env :: Env [("m" := Double), ("c" := Double), ("y" := a)]
example_env = (#m := [0.5]) • (#c := [3.0]) • (#y := []) • ENil
```

Finally, type-safe access (get) and updates (set) to conditionable variables is done via the type class Conditionable env x a, which asserts that the association x := a is present in env. Many conditionable variables of the same type can be specified with the type family Conditionables env xs a, which returns a nested tuple of constraints Conditionable env x a for each variable x in xs.

3.3 Semantics for multimodal models

We now turn to using effect handlers for specialising, or *conditioning*, a multimodal model according to a given model environment. This has two stages: (i) handling EnvRW env by reading observed values from a given environment (Section 3.3.1), and (ii) handling MulDist by interpreting primitive distributions in response to whether observed values have been provided (Section 3.3.2). We illustrate how these two handlers compose to produce a concrete model that samples and observes (Section 3.3.3), which can then be executed by a simulation or inference algorithm (Section 3.3.4).

3.3.1 Effect handler: reading from model environments

The handler handleEnvRW (top, Fig. 3.6) interprets EnvRW env by maintaining a pair of input and output environments (env_{in}, env_{out}), where env_{out} is initially the emptied input environment via initEmpty.

Upon encountering a request EnvRead x to read from x, the list of values associated with x is looked up in the input environment env_{in}. If the list is non-empty in (v:vs), the head value v becomes the current observation of x and is removed from env_{in}; this ensures that no observed value is conditioned on more than once during an execution, and that the order in which observations are consumed matches the execution order. Otherwise for the empty list [], there is no observation for x and env_{in} is unchanged.

For the request EnvWrite x v to write a value v to x, the current values for x in the output environment env_{out} are prepended by v. The final output environment is returned alongside the computation's result, discarding the input environment.

3.3.2 Effect handler: multimodal distributions

The handler handleMulDist (middle, Fig. 3.6) handles the MulDist effect by interpreting each multimodal distribution as either a sampling or observing operation, depending on the presence or absence of an

Effect handler: reading and writing for model environments

```
handleEnvRW :: Env env → Handler (EnvRW env) es a (a, Env env)
handleEnvRW env_in = handleWith (env_in, initEmpty env_in) hval hop where
  hval (_, env_out) x                  = Val (x, env_out)
  hop  (env_in, env_out) (EnvRead x)  k = case get x env_in of
                                            (v:vs)  → k (set x vs env_in, env_out) (Just v)
                                            []      → k (env_in, env_out) Nothing
  hop  (env_in, env_out) (EnvWrite x v) k = let vs = get x env_out
                                            in  k (env_in, set x (v:vs) env_out) ()

initEmpty :: Env env → Env env
initEmpty (ECons _ env) = ECons [] (initEmpty env)
```

Effect handler: interpreting multimodal distributions as sampling or observing

```
handleMulDist :: (Sample ∈ es, Observe ∈ es) ⇒ Handler MulDist es a a
handleMulDist = handle Val hop where
  hop (MulDist d maybey) k = case maybey of Just y  → call (Observe d y) >>= k
                                            Nothing → call (Sample d)    >>= k

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

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

Conditioning multimodal models to concrete models

```
conditionWith :: Env env → MulModel env [EnvRW env, MulDist, Observe, Sample, IO] a
              → Model (a, Env env)
conditionWith env_in = handleMulDist ∘ handleEnvRW env_in ∘ runMulModel

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

Figure 3.6: Specialising multimodal models to concrete models

observed value. This introduces two new effects, Sample and Observe, each with a single operation; Sample takes a distribution to sample from, whereas Observe takes a distribution and an observed value. Then handleMulDist will try to retrieve the observed value of each MulDist call; if there is such a value, we call a corresponding Observe operation, and otherwise we call Sample.

3.3.3 Conditioning multimodal models to concrete models

The composition of these two handlers, conditionWith (bottom, Fig. 3.6), will condition a multimodal model under a given model environment, producing a *concrete model* of type Model that can Sample and Observe. These two effects characterise the minimal interface assumed by most inference algorithms, and for simplicity here, we assume they are the *only* model effects required, along with IO for random number generation. [1] Hence conditionWith fixes the effect signature of MulModel, a detail which is kept hidden from the user when writing multimodal models.

To illustrate, consider applying conditionWith to the coinFlip model from earlier, written again

[1] Any monad for random number generation is fine. It is also possible to remove IO entirely from the signature, or keep the signature polymorphic, or extend (*weaken*) the signature with new effects after conditioning.

at the top of Fig. 3.7, and using the input environment (#p := [0.5]) • (#y := []) • ENil. First applying handleEnvRW would produce coinFlip′ as an intermediate program, in which all distributions calls have been parameterised by either a concrete observed value or Nothing, and their results written to an output environment. [2] Then applying handleMulDist would yield coinFlip″.

```
coinFlip  ::  ...  ⇒ MulModel ...
coinFlip  = do
  maybep  ← call (EnvRead #p)
  p       ← call (MulDist (Uniform 0 1) maybep)
  call (EnvWrite #p p)

  maybey  ← call (EnvRead #y)
  y       ← call (MulDist (Bernoulli p) maybey)
  call (EnvWrite #y y)
  return y

            ⇓  handleEnvRW ((#p := [0.5]) • (#y := []) • ENil)

coinFlip′ = do
  p ← call (MulDist (Uniform 0 1) (Just 0.5))
  y ← call (MulDist (Bernoulli p) Nothing)
  return (y, (#p := [p]) • (#y := [y]) • ENil)

            ⇓  handleMulDist

coinFlip″ ::  ...  ⇒ Model ...
coinFlip″ = do
  p ← call (Observe (Uniform 0 1) 0.5)
  y ← call (Sample (Bernoulli p))
  return (y, (#p := [p]) • (#y := [y]) • ENil)
```

Figure 3.7: Applying conditionWith to a multimodal model and model environment

3.3.4 Effect handlers for executing models

Finally, having specialised a multimodal model, we show how the resulting Model can be fully executed as a top-level program: by assigning a semantics to its Observe effect and to its Sample effect. These two steps are sufficient to implement our simple examples of Simulation and Likelihood Weighting inference below. Sophisticated inference tasks require more thought, and is the topic of Part II where we embark on programmable inference.

3.3.4.1 Simulation

The most basic interpretation of a model, as a generative process with no inference, is usually called *simulating* the model: it runs the provided model by using observed data when available and otherwise drawing new samples. We define this as the handler composition simulate in Fig. 3.8 (and the variant simulateWith for multimodal models). The handler defaultObserve performs no conditioning side-effects,

[2] The discarded input environment would be (#p := []) • (#y := []) • ENil, where #p is fully consumed and #y is unchanged.

and instead trivially interprets Observe d y operations to return the observed value y via the continuation k. 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. Then runImpure discharges the final effect IO, yielding a top-level Haskell program which can be executed.

Model execution

```
simulateWith :: Env env → MulModel env [EnvRW env, MulDist, Observe, Sample, IO] a
                  → IO (a, Env env)
simulateWith env_in = simulate ∘ conditionWith env_in

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

Auxiliary handlers

```
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)

random :: IO Double
```

Figure 3.8: Simulation

3.3.4.2 Likelihood Weighting inference

Given a model, and some observed data for a subset of its variables, *inference* tries to learn how the *other* variables are distributed with respect to those observations. The most vanilla form of inference is perhaps Likelihood Weighting (LW) [Meent et al. 2018]; the idea is that if one uses simulation as a process for randomly proposing new samples, then the LW algorithm assigns to each proposal a *weight*, that is, the total likelihood of them having generated the observed data. To implement this in Fig. 3.9, we interpret Observe using the likelihood handler, which sums the log likelihood w over all observations with 0 as the starting value. The function lw then repeats this for n iterations, generating a list of random proposals and their associated weights.

Model execution

```
lwWith :: Int → Env env → MulModel env [EnvRW env, MulDist, Observe, Sample, IO] a
         → IO [((a, Env env), LogP)]
lwWith n env_in = lw n ∘ conditionWith env_in

lw :: Int → Model a → IO [(a, LogP)]
lw n = replicateM n ∘ runImpure ∘ defaultSample ∘ likelihood
```

Auxiliary handlers

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

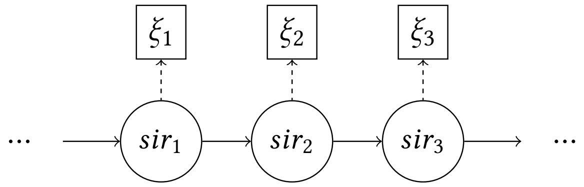
Figure 3.9: Likelihood Weighting inference

3.4 A case study in modular, multimodal models

This section demonstrates our support for modular, first-class, multimodal models, by implementing a realistic case study with real-world applications: a compartmental model for the spread of infectious diseases. Section 3.4.1 introduces our running example, the SIR (Susceptible-Infected-Recovered) model for the spread of disease [F. Liang and Li 2021]. Section 3.4.2 and Section 3.4.3 use the SIR model to show how our language supports higher-order models which can be easily extended and adapted. Section 3.4.4 shows how a multimodal model can be used for both simulation and inference in the same application to facilitate Bayesian bootstrapping.

3.4.1 The SIR model for epidemic modelling

The SIR model predicts the spread of disease in a fixed population of size n partitioned into three groups: s for *susceptible to infection*, i for *infected*, or r for *recovered*. The model tracks how s, i, and r vary over discrete time t measured in days. Because testing is both incomplete and unreliable, the true sir values for the population cannot be directly observed; however, we can observe the number of reported infections ξ. This problem is thus a good fit for the Hidden Markov Model introduced in Section 2.1.1, where the sir values play the role of latent states of type Population, and ξ as the observations of type Reported:



```
type Reported
  = Int

data Population
  = Population { s :: Int , i :: Int , r :: Int }
```

We now show how our language can be used to implement the SIR model as a modular HMM, starting with the transition (⟶) and observation (⇡) models.

3.4.1.1 SIR transition model

The transition model describes how the sir values change over a single day. We model two specific dynamics. First, susceptible individuals s transition to infected i at a rate determined by the values of s and i and the contact rate β between the two groups. We use a binomial distribution to model each person in s having a $1 - e^{-\beta i/n}$ probability of becoming infected, and update s and i accordingly:

```
trans_si :: Double → Population → MulModel env es Population
trans_si β (Population s i r) = do
  let n = s + i + r
  δ_si ← binomial′ s (1.0 − exp ((−β ∗ i) / n))
  return (Population (s − δsi) (i + δsi) r)
```

Second, infected individuals i transition to the recovered group r, where a fixed fraction γ of people will recover in a given day. Again we use a binomial to model each person in i as having a probability of $1 - e^{-\gamma}$ of recovering, and use this to update i and r:

```
trans_ir :: Double → Population → MulModel env es Population
trans_ir γ (Population s i r) = do
  δ_ir ← binomial′ i (1.0 − exp (−γ))
  return (Population s (i − δir) (r + δir))
```

The overall transition model trans_{sir} is simply the sequential composition of trans_{si} and trans_{ir}. Given β and γ, aggregated into the type TransParams, trans_{sir} computes the changes from s to i and then i to r to yield the updated *sir* population over a single day:

```
data TransParams = TransParams { β :: Double, γ :: Double }

trans_sir :: TransParams → Population → MulModel env es Population
trans_sir (TransParams β γ) = trans_si γ >==> trans_ir β
```

3.4.1.2 SIR observation model

For the observation model, we assume that the reported infections ξ depends only on the number of infected individuals i, of which a fixed fraction ρ will be reported. We use the Poisson distribution to model reports occurring with a mean rate of $\rho * i$:

```
type ObsParams = Double

obs_sir :: Conditionable env "ξ" Int ⇒ ObsParams → Population → MulModel env es Reported
obs_sir ρ (Population _ i _) = do
  ξ ← poisson (ρ ∗ i) #ξ
  return ξ
```

Since we intend this as the observation model, we declare conditionable variable #ξ with type Int in the Conditionable constraint, and attach it to the Poisson distribution so we can provide it observations to condition on later.

3.4.1.3 HMM for the SIR model

Now the transition and observation models can be combined into a HMM. We build on our modular HMM design from Fig. 2.2, but go a step further by defining it as a *higher-order model* in Fig. 3.10, parameterised by its two abstract sub-models of type TransModel and ObsModel; here, ps represents the types of the model parameters, and lat and obs are the types of latent states and observations. Then in hmm, the input models transPrior and obsPrior are first used to generate the model parameters θ and ϕ, which are provided to transModel and obsModel respectively.

```
type TransModel env es ps  lat       = ps → lat → MulModel env es lat
type ObsModel   env es ps  lat  obs  = ps → lat → MulModel env es obs

hmm :: MulModel env es ps_1         → MulModel env es ps_2
    → TransModel env es ps_1 lat → ObsModel env es ps_2 lat obs
    → Int → lat → MulModel env es lat
hmm transPrior obsPrior  transModel obsModel n x_0 = do
  θ ← transPrior
  ϕ ← obsPrior
  hmmNode x_{i−1} = do  x_i ← transModel θ x_{i−1}
                        y_i ← obsModel ϕ x_i
                        return  x_i
  foldl  (>=>) return  (replicate   n hmmNode) x_0
```

Figure 3.10: A higher-order Hidden Markov Model

We can now define the complete SIR model in Fig. 3.11. From an initial population *sir* of susceptible, infected, and recovered individuals, hmm$_{sir}$ models the change in *sir* over *t* days given reported infections ξ. Its transition and observation parameters are provided by models transPrior$_{sir}$ and obsPrior$_{sir}$ using primitive distributions gamma and beta; their conditionable variables #β, #γ, and #ρ will let us condition on those parameters later:

```
hmm_sir :: (Conditionable env "ξ" Int,  Conditionables  env  ["β",  "γ",  "ρ"]  Double)
        ⇒ Int → Population → MulModel env es Population
hmm_sir = hmm transPrior_sir obsPrior_sir trans_sir obs_sir

transPrior_sir :: Conditionables env  ["β",  "γ"]  Double       obsPrior_sir :: Conditionable env "ρ"  Double
               ⇒ MulModel env es TransParams                                 ⇒ MulModel env es ObsParams
transPrior_sir = do                                             obsPrior_sir = do
  β ← gamma 2 1 #β                                                ρ ← beta 2 7 #ρ
  γ ← gamma 1 (1/8) #γ                                            return  ρ
  return  (TransParams  β γ)
```

Figure 3.11: SIR model, using a higher-order Hidden Markov Model

We can simulate over this model, perhaps to explore some expected model behaviours, by specifying an input model environment sim_env$_{\text{in}}$ of type Env SIRenv that provides specific values for #β, #γ, and #ρ, but provides no values for reported infections #ξ (ensuring that we always sample for ξ). Applying simulateWith sim_env$_{\text{in}}$ to hmm$_{sir}$ 100 with an input population simulates the spread of the disease over 100 days.

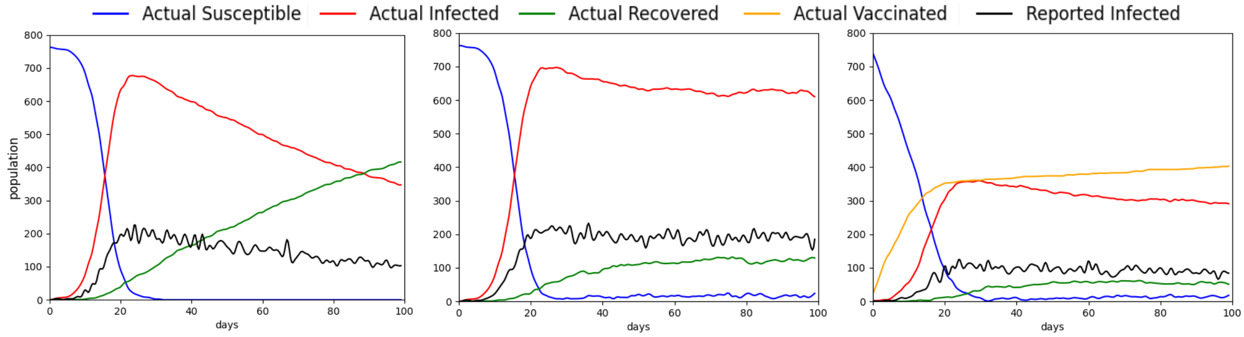
```
type SIRenv = ["β"  :=  Double, "γ"  :=  Double, "ρ"  :=  Double, "ξ"  :=  Int]

simulateSIR  ::  IO (Population,  Env SIRenv)
simulateSIR  = do
  let  sim_env_in = (#β  := [0.7]) • (#γ  := [0.009]) • (#ρ  := [0.3]) • (#ξ  := []) • ENil
  simulateWith sim_env_in (hmm_sir 100 (Population 762 1 0))
```

This returns the final population sir_{100} plus an *output* model environment env$_{\text{out}}$ mapping each conditionable variable to the values sampled for that variable during simulation. From this we can extract the reported infections ξs:

```
do (sir₁₀₀ :: Population, sim_env_out :: Env SIRenv) ⟵ simulateSIR
   let ξs :: [Reported] = get #ξ env_out
   ...
```

Fig. 3.12a shows a plot of these ξ values and their corresponding latent (population) states; but note that as it stands, the model provides no external access to the latent states shown in the plot, except the final one sir_{100}. Instrumenting the model with new effects, such as for recording the intermediate sir states, is straightforward with algebraic effects, shown in Section 3.4.3.



(a) SIR (b) with resusceptible (c) with resusceptible + vacc

Figure 3.12: SIR model: Simulation

3.4.2 Extending the SIR model with new behaviours

Although the SIR model is simplistic, realistic models may be uneconomical to run or too specific to be useful. When designing models, statisticians aim to strike a balance between complexity and precision, and modular models make it easier to incrementally explore this trade-off. We support this claim by showing how two possible extensions of the SIR model are made easy in our language; while these are by no means the most modular solutions possible, they should suffice to make our point.

Suppose our disease does not confer long-lasting immunity, so that recovered individuals r transition back to being susceptible s after a period of time [Shi et al. 2008]. We can model this *resusceptibility* as a new transition behaviour:

```
data TransParams = TransParams { β :: Double, γ :: Double, ρ :: Double, η :: Double }

trans_rs :: TransModel env es Double Population
trans_rs η (Population s i r) = do
  δrs ⟵ binomial′ r (1.0 − exp (−η))
  return (Population (s + δrs) i (r - δrs))

trans_sirs :: TransModel env es TransParams Population
trans_sirs (TransParams β γ η) = trans_si β >==> trans_ir γ >==> trans_rs η
```

We need only modify TransParams to include a new parameter η; define a new transition sub-model trans_{rs} (parameterised by η) that stochastically moves individuals from recovered r to susceptible s; and then define trans_{sirs} to compose trans_{rs} with our existing transition behaviours. A simulation of the resulting system is shown in Fig. 3.12b.

Now consider adding a variant where susceptible individuals s can become *vaccinated* v as a new sub-population [Ameen et al. 2020].

```
data Population   = Population { s :: Int , i :: Int , r :: Int , v :: Int }
data TransParams  = TransParams { β :: Double, γ :: Double, ρ :: Double, η :: Double, ω :: Double }

trans_sv :: TransModel env es Double Population
trans_sv ω (Population s i r v) = do
  δsv ← binomial′ s (1.0 − exp (−ω))
  return (Population (s - δsv) i r (v + δsv))

trans_sirsv :: TransModel env es TransParams Population
trans_sirsv (TransParams β γ η ω) = trans_si β >==> trans_ir γ >==> trans_rs η >==> trans_sv ω
```

We add field v to Population for vaccinated individuals, and add ω to TransParams representing the rate at which individuals s transition to v; the new behaviour is then expressed by trans_{sv}, composed into the overall transition model trans_{sirsv}. A simulation of this is shown in Fig. 3.12c.

3.4.3 Extending the SIR model with additional effects

When building a multimodal model, users are not restricted to using only the two base effects EnvRW and MulDist; the effect signature es in MulModel env es a can be easily extended with an arbitrary desired effect e, by (i) constraining the model with e ∈ es, and then (ii) handling e with a corresponding handler.

As an example, the SIR model simulation in Fig. 3.12a plotted *all* intermediate *sir* values of type Population over t days, despite our implementation only returning the final one:

```
−− previously
hmm_sir :: (Conditionable env "ξ" Int, Conditionables env ["β", "γ", "ρ"] Double)
        ⇒ Int → Population → MulModel env es Population
hmm_sir = hmm transPrior_sir obsPrior_sir trans_sir obs_sir
```

To record all *sir* values, we introduce the constraint Writer w ∈ es to require es contain the well-known effect Writer w, representing computations that produce a stream of data of type w; here we choose w to be a list of *sir* values [Population]. The transition model trans_{sir} can now call the Writer operation Tell to concatenate each new *sir* value to the existing trace of values.

```
data Writer w a where
  Tell  :: w → Writer w ()

trans_sir :: Writer [Population] ∈ es ⇒ TransModel env es TransParams Population
trans_sir (TransParams β γ) sir = do
  sir′ ← (trans_si β >==> trans_ir γ) sir
  call (Tell [sir′])
  return sir′
```

Models with user-specified effects can then be reduced into a form suitable for conditioning under a model environment (Section 3.3.3), by handling those effects beforehand with a suitable handler. Assuming hmm_{sir} now uses the new transition model above, composing hmm_{sir} with handleWriter will interpret the Tell operations, producing a new SIR model hmm'_{sir} that returns the trace of *sir* values as an additional output of type [Population]. [3]

[3] An omitted detail is the lifting of call and handleWriter from type Comp into MulModel; see Fig. A.1.

```
handleWriter  ::  Monoid w ⇒ Handler (Writer w) es a  (a,  w)
handleWriter  =  handleWith mempty (λw x ⟶ Val (x, w)) hop where
  hop w (Tell w') k = k (w `mappend` w') ()

hmm'_sir :: (Conditionable env "ξ" Int,  Conditionables  env  ["β",  "γ",  "ρ"]  Double)
          ⇒ Int ⟶ Population ⟶ MulModel env es (Population, [Population])
hmm'_sir t = handleWriter ∘ hmm_sir t
```

3.4.4 Exploring multimodality in the SIR model

We now show how our support for higher-order, modular models is complemented by the flexibility of multimodal models. Suppose the goal were to infer SIR model parameters β, γ, ρ given data on reported infections ξ. Ideally we would have a real dataset of reported infections to condition on. But what if our dataset were sparse, or if we were interested in quick hypothesis testing? A common option is to use simulated data as observed data, a method called Bayesian bootstrapping [Fushiki 2010]. This task is made simple with multimodal models, because we can take the outputs from simulation over a model and plug them into an environment that specifies inference over the same model:

```
inferSIR   ::  Env SIRenv ⟶ IO [(Population, Env SIRenv)]
inferSIR  sim_env_out = do
  let  ξs            =  get  #ξ  sim_env_out
       infer_env_in = (#β := []) • (#γ  := [0.0085]) • (#ρ  := []) • (#ξ  := ξs) • ENil
  ssmhWith 50000 infer_env_in (hmm_sir  100 (Population 762 1 0))
```

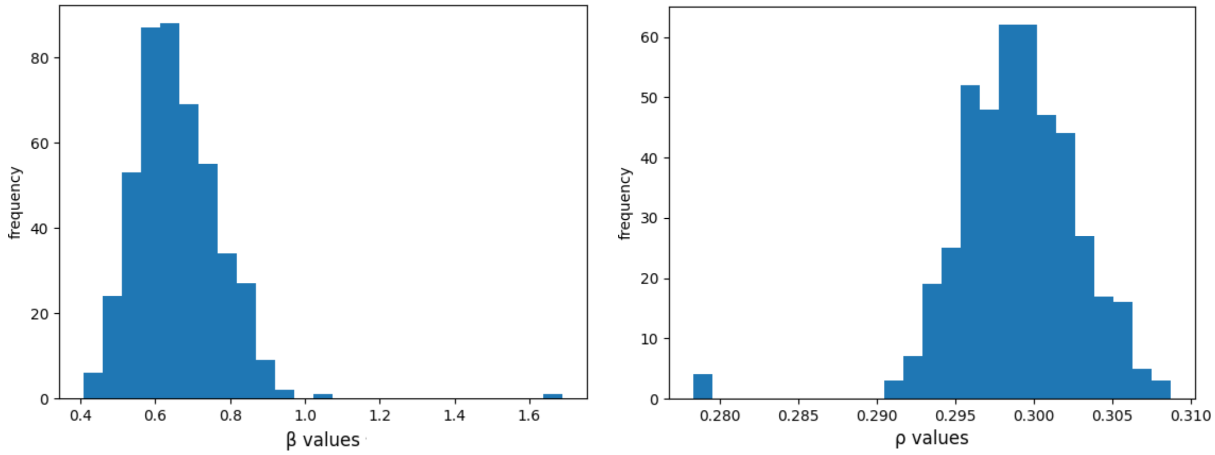
Here we take the output model environment sim_env$_{out}$ produced by simulateSIR, and use its ξ values to define an input model environment infer_env$_{in}$ that conditions against $\#\xi$. Moreover, suppose we already have some confidence about a particular model parameter, such as the recovery rate γ; for efficiency, we can avoid inference on γ by setting $\#\gamma$ to an estimate 0.0085 and sampling only for the remaining parameters $\#\beta$ and $\#\rho$. We then run Single-Site Metropolis-Hastings inference [Wingate, Stuhlmueller, et al. 2011] for 50,000 iterations, which returns a list of output environments containing the values sampled from all iterations. [4]

The inferred distributions for β and ρ can then be obtained simply by extracting their samples from those environments:

```
do (_,  infer_envs_out) ⟵ fmap unzip (inferSIR sim_env_out)
   let  βs = concatMap (get #β)  infer_envs_out
        ρs = concatMap (get #ρ)  infer_envs_out
```

These are visualised in Fig. 3.13. Values around $\beta = 0.7$ and $\rho = 0.3$ occur more frequently, because these were the parameter values provided in simulateSIR in Section 3.4.1.

[4] The core algorithm is implemented in Part II: Fig. 5.5, and is used to define ssmhWith in Fig. A.2.



(a) SIR β inferred distribution

(b) SIR ρ inferred distribution

Figure 3.13: SIR model: Single-Site Metropolis-Hastings inference

3.5 Qualitative evaluation and related work

This section empirically evaluates our modelling language in terms of supported model features, comparing with a wide range of modern PPLs in Table 3.1. To the best of our knowledge, ours is the first PPL to fully support both multimodal and higher-order models. It is also the first general-purpose PPL with multimodal models in a statically typed paradigm. [5] We next discuss some of the PPLs' approaches for implementing models, and explain the presence or absence of their supported modelling features.

Table 3.1: Comparison of PPLs in terms of supported model features, where ● is full support, ◐ is partial support, and ○ is no support. Modular models are those that are definable in terms of other models.

Modelling features	ProbFX	Gen	Turing	Pyro	Edward	Stan	MonadBayes	Anglican	WebPPL
Multimodal	●	●	●	●	●	●	○	○	○
Modular	●	●	◐	●	●	○	●	●	●
Higher-order	●	◐	○	◐	◐	○	●	◐	●
Type-safe	●	○	○	○	○	●	●	○	○

3.5.1 Approaches for implementing probabilistic models

3.5.1.1 Gen

Gen [Cusumano-Towner et al. 2019], embedded in Julia as a host language, implements multimodal models via *macros*, which are procedures that take input expressions and generate modified expressions at parse-time. In Fig. 3.14a, the @gen macro transforms a regular Julia function into a "dynamic model" object written in a special shallow-embedded DSL. The (~) macro in x ~ d says that x is a random variable distributed by d, and that its random behaviour is *traced*. A traced variable can later be interpreted as either observe or sample by executing the model with Gen.generate: a runtime DSL interpreter that constrains @gen-type functions by a dictionary of observed values, and produces a sample trace subject to those constraints [Cusumano-Towner 2020].

[5] Stan [Carpenter et al. 2017] is type-safe but special-purpose.

The support for higher-order models is limited in a sense. Although @gen functions can interact with each other in a higher-order way, applying a *regular* function to one will "escape" the tracing of its random variables. For example with mapLinRegr in Fig. 3.14b, applying map to the linRegr model will demote that model to a regular function, preventing it from being multimodal or used in any inference procedures. Circumventing this requires some programmer intervention, for example via mapLinRegr′ which uses a top-level (~) to trace the linRegr model itself. Alternatively, Gen provides special model combinators like Gen.Map, that take @gen functions as input and return new @gen functions.

```
@gen function linRegr(x)
  m ~ normal(0, 1)
  c ~ normal(0, 2)
  y ~ normal(m * x + c, 1)
end

# observe y = 2.3, and sample for m and c
constraints  = Gen.choicemap((:y, 2.3))
(trace, _)   = Gen.generate(linRegr, (1.0,)
                            , constraints)
```

(a) Writing and specialising multimodal models

```
# treats linRegr as black box function
@gen function mapLinRegr(xs)
  ys = map(linRegr, xs)
end

# treats linRegr as multimodal model
@gen function mapLinRegr′(xs)
  ys = map((i, x) → {i} ~ linRegr(x)
           , enumerate(xs))
end
```

(b) Higher-order treatment of multimodal models

Figure 3.14: Gen: linear regression

3.5.1.2 Turing

Turing [Ge et al. 2018] is another PPL in Julia that uses macros to implement models, in Fig. 3.15a. The @model macro is used rewrite Julia functions into model objects, similar to Gen. Whereas Gen uses a DSL interpreter to support multimodality, Turing does this entirely through the (~) macro. In particular, Turing allows each argument of the @model function to be used/treated as a random variable inside the model (Fig. 3.15a); by providing the **missing** argument for m, this causes m ~ Normal(0, 1) to be rewritten to a Sample operation, whereas providing the observed value 2.3 for y results in Observe. An interesting feature of the approach is that it allows random variables to be indexed like regular variables; for example, by passing a list of observed values for argument y, and then writing y[i] ~ Normal (...) inside the model. This becomes convenient for expressing multiple observations for the same random variable name.

These models are not completely modular. Consider Fig. 3.15b which tries to modularise linRegr into calling a sub-model prior for generating parameters m and c. The refactoring linRegrModular is illegal because, unlike primitive distributions such as Normal, Turing's @model terms cannot be bound (~) to random variables; this is in contrast to @gen models in Gen (Fig. 3.14b). The alternative is linRegrModular′ which assigns (=) the result of prior to a regular variable, demoting prior from a model to a regular function, unable to be used for inference (similar to when using higher-order functions with Gen models).

```
@model function linRegr(x, y, m, c)
  m ∼ Normal(0, 1)
  c ∼ Normal(0, 2)
  y ∼ Normal(m ∗ x + c, 1)
end

# observe y = 2.3, and sample for m and c
linRegr (1.0,   2.3,  missing, missing)
```

(a) Writing and specialising multimodal models

```
@model function prior(m, c)
  m ∼ Normal(0, 1)
  c ∼ Normal(0, 2)
  return (m, c)
end

# invalid, cannot bind models to random variables
@model function linRegrModular(x, y, m, c)
  (m, c) ∼ prior (m, c)
  y ∼ Normal(m ∗ x + c, 1)
end

# treats prior as black-box function
@model function linRegrModular′(x, y, m, c)
  (m, c) = prior (m, c) ()
  y ∼ Normal(m ∗ x + c, 1)
end
```

(b) Modular treatment of multimodal models

Figure 3.15: Turing: linear regression

3.5.1.3 Pyro and Edward

Pyro [Bingham et al. 2019] and Edward [Moore and Gorinova 2018] are PPLs in Python that use Python coroutines to support multimodal models. We explain this for Pyro (but the same description holds for Edward). In Fig. 3.16, models are regular Python functions that use the Pyro primitive sample(s, d), where s is a string representing a random variable name, and d is a distribution. The programmer can later choose to interpret sample operations as observe operations instead, by wrapping their model with the coroutine ConditionMessenger, and providing a dictionary from variable names to observed values. Then upon model execution, each call to sample will defer control to the ConditionMessenger coroutine, which overrides the default behaviour of random sampling to return the observed value (if provided in the dictionary). While the approach of Python coroutines initially resembles effect handlers, the details are quite different; we discuss this when comparing with Pyro and Edward for programmable inference (Section 5.6.1).

Because functions in Python are first-class, so are models in Pyro. However, the support for higher-order models is slightly limited. Pyro uses the string argument s in sample(s, d) to uniquely identify each probabilistic operation invoked at runtime, and so the same random variable name may arise no more than once during model execution. Violating this, such as mapLinRegr in Fig. 3.16b, results in a runtime crash. This prevents models in general from being iterated over; instead, potential iterative behaviour must be foreseen when defining the model itself, for example mapLinRegrIdx which creates a new variable name for each iteration.

```
def linRegr (x ):
  m = sample("m", Normal(0, 1))
  c = sample("c" , Normal(0, 2))
  y = sample("y", Normal(m ∗ x + c, 1))
  return y

# observe y = 2.3, and sample for m and c
with ConditionMessenger(data={"y" =2.3}):
  y = linRegr (1.0)
```

(a) Writing and specialising multimodal models

```
# crashes, if length(xs) > 1
def mapLinRegr(xs):
  ys = map(lambda x: linRegr(x ), xs)

# good
def mapLinRegrIdx(xs):
  ys = map(linRegrIdx, enumerate(xs))

def linRegrIdx(x, i):
  m = sample("m" + str ( i ), Normal(0, 1))
  c = sample("c" + str ( i ), Normal(0, 2))
  y = sample("y" + str ( i ), Normal(m ∗ x + c, 1))
  return y
```

(b) Higher-order treatment of multimodal models

Figure 3.16: Pyro: linear regression

3.5.1.4 MonadBayes

MonadBayes [Ścibior, Kammar, and Ghahramani 2018] is a Haskell-embedded PPL like our language, but based on an effect framework called the *Monad Transformer Library* (MTL) [S. Liang et al. 1995; Gill 2022], which is the main alternative to algebraic effects for typed functional programming languages. The idea in Fig. 3.17, is to represent the model type m as an abstract stack of monads (effect types) that implement the type classes MonadSample and MonadCond, providing modelling operations for sampling (e.g. normal) and observing (score). Upon model execution, this abstract stack is specialised to some concrete arrangement of monads, determining the specific type class instances for sampling and observing that are called.

Embedding into Haskell means that models in MonadBayes are both first-class and statically typed. Models are however not multimodal, being defined explicitly in terms of sampling and observing operations, and thus requiring each "specialisation" of the same underlying model in Fig. 3.17 to be written from scratch. We are not aware whether the MTL approach can be used to support multimodal models. As the focus of MonadBayes is on designing modular inference algorithms rather than models, we defer the main discussion of MTL for probabilistic programming to Part II, after implementing inference.

```
linRegrSim :: (MonadSample m, MonadCond m)
  ⇒ Double → Double → Double → m Double
linRegrSim x m c = do
  y ← normal (m ∗ x + c) 1
  return y
```

(a) Simulation

```
linRegrInf :: (MonadSample m, MonadCond m)
  ⇒ Double → Double → m (Double, Double)
linRegrInf x y = do
  m ← normal 0 1
  c ← normal 0 2
  score (normalPdf (m ∗ x + c) 1 y)
  return (m, c)
```

(b) Inference

Figure 3.17: MonadBayes: linear regression

3.5.1.5 Anglican and WebPPL

In Anglican [Tolpin et al. 2016] (in Clojure) and WebPPL [Goodman and Stuhlmüller 2014] (in Javascript), the user defines models in terms of explicit sample and observe operations, and so like MonadBayes, these models are not multimodal. The examples in Fig. 3.18 and Fig. 3.19 hence follow the same pattern of model duplication as Fig. 3.17.

Their embedding of models is similar in some ways to our algebraic effect representation. Both languages compile models using a continuation-passing-style (CPS) transformation, creating breakpoints at each sample and observe operation which store their continuations. Executing the model then consists of intercepting at those operations, performing some desired side computation, and then using the continuations to resume execution – akin to how effect handlers work. As far as we know, compilation of models into CPS has not been used for multimodality.

```
(defquery linRegrSim [x m c]
(let [y  (sample (normal (m * x + c) 1))]
      {:y y}))
```

(a) Simulation

```
(defquery linRegrInf [x y]
(let [m    (sample (normal 0 1))
      c    (sample (normal 0 2))
      y    (observe (normal (mu * x + c) 1) y)]
     {:mu mu :c c :y y}))
```

(b) Inference

Figure 3.18: Anglican: linear regression

```
var linRegrSim = function(x, m, c) {
  y = sample(Normal(m * x + c, 1), y)
  return y
}
```

(a) Simulation

```
var linRegrInf = function(x, y) {
  m    = sample(Normal(0, 1))
  c    = sample(Normal(0, 2))
  observe(Normal(m * x + c, 1), y)
  return (m, c)
}
```

(b) Inference

Figure 3.19: WebPPL: linear regression

3.5.1.6 Type safety

Multimodal PPLs require a way of naming random variables and providing them observed values. Our language takes a typed approach to this, using type-level strings to statically identify random variables, and type classes to constrain their types of observations (Section 3.2). The other discussed approaches to multimodality use just-in-time compilation (Turing, Gen) or are dynamically typed (Pyro, Edward), and so do not guarantee that certain random variables exist, or that the observed values provided to them are of the correct type.

3.5.1.7 Performance

We quantatively evaluate our language later in Part II. Although those results mainly compare inference, they also suggest our approach to multimodality does not notably affect performance, competing well against Gen (which supports multimodal models) and MonadBayes (which does not).

3.5.2 Other related work

Tagless-final shallow embedding as conceived by Kiselyov [2010] has been used as an embedding technique for typed functional PPLs, demonstrated by Kiselyov and Shan [2009] with OCaml as a host language, and Narayanan et al. [2016] who embed into Haskell. The idea is to capture the syntax of the

PPL as type class methods, where different class instances map the probabilistic program to different inference semantics. This is well suited to mapping entire programs uniformly into a semantic domain, but we found the approach difficult to compose semantics with; in particular, for composing the steps needed to interpret a multimodal model as a concrete model execution.

Monads The first use of Haskell for probabilistic programming is perhaps by Erwig and Kollmansberger [2006]. They restrict models to being discrete distributions implemented as the probability monad [Giry 2006]: a list that pairs all possible samples and their probabilities. By sequencing the operations of the monad with (>>=), this evolves the overall distribution analogously to how a probability tree grows.

Free monads The underlying data structure used to implement our algebraic effect framework was the free monad. Ścibior, Ghahramani, et al. [2015] embed primitive and conditional distributions in Haskell using an intermediate free monad representation; this bears an initial resemblance to our approach but the semantics are instead provided using type classes. Because their language directly implements conditional distributions, their models are also not multimodal. Later work by Ścibior, Kammar, and Ghahramani [2018] more closely coincides with our approach: they use free monad transformers [Schrijvers et al. 2019] to encode sample operations as syntax, allowing sample to be later interpreted as either drawing a new random value, or as reusing an old one instead.

Probabilistic logic languages ProbLog [De Raedt et al. 2007] is an extension of the language Prolog [Colmerauer 1990] which operates in a declarative logic-based paradigm, in contrast to the functional and imperative languages discussed in Section 3.5. In Prolog, programs are defined in terms of two types of clauses: facts and rules. Models in ProbLog are then programs with facts that only hold with a specified probability, and with rules that express the conjunction or disjunction of probabilistic facts. A key benefit of this declarative setting is the natural ability to specify models as relations between random variables; ProbLog is also modular, allowing clauses to be individually defined and reused, and multimodal, by allowing the user to specify observations with the special evidence keyword. A possible limitation is that models can be lengthy, often containing a large sequence of facts and rules as primitives [Vidal 2022]. However, the comparison of probabilistic logic languages with PPLs in general is not straightforward, as the former moreso targets models expressable in terms of exact probability values, rather than in terms of probability distributions.

CHAPTER 4

A Formal Calculus for Multimodal Models

This chapter presents an idealised minimal calculus for a statically typed functional language with multimodal models, capturing only the key ideas from the embedding in Chapter 3. The calculus provides a type-and-effect system for algebraic effects and handlers in a fine-grain call-by-value setting [Levy et al. 2003], extended with built-in constructs for conditionable variables, multimodal distributions, and model environments.

We introduce syntax of the kinds, types, and terms of the language in Section 4.1; the kinding and typing rules in Section 4.2 and Section 4.3; and finally, the small-step operational semantics of the language in Section 4.4, along with some of its formal properties.

4.1 Syntax

4.1.1 Type syntax

Fig. 4.1 gives the grammar for the kinds and types of the language.

Kinds K consist of: Type for value types, Comp for computation types, Effect and EffectRow for effects and rows of effects, and Handler for effect handlers.

Value types A, B include constants ι (e.g. Booleans, integers), lists `List` A, and products $A \times B$. Functions have type $A \to \underline{C}$ and map a value of type A to a computation of type $\underline{C}$. The type system also supports parametric polymorphism via the type $\forall \alpha : K.\ \underline{C}$; this describes a computation of type $\underline{C}$ that universally quantifies over a type variable α of unspecified kind K.

Computation types $\underline{C}, \underline{D}$ have the form $A \,!\, R$, where A represents the return value, and R is the *effect row* specifying the effects that the computation may perform.

Effects E are sets of type-assigned operations $\mathsf{op} : A \to B$, specifying the argument type A and return type B of operation op.

Effect rows R are *unordered* collections of effects E used to represent effect rows, where closed rows end in empty ε and open rows end in row variables r; using polymorphic (i.e. open) rows to capture effects is one way of enabling modular programming in a type-and-effect system.

Handler types F have the form $C \Rightarrow^{E} D$, which denotes that they interpret an effect E in an input computation of type $\underline{C}$, to produce an output computation of type $\underline{D}$.

$$
\begin{array}{lrcl}
\text{Kinds} & K & ::= & \mathsf{Type} \mid \mathsf{Comp} \mid \mathsf{Effect} \mid \mathsf{EffectRow} \mid \mathsf{Handler} \\
\text{Type} & T & ::= & A \mid \underline{C} \mid F \mid R \mid E \\
\text{Value type} & A, B & ::= & \iota \mid \texttt{Maybe}\ A \mid \texttt{List}\ A \mid A \times B \mid A \to \underline{C} \mid \forall \alpha : K.\, \underline{C} \mid \alpha \\
\text{Computation type} & \underline{C}, \underline{D} & ::= & A\,!\,R \\
\text{Effect} & E & ::= & \emptyset \mid \{\mathrm{op} : A \longrightarrow B\} \uplus E \\
\text{Effect row} & R & ::= & E; R \mid \varepsilon \mid r \\
\text{Effect handler type} & F & ::= & \underline{C} \Rightarrow^{E} \underline{D} \\
\text{Variables} & x, y, k & & \\
\text{Row variables} & r & & \\
\text{Conditionable variables} & \tilde{x}, \tilde{y} & & \\
\text{Kind environment} & \Delta & ::= & \Delta \cdot (\alpha \colon K) \mid \varepsilon \\
\text{Type environment} & \Gamma & ::= & \Gamma \cdot (x \colon A) \mid \varepsilon \\
\text{Model type environment} & \Omega & ::= & \Omega \cdot (\tilde{x} \colon A) \mid \varepsilon
\end{array}
$$

Figure 4.1: Syntax: types

Type and kind environments For environments, we write $(\cdot)$ to express extending an environment with a new variable binding, and ε for the empty environment. The standard kind environment Δ associates type variables α to their kinds K, and the type environment Γ maps term variables x to their types A (where Γ is well-kinded under Δ if Δ maps every type in Γ to a kind). As well as regular variables x, the language also has *conditionable variables* $\tilde{x}$. The *model type environment* Ω then maps each $\tilde{x}$ to its value type A (where Ω is well-kinded under Δ if Δ maps every type in Ω to a kind), and is used for typing the model environments $\tilde{\rho}$ introduced next in Fig. 4.2.

4.1.2 Term syntax

Fig. 4.2 gives the term syntax, which distinguishes between values V, effectful computations M, and effect handlers H.

Model environment $\tilde{\rho}$. A model environment contains bindings $(\tilde{x} \colon V)$ that map conditionable variables to their values, with ε being the empty environment. We consider $\tilde{\rho}$ as a set which is equivalent up to reordering of its variables.

Values V, U. Primitive values include variables x, constants c, the "maybe" constructors `Just` V and `Nothing`, and the list constructors $(V_1 :: V_2)$ and `Nil`. We also have value abstraction $\lambda x : A.\, M$ and type abstraction $\Lambda\alpha : K.\, M$.

Computations M, N. Basic computation terms include pure values `return` V, regular application $V_1\, V_2$, type application $V\ [T]$, and let-bindings `let` $x \leftarrow M$ `in` N.

An arbitrary operation op is called by providing an appropriate argument V. A primitive distribution ϕ can be thought of as a special kind of operation, and is called by providing its associated distribution parameters V. This can also be used in the special let-binding, `let` $\tilde{x} \sim \phi\ V$ `in` M, to bind the result to a conditionable variable $\tilde{x}$; if the current model environment $\tilde{\rho}$ contains a value for $\tilde{x}$, then the expression denotes *conditioning* against the likelihood of $\phi\ V$ having generated that value, otherwise representing *sampling* from $\phi\ V$. Finally, the handle term, `with` H `handle` M, interprets all the operations raised in

Model environment	$\tilde{\rho}$	::=	$\tilde{\rho} \cdot (\tilde{x} : V) \mid \varepsilon$	
Value	V, U	::=	x	variable
			c	constant
			$\texttt{Just}\ V \mid \texttt{Nothing}$	just, nothing
			$V_1 :: V_2 \mid \texttt{Nil}$	cons, nil
			$\lambda x : A.\, M$	function
			$\Lambda \alpha : K.\, M$	type abstraction
Handler	H	::=	$\{\texttt{return}\ x \to M\}$	return clause
			$\{\mathrm{op}\, x\, k \to M\} \uplus H$	operation clause
Computation	M, N	::=	$\texttt{return}\ V$	return
			$V_1\, V_2$	application
			$V\ [T]$	type application
			$\mathrm{op}\ V$	operation call
			$\phi\ V$	distribution call
			$\texttt{let}\ x \leftarrow M\ \texttt{in}\ N$	let-bind
			$\texttt{let}\ \tilde{x} \sim \phi\ V\ \texttt{in}\ M$	let-bind (~)
			$\texttt{with}\ H\ \texttt{handle}\ M$	handle

Figure 4.2: Syntax: terms

computation M that have an associated “operation clause” defined in the effect handler H.

Handlers H. An effect handler consists of exactly one return clause and a set of operation clauses, where the notation $\uplus$ requires the clauses to be mutually exclusive in their associated operation. The return clause $\{\texttt{return}\ x \to M\}$ describes how to handle the final return value of a handled computation: by binding that value to x in the returned computation M. An operation clause $\{\mathrm{op}\ x\ k \to M\}$ describes how to interpret an operation op: by binding the operation’s argument to x, the operation’s continuation to k, and returning the computation M.

4.2 Kinding

Fig. 4.3 gives the kinding rules for the language’s type system, where the judgement $\Delta \vdash T : K$ says that type T may be assigned kind K under a kind environment Δ.

The kind Type describes value types A, which include: the type constant ι for constant values, function types $A \to \underline{C}$, and polymorphic computation types $\forall \alpha : K.\, \underline{C}$ where K is determined by the entry for type variable α in the kind environment. Then, computation types $A\,!\,R$ have kind Comp, handler types $C \Rightarrow^{E} D$ with kind Handler, and effect rows $E; R$ and ϵ with kind EffectRow. Lastly, an effect E has kind Effect where the built-in effects of the language include `Dist`, `Sample`, and `Observe`, explained in Section 4.3.

$\boxed{\Delta \vdash T : K}$ *Type T has kind K under a kind environment Δ.*

type constant
$$\frac{}{\Delta \vdash \iota : \mathsf{Type}}$$

function
$$\frac{\Delta \vdash A : \mathsf{Type} \qquad \Delta \vdash \underline{C} : \mathsf{Comp}}{\Delta \vdash A \longrightarrow \underline{C} : \mathsf{Type}}$$

type variable
$$\frac{\alpha : K \in \Delta}{\Delta \vdash \alpha : K}$$

polymorphic computation
$$\frac{\Delta \cdot (\alpha : K) \vdash \underline{C} : \mathsf{Comp}}{\Delta \vdash \forall \alpha : K.\, \underline{C} : \mathsf{Type}}$$

computation
$$\frac{\Delta \vdash A : \mathsf{Type} \qquad \Delta \vdash R : \mathsf{EffectRow}}{\Delta \vdash A\,!\,R : \mathsf{Comp}}$$

handler
$$\frac{\Delta \vdash \underline{C} : \mathsf{Comp} \qquad \Delta \vdash E : \mathsf{Effect} \qquad \Delta \vdash \underline{D} : \mathsf{Comp}}{\Delta \vdash \underline{C} \Rightarrow^{E} \underline{D} : \mathsf{Handler}}$$

effect row (non-empty)
$$\frac{\Delta \vdash E : \mathsf{Effect} \qquad \Delta \vdash R : \mathsf{EffectRow}}{\Delta \vdash E; R : \mathsf{EffectRow}}$$

effect row (empty)
$$\frac{}{\Delta \vdash \varepsilon : \mathsf{EffectRow}}$$

distribution effect
$$\frac{}{\Delta \vdash \mathtt{Dist} : \mathsf{Effect}}$$

sample effect
$$\frac{}{\Delta \vdash \mathsf{Sample} : \mathsf{Effect}}$$

observe effect
$$\frac{}{\Delta \vdash \mathtt{Observe} : \mathsf{Effect}}$$

Figure 4.3: Kinding rules

4.3 Typing

This section describes the type system of the language for model environments, values, operations, computations, and effect handlers.

4.3.1 Model environment types

Fig. 4.4 defines the typing rules for *model environments*. The judgement $\Delta; \Gamma \vdash \tilde{\rho} : \Omega$ says that if Γ and Ω are well-kinded under Δ, then model environment $\tilde{\rho}$ may be assigned type Ω under $\Delta; \Gamma$.

The model env (non-empty) rule for non-empty $\tilde{\rho}$ is non-standard w.r.t regular environments. It requires the value of a given conditionable variable $\tilde{x} : A$ to be a *list* of type $\mathtt{List}\ A$, thus associating each $\tilde{x}$ to a trace of observed values; these values cannot be conditionable variables themselves.

4.3.2 Value types

Fig. 4.5 defines the typing rules for *values*. The judgement $\Delta; \Gamma; \Omega \vdash V : A$ says that if Γ and Ω are well-kinded under Δ, and $\Delta \vdash A : \mathsf{Type}$, then value V may be assigned type A under $\Delta; \Gamma; \Omega$.

A variable x has type A if the binding $x : A$ is contained in the type environment Γ. A function $\lambda x : A.\, M$ takes a value argument $x : A$ to return an arbitrary computation type $M : \underline{C}$, and likewise for type abstraction $\Lambda \alpha : K.\, M$ but instead taking a type argument $\alpha : K$. There are also the just/nothing cases of type $\mathtt{Maybe}\ A$, and cons/nil cases of type $\mathtt{List}\ A$, as usual; the decomposition $\Omega_1 \uplus \Omega_2$ in the cons rule is explained later when typing computations (Fig. 4.7).

4.3.3 Operation types

Fig. 4.6 gives the type signatures for the language's *core operations* – by first assuming a fixed set Φ of primitive distributions $\phi : A \longrightarrow B$, where A is the type of parameters usually associated with the distribution, and B is the type of values the distribution generates.

Primitive distributions are not technically operations, but rather act as a name in the syntax for characterising three core effects. In particular, for each primitive distribution $\phi : A \rightarrow B \in \Phi$:

- The effect $\texttt{Dist}$ contains an operation $\texttt{dist}_\phi$ that takes an extra argument of type $\texttt{Maybe}\ B$ which may or may not contain an observed value to condition ϕ on.

- The effect $\texttt{Observe}$ contains an operation $\texttt{observe}_\phi$ that takes an extra argument of type B as an observed value to condition ϕ on.

- The effect $\texttt{Sample}$ contains an operation $\texttt{sample}_\phi$ that simply samples from ϕ.

The operations $\texttt{sample}_\phi$ and $\texttt{observe}_\phi$ can thus be viewed as specialisations of $\texttt{dist}_\phi$ with respect to whether its argument of type $\texttt{Maybe}\ B$ contains a value of type B.

4.3.4 Computation types

Fig. 4.7 defines the typing rules for *computations.* The judgement $\Delta; \Gamma; \Omega \vdash M : \underline{C}$ says that if Γ and Ω are well-kinded in Δ, and $\Delta \vdash \underline{C} : \text{Comp}$, then computation M can be assigned type $\underline{C}$ under $\Delta; \Gamma; \Omega$.

Definition* 1** (Disjoint merge):*** Let $\Omega \uplus \Omega' = \Omega \cdot \Omega'$ iff $\text{dom}(\Omega) \cap \text{dom}(\Omega') = \varnothing$

For computations that decompose into multiple sub-terms, Definition (Disjoint merge) is used to type each sub-term under a mutually exclusive model environment, preventing the same conditionable variable $\tilde{x}$ from being referred to statically more than once in a program. This is used for application, let-bind, and let-bind ($\sim$), the last of which requires a bit more explanation.

A return expression $\texttt{return}\ V$ has type $A\,!\,R$ for arbitrary effect row R whenever $V : A$. The application rule for $V_1\ V_2$ is standard, providing the function $V_1 : A \rightarrow \underline{C}$ with appropriate input $V_2 : A$ to output a computation of type $\underline{C}$. The type application rule for $V\ [T]$ is similar, but where the type-level function $V : \forall\alpha : K.\ \underline{C}$ is provided a type T as an input, binding this to type variable α in $\underline{C}$.

In operation call for op V, given op $: A \rightarrow B \in E$ and argument $V : A$, this outputs a computation of type $B\,!\,(E; R)$ where R captures the rest of the possible effects in the containing computation; as row types are unordered, $E; R$ only specifies that E is a member of the effect row. For the distribution call rule for $\phi\ V$, which provides $\phi : A \rightarrow B \in \Phi$ with argument $V : A$, this produces a computation of type $B\,!\,(\texttt{Dist}; R)$ implying the operation $\texttt{dist}_\phi \in \texttt{Dist}$ is invoked implicitly.

The standard let-bind, $\texttt{let}\ x \leftarrow M\ \texttt{in}\ N$, binds the return value of the bound computation $M : A\,!\,R$ to the local variable $x : A$, before proceeding with the body $N : B\,!\,R$ which is typed under the same effect row as M.

The special let-bind ($\sim$) rule for $\texttt{let}\ \tilde{x} \sim \phi\ V\ \texttt{in}\ M$, given that $\phi : A \rightarrow B$, specifically decomposes the model type environment into $\Omega \uplus \Omega' \cdot (\tilde{x} : B)$. First, Ω is used to type the distribution's argument $V : A$. Then, the type of values the distribution generates is associated to the bound conditionable variable $\tilde{x} : B$, which is in turn used to type the regular variable $x : B$ sharing the same name; this expresses that x may be one of the observed values of $\tilde{x}$ (recalling that conditionable variables are assigned lists of values). Lastly, the body M is typed using the new local variable $x : B$ and the remaining model type environment Ω'.

The final rule, handle for handling computations, is similar to application. Given a handler of type $A\,!\,(E; R) \Rightarrow^E B\,!\,R$ for some effect E, applying this to an input of type $A\,!\,(E; R)$ will yield an output of type $B\,!\,R$ where E has been discharged from the effect row.

4.3.5 Handler types

Fig.4.8 defines the typing rule for *handlers*. The judgement $\Delta; \Gamma \vdash H : \underline{C} \Rightarrow^{E} \underline{D}$ says that if Γ is well-kinded under Δ, and $\Delta \vdash \underline{C} \Rightarrow^{E} \underline{D} : \mathsf{Handler}$, then handler H can be assigned type $\underline{C} \Rightarrow^{E} \underline{D}$ under $\Delta; \Gamma$.

A handler for the effect E takes an input computation of type $A \,!\, (E; R)$ and produces an output computation of type $B \,!\, R$ – thus mapping the type of the return value from A to B, and discharging E from the effect row $(E; R)$. (This implies the input computation may only perform operations in $(E; R)$ and the output computation may only perform operations in R.)

The return clause $\{\mathtt{return}\ x \longrightarrow M\}$ is typed by the first premise: given a value of type A bound to x, the returned computation M must be of type $B \,!\, R$. In the second premise, an operation clause $\{\mathrm{op}_i\ x_i\ k_i \longrightarrow M_i\}$ is provided for each $\mathrm{op}_i : A_i \longrightarrow B_i \in E$: given the operation's argument $x_i : A_i$, and a continuation $k_i : B_i \longrightarrow B \,!\, R$ that is parameterised by the operation's result, the returned computation M_i must again be of type $B \,!\, R$.

Remark* 1** (Closed handlers):*** We prevent handlers from containing any conditionable variables, by requiring each clause to return a computation M that is well-typed under the empty model type environment $\Omega = \varepsilon$.

4.3.6 Opening and closing row types

To make computations easy to reuse in different contexts, the user can define them as polymorphic in the effects they do not use. To avoid having to specify effect rows as being explicitly polymorphic, we introduce two type rules in Fig. 4.9.

The first rule opens a (possibly empty) closed effect row with a polymorphic row variable, and the second simplifies open effect rows. Although close effect row can lead to some programs becoming ill-typed by assigning a less general type to computations, open effect row can always be applied to recover the original most general type; thus, even if types become simplified by close effect row, the set of typeable programs remains unchanged [Leijen 2017].

$\boxed{\Delta; \Gamma \vdash \tilde{\rho} : \Omega}$ *Model environment $\tilde{\rho}$ has type Ω under a kind and type environment $\Delta; \Gamma$*

model env (non-empty)

$$\frac{\Delta; \Gamma \vdash \tilde{\rho} : \Omega \qquad \Delta; \Gamma; \varepsilon \vdash V : \mathtt{List}\ A}{\Delta; \Gamma \vdash \tilde{\rho} \cdot (\tilde{x} : V) : \Omega \cdot (\tilde{x} \colon A)}$$

model env (empty)

$$\frac{}{\Delta; \Gamma \vdash \varepsilon : \varepsilon}$$

Figure 4.4: Type rules: model environments

$\boxed{\Delta; \Gamma; \Omega \vdash V : A}$ *Value V has type A under a kind, type, and model type environment $\Delta; \Gamma; \Omega$*

var
$$\frac{x : A \in \Gamma}{\Delta; \Gamma; \Omega \vdash x : A}$$

function
$$\frac{\Delta; \Gamma \cdot (x : A); \Omega \vdash M : \underline{C}}{\Delta; \Gamma; \Omega \vdash \lambda x : A.\, M : A \rightarrow \underline{C}}$$

type abstraction
$$\frac{\Delta \cdot (\alpha : K); \Gamma; \Omega \vdash M : \underline{C}}{\Delta; \Gamma; \Omega \vdash \Lambda \alpha : K.\, M : \forall \alpha : K.\, \underline{C}}$$

just
$$\frac{\Delta; \Gamma; \Omega \vdash V : A}{\Delta; \Gamma; \Omega \vdash \texttt{Just}\ V : \texttt{Maybe}\ A}$$

nothing
$$\frac{}{\Delta; \Gamma; \Omega \vdash \texttt{Nothing} : \texttt{Maybe}\ A}$$

cons
$$\frac{\Delta; \Gamma; \Omega_1 \vdash V_1 : A \qquad \Delta; \Gamma; \Omega_2 \vdash V_2 : \texttt{List}\ A}{\Delta; \Gamma; \Omega_1 \uplus \Omega_2 \vdash (V_1 :: V_2) : \texttt{List}\ A}$$

nil
$$\frac{}{\Delta; \Gamma; \Omega \vdash \texttt{Nil} : \texttt{List}\ A}$$

Figure 4.5: Type rules: values

$\boxed{\phi : A \rightarrow B \in \Phi}$ *Primitive distribution ϕ has parameters of type A and generates values of type B*

$$\begin{aligned}
\texttt{normal} &: \texttt{Double} \times \texttt{Double} &&\rightarrow \texttt{Double} \in \Phi \\
\texttt{bernoulli} &: \texttt{Double} &&\rightarrow \texttt{Bool} \in \Phi \\
\texttt{discrete}_A &: \texttt{List}\ A \times \texttt{Double} &&\rightarrow A \in \Phi \\
&\ldots
\end{aligned}$$

$\boxed{\mathrm{op} : A \rightarrow B \in E}$ *Operation op has argument type A and output type B, and belongs to effect E*

$$\begin{aligned}
\texttt{dist}_\phi &: A \times \texttt{Maybe}\ B &&\rightarrow B \in \texttt{Dist} \\
\texttt{observe}_\phi &: A \times B &&\rightarrow B \in \texttt{Observe} \\
\texttt{sample}_\phi &: A &&\rightarrow B \in \texttt{Sample}
\end{aligned}$$

Figure 4.6: Type signatures: primitive distributions and core operations

$\boxed{\Delta; \Gamma; \Omega \vdash M : \underline{C}}$ *Computation M has type $\underline{C}$ under kind, type, and model type environment $\Delta; \Gamma; \Omega$*

return
$$\frac{\Delta; \Gamma; \Omega \vdash V : A}{\Delta; \Gamma; \Omega \vdash \texttt{return}\ V : A\,!\,R}$$

application
$$\frac{\Delta; \Gamma; \Omega \vdash V_1 : A \rightarrow \underline{C} \qquad \Delta; \Gamma; \Omega' \vdash V_2 : A}{\Delta; \Gamma; \Omega \uplus \Omega' \vdash V_1\, V_2 : \underline{C}}$$

type application
$$\frac{\Delta; \Gamma; \Omega \vdash V : \forall \alpha : K.\, \underline{C} \qquad \Delta \vdash T : K}{\Delta; \Gamma; \Omega \vdash V\ [T] : \underline{C}[\alpha \mapsto T]}$$

operation call
$$\frac{\mathrm{op} : A \rightarrow B \in E \qquad \Delta; \Gamma; \Omega \vdash V : A}{\Delta; \Gamma; \Omega \vdash \mathrm{op}\ V : B\,!\,(E; R)}$$

distribution call
$$\frac{\phi : A \rightarrow B \in \Phi \qquad \Delta; \Gamma; \Omega \vdash V : A}{\Delta; \Gamma; \Omega \vdash \phi\ V : B\,!\,(\texttt{Dist}; R)}$$

let-bind
$$\frac{\Delta; \Gamma; \Omega \vdash M : A\,!\,R \qquad \Delta; \Gamma \cdot (x : A); \Omega' \vdash N : B\,!\,R}{\Delta; \Gamma; \Omega \uplus \Omega' \vdash \texttt{let}\ x \leftarrow M\ \texttt{in}\ N : B\,!\,R}$$

let-bind ($\sim$)
$$\frac{\phi : A \rightarrow B \in \Phi \qquad \Delta; \Gamma; \Omega \vdash V : A \qquad \Delta; \Gamma \cdot (x : B); \Omega' \vdash M : C\,!\,(\texttt{Dist}; R)}{\Delta; \Gamma; \Omega \uplus \Omega' \cdot (\tilde{x} : B) \vdash \texttt{let}\ \tilde{x} \sim \phi\ V\ \texttt{in}\ M : C\,!\,(\texttt{Dist}; R)}$$

handle
$$\frac{\Delta; \Gamma \vdash H : A\,!\,(E; R) \Rightarrow^{E} B\,!\,R \qquad \Delta; \Gamma; \Omega \vdash M : A\,!\,(E; R)}{\Delta; \Gamma; \Omega \vdash \texttt{with}\ H\ \texttt{handle}\ M : B\,!\,R}$$

Figure 4.7: Type rules: computations

$\boxed{\Delta;\Gamma \vdash H : \underline{C} \Rightarrow^{E} \underline{D}}$ *Handler H for effect E has type $\underline{C} \Rightarrow^{E} \underline{D}$ under kind and type environment $\Delta;\Gamma$*

handler

$$\frac{\Delta;\Gamma\cdot(x\colon A);\varepsilon \vdash M : B\,!\,R \qquad \left[\Delta;\Gamma\cdot(x_i\colon A_i)\cdot(k_i\colon B_i \longrightarrow B\,!\,R);\varepsilon \vdash M_i : B\,!\,R\right]_{\forall \mathrm{op}_i\,:\,A_i\to B_i\in E}}{\Delta;\Gamma \vdash \{\mathtt{return}\ x \longrightarrow M\} \uplus \{\mathrm{op}_i\ x_i\ k_i \longrightarrow M_i\}_{\mathrm{op}_i\in E} : A\,!\,(E;R) \Rightarrow^{E} B\,!\,R}$$

Figure 4.8: Type rules: handler

open effect row

$$\frac{\Delta;\Gamma;\Omega \vdash M : A\,!\,(E_1;\ldots;E_n;\varepsilon)}{\Delta\cdot(r : \mathsf{EffectRow});\Gamma;\Omega \vdash M : A\,!\,(E_1;\ldots;E_n;r)}$$

close effect row

$$\frac{\Delta\cdot(r : \mathsf{EffectRow});\Gamma;\Omega \vdash M : A\,!\,(E_1;\ldots;E_n;r)}{\Delta;\Gamma;\Omega \vdash M : A\,!\,(E_1;\ldots;E_n;\varepsilon)}$$

Figure 4.9: Opening and closing effect rows

4.4 Semantics

Fig. 4.10 defines the semantics for our language using a small-step operational style in the relation $\tilde{\rho}, M \rightsquigarrow \tilde{\rho}', M'$. This relation states that $\tilde{\rho}$, M reduces to $\tilde{\rho}'$, M' in a single step, for any $\Delta;\Gamma \vdash \tilde{\rho} : \Omega$ and $\Delta;\Gamma;\Omega \vdash M : \underline{C}$ where Γ and Ω are well-kinded in Δ.

The semantics uses an evaluation context $\mathcal{E}$ whose grammar allows us to focus on reducible sub-terms in let-bindings ($\mathtt{let}\ x \leftarrow \mathcal{E}\ \mathtt{in}\ M$) and handle terms ($\mathtt{with}\ H\ \mathtt{handle}\ \mathcal{E}$). It does this by letting a reducible sub-term M be hoisted up through a program to yield the form $\mathcal{E}[M]$, where the LIFT rule ($\overset{\mathrm{L}}{\rightsquigarrow}$) can then apply a top-level reduction ($\rightsquigarrow$) to the nested M.

The rules APP, TY-APP, and LET-BIND (RET) use standard beta reduction to replace the bound variable of a computation with an argument.

Definition* 2** (!):*** For any closed value of type $\mathtt{List}\ A$, the function (!) returns a pair of closed values of type $\mathtt{Maybe}\ A \times \mathtt{List}\ A$, and is defined by the equations below. (A value V is closed when $\varepsilon;\varepsilon;\varepsilon \vdash V : A$ for some type A.)

$$(V_1 :: V_2)\,! = (\mathtt{Just}\ V_1, V_2) \qquad \mathtt{Nil}\,! = (\mathtt{Nothing}, \mathtt{Nil}).$$

The rule LET-BIND ($\sim$) shows how $\mathtt{let}\ \tilde{x} \sim \phi\ V\ \mathtt{in}\ M$ reduces to a regular let-binding of the form $\mathtt{let}\ x \leftarrow \mathtt{dist}_{\phi}(V, V')\ \mathtt{in}\ M$ – by calling the operation $\mathtt{dist}_{\phi}$ corresponding to ϕ, and then binding the result to regular variable x whose name corresponds to $\tilde{x}$. The argument V' is the current observed value of $\tilde{x}$ in the model environment $\tilde{\rho}\cdot(\tilde{x}\colon U)$, i.e. the head element (if it exists) of the list U; this is safely looked up using (!) in Definition (!), and removed from that list in the output environment.

The rule DIST CALL, for when $\phi\ V$ is not bound to a conditionable variable, reduces to the operation call $\mathtt{dist}_{\phi}$ (V, $\mathtt{Nothing}$) where $\mathtt{Nothing}$ indicates no observed value to condition against.

The rule HANDLE (RET) reduces $\mathtt{with}\ H\ \mathtt{handle}\ (\mathtt{return}\ V)$ by beta reducing the return clause $\{\mathtt{return}\ x \longrightarrow M\}$ of H, and then continuing with the variable assignment $x \mapsto V$ in M. [1]

Finally, the rule HANDLE (OP) shows how a program containing an operation call, $\mathcal{E}[\mathrm{op}\ V]$, is evaluated by a corresponding handler clause $\{\mathrm{op}\ x\ k \longrightarrow M\}$. It does this by returning the clause body M,

[1] The terms $(\lambda x : A.\, M)\ V$ and $\mathtt{let}\ x \leftarrow \mathtt{return}\ V\ \mathtt{in}\ M$ can be subsumed by $\mathtt{with}\ \{\mathtt{return}\ x \longrightarrow M\}\ \mathtt{handle}\ (\mathtt{return}\ V)$, but are retained in the language for convenience.

$\boxed{\tilde{\rho}, M \rightsquigarrow \tilde{\rho}', M'}$ *Model environment $\tilde{\rho}$ and computation M reduce to $\tilde{\rho}'$ and M' in one step*

APP $\quad \tilde{\rho}, (\lambda x : A.\, M)\ V \rightsquigarrow \tilde{\rho}, M[x \mapsto V]$

TY-APP $\quad \tilde{\rho}, (\Lambda \alpha : K.\, M)[T] \rightsquigarrow \tilde{\rho}, M[\alpha \mapsto T]$

LET-BIND (RET) $\quad \tilde{\rho}, \texttt{let}\ x \leftarrow \texttt{return}\ V\ \texttt{in}\ M \rightsquigarrow \tilde{\rho}, M[x \mapsto V]$

LET-BIND ($\sim$) $\quad \tilde{\rho} \cdot (\tilde{x} : U), \texttt{let}\ \tilde{x} \sim \phi\ V\ \texttt{in}\ M \rightsquigarrow \tilde{\rho} \cdot (\tilde{x} : U'), \texttt{let}\ x \leftarrow \texttt{dist}_{\phi}(V, V')\ \texttt{in}\ M$
$\quad$ where $(V', U') = U\,!$

DIST CALL $\quad \tilde{\rho}, \phi\, V \rightsquigarrow \tilde{\rho}, \texttt{dist}_{\phi}(V, \texttt{Nothing})$

HANDLE (RET) $\quad \tilde{\rho}, \texttt{with}\ H\ \texttt{handle}\ (\texttt{return}\ V) \rightsquigarrow \tilde{\rho}, M[x \mapsto V]$
$\quad$ if $\{\texttt{return}\ x \rightarrow M\} \in H$

HANDLE (OP) $\quad \tilde{\rho}, \texttt{with}\ H\ \texttt{handle}\ \mathcal{E}[\texttt{op}\ V] \rightsquigarrow \tilde{\rho}, M[x \mapsto V, k \mapsto \lambda y.\, \texttt{with}\ H\ \texttt{handle}\ \mathcal{E}[\texttt{return}\ y]]$
$\quad$ if $\{\texttt{op}\ x\ k \rightarrow M\} \in H$
$\quad$ and $\texttt{op} \notin \texttt{Handled}(\mathcal{E})$

$$\begin{aligned}
\texttt{Handled}([]) &= \emptyset \\
\texttt{Handled}(\texttt{let}\ x \leftarrow \mathcal{E}\ \texttt{in}\ M) &= \texttt{Handled}(\mathcal{E}) \\
\texttt{Handled}(\texttt{with}\ H\ \texttt{handle}\ \mathcal{E}) &= \texttt{Handled}(\mathcal{E}) \cup \texttt{dom}(H)
\end{aligned}$$

$\boxed{\tilde{\rho}, \mathcal{E}[M] \overset{\text{L}}{\rightsquigarrow} \tilde{\rho}', \mathcal{E}[M']}$ *Model environment $\tilde{\rho}$ and evaluation context $\mathcal{E}[M]$ reduce to $\tilde{\rho}'$ and $\mathcal{E}[M']$ in one step*

LIFT $\quad \tilde{\rho}, \mathcal{E}[M] \overset{\text{L}}{\rightsquigarrow} \tilde{\rho}', \mathcal{E}[M']\quad$ if $\tilde{\rho}, M \rightsquigarrow \tilde{\rho}', M'$

Evaluation context $\quad \mathcal{E} ::= [\cdot] \mid \texttt{let}\ x \leftarrow \mathcal{E}\ \texttt{in}\ M \mid \texttt{with}\ H\ \texttt{handle}\ \mathcal{E}$

Figure 4.10: Small-step operational semantics

where the operation's argument V is bound to x, and its continuation $\lambda y.\, \texttt{with}\ H\ \texttt{handle}\ \mathcal{E}[\texttt{return}\ y]$ to k. Here, k continues by recursively handling the containing program $\mathcal{E}$, thus describing the semantics of a *deep* handler [Hillerström and Lindley 2018], where the original operation is replaced by its result bound to y. The side condition 'if $\texttt{op} \notin \texttt{Handled}(\mathcal{E})$' enforces that each operation can only be handled by its *nearest* (appropriate) enclosing handler, with $\texttt{Handled}(\mathcal{E})$ being the set of operations in $\mathcal{E}$ that already have a handler in $\mathcal{E}$.

4.4.1 Formal properties

The following formal properties hold for these semantics, assuming no usage of the typing rules close effect row and close effect row from Fig. 4.9.

***Theorem* 1** (Determinism)**:** Suppose $\tilde{\rho}$ and M are well-typed.
If $\tilde{\rho}, M \overset{\text{L}}{\rightsquigarrow} \tilde{\rho}_1, M_1$ and $\tilde{\rho}, M \overset{\text{L}}{\rightsquigarrow} \tilde{\rho}_2, M_2$ then $\tilde{\rho}_1 = \tilde{\rho}_2$ and $M_1 = M_2$.

***Definition* 3** (Redex)**:** $\text{redex}(M)$ if and only if $\exists M'.\ M \rightsquigarrow M'$.

***Definition* 4** (Decomposition)**:** $M \sim \mathcal{E}[N]$ if and only if $M = \mathcal{E}[N]$ and $\text{redex}(N)$.

$\boxed{M \sim \mathcal{E}[N]}$ *Computation M decomposes into evaluation context $\mathcal{E}[N]$.*

$$\frac{\mathsf{redex}(M)}{M \sim [M]} \qquad \frac{M \sim \mathcal{E}[N]}{\texttt{let}\ x \leftarrow M\ \texttt{in}\ M' \sim \texttt{let}\ x \leftarrow \mathcal{E}[N]\ \texttt{in}\ M'} \qquad \frac{M \sim \mathcal{E}[N]}{\texttt{with}\ H\ \texttt{handle}\ M \sim \texttt{with}\ H\ \texttt{handle}\ \mathcal{E}[N]}$$

***Lemma* 1** (Unique decomposition)**:** Suppose M is well-typed.
If $M \sim \mathcal{E}[N]$ and $M \sim \mathcal{E}'[N']$, then $\mathcal{E} = \mathcal{E}'$ and $N = N'$.

Theorem (Determinism) guarantees that reduction ($\overset{\mathrm{L}}{\rightsquigarrow}$) on a well-typed computation and model environment is deterministic. In real-world applications such as simulation or inference, determinism may not generally be true as the user may introduce stochastic behaviour when implementing an effect handler for `Sample`. This calculus, primarily for translating multimodal models to probabilistic programs that sample and observe, does not have/require a notion of randomness and so is deterministic. However, it can sufficiently describe a probabilistic model as a deterministic system parameterised by a sequence of stochastic inputs (illustrated in Section 4.5), which is sometimes called a trace-based semantics for PPLs [Kozen 1979; Borgström et al. 2016; Dahlqvist et al. 2023].

Proof of Determinism. The proof in Appendix B.1 proceeds by: (i) proving the Unique decomposition property of the relation ($\sim$) in Definition (Decomposition), which describes how a computation can be decomposed into an evaluation context with a hole plugged by a Redex, and then (ii) using the fact that there is only single top-level reduction rule ($\rightsquigarrow$) per redex. □

***Theorem* 2** (Progress)**:** Suppose $\tilde{\rho}$ and M are well-typed such that $\varepsilon; \varepsilon \vdash \tilde{\rho} : \Omega$ and $\varepsilon; \varepsilon; \Omega \vdash M : \underline{C}$.
Either M is in canonical form, or $\exists \tilde{\rho}', M'.\ \tilde{\rho}, M \overset{\mathrm{L}}{\rightsquigarrow} \tilde{\rho}', M'$.

***Definition* 5** (Canonical form)**:** The canonical computations are $\texttt{return}\ V$ and $\mathcal{E}[\mathsf{op}\ V]$ where $\mathsf{op} \notin \mathsf{Handled}(\mathcal{E})$.

Theorem (Progress) guarantees that if reduction ($\overset{\mathrm{L}}{\rightsquigarrow}$) cannot be applied to a well-typed computation and model environment, the computation must be in Canonical form; that is, it returns a value or gets stuck on an unhandled operation.

Proof of Progress. The proof in Appendix B.2 is by induction on the type derivations of computations. □

***Theorem* 3** (Type preservation)**:** Suppose $\varepsilon; \varepsilon \vdash \tilde{\rho} : \Omega$ and $\varepsilon; \varepsilon; \Omega \vdash M : \underline{C}$.
If $\tilde{\rho}, M \overset{\mathrm{L}}{\rightsquigarrow} \tilde{\rho}', M'$, then $\varepsilon; \varepsilon \vdash \tilde{\rho}' : \Omega$ and $\varepsilon; \varepsilon; \Omega \vdash M' : \underline{C}$.

***Lemma* 2** (Value substitution)**:** Suppose $\varepsilon; (x : A); \Omega \vdash M : \underline{C}$.
If $\varepsilon; \varepsilon; \Omega' \vdash V : A$ then $\varepsilon; \varepsilon; \Omega \uplus \Omega' \vdash M[x \mapsto V] : \underline{C}$.

***Lemma* 3** (Type substitution)**:** Suppose $(\alpha : K); \varepsilon; \Omega \vdash M : \underline{C}$.
If $\varepsilon \vdash T : K$ then $\varepsilon; \varepsilon; \Omega \vdash M[\alpha \mapsto T] : \underline{C}$.

***Theorem* 4** (Type preservation of $\rightsquigarrow$)**:** Suppose $\varepsilon; \varepsilon; \vdash \tilde{\rho} : \Omega$ and $\varepsilon; \varepsilon; \Omega \vdash M : \underline{C}$.
If $\tilde{\rho}, M \rightsquigarrow \tilde{\rho}', M'$, then $\varepsilon; \varepsilon; \vdash \tilde{\rho}' : \Omega$ and $\varepsilon; \varepsilon; \Omega \vdash M' : \underline{C}$.

***Lemma* 4** (Context invariance)**:** Suppose $\varepsilon; \varepsilon; \Omega \uplus \Omega_1 \vdash \mathcal{E}[M_1] : \underline{C}$ and $\varepsilon; \varepsilon; \Omega_1 \vdash M_1 : \underline{D}$.
If $\varepsilon; \varepsilon; \Omega_2 \vdash M_2 : \underline{D}$ then $\varepsilon; \varepsilon; \Omega \uplus \Omega_2 \vdash \mathcal{E}[M_2] : \underline{C}$.

Theorem (Type preservation) guarantees that reduction ($\overset{\text{L}}{\rightsquigarrow}$) on a well-typed computation and model environment produces a computation and environment of the same type.

Proof of Type preservation. The proof in Appendix B.3 proceeds by: (i) proving Type preservation of $\rightsquigarrow$ for $\tilde{\rho}, M$, by induction on the reduction rules and appealing to Value substitution and Type substitution, and then (ii) using Context invariance to lift to type preservation of $\overset{\text{L}}{\rightsquigarrow}$ for $\tilde{\rho}, \mathcal{E}[M]$. □

Theorem* 5** (Type soundness):*** Suppose $\varepsilon; \varepsilon \vdash \tilde{\rho} : \Omega$ and $\varepsilon; \varepsilon; \Omega \vdash M : \underline{C}$.
Then either reduction of M diverges, or there exists $\varepsilon; \varepsilon; \Omega \vdash N : \underline{C}$ such that $\tilde{\rho}, M \overset{\text{L}}{\rightsquigarrow}{}^{*} \tilde{\rho}', N \not\overset{\text{L}}{\rightsquigarrow}$, and N is in canonical form.

Theorem (Type soundness) guarantees that iterative reduction ($\overset{\text{L}}{\rightsquigarrow}{}^{*}$) over a well-typed computation and model environment is both progressing (except when in canonical form) and type-preserving.

Proof of Type soundness. By proof of Theorem (Progress) we have that either: (i) the computation is already in canonical form where type preservation trivially holds, or (ii) the computation can be reduced one step in $\tilde{\rho}, M \overset{\text{L}}{\rightsquigarrow} \tilde{\rho}', N$ where type preservation holds by Theorem (Type preservation). In case (ii), if reduction of M does not diverge, then by induction of Theorem (Progress) we have that $\exists N . \tilde{\rho}, M \overset{\text{L}}{\rightsquigarrow}{}^{*} \tilde{\rho}', N$ where N is in canonical form, and type-preservation holds by induction of Theorem (Type preservation) on the number of reduction steps taken. □

4.5 Example: linear regression

To illustrate a probabilistic program in this calculus, we assume the existence of the term `match` as for pattern-matching on values of type `Maybe` A and `List` A, and the operator `draw` (c.f. Section 3.1) for drawing a sample from a primitive distribution given a random value $\mathsf{r} \in [0, 1]$. These constructs are given fully in the appendix in Fig. B.1.

Fig. 4.11 then implements linear regression as a multimodal model (c.f. Section 1.1), along with the handlers for multimodal distributions (c.f. Section 3.3.2) and for observing and sampling used by model simulation (c.f. Section 3.3.4); each of their type signatures make implicit use of the open effect row rule for extending closed effect rows ε by the row variable r. The handler defaultSample is deterministic, using a user-provided list of real numbers $\mathsf{r} \in [0, 1]$ to draw samples with; to thread the list as an internal state, each of its handler clauses returns a function from that list to the rest of the recursively handled program. The top-level program simLinRegr then applies the three handlers to the linear regression model, and uses the returned function f to simulate the model under a random trace.

```
linRegr  :  Double → Double ! (Dist; ε)
linRegr  =  λ(x : Double).
         let m̃u ∼ normal (0.0, 2.0) in
         let c̃ ∼ normal (0.0, 3.0) in
         let ỹ ∼ normal (mu ∗ x + c, 1.0)
         in y

handleMulDist :  ∀(α : Type).  α ! (Dist; Observe; Sample; ε) ⇒ ^Dist α ! (Observe; Sample; ε)
handleMulDist = Λ(α : Type).
       {  return x
               → return x }
    ⊎ { dist_ϕ (args_ϕ, obs_val) κ
               → match obs_val as { Nothing → let x ← sample_ϕ args_ϕ in κ x,
                                    Just  y   → let x ← observe_ϕ (args_ϕ, y) in κ x } }

defaultObserve  :  ∀(α : Type).  α ! (Observe; ε) ⇒ ^Observe α ! ε
defaultObserve  =  Λ(α : Type).
       {  return x
               → return x }
    ⊎ { observe_ϕ (args_ϕ, y) κ
               → κ y }

defaultSample  :  ∀(α : Type).  α ! (Sample; ε) ⇒ ^Sample (List Double → (α ! ε)) ! ε
defaultSample  =  Λ(α : Type).
       {  return x
               → return (λ(rss : List Double). return x) }
    ⊎ { sample_ϕ args_ϕ κ
               → return (λ(rss : List Double).
                           match rss as { (r::rs) → let y ← return (draw (ϕ args_ϕ) r)
                                                         κ′ ← κ y
                                                    in   (κ′ rs)),
               - - default to r = 0.0        Nil   → let y ← return (draw (ϕ args_ϕ) 0.0)
                                                         κ′ ← κ y
                                                    in   (κ′ Nil) } }

simLinRegr :  Double ! ε
simLinRegr =  let f : List Double → (Double ! ε)
                  f  ← with (defaultSample  Double) handle
                          (with (defaultObserve  Double) handle
                             (with (handleMulDist  Double) handle (linRegr 5.0)))
              in  f (0.36  ::  0.28  ::  0.76  ::  Nil)
```

Figure 4.11: Linear regression and handlers for model simulation.

4.6 Related work

4.6.1 Calculi for algebraic effect oriented programming

The original calculus for algebraic effects with handlers, by Plotkin and Pretnar [2009], was built on top of Levy [1999]'s paradigm of call-by-push-value which distinguishes computations M from values V. [2] This has since given rise to a range of approaches for formalising languages based on algebraic effects, two of which our calculus primarily draws from.

Kammar et al. [2013] present a type-and-effect system for a higher-order calculus of effect handlers called λ_{eff}. Like us (Fig. 4.1), they denote effects as sets of type-annotated operations, $\{\text{op} : A \longrightarrow B\} \subseteq E$, and handlers as (containing) sets of operation clauses, $\{\text{op}\, x\, k \longrightarrow M\} \subset H$. In contrast to us, they annotate the types of their handlers ($A \ {}^{E}\!\Rightarrow^{E'}\ \underline{C}$) by *both* the input and output sets of unhandled operations i.e. effect signatures. They also introduce the first operational semantics of algebraic effects *with* effect handlers, leading to later simpler variants (next) that our semantics more closely resembles.

Hillerström and Lindley [2016] describe a calculus called $\lambda_{\text{eff}}^{\rho}$ that is novel in structuring effect signatures as polymorphic rows of effects, following ideas of row-based effect types for databases [Lindley and Cheney 2012]. They, and others [Leijen 2017], have recognised row polymorphism as a natural abstraction for modelling extensibility of effects at the type-level. They thus use a kind system that, in addition to the standard Type kind for value types, includes richer kinds for rows and effects (as well as computations and handlers). Our kind system (Fig. 4.3) is a minimal version of $\lambda_{\text{eff}}^{\rho}$, e.g. excluding "presence kinds" [Rémy 1994] for denoting whether an effect is present or absent in a row, and our type language is also close. They also present a small-step operational semantics for deep effect handlers, which we have found useful (Fig. 4.10) for expressing our evaluation contexts $\mathcal{E}$, the notion of "nearest enclosing handler" in $\mathsf{Handled}(\mathcal{E})$, and the reduction of $\texttt{with}\ H\ \texttt{handle}\ M$ terms.

There are some other design considerations which calculi based on algebraic effects and effect handlers can make. One detail is the flavour of effect handler. We use *deep handlers* which recursively attempt to handle all operation calls in the program (Fig. 4.10). *Shallow handlers* instead let the programmer choose whether to resume an operation's continuation under the same handler or a different one. The deep approach is thus defined as a fold over a computation tree, which is important for the categorical semantics of handlers as "folds over algebras" [Plotkin and Pretnar 2013], whereas shallow handlers (†) are defined as case-splits which do only one level of the fold [Hillerström and Lindley 2018]:

$$\begin{array}{lll} \textsc{handle}^{\dagger}\ (\textsc{op}) & \texttt{with}\ H\ \texttt{handle}^{\dagger}\ \mathcal{E}[\text{op}\ V] \rightsquigarrow M[x \mapsto V, k \mapsto \lambda y.\, \mathcal{E}[\texttt{return}\ y]] \\ & \qquad\qquad\qquad\qquad\qquad \text{if}\ \ \{\text{op}\ x\ k \longrightarrow M\} \in H \\ & \qquad\qquad\qquad\qquad\qquad \text{and}\ \ \text{op} \notin \mathsf{Handled}(\mathcal{E}) \end{array}$$

Another design choice is how operations are called. We use *indirect style* operation calls [Bauer and Pretnar 2013], op V, where the programmer simply provides an argument V to the operation (Fig. 4.7). The alternative is *direct style* [Kammar et al. 2013], $\widehat{\text{op}}\ V\ (\lambda y.\, M)$, which allows the programmer to manually provide a continuation (it is simple to desugar op V into this form). Because direct style provides immediate access to the operation's continuation, programs that contain these can be reduced

[2] As does Levy et al. [2003]'s paradigm of fine-grain call-by-value, which we used. One difference is that call-by-push-value considers functions as computations rather than values.

without a handler in the current context:

$$\textsc{lift}\ (\widehat{\textsc{op}}) \qquad\qquad \mathcal{E}[\widehat{\mathrm{op}}\ V\ (\lambda y.\,M)] \rightsquigarrow \widehat{\mathrm{op}}\ V\ (\lambda y.\,\mathcal{E}[M])$$

There are also different ways of formalising how operations are relayed to their suitable handlers. Our use of evaluation contexts (Fig. 4.10), in the reduction rule for `with` H `handle` $\mathcal{E}[\mathrm{op}\ V]$, allows operations to be automatically relayed into the scope of the nearest appropriate handler H [Hillerström and Lindley 2016]. A semantics *without* an evaluation context requires an extra rule that specifically relays *direct style* operation calls $\widehat{\mathrm{op}}\ V\ k$, which lets handlers apply themselves to the inside of exposed continuation k if they cannot handle $\widehat{\mathrm{op}}$ [Pretnar 2015]:

$$\textsc{handle}\ (\textsc{relay}\ \widehat{\textsc{op}}) \qquad \mathtt{with}\ H\ \mathtt{handle}\ \widehat{\mathrm{op}}\ V\ (\lambda y.\,M) \rightsquigarrow \widehat{\mathrm{op}}\ V\ (\lambda y.\,\mathtt{with}\ H\ \mathtt{handle}\ M)$$
$$\text{if } \widehat{\mathrm{op}} \notin \mathrm{dom}(H)$$

Finally, there are also calculi with first-class support for operations and handlers as values. In Eff [Bauer and Pretnar 2015], deep handlers and unsaturated operation calls (i.e. op without an argument) are considered as regular values that produce computations, similarly to functions. In core Eff [Bauer and Pretnar 2013], handlers are values but unsaturated operation calls do not exist. The functional language Frank [Lindley, McBride, et al. 2017] is based solely on shallow effect handlers where there is no primitive notion of a function, but rather, a function is a special case of a handler.

4.6.2 Formalising row types for model environments

Our use of polymorphic rows was mainly for expressing extensible effect signatures (Fig. 4.9) which is key for our modular treatment of programs annotated with effect types; however, it is also easy to extend the calculus to include the usual row-typed terms such as records (products) and variants (coproducts) [Leijen 2005]. We additionally explored row polymorphism for model environments $\tilde{\rho}$, using row types to represent them as records that map conditionable variable names to values. This was successful in describing an embedding of multimodal models inside a generic host language with row types, but introduced a lot of notational overhead and intricate type rules; in particular, when viewing models as computations indexed by an effect row, that in turn, contains an effect indexed by a row of conditionable variables. Rather, we found that treating model environments as part of the metalanguage (Fig. 4.4) resulted in a more minimal, workable calculus for multimodal models. Closely related to the idea of row types for random variables is work by Lew et al. [2019]; they formalise a *trace type system* that assigns types to the named random choices made by models, thus tracking the space of possible execution traces, and then prototype this in Haskell using a package for emulating row types. We suspect this could be used to characterise multimodal model execution under a fixed model environment.

Part II

Effects and Effect Handlers for Probabilistic Inference

CHAPTER 5

A Framework for Programmable Inference

Chapter 3 used algebraic effects to support the idea of a multimodal model, which when conditioned on, specialised into a concrete probabilistic Model that samples and observes (Fig. 3.6). The definition of Model is summarised in Fig. 5.1.

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

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

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

Figure 5.1: Models as computations that sample and observe

The insight of this chapter, is that algebraic effects also offer a powerful framework for *programmable inference* over these models, as alluded to in Section 2.2. We outline our approach for this next.

5.1 Inference patterns

Programmable inference is the ability to program new inference algorithms out of reusable parts of existing ones. Our key insight is that algebraic effects seem to be a natural fit for two kinds of extensibility central to programmable inference.

First, by representing models as computations in terms of reinterpretable Sample and Observe operations (Fig.5.1), we allow them to be assigned semantics tailored to specific inference algorithms. For example, we can instrument models to record the sample traces needed for Metropolis-Hastings (Section 5.2), or arrange for models to execute stepwise rather than to completion, for particle filters (Section 5.3).

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 Propose and Accept, in the case of Metropolis-Hastings (Section 2.2) — we can turn these operations into extension points, able to be given different meanings by different members of the same broad algorithmic family. 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.

Inference Pattern	Pattern Instance
Inference skeleton Abstract inference algorithm. Given a model and a model interpreter, yields a computation expressed in terms of inference operations.	*Concrete algorithm* Instantiates the inference skeleton with a model interpreter, and post-composes with an inference handler to yield a concrete inference algorithm.
Inference operations Operations specific to inference pattern, such as proposal or resampling.	*Inference handler* Assigns a meaning to each inference operation.
Model interpreter type Pattern-specific type of model interpreters. A model interpreter assigns meaning to Sample and Observe, and executes a model to an IO action.	*Model interpreter* Model interpreter containing some behaviours specific to a concrete algorithm, and others provided by the pattern.
Auxiliary definitions Additional definitions supporting the pattern or for reuse by concrete instances.	*Auxiliary definitions* Additional definitions supporting the above.

Figure 5.2: Informal structure of inference patterns (left) and pattern instances (right)

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, shown on the left-hand side of Fig. 5.2, is what we earlier called (Section 2.2) an *inference pattern*; a library designer developing their own abstract inference algorithms using our approach would most likely follow this high-level template. Below fleshes out the idea of an inference pattern a little.

5.1.1 Inference patterns

The core of an inference pattern (Fig. 5.2, 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 (Fig. 5.1) into an IO action on each iteration. Each pattern may also provide one or more *auxiliary definitions*, including reusable inference components that make it easier to define instances.

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. Although it is possible to unify the effect signatures of the model and inference algorithm under the same computation, we find that keeping them distinct allows for a more modular design, whilst also being more performant (Section 5.5).

5.1.2 Pattern instances

A pattern instance (Fig. 5.2, 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 three inference patterns: Metropolis-Hastings (Section 5.2), Particle Filter (Section 5.3), and Guided Optimisation (Section 5.4), along with concrete pattern instances illustrating the compositionality and programmability of the approach.

5.2 Inference pattern: Metropolis-Hastings

Metropolis-Hastings algorithms [Beichl and Sullivan 2000], introduced briefly in Section 2.2, repeatedly draw samples from a chosen "proposal" distribution. By using an accept/reject scheme that determines whether to accept a new proposal and thus move to a new configuration, or to reject it and remain in the current configuration, the algorithm controls how samples are generated. 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 points, we represent them by the inference effect type Propose w in Fig. 5.3. 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, τ)) 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, 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 (and Observe) nodes are now identified by *addresses* α [Tolpin et al. 2016] of abstract type Addr, either

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 = return 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 τ = handleWith τ (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′)

random :: IO Double
```

Figure 5.3: Inference Pattern: Metropolis-Hastings

generated behind the scenes or manually assigned by the user. (See related work, Section 5.6.3.)

A trace is then a map from addresses to random values r ∈ [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, allowing the generative behaviour of the model to be controlled by providing it with specific traces. 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 (Section5.2.1) and Single-Site Metropolis-Hastings (Section 5.2.2). Our third example, Particle Metropolis-Hastings, will be defined in Section 5.3.3 after we have introduced Particle Filter.

Concrete algorithm

```
im  ::  Int  → Model a → IO [(a, (LogP, Trace ))]
im n = runImpure  ∘ 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  xwτ  xwτ′) k = do  let  (w, w′) = ((fst ∘ snd) xwτ, (fst ∘ snd) xwτ′)
                                       ratio      = w′ − w
                                  u ← call  random
                                  k  (if  exp ratio ≥ u then xwτ else xwτ′)
```

Model interpreter

```
execModel_im :: ModelExec LogP a
execModel_im τ = rassoc ∘ runImpure ∘ 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 (λ((x, w), τ) →  (x, (w, τ)))
```

Figure 5.4: Independence Metropolis as an instance of Metropolis-Hastings

5.2.1 Pattern instance: Independence Metropolis

Fig. 5.4 defines a simple Metropolis-Hastings variant called Independence Metropolis [Holden et al. 2009], 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$_{\mathrm{im}}$ 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 Section 5.3.3.) 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, the likelihood handler interprets 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, then composing the result with handlePropose$_{\mathrm{im}}$ and runImpure to yield a Markov chain of n proposals for a given model.

5.2.2 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. 5.5 defines Single-Site Metropolis-Hastings [Wingate, Stuhlmueller, et al.

Concrete algorithm

```
ssmh :: Int → Trace → Model a → IO [(a, (LPTrace, Trace ))]
ssmh n τ = runImpure ∘ handlePropose_ssmh ∘ mh n τ execModel_ssmh
```

Inference handler

```
handlePropose_ssmh :: IO ∈ fs ⇒ Handler (Propose LPTrace) fs a a
handlePropose_ssmh = handleWith α₀ (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 ∘ runImpure ∘ 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 5.5: Single-Site Metropolis-Hastings as an instance of 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$_{\text{ssmh}}$ 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 trace. [1] 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

[1] 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.

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, $\text{execModel}_{\text{ssmh}}$ records the log probabilities of both Observe and Sample operations into the map w of type LPTrace, via the handler traceLP shown in Fig. 5.5 (written using Haskell's pattern guard syntax). This deviates from the normal handler pattern, instead matching on the result of prj op from Section 2.3.1 to intercept operations of different effect types, and leaving them unhandled. Here we simply modify the continuation k to store the log probability of the value returned by the operation. The case of prj returning Nothing in **otherwise** follows the same pattern as decomp returning Left in Section 2.3.3.

All conditioning side-effects are in fact taken care of by traceLP, so the residual Observe operations are handled by defaultObserve (Fig. 3.8) 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 necessary components to derive Single-Site Metropolis-Hastings from the mh pattern.

5.3 Inference pattern: Particle Filter

Particle filters [Doucet and Johansen 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. 5.6. 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.

The inference skeleton pfilter n w_0 describes a generic particle filter, recursively running a set of n particles with a starting weight of w_0 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 intended 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

Inference skeleton

```
  pfilter   ::  (Resample w ∈ fs, IO ∈ fs)
            ⇒ Int → w → ModelStep w a → Model a → Comp fs [(a, w)]
  pfilter  n w_0 step model = do
    let  pfStep  pws = do pws′ ← call  (mapM step pws)
                          case done pws′ of
                            Just  rs    → Val rs
                            Nothing   → call  (Resample  pws′) >>= pfStep
    pfStep  (replicate  n (model, w_0))
```

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  op_es               →  Op op_es (advance w ∘ k)
```

Figure 5.6: Inference Pattern: Particle Filter

discharges the handled effect from the effect signature, and so does not support the shallow (partial) handling needed for suspensions.

We now present two concrete instances of Particle Filter, namely Multinomial Particle Filter (Section 5.3.1) and Resample-Move Particle Filter (Section 5.3.2), the latter constructed using Metropolis-Hastings. We also present Particle Metropolis-Hastings (Section5.3.3), an instance of Metropolis-Hastings constructed using Particle Filter.

5.3.1 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. 5.7. To interpret Resample pws, containing n particles ps and their weights ws, we use the categorical distribution to draw n integers from the range $[0, ..., n-1]$ with log probabilities corresponding to the normalised weights ws_{norm}. These integers, idxs, indicate the positions of particles to continue executing with, which are extracted by indexing with (!!), and then uniformly paired with the mean weight $\overline{ws}$; it is expected for particles with higher weight to be selected more than once, and unlikely ones pruned.

Concrete algorithm

```
mulpfilter   ::  Int  → Model a → IO [(a, LogP)]
mulpfilter  n  =  runImpure  ∘ 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, w̅s̅)  = (normalise ws, logMeanExp ws)
                              idxs  ← call  (replicateM  (length  ps)   (categorical  ws_norm))
                              k  (map ((, w̅s̅) ∘ ps!!) idxs)
```

Model interpreter

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

Auxiliary definitions

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

Figure 5.7: Multinomial Particle Filter as an instance of Particle Filter

For model execution, Observe is handled with advance, and Sample can be interpreted simply with defaultSample (Fig. 3.8) 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 (runImpure ∘ $\text{handleResample}_{\text{mul}}$) to specialise to a multinomial particle filter that generates n samples from the posterior and their final weights.

5.3.2 Pattern instance: Resample-Move Particle Filter

Complex inference problems often require the programmer to combine different top-level inference procedures, each 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 in Fig. 5.8 from the pfilter skeleton, providing a Resample handler which calls Single-Site Metropolis-Hastings (Section 5.2.2).

To specialise Particle Filter to use Metropolis-Hastings, we set the weight parameter w to PState, storing an execution trace alongside each particle's weight to allow for proposals. To know how far to execute a particle under a given proposal, the state variable t in the Resample handler tracks the number of observations encountered so far in the model, incremented at each resample.

To handle Resample, we unzip the particle states into their weights ws and traces τs, and use the weights to resample a selection of traces τs_{res} via the same multinomial resampling procedure used in Fig. 5.7. 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

Concrete algorithm

```
rmpf :: Int → Int → Model a → IO [(a, PState)]
rmpf n m model = (runImpure ∘ 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, w̅s̅)    = (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)
                                      return (p_mov, (w̅s̅, τ_mov)))
```

Model interpreter

```
stepModel_rmpf :: ModelStep PState a
stepModel_rmpf (p, (w, τ)) = (rassoc ∘ runImpure ∘ 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 5.8: Resample-Move Particle Filter as an instance of Particle Filter

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 corresponding trace are used.

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

5.3.3 Pattern instance: Particle Metropolis-Hastings

We are now able to revisit the Metropolis-Hastings inference pattern from Section 5.2, and show that our framework makes it equally easy to reuse a particle filter inside Metropolis-Hastings. Section 5.2 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, τ_θ 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

Concrete algorithm

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

Model interpreter

```
execModel_pmh :: Int → ModelExec LogP a
execModel_pmh n τ_θ model = do
  let stepModel_pmh :: ModelStep LogP a
      stepModel_pmh (p, w) = (fmap fst ∘ runImpure ∘ reuseTrace τ_θ ∘ advance w) p

  (xs, ws) ← (fmap unzip ∘ runImpure ∘ handleResample_mul ∘ pfilter n 0 stepModel_pmh) model
  let (ws_norm, ws̄) = (normalise ws, logMeanExp ws)
  idx ← categorical ws_norm

  return (xs !! idx, (ws̄, τ_θ))
```

Figure 5.9: Particle Metropolis-Hastings as an instance of Metropolis-Hastings

Metropolis-Hastings [Dahlin et al. 2015] and is used to reduce the variance of likelihood estimates of proposals. Fig. 5.9 derives a version of this algorithm from the Metropolis-Hastings pattern, providing a ModelExec that also calls a multinomial particle filter (Section 5.3.1), and reusing the Propose handler from Independence Metropolis (Section 5.2.1).

The model interpreter takes a number of particles n and trace τ_θ 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 τ_θ 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 τ_θ rather than the possibly extended trace returned by reuseTrace, to avoid fixing stochastic choices other than those in τ_θ (when handling Propose in Section 5.2.1).

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 of pmh initialise τ_θ, using reuseTrace empty to populate an empty trace, and then filtering to the addresses in θ.

5.4 Inference pattern: Guided Optimisation

The objective of inference so far has been to approximate the posterior distribution of a model by *generating samples* from it, a broad family of approaches known as Monte Carlo methods [Cranmer et al. 2020]. A different objective, known as *parameter estimation*, aims to numerically optimise a set of distribution parameters, which can then be interpreted as a so-called "point estimate" of the posterior. In this section, we present a final inference pattern called Guided Optimisation, an approach to parameter estimation following Bingham et al. [2019]. This pattern operates over a special kind of model that we call a *guided* model, which supports an operation called GuidedSample in addition to the usual Sample and Observe. We introduce this idea before turning to the pattern itself.

5.4.1 Guided models

Consider the linear regression model below, identical to the one in Section 2.2 except that latent variable m is described by a new operation GuidedSample, taking two distributions. The first is the prior distribution, representing the model's prior beliefs about that latent variable, as usual; the second is a *guide distribution*, or simply *guide*. Unlike the prior, which is intended to be fixed, the guide's parameters can be changed during inference. Here the parameters of the guide are initially 0 and 3; it is common (although not required) for the guide to initially match the prior, as it does below.

```
linRegr  ::  Double → Double → GuidedModel (Double, Double)
linRegr  x y = do
    m  ← call (GuidedSample (Normal 0 3) (Normal 0 3))
    c  ← call (Sample (Normal 0 2))
    call (Observe (Normal (m ∗ x + c) 1)  y)
    return  (m, c)
```

The idea of *guided optimisation* is to iteratively propose new parameters for the guide, and then, during model execution, sample from the guide rather than the prior. The extent to which these samples align with the prior and observations, provides a weighting of how well the current guide approximates the model's posterior. The algorithm then uses a *gradient update* operation, which combines the weight with some additional gradient information to determine revised values for the guide's parameters. Ideally, over many iterations, the parameters of the guide will converge on an accurate estimate of the posterior.

The type GuidedModel, given in Fig. 5.10, extends the effect signature of a probabilistic model with a new effect type GuidedSample, representing distributions with optimisable parameters which can be used to "guide" the samples of a model in this way.

```
type GuidedModel a = Comp [GuidedSample, Observe, Sample, IO] a

data GuidedSample a where
  GuidedSample  ::  (Dist  d a,  DiffDist  q  n  a)  ⇒ d → q → Addr → GuidedSample a

class  Dist  q  a  ⇒  DiffDist  q  (n  ::  Nat)  a  |  q  → n where
  δlogProb  ::  q  → a → Vec n Double
  (⊕)        ::  q  → Vec n Double → q
```

Figure 5.10: Guided models: models with optimisable distributions

Its single operation GuidedSample d q takes a regular distribution d generating samples of type a, and a guide distribution q which generates samples of the same type and has n parameters that can be optimised. The latter property is expressed by the constraint DiffDist q (n :: Nat) a, asserting that q is a *differentiable* distribution with n parameters, where n is a type-level natural number.

Instances of DiffDist q n a can use δlogProb, representing the *score function* [Fu 2006]. It is defined as follows: given that function logProb q y computes the log probability of q generating y (Fig. 5.1), then δlogProb q y computes the *partial derivatives* of the function at that point with respect to q's parameters, returning these as a vector of type Vec n Double. A vector of partial derivatives is known as a *gradient*, and can be used to adjust the distribution's n parameters via the operator (⊕) for element-wise addition (requiring those n parameters to also have type Double).

For example, below defines the normal distribution with its usual parameters of mean μ and standard deviation σ, and its type class instances for Dist and DiffDist.

```
data Normal = Normal { μ :: Double, σ :: Double }

instance Dist Normal Double where
  logProb  :: Normal → Double → Double
  logProb (Normal μ σ) y = −((y − μ)^2 / (2 ∗ σ^2)) − log (sqrt (2 ∗ pi)) − log σ

  draw :: Normal → Double → Double
  draw (Normal μ σ) r = (−invErfc (2 ∗ r)) ∗ (sqrt 2) ∗ σ + μ        where invErfc = inverse error function

instance DiffDist Normal Nat2 Double where
  δlogProb :: Normal → Double → Vec Nat2 Double
  δlogProb (Normal μ σ) y = (dμ ::: dσ ::: VNil) where
    dμ = (y − μ)/(σ^2)
    dσ = −1/σ + (y − μ)^2/(σ^3)

  (⊕) :: Normal → Vec Nat2 Double → Normal
  (Normal μ σ) ⊕ (dμ ::: dσ ::: VNil) = Normal (μ + dμ) (σ + dσ)
```

Figure 5.11: Normal distribution + type class instances for Dist and DiffDist

The type-level natural Nat2 indicates that the normal distribution has two parameters. The function δlogProb (Normal μ σ) y computes the partial derivatives of logProb (Normal μ σ) y with respect to μ and σ, returning these inside a gradient vector of size two.

5.4.2 Inference pattern: Guided Optimisation

The Guided Optimisation pattern in Fig. 5.12 learns the parameters of the guides in a GuidedModel. A configuration in Guided Optimisation is the set of guide distributions qs in the model, represented by the type Guides. The model interpreter type GuidedExec runs a model under a configuration qs, returning a weight and some *gradient information* about qs, of type ΔGuides. We explain Guides and ΔGuides next.

To store guide distributions of different types in the same data structure, and similarly for their gradients, we use Haskell's dependent map type DMap k f, which for any a, maps keys of type k a to values of type f a. Here, we specialise keys to the datatype DiffDist q n a ⇒ Key q with a constructor that stores an address α, identifying a specific GuidedSample operation. The synonym Guides then maps these keys to values of type Identity q, containing the guide distribution of type q associated with that key's address. The synonym ΔGuides maps the same keys to values of type VecFor q, containing the guide q's gradient vector of type Vec n Double. The operator ($\hat{\oplus}$) lifts (⊕) to work over values from these maps.

Inference skeleton

```
guidedOpt :: (GradUpdate ∈ fs, IO ∈ fs)
          ⇒ Int          → Int            →  Guides
          → GuidedExec a → GuidedModel a → Comp fs Guides
guidedOpt t n qs0 exec model = guidedStep 0 qs0
  where guidedStep i qs
          | i < t       = do δws ← call ((replicateM n ∘ fmap snd ∘ exec qs) model)
                             call (GradUpdate δws qs) >>= guidedStep (i + 1)
          | otherwise = return qs
```

Inference operations

```
data GradUpdate a where
  GradUpdate :: [(ΔGuides, LogP)] → Guides → GradUpdate Guides
```

Model interpreter type

```
type GuidedExec a = Guides → GuidedModel a → IO (a, (ΔGuides, LogP))
```

Auxiliary definitions

```
type Guides    = DMap Key Identity
type ΔGuides   = DMap Key VecFor
data Key q     = forall n a. DiffDist q n a ⇒ Key { α :: Addr }
data VecFor q  = forall n a. DiffDist q n a ⇒ VecFor { δ :: Vec n Double }

(⊕̂) :: DiffDist q n a ⇒ Identity q → VecFor q → Identity q
Identity q ⊕̂ VecFor δ = Identity (q ⊕ δ)

useGuides :: GuidedSample ∈ es ⇒ Guides → Comp es a → Comp es (a, ΔGuides)
useGuides qs = loop empty where
  loop δs (Val x)   = Val (x, δs)
  loop δs (Op op k) = case prj op of
    Just (GuidedSample d (q :: tyq) α) → do let kα          = (Key α :: Key tyq)
                                                Identity q′ = findWithDefault kα (Identity q) qs
                                            x ← call (GuidedSample d q′ α)
                                            let δs′ = insert kα (VecFor (δlogProb q′ x)) δs
                                            loop δs′ (k x)
    Nothing → Op op (loop δs ∘ k)
```

Figure 5.12: Inference Pattern: Guided Optimisation

The inference skeleton guidedOpt t n qs$_0$ performs t abstract updates to an initial configuration qs$_0$ by iterating guidedStep, returning the final configuration at the end. Each update has two phases. The first phase is simulation, where the model is executed n times under the current configuration qs, producing a list of n weighted gradients δws :: [ΔGuides, LogP]. The second phase is optimisation, which calls the inference pattern's key operation, GradUpdate of effect type GradUpdate, representing an approach to gradient updating. In general, GradUpdate δws qs will compute a chosen form of *gradient estimate* from the values in δws, and then combine this with a choice of *gradient descent* method to update qs.

The auxiliary function useGuides is a reusable inference component. It runs a model under a configuration qs, intercepting any GuidedSamples to use the guide parameters found in qs (in preference to those in the model), and recording the gradients of the guides at their samples. Note that useGuides does not handle any effects, and so the task of actually interpreting GuidedSample is left to a later stage.

Concrete algorithm

```
bbvi  ::  Int  → Int → Guides → GuidedModel a → IO Guides
bbvi  t  n qs_0 = runImpure ∘ handleGradEst_bbvi ∘ guidedOpt t n qs_0 execModel_bbvi
```

Inference handler

```
handleGradEst_bbvi :: Handler GradUpdate fs a a
handleGradEst_bbvi = handleWith 1 (const Val) hop where
  hop t  (GradUpdate  δws qs) k = let Δqs = fmap (map (/t)) (scoreEstimator  δws)
                                  in  k (t + 1) (intersectionWith  (⊕̂)   qs  Δqs)
```

Model interpreter

```
execModel_bbvi :: GuidedExec a
execModel_bbvi qs = sumw ∘ runImpure ∘ defaultSample ∘ likelihood ∘ latentDiff ∘ useGuides qs
  where sumw = fmap (λ ((x,  w_obs), w_lat) → (x, w_obs + w_lat))
```

Auxiliary definitions

```
latentDiff   ::  Sample ∈ es ⇒ Handler GuidedSample es a (a, LogP)
latentDiff   =  handleWith 0 (λw x → Val (x, w)) hop where
  hop w (GuidedSample d q α) k = do x ← call (Sample q α);   k (w + logProb d x - logProb q x) x

scoreEstimator   ::   [(ΔGuides,  LogP)]  → ΔGuides
```

Figure 5.13: Black Box Variational Inference as an instance of Guided Optimisation

We now present a concrete instance of Guided Optimisation called Black Box Variational Inference in Section 5.4.3. We discuss other potential instances as future work in Section 6.1.

5.4.3 Pattern instance: Black Box Variational Inference

Black Box Variational Inference [Wingate and Weber 2013; Ranganath et al. 2014] in Fig. 5.13 learns the parameters of the guides by maximising a weight called the *evidence lower bound*, which is the product of (i) the probability ratio of the guide distributions relative to the prior distributions for latent variables, and (ii) the likelihood of observed variables. These are respectively computed during model execution in $\mathsf{execModel}_{\mathsf{bbvi}}$, by using latentDiff to handle GuidedSample, and likelihood to handle Observe. Here, latentDiff replaces each operation GuidedSample d q with Sample q — to be later handled by defaultSample (Fig. 3.8) for simply drawing random samples — and accumulates the differences in log weights of d and q generating each sample.

To handle GradUpdate in $\mathsf{handleGradEst}_{\mathsf{bbvi}}$, Black Box Variational Inference uses a *score function estimator* [2] to estimate the guides' gradients; we omit the implementation of scoreEstimator, which is a standard gradient estimation method [Glynn 1990; Fu 2006]. The gradient estimates are then scaled by a learning rate of 1/t and added to the current guide parameters using intersectionWith (⊕̂), which is a vanilla approach to gradient descent.

Deriving bbvi from guidedOpt, given an initial configuration of guides qs_0, is then similar to other pattern instances. It is also straightforward to derive variants of bbvi that plug in other gradient descent approaches, like Adam [Kingma and Ba 2014] or Adagrad [Lydia and Francis 2019].

[2] also known as likelihood-ratio estimator or reinforce-style estimator

5.5 Performance evaluation

This section shows that our implementation is capable of competing with state-of-the-art probabilistic programming systems, suggesting that the choice of algebraic effects as a foundation does not imply a compromise on performance. We compare with two systems: MonadBayes[3] [Ścibior, Kammar, and Ghahramani 2018], a Haskell library that uses a monad transformer effect system, and Gen[4] [Cusumano-Towner et al. 2019], an embedded language in Julia. Both are written in general-purpose languages and were designed with programmable inference as an explicit goal.

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 models: 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 in Fig. 5.14a, our performance scales linearly across all models. Our performance remains linear when varying the number of observations provided to models in Fig. 5.14b, except for RMPF where, like MonadBayes and Gen, we scale quadratically.

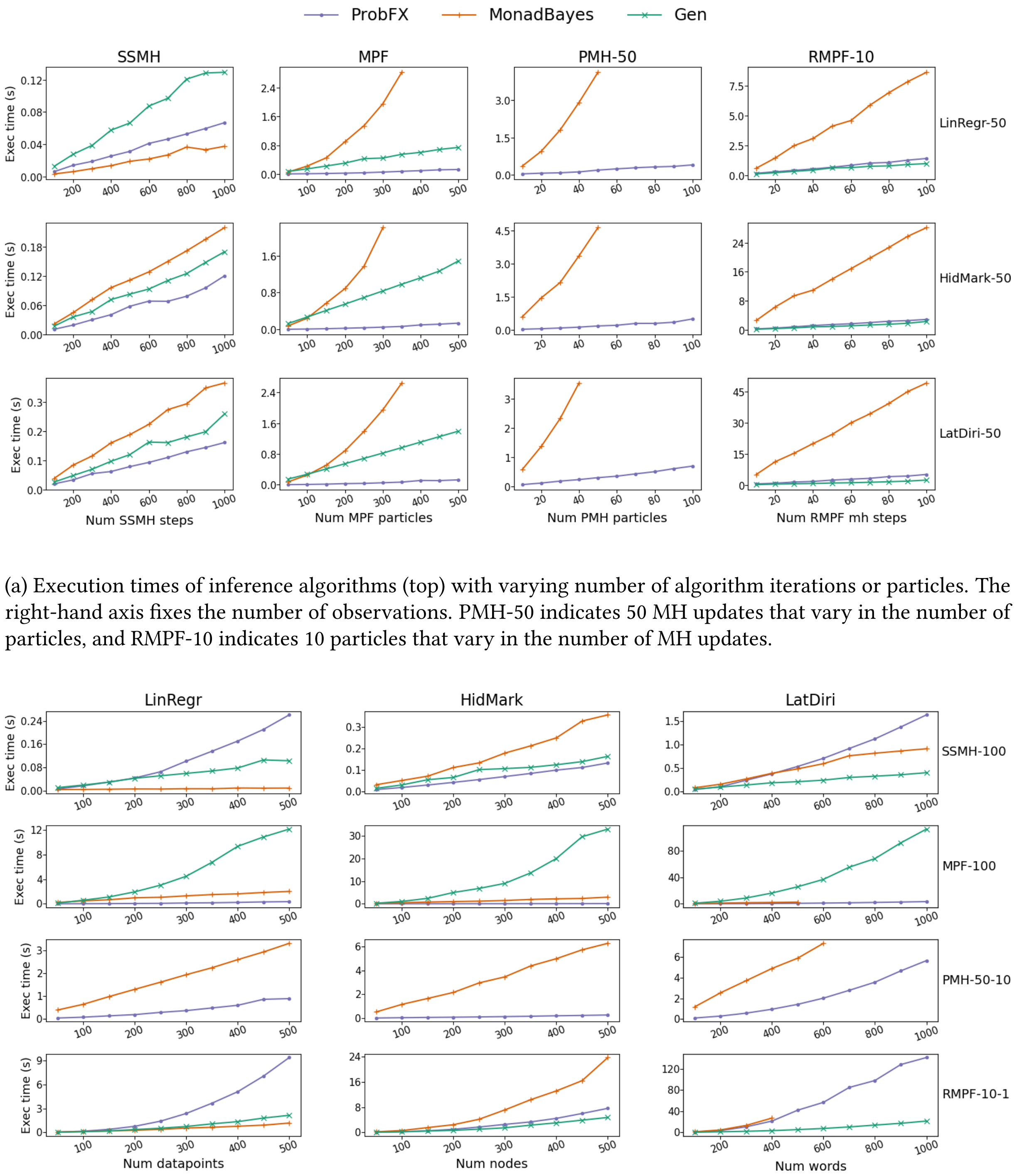
Against MonadBayes, for SSMH we are on average 15x slower for linear regression, and 1.8x faster for other models. The former result is likely because of the specific linear regression model used, which varies only in the number of observe operations, and in contrast to our implementation, their version of SSMH does not store log weights for individual observations, but instead simply sums over them. For MPF, PMH, and RMPF, we are on 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. Another task is to compare the performance of BBVI, which Gen does provide. However, their BBVI implementation is much more sophisticated than ours, for example allowing individual distribution parameters to be optimised (we default to all), and supporting several gradient descent algorithms (we use vanilla gradient descent); to derive a meaningful comparison of such inference algorithms that rely on differentiation, further work is needed (Section 6.1). Exploring performance of effect handlers with PPLs in general is also a challenging topic we plan to investigate separately, including techniques specific to algebraic effects, such as the codensity monad [Voigtländer 2008], and alternative representations of effectful programs, for example by Wu and Schrijvers [2015].

In terms of optimisations to our framework, a key performance improvement resulted by separating the effect signatures of the model and inference algorithm, allowing each one to be handled (executed) individually. If the implementation of inference were to incorporate the model's effects into its own

[3] https://github.com/tweag/monad-bayes

[4] https://github.com/probcomp/Gen.jl



(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 5.14: 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.

computation, and the interpretation of the model deferred until the handling of the inference operations, then the effect signature for inference would need to include model operations like Observe and Sample, and the resulting computation trees would be much larger; the size of the tree would then be further exacerbated by the model being run (i.e. inlined into the tree) for typically thousands of algorithm iterations. We noticed that having the effect signatures distinct made for a more efficient design, where each algorithm iteration instead fully executes the model all the way to an IO action, keeping the computation trees that are executed relatively small.

5.6 Qualitative comparison and related work

This section investigates how successfully our approach meets the needs of inference programming, contrasting the qualities of our design with some of the leading PPLs that also target this. Chapter 5 identified two key forms of extensibility central to programmable inference, of which we have seen several examples in the preceding sections:

1) *Reinterpretable models.* Inference algorithms need to be able to provide custom semantics for models. Specific algorithms require specific interpretations of Sample and Observe, as well as fine-grained control over model execution, so they can 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 different strategies for key behaviours like resampling or proposing new samples. "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, first briefly considering mainstream techniques in Section 5.6.1, which are dynamically typed, and then turning in more detail to MonadBayes in Section 5.6.2, the main existing system based on typed effects.

5.6.1 Dynamically typed approaches

The majority of programmable inference systems to date have been implemented in dynamically typed languages, which often offer flexibility at the price of a less structured approach. Here we consider Venture, Gen, Pyro, and Edward. The inference frameworks of most other mainstream PPLs, like Stan [Carpenter et al. 2017], Anglican [Tolpin et al. 2016], and Turing [Ge et al. 2018], are non-modular or black box, and not designed with inference programming in mind.

5.6.1.1 Venture

1) Reinterpretable models. Venture [Mansinghka, Selsam, et al. 2014] introduced the idea of inference programming, and represents top-level programs as interleaved sequences of *modelling instructions* and *inference instructions*. The inference instructions then affect the semantics of prior modelling code. For example, Fig. 5.15a shows a top-level program for inference on a linear regression model; it first binds the model's variables to the global environment (assume), and then interleaves new observations (observe) with invocations of Single-Site Metropolis-Hastings inference (infer), thus incorporating the observations to evolve the probability distribution over the model's executions. Although flexible, e.g. permitting sub-problems of the model to be easily inferred in isolation, it lacks a clear delination between modelling and inference provided by a more modular approach.

2) Modular, reusable algorithms. Venture achieves reuse of inference components mainly via its set of built-in *inference expressions* in Fig. 5.15b. These express a range of high-level inference procedures, like rejection sampling and Metropolis-Hastings, as low-level primitives that can be passed to the infer instruction. The idea is that given the right primitives and means of combination, more complex inference strategies can be assembled. While it is possible to add new inference primitives, these must be written in Venture's specialised DSL, and can only reuse code from that DSL. Inference programs embedded in richer languages can reuse familiar host abstractions, like pattern matching and recursion in suspendAfter (Section 5.3.2), thus avoiding the need for new DSL constructs; they can also incorporate external inference code and data structures, such as our use of dependent maps in useGuides (Section5.4.2),

```
assume m       = normal(0, 2)
assume c       = normal(0, 3)
assume linRegr = proc(x) { normal(m * x + c, 1) }
observe (linRegr 0) = 0.3
infer (mh ...)
observe (linRegr 1) = 2)4
infer (mh ...)
```

(a) Reinterpretable models: via modelling (assume, observe) and inference instructions (infer)

```
(rejection <args>)
(mh <args>)
(pgibbs <args>)
(meanfield <args>)
(nesterov <args>)
```

(b) Modular, reusable algorithms: via built-in inference expressions as primitives

Figure 5.15: Inference programming in Venture

5.6.1.2 Gen

1) Reinterpretable models. Gen [Cusumano-Towner et al. 2019] (in Julia) provides a closed interface of methods for interacting with models as black boxes, exposing certain capabilities intended for inference. Fig. 5.16a shows a subset of the interface; for example, Gen.propose selects variables of the model to propose samples for, and Gen.update updates a model's execution trace according to such a selection. The operations of the Gen interface are, however, non-programmable (have fixed meanings).

2) Modular, reusable algorithms. Gen expresses inference in terms of regular host language functions, either defined by the user or provided in the library. To illustrate, Fig.5.16b shows an excerpt of a possible Metropolis-Hastings function; the general approach is to interact with the model via Gen's bespoke model interface, and use the results to describe an algorithmic procedure. While functions are technically reusable, the lack of a type discipline means that they tend to mix arbitrary, possibly side-effectful code with model interactions, rather than being organised explicitly around the key operations of the algorithm. This can make them challenging to reuse in new contexts, and shifts the burden of identifying possible extension points onto the programmer.

```
function Gen.simulate(model, ..)

function Gen.generate(model, obs, ..)

function Gen.propose(model, ..)

function Gen.update(trace, choices, ..)
```

(a) Reinterpretable models: via the generative function interface

```
function mh(n, model, obs)
  (trace, ..) = Gen.generate(model, obs, ..)
  for i=1:n
      (trace, ..) = mhStep(trace, model)
  end

function mhStep(trace, model)
  (choices, ...) = Gen.propose(model, ...)
  (trace, ...) = Gen.update(trace, choices)
```

(b) Modular, reusable algorithms: via lightweight host language functions

Figure 5.16: Inference programming in Gen

5.6.1.3 Pyro and Edward

1) Reinterpretable models. Pyro and Edward [Bingham et al. 2019; Moore and Gorinova 2018] (in Python) interpret models by using a stack of programmable coroutines known as *context managers* [Yang et al. 2022]; the Sample calls invoked by a model's execution will then sequentially trigger each member in the stack. For instance, Fig. 5.17a shows the context manager ReplayMessenger, which when active on the stack, responds to sampling by reusing values from a given trace (akin to our reuseTrace handler).

These coroutines have some of the flavour of effect handlers, but offer a less structured approach (for assigning semantics to models). For example, the approach relies on global state to maintain the coroutine stack, where the model itself triggers each coroutine by propagating a mutable message along them; this contrasts with the algebraic effect embedding where handlers are applied directly to the model, at the top-level, using function composition. Additionally, the effectful operations (e.g. Sample) are regular Python methods with their own behaviours, in contrast to being pure syntax. There is also no type discipline for tracking effects and associating them to coroutines; thus, the operations that a coroutine interprets are arbitrary, and there is no notion of an effect being handled (fully interpreted).

2) Modular, reusable algorithms. Pyro and Edward adopt an object-oriented design that supports algorithm reuse through class inheritance. For example, the abstract class ELBO in Fig. 5.17b describes algorithms that optimise the evidence lower bound; concrete examples like Black Box Variational Inference (BBVI) are then subclasses that inherit and implement specific components, also making use of Python coroutines to interact with the model, similar to how Gen's interface is used. Using class inheritance to structure algorithms is sometimes known as the *template method* [Tokuda and Batory 2001], and as far as we know, is the closest analogue to our approach of inference patterns. As both Pyro and Edward are untyped, it can be difficult to recognise the key behaviours and effects of algorithms when implemented as programs, in a similar way to Gen.

```
class ReplayMessenger(Messenger):
  def sample(self, msg):
    variable = msg["name"]
    if variable in self.trace:
      msg["value"] = self.trace[variable]["value"]
      msg["infer"] = self.trace["infer"]
    return None
```

(a) Reinterpretable models: via programmable coroutines of the Messenger class

```
class ELBO:
  def loss(self, model, guide):

class BBVI(ELBO):
  def loss(self, model, guide):
    with SeedMessenger(...):
      with ReplayMessenger(...):
        ...
```

(b) Modular, reusable algorithms: via object-oriented class inheritance

Figure 5.17: Inference programming in Pyro

5.6.1.4 Delimiting model execution

A more specific requirement of inference programming, is control over model execution for particle stepping. This is also realised in different ways. In Gen, the programmer must parameterise their model on the number of observations to be executed (Fig. 5.18a), and manage this aspect of execution themselves when later using the particle filter. In Pyro, the situation is similar; any model executed this way must provide a method called step (Fig. 5.18b) which Pyro's particle filter relies essentially on. Other dynamically typed probabilistic languages rely on continuation-passing-style transformations of models [Tolpin et al. 2016; Goodman and Stuhlmüller 2014]. This seems to be one aspect in which algebraic effects offer a clear advantage, 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.

```
@gen function linRegr(xs, num_obs)
  m ~ normal(0, 2)
  c ~ normal(0, 3)
  for t=1:num_obs
      ys[t] ~ normal(m * xs[t] + c, 1)
  end
end
```

(a) Gen: via defining models to be parameterised by a delimiter index

```
class LinRegr:
  def init(self, xs):
    self.xs, self.t = xs, 0
    self.m = sample("m", Normal(0, 2))
    self.c = sample("c", Normal(0, 3))

  def step(self, y_t=None):
    self.t += 1
    y_t = sample("y" + str(self.t)
            , Normal(self.m * self.xs[self.t] + self.c, 1)
            , obs=y_t)
    return y_t
```

(b) Pyro: via defining models to implement the step method

Figure 5.18: Approaches to delimiting model execution

5.6.2 Monad transformer approach: MonadBayes

MonadBayes [Ścibior, Kammar, and Ghahramani 2018] is a Haskell library for typed functional programmable inference based on the Monad Transformer Library (MTL).

MTL [Gill 2022], a functional programming framework for imperative programming, lets the programmer stack monads on top of each other, producing a combined effect consisting of "layers" of elementary monadic effects called *monad transformers* [S. Liang et al. 1995]. A given set of monads may be layered in different ways; moreover 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 the library) must define how each monad 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.

5.6.2.1 Reinterpretable models

In MonadBayes, reinterpretable models are provided by MTL's support for abstract monad stacks. The constrained type (MonadSample m, MonadCond 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 MonadSample and MonadCond in Fig. 5.19a. 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 MonadSample and MonadCond in our library are the concrete datatypes Sample and Observe in Fig. 5.19b, 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 (Section 2.3.3).

While the monad transformer approach is both compositional and type-safe, the network of relaying that arises in the semantics of a model in MonadBayes is non-trivial. More than one concrete monad in the stack may perform its own Sample and Observe behaviours, such as the Traced monad in Fig. 5.19a which recursively applies rand and score to its monadic components, whereas others may choose *not* to relay their operations. As this relaying is carried out implicitly, via type class resolution, the eventual runtime behaviour of a model is sometimes far from obvious. With algebraic effects, the correspondence between modelling operations and their semantics is perhaps more obvious in the form of handlers, such as the reuseTrace and likelihood handlers in Fig. 5.19b which provide semantics for Sample and Observe.

5.6.2.2 Modular, reusable algorithms

In MonadBayes, the reusable building blocks are datatypes that implement the type classes MonadSample and MonadCond from Fig. 5.19a, 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. For example, the (simplified) type of rmpf in Fig. 5.20a, read inside-out, instantiates the supplied model to "a 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 to contain 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 determines the specific sampling and conditioning behaviours that occur at runtime and the order in which those effects should 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, *Sampling and observing as type class methods*

```
class Monad m ⇒ MonadSample m where
  rand   :: m Double

class Monad m ⇒ MonadCond m where
  score  :: LogP → m ()
```

Monad for weighting a model

```
data Weighted m a
  = W (StateT LogP m a)

instance MonadSample m
      ⇒ MonadSample (Weighted m) where
  rand = lift ∘ rand

instance Monad m
      ⇒ MonadCond (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 MonadSample m
      ⇒ MonadSample (Traced m) where
  rand = Tr rand (fmap singleton rand)

instance MonadCond m
      ⇒ MonadCond (Traced m) where
  score w = Tr (score w) (score w » return (score w))
```

(a) MonadBayes: via type classes and class instances

Sampling and observing as data constructors

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

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

Handler for weighting a model

```
likelihood  :: Handler Observe es a (a, LogP)
likelihood  = handleWith 0 hval hop where
  hval w x = Val (x, w)
  hop  w (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 hval hop where
  hval τ x = Val (x, τ)
  hop  τ (Sample d α) k =
    do r ← call random
       let (r′, τ′) = findOrInsert α r τ
       k τ′ (draw d r′)
```

(b) ProbFX: via operations and handlers

Figure 5.19: Reinterpretable models (sampling and observing) in MonadBayes vs. ProbFX

the algorithm rmpf in Fig. 5.20b is implemented by choosing a composition of handlers $\text{stepModel}_{\text{rmpf}}$ for executing the model, plus a handler $\text{handleResample}_{\text{rmpf}}$ for the inference effect. Here, constructing inference algorithms out of reusable parts is expressed mainly at the value level, via effect handler composition, without the need to adjust the types.

While each of the concrete monads in MonadBayes is by itself intuitive to use, 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 datatype in turn to provide an implementation of each supported operation in the stack. Thus programmability in MonadBayes comes with a certain cost in terms of the amount of boilerplate required. With algebraic effects, support for new semantics is possibly more lightweight by requiring only a new handler to define the relevant operations. For example, by swapping out the Resample handler in the multinomial particle filter (Section 5.3.1), our library is able to provide a number of other particle filter variants not discussed in the thesis, such as *residual* and *systematic* [Doucet and Johansen 2009], and also compose these parts to form other algorithms like Resample-Move Particle Metropolis-Hastings [Chopin et al. 2013].

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

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

(a) MonadBayes: via type-level composition of monads

```
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) ProbFX: via value-level composition of handlers

Figure 5.20: Modular, reusable algorithms in MonadBayes vs. ProbFX

5.6.2.3 Delimiting model execution

For control over model execution, like 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 (in contrast to Gen and Pyro in Section 5.6.1), the inference code requires a significant amount of plumbing which can obscure the key operations of the algorithm. Algebraic effects offer more native control, by providing access to the continuation in each handler, allowing the advertised effect signature to be more domain-specific.

5.6.3 Other related work

Algebraic effects for inference Algebraic effects as a typed functional approach to implementing inference is relatively unexplored. Some efforts have been made towards an algebraic effect translation of the MonadBayes library [Goldstein 2019], written using Koka [Leijen 2017] (instead of Haskell) which provides a native type-and-effect system. Aside from this, work by Ścibior and Kammar [2015] is the only precedent we are aware of. In Haskell, they present a basic algorithm known as rejection sampling, defining an abstract inference operation Rejection and a corresponding handler that interprets rejections by restarting model execution. In contrast to us, their inference operations are called *internally* to the model, thus growing a single computation tree that describes both model and inference algorithm; when scaling to realistically large applications, for performance, we found it necessary to keep the effect signatures (and hence computation trees) of the model and inference separate.

Object-oriented design patterns for inference Our framework of inference patterns as abstract computations, and pattern instances which handle (interpret) these computations, bears some resemblence to object-oriented programming (OOP) designs [Gamma et al. 1995] – in particular, *behavioural design patterns* for algorithms. Some of the closest examples include the Template Method, which captures the skeleton of an algorithm in the superclass and lets subclasses override specific components; the Strategy Pattern, which defines a family of algorithms as separate classes and lets their objects be interchanged; and Chain of Responsibility, which represents particular behaviors of algorithms as request handlers. We suspect OOP patterns, which are based on class hierarchies and tight coupling of data and functions, may result in overly verbose implementations of the algorithms in our (small) framework, but could be beneficial for scaling to a more complex inference system. Some key advantages of effect handlers (in typed functional programming) include its native access to the model's continuation, which is used frequently during inference (Section 5.6.1), and a structured approach for managing the algorithms' side-effects, perhaps making inference easier to reason about than in an OOP setting which relies heavily on mutability.

Addresses for random choices Section 5.2 introduced the notion of addresses α for probabilistic operations, which allow PPLs to identify the stochastic choices made during model execution. Depending on the language, addresses are either specified manually by the user or generated automatically by the language backend. Our full implementation follows Anglican [Tolpin et al. 2016], using addresses of the shape (*name, integer*), with the second element being the number of addresses so far with the same name; we generate these addresses automatically when specialising multimodal models (MulModel) to concrete models (Model), via an effect handler that implements Anglican's addressing scheme.

Languages Pyro [Bingham et al. 2019] and PyMC3 [Salvatier et al. 2016] use string values to represent addresses, and require the user to address *all* random choices in a model. Model execution is then terminated early if the same address arises twice at runtime. Our system does not respond to duplicate addresses, but doing so would be simple.

In Anglican [Tolpin et al. 2016] and Gen [Cusumano-Towner et al. 2019], the user can *choose* whether to provide an address. In Anglican, probabilistic operations without addresses are generated an address at runtime. In Gen, users can provide any value of the host language (Julia) as an address; all unaddressed operations are then treated as black box randomness and so cannot be referred to by any inference algorithms or the user. A compelling feature of Gen is its novel design of *hierarchical address spacing* that automatically distinguishes the namespaces of nested models, making it easier to compose models.

Lastly, Venture [Mansinghka, Selsam, et al. 2014] and Church [Goodman, Mansinghka, et al. 2012] keep addresses fully hidden from the user and has them generated behind the scenes. Monad-Bayes [Ścibior, Kammar, and Ghahramani 2018] has no notion of addresses at all but simply accumulates all random choices, as real numbers from the interval $[0, 1]$, into a list.

CHAPTER 6

Conclusion

Probabilistic programming languages support the construction of principled, generative models which can then be subject to Bayesian inference. But existing practical languages do not fully support reusability, abstraction, and type-safety. This thesis showed how ideas from typed functional programming can provide a solid foundation for these desirables, by using techniques of algebraic effects and effect handlers to develop a modular, type-safe language for modelling and inference.

In Chapter 3, we implemented a language for probabilistic models that are modular, first-class, and "multimodal" i.e. reusable for both simulation and inference. To support multimodality, we used algebraic effects to capture models as syntax, and effect handlers to defer their semantics to a choice of "model environment". By embedding into a functional language (Haskell), we were able to manipulate these models as first-class functions that could leverage all the features of the host language.

In Chapter 4, we formalised a minimal calculus for the modelling language based on a type-and-effect system for algebraic effects and handlers. We characterised some key abstractions of multimodal models in the metalanguage, such as model environments and conditionable variables, along with a small-step operational semantics describing how those models are executed under a model environment.

In Chapter 5, we developed a technique for type-driven and modular programmable inference. We used effect signatures to specify the key operations of abstract inference algorithms, called "inference patterns". We then used effect handlers as an intuitive interface for interpreting those operations to derive specific algorithm variants, called "pattern instances". We showed how this technique enabled some off-the-shelf algorithms to be implemented in a modular way, by tailoring and recombining their parts.

6.1 Future work and discussion

Conditionable variables: naming conflicts, multiple static uses, and matrices In Section 3.2 we introduced model environments, which express a model's conditionable variables, such as #x, as type-level strings. One topic of investigation is how naming conflicts between conditionable variables should be resolved when combining models. This is not considered an issue by most existing PPLs; many demand programmers to uniquely name each dynamic random variable instance [Salvatier et al. 2016], perhaps failing at runtime if this requirement is not met [Bingham et al. 2019]. Our language allows model environments to have typed, orthogonally combinable variables (via constraint kinds), but for further modularity, we are also considering a renaming mechanism for rebinding conditionable variables when name clashes arise.

A related topic is when the programmer statically refers to the same conditionable variable more than once in a model:

```
do x1 ⟵ normal 0 1 #x
   x2 ⟵ normal 0 2 #x
   return  (x1 + x2)
```

Technically, this is an invalid use of a random variable, resulting in an ill-formed model where #x is (confusingly) distributed according to two different distributions. For now, programmers must take care not to misuse conditionable variables in this way; a solution we intend to explore is an affine type system for model contexts which will disallow multiple static uses of the same conditionable variable.

It is also possible to use conditionable variables to represent matrices of observed values e.g. for modelling Bayesian neural networks. To do so, the programmer can define their own distribution that generates a matrix via the Dist type class (Fig. 3.1), and then associate a conditionable variable with that distribution. An alternative is to use a conditionable variable that represents single values (e.g. of type Double) and refer to it many times at runtime to manually build up a matrix. However, the second approach conflates a *matrix*, as a single observation, with a *list*, containing multiple observations (inside a model environment); this requires the programmer to carefully manage this correspondence themselves, such as avoiding providing an insufficient number of observed values, discussed next.

Running out of observations In Section 3.3, we defined effect handlers for specialising a multimodal model under an environment of observed values. Upon running out of observed values in a list (for a given conditionable variable), our implementation currently responds by calling Sample by default (Section3.3.2). While this flexibility could certainly obscure programming errors, this approach is typical in PPLs such as Turing [Ge et al. 2018] and Pyro [Bingham et al. 2019], and we also note that the correctness of inference is unaffected. Relatedly, our use of lists for representing observations is justified by the fact that, for correctness, general-purpose inference algorithms must compute the same distribution on traces i.e. sequences of sampled or observed values [Tolpin et al. 2016]. To instead statically constrain the number of observations, it may be possible in some settings to use type-level naturals; or, a dynamic check to signal when too many or too few observations are provided would be easy to implement.

Interfacing multimodal models with alternative backends We suspect it possible for our language for multimodal models (Chapter 3) to target other inference backends in general, which would allow users to work with first-class multimodal models in a typed setting, and then later execute them with more mature or preferred implementations of inference. This has been shown with MonadBayes [Ścibior, Kammar, and Ghahramani 2018] by defining effect handlers that interpret multimodal models into a Monad Transformer Library setting [Gill 2022], demonstrated in Haskell and Idris [Brady 2021] as host languages. [1] A future goal is to investigate how well this translates to inference frameworks in untyped or imperative languages, such as Anglican [Tolpin et al. 2016] or Pyro [Bingham et al. 2019].

Inference for differentiable models In Section 5.4.3, we defined a Guided Optimisation pattern instance called Black Box Variational Inference (BBVI). This is also known as *Mean Field* Variational Inference because it makes the *mean field assumption* that all sampled variables in the (guide) model

[1] https://github.com/idris-bayes/prob-fx

are independent. This assumption means that, in contrast to many other inference methods that use differentiation, BBVI only requires the score function δlogProb d y (Fig. 5.10) for differentiating logProb d y with respect to the parameters of distribution d [Wingate and Weber 2013]. Other possible instances of Guided Optimisation, such as Automatic Differentiation Variational Inference [Kucukelbir et al. 2017], Variational Autoencoding [Kingma and Welling 2013], and Maximum Likelihood Estimation [Myung 2003], do not make this assumption, and instead need to be able to differentiate logProb d y with respect to *any variable* in the model. Achieving this is generally done in PPLs by building on top of existing frameworks for automatic differentation (AD): a procedure for interpreting standard programs as differentiable functions. For instance, Gen [Cusumano-Towner et al. 2019] and Pyro [Bingham et al. 2019] build on top of the AD capabilities of Tensorflow [Abadi et al. 2016] and PyTorch [Paszke et al. 2017].

For our language to support differentiable models, one approach is to interpret a model into a form suitable for use by an existing AD library. An obstacle of this is the current lack of suitable tools in Haskell. The ad library [Kmett et al. 2010] was experimented with by Ścibior [2019], but found the resulting models to have extremely complex types; recent work by Berg et al. [2022] provides a more lightweight abstraction which seems promising. Alternatively, it may be possible to implement AD using an algebraic effect infrastructure, perhaps building on ideas from Sigal [2021] which show how to express AD with effect handlers. In all cases, we would need to investigate the set of valid operations for models in our language that are amenable to AD.

Semantics for probabilistic programs We aim to formalise some properties of our language's implementation, regarding the construction of models and the semantics of handlers that execute them during inference. For example, a restricted version of Gen [Cusumano-Towner et al. 2019] can be characterised by the trace-based denotational semantics of Lew et al. [2019]. MonadBayes [Ścibior, Kammar, and Ghahramani 2018] is based on the denotational semantics of Ścibior, Kammar, Vákár, et al. [2017], which provides modular proofs that ensure every combination of monad transformers they use is correct in producing an "unbiased sampler" for inference. It may be possible to transfer the semantics of monad transformers to the setting of algebraic effects, perhaps using the work of Schrijvers et al. [2019] that specifies when one is expressable in terms of the other. However, our use of effect handlers is not intended to correspond to the use of monads in MonadBayes, and while the algorithms we implemented are replicated from the statistical literature, working out a semantic foundation is a substantial task.

Bibliography

Abadi, Martín, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. (2016). "Tensorflow: a system for large-scale machine learning." In: *OSDI*. Vol. 16. 2016. Savannah, GA, USA, pp. 265–283.

Ackerman, Nathanael L, Cameron E Freer, and Daniel M Roy (2011). "Noncomputable conditional distributions". In: *2011 IEEE 26th Annual Symposium on Logic in Computer Science*. IEEE, pp. 107–116.

Ameen, I, Dumitru Baleanu, and Hegagi Mohamed Ali (2020). "An efficient algorithm for solving the fractional optimal control of SIRV epidemic model with a combination of vaccination and treatment". In: *Chaos, Solitons & Fractals* 137, p. 109892.

Andrieu, Christophe, Nando De Freitas, Arnaud Doucet, and Michael I Jordan (2003). "An introduction to MCMC for machine learning". In: *Machine learning* 50.1, pp. 5–43.

Bauer, Andrej and Matija Pretnar (2013). "An effect system for algebraic effects and handlers". In: *International Conference on Algebra and Coalgebra in Computer Science*. Springer, pp. 1–16.

Bauer, Andrej and Matija Pretnar (2015). "Programming with algebraic effects and handlers". In: *Journal of Logical and Algebraic Methods in Programming* 84.1. Special Issue: The 23rd Nordic Workshop on Programming Theory (NWPT 2011) Special Issue: Domains X, International workshop on Domain Theory and applications, Swansea, 5-7 September, 2011, pp. 108–123. ISSN: 2352-2208.

Beichl, Isabel and Francis Sullivan (2000). "The metropolis algorithm". In: *Computing in Science & Engineering* 2.1, pp. 65–69.

Berg, Birthe van den, Tom Schrijvers, James McKinna, and Alexander Vandenbroucke (2022). "Forward-or Reverse-Mode Automatic Differentiation: What's the Difference?" In: *arXiv preprint arXiv:2212.11088*.

Bingham, Eli, Jonathan P. Chen, Martin Jankowiak, Fritz Obermeyer, Neeraj Pradhan, Theofanis Karaletsos, Rohit Singh, Paul A. Szerlip, Paul Horsfall, and Noah D. Goodman (2019). "Pyro: Deep Universal Probabilistic Programming". In: *J. Mach. Learn. Res.* 20, 28:1–28:6.

Borgström, Johannes, Ugo Dal Lago, Andrew D Gordon, and Marcin Szymczak (2016). "A lambda-calculus foundation for universal probabilistic programming". In: *ACM SIGPLAN Notices* 51.9, pp. 33–46.

Brady, Edwin (2013). "Programming and reasoning with algebraic effects and dependent types". In: *Proceedings of the 18th ACM SIGPLAN international conference on Functional programming*, pp. 133–144.

Brady, Edwin (2021). "Idris 2: Quantitative type theory in practice". In: *arXiv preprint arXiv:2104.00480*.

Carpenter, Bob, Andrew Gelman, Matthew Hoffman, Daniel Lee, Ben Goodrich, Michael Betancourt, Marcus Brubaker, Jiqiang Guo, Peter Li, and Allen Riddell (2017). "Stan : A Probabilistic Programming Language". In: *Journal of Statistical Software* 76.

Chopin, Nicolas, Pierre E Jacob, and Omiros Papaspiliopoulos (2013). “SMC2: an efficient algorithm for sequential analysis of state space models”. In: *Journal of the Royal Statistical Society: Series B (Statistical Methodology)* 75.3, pp. 397–426.

Colmerauer, Alain (1990). “An introduction to Prolog III”. In: *Communications of the ACM* 33.7, pp. 69–90.

Cranmer, Kyle, Johann Brehmer, and Gilles Louppe (2020). “The frontier of simulation-based inference”. In: *Proceedings of the National Academy of Sciences* 117.48, pp. 30055–30062.

Cusumano-Towner, Marco (2020). “Gen: a high-level programming platform for probabilistic inference”. PhD thesis. Massachusetts Institute of Technology.

Cusumano-Towner, Marco, Feras A. Saad, Alexander Lew, and Vikash 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.* PLDI 2019. Phoenix, AZ, USA: Association for Computing Machinery, pp. 221–236. ISBN: 9781450367127.

Dahlin, Johan, Fredrik Lindsten, and Thomas B Schön (2015). “Particle Metropolis–Hastings using gradient and Hessian information”. In: *Statistics and computing* 25.1, pp. 81–92.

Dahlqvist, Fredrik, Alexandra Silva, and William Smith (2023). “Deterministic stream-sampling for probabilistic programming: semantics and verification”. In: *arXiv preprint arXiv:2304.13504.*

Darlington, John, Yi-ke Guo, Hing Wing To, and Jin Yang (Aug. 1995). “Parallel Skeletons for Structured Composition”. In: *SIGPLAN Not.* 30.8, pp. 19–28. ISSN: 0362-1340.

De Raedt, Luc, Angelika Kimmig, Hannu Toivonen, and M Veloso (2007). “ProbLog: A probabilistic Prolog and its application in link discovery”. In: *IJCAI 2007, Proceedings of the 20th international joint conference on artificial intelligence.* IJCAI-INT JOINT CONF ARTIF INTELL, pp. 2462–2467.

Ding, Zhenghao, Jun Li, and Hong Hao (2019). “Structural damage identification using improved Jaya algorithm based on sparse regularization and Bayesian inference”. In: *Mechanical Systems and Signal Processing* 132, pp. 211–231.

Djuric, P.M., J.H. Kotecha, Jianqui Zhang, Yufei Huang, T. Ghirmai, M.F. Bugallo, and J. Miguez (2003). “Particle filtering”. In: *IEEE Signal Processing Magazine* 20.5, pp. 19–38.

Doucet, Arnaud and Adam M Johansen (2009). “A tutorial on particle filtering and smoothing: Fifteen years later”. In: *Handbook of nonlinear filtering* 12.656-704, p. 3.

Erwig, Martin and Steve Kollmansberger (2006). “Functional Pearls: Probabilistic functional programming in Haskell”. In: *J. Funct. Program.* 16.1, pp. 21–34.

Fox, Charles W and Stephen J Roberts (2012). “A tutorial on variational Bayesian inference”. In: *Artificial intelligence review* 38.2, pp. 85–95.

Freeman, Phil (2017). *PureScript by Example.* URL: https://book.purescript.org.

Fu, Michael C. (2006). “Chapter 19 Gradient Estimation”. In: *Simulation.* Ed. by Shane G. Henderson and Barry L. Nelson. Vol. 13. Handbooks in Operations Research and Management Science. Elsevier, pp. 575–616.

Fushiki, Tadayoshi (2010). “Bayesian bootstrap prediction”. In: *Journal of statistical planning and inference* 140.1, pp. 65–74.

Gamma, Erich, Richard Helm, Ralph Johnson, and John Vlissides (1995). *Design patterns: elements of reusable object-oriented software.* Pearson Deutschland GmbH.

Ge, Hong, 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*, pp. 1682–1690.

Gelman, Andrew, John B Carlin, Hal S Stern, and Donald B Rubin (1995). *Bayesian data analysis.* Chapman and Hall.

Gelman, Andrew and Jennifer Hill (2006). *Data analysis using regression and multilevel/hierarchical models.* Cambridge university press.

Gilks, Walter R and Carlo Berzuini (2001). "Following a moving target—Monte Carlo inference for dynamic Bayesian models". In: *Journal of the Royal Statistical Society: Series B (Statistical Methodology)* 63.1, pp. 127–146.

Gill, Andy (2022). *Monad transformer library.* URL: https://hackage.haskell.org/package/mtl.

Giry, Michele (2006). "A categorical approach to probability theory". In: *Categorical Aspects of Topology and Analysis: Proceedings of an International Conference Held at Carleton University, Ottawa, August 11–15, 1981.* Springer, pp. 68–85.

Glynn, Peter W (1990). "Likelihood ratio gradient estimation for stochastic systems". In: *Communications of the ACM* 33.10, pp. 75–84.

Goldstein, Oliver (2019). "Modular probabilistic programming with algebraic effects". Master's thesis. University of Edinburgh.

Goodman, Noah, Vikash Mansinghka, Daniel M Roy, Keith Bonawitz, and Joshua B Tenenbaum (2012). "Church: a language for generative models". In: *arXiv preprint arXiv:1206.3255.*

Goodman, Noah and Andreas Stuhlmüller (2014). *The Design and Implementation of Probabilistic Programming Languages.* URL: http://dippl.org.

Gordon, Andrew D, Thomas A Henzinger, Aditya V Nori, and Sriram K Rajamani (2014). "Probabilistic programming". In: *Future of Software Engineering Proceedings*, pp. 167–181.

Hillerström, Daniel and Sam Lindley (2016). "Liberating effects with rows and handlers". In: *Proceedings of the 1st International Workshop on Type-Driven Development*, pp. 15–27.

Hillerström, Daniel and Sam Lindley (2018). "Shallow effect handlers". In: *Asian Symposium on Programming Languages and Systems.* Springer, pp. 415–435.

Hol, Jeroen D, Thomas B Schon, and Fredrik Gustafsson (2006). "On resampling algorithms for particle filters". In: *2006 IEEE nonlinear statistical signal processing workshop.* IEEE, pp. 79–82.

Holden, Lars, Ragnar Hauge, and Marit Holden (2009). "Adaptive independent metropolis–hastings". In: *The Annals of Applied Probability* 19.1, pp. 395–413.

Idreos, Stratos, Olga Papaemmanouil, and Surajit Chaudhuri (2015). "Overview of data exploration techniques". In: *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pp. 277–281.

Kammar, Ohad, Sam Lindley, and Nicolas Oury (2013). "Handlers in action". In: *ACM SIGPLAN Notices* 48.9, pp. 145–158.

Kingma, Diederik P and Jimmy Ba (2014). "Adam: A method for stochastic optimization". In: *arXiv preprint arXiv:1412.6980.*

Kingma, Diederik P and Max Welling (2013). "Auto-encoding variational bayes". In: *arXiv preprint arXiv:1312.6114.*

Kiselyov, Oleg (2010). “Typed Tagless Final Interpreters”. In: *Proceedings of the 2010 International Spring School Conference on Generic and Indexed Programming*. SSGIP’10. Oxford, UK: Springer-Verlag, pp. 130–174. ISBN: 9783642322013.

Kiselyov, Oleg and Hiromi Ishii (2015). “Freer Monads, More Extensible Effects”. In: *Proceedings of the 2015 ACM SIGPLAN Symposium on Haskell*. Haskell ’15. Vancouver, BC, Canada: Association for Computing Machinery, pp. 94–105. ISBN: 9781450338080.

Kiselyov, Oleg, Amr Sabry, and Cameron Swords (2013). “Extensible Effects: An Alternative to Monad Transformers”. In: *Proceedings of the 2013 ACM SIGPLAN Symposium on Haskell*. Haskell ’13. Boston, Massachusetts, USA: Association for Computing Machinery, pp. 59–70. ISBN: 9781450323833.

Kiselyov, Oleg and Chung-Chieh Shan (2009). “Embedded Probabilistic Programming”. In: *Proceedings of the IFIP TC 2 Working Conference on Domain-Specific Languages*. DSL ’09. Oxford, UK: Springer-Verlag, pp. 360–384. ISBN: 9783642030338.

Kline, Brendan and Elie Tamer (2016). “Bayesian inference in a class of partially identified models”. In: *Quantitative Economics* 7.2, pp. 329–366.

Kmett, Edward, Barak Pearlmutter, and Jeffrey Mark Siskind (2010). *ad: Automatic Differentiation*. URL: https://hackage.haskell.org/package/ad.

Kozen, Dexter (1979). “Semantics of probabilistic programs”. In: *20th Annual Symposium on Foundations of Computer Science (sfcs 1979)*. IEEE, pp. 101–114.

Kucukelbir, Alp, Dustin Tran, Rajesh Ranganath, Andrew Gelman, and David M Blei (2017). “Automatic differentiation variational inference”. In: *Journal of machine learning research*.

Leijen, Daan (2005). “Extensible records with scoped labels”. In: *Proceedings of the 2005 Symposium on Trends in Functional Programming*, pp. 297–312.

Leijen, Daan (2017). “Type directed compilation of row-typed algebraic effects”. In: *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, pp. 486–499.

Leroy, Xavier, Damien Doligez, Alain Frisch, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon (2020). *The OCaml system release 4.11: Documentation and user’s manual*.

Levy, Paul (1999). “Call-by-push-value: A subsuming paradigm”. In: *International Conference on Typed Lambda Calculi and Applications*. Springer, pp. 228–243.

Levy, Paul, John Power, and Hayo Thielecke (2003). “Modelling environments in call-by-value programming languages”. In: *Information and computation* 185.2, pp. 182–210.

Lew, Alexander, Marco Cusumano-Towner, Benjamin Sherman, Michael Carbin, and Vikash Mansinghka (2019). “Trace types and denotational semantics for sound programmable inference in probabilistic languages”. In: *Proceedings of the ACM on Programming Languages* 4.POPL, pp. 1–32.

Liang, Feng and Ziyuan Li (2021). “Statistical Analysis on COVID-19 Based on SIR Model”. In: *2020 4th International Conference on Computational Biology and Bioinformatics*. ICCBB 2020. Bali Island, Indonesia: Association for Computing Machinery, pp. 14–19. ISBN: 9781450388443.

Liang, Sheng, 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*. POPL ’95. San Francisco, California, USA: Association for Computing Machinery, pp. 333–343. ISBN: 0897916921.

Lindley, Sam and James Cheney (2012). “Row-based effect types for database integration”. In: *Proceedings of the 8th ACM SIGPLAN Workshop on Types in Language Design and Implementation*, pp. 91–102.

Lindley, Sam, Conor McBride, and Craig McLaughlin (2017). “Do Be Do Be Do”. In: *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages.* POPL ’17. Paris, France: Association for Computing Machinery, pp. 500–514. ISBN: 9781450346603.

Lunn, David J., Andrew Thomas, Nicky Best, and David Spiegelhalter (2000). “WinBUGS – A Bayesian Modelling Framework: Concepts, Structure, and Extensibility”. In: *Statistics and Computing* 10.4, pp. 325–337. ISSN: 0960-3174.

Lydia, Agnes and Sagayaraj Francis (2019). “Adagrad – an optimizer for stochastic gradient descent”. In: *Int. J. Inf. Comput. Sci* 6.5, pp. 566–568.

Mansinghka, Vikash, Ulrich Schaechtle, Shivam Handa, Alexey Radul, Yutian Chen, and Martin Rinard (June 2018). “Probabilistic Programming with Programmable Inference”. In: *SIGPLAN Not.* 53.4, pp. 603–616. ISSN: 0362-1340.

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

Meent, Jan-Willem van de, Brooks Paige, Hongseok Yang, and Frank Wood (2018). “An introduction to probabilistic programming”. In: *arXiv preprint arXiv:1809.10756.*

Moon, Todd K (1996). “The expectation-maximization algorithm”. In: *IEEE Signal processing magazine* 13.6, pp. 47–60.

Moore, Dave and Maria I Gorinova (2018). “Effect handling for composable program transformations in Edward2”. In: *arXiv preprint arXiv:1811.06150.*

Myung, In Jae (2003). “Tutorial on maximum likelihood estimation”. In: *Journal of mathematical Psychology* 47.1, pp. 90–100.

Narayanan, Praveen, Jacques Carette, Wren Romano, Chung-chieh Shan, and Robert Zinkov (2016). “Probabilistic Inference by Program Transformation in Hakaru”. In: *International Symposium on Functional and Logic Programming.* Springer, pp. 62–79. ISBN: 978-3-319-29603-6.

Nguyen, Minh, Roly Perera, Meng Wang, and Steven Ramsay (2023). “Effect handlers for programmable inference”. In: *Proceedings of the 16th ACM SIGPLAN International Symposium on Haskell.* Haskell ’23. Seattle, WA, USA: Association for Computing Machinery, pp. 44–58. ISBN: 9798400702983.

Nguyen, Minh, Roly Perera, Meng Wang, and Nicolas Wu (2022). “Modular probabilistic models via algebraic effects”. In: *Proceedings of the ACM on Programming Languages* 6.ICFP, pp. 381–410.

Paszke, Adam, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer (2017). “Automatic differentiation in pytorch”.

Plotkin, Gordon and John Power (2003). “Algebraic operations and generic effects”. In: *Applied categorical structures* 11.1, pp. 69–94.

Plotkin, Gordon and Matija Pretnar (2009). “Handlers of algebraic effects”. In: *European Symposium on Programming.* Springer, pp. 80–94.

Plotkin, Gordon and Matija Pretnar (2013). “Handling Algebraic Effects”. In: *Logical Methods in Computer Science* 9.4. Ed. by AndrzejEditor Tarlecki. ISSN: 1860-5974.

Polson, Nicholas G, James G Scott, and Jesse Windle (2013). “Bayesian inference for logistic models using Pólya–Gamma latent variables”. In: *Journal of the American statistical Association* 108.504, pp. 1339–1349.

Pretnar, Matija (2015). “An introduction to algebraic effects and handlers. invited tutorial paper”. In: *Electronic notes in theoretical computer science* 319, pp. 19–35.

Rabiner, L. and B. Juang (1986). "An introduction to hidden Markov models". In: *IEEE ASSP Magazine* 3.1, pp. 4–16.

Ranganath, Rajesh, Sean Gerrish, and David Blei (2014). "Black box variational inference". In: *Artificial intelligence and statistics*. PMLR, pp. 814–822.

Rémy, Didier (1994). "Type inference for records in a natural extension of ML". In: *Theoretical aspects of object-oriented programming*, pp. 67–96.

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

Schrijvers, Tom, Maciej Piróg, Nicolas Wu, and Mauro Jaskelioff (2019). "Monad Transformers and Modular Algebraic Effects: What Binds Them Together". In: Haskell 2019, pp. 98–113.

Ścibior, Adam (2019). "Formally justified and modular Bayesian inference for probabilistic programs". PhD thesis. University of Cambridge.

Ścibior, Adam, Zoubin Ghahramani, and Andrew D. Gordon (2015). "Practical Probabilistic Programming with Monads". In: *Proceedings of the 2015 ACM SIGPLAN Symposium on Haskell*. Haskell '15. Vancouver, BC, Canada: Association for Computing Machinery, pp. 165–176. ISBN: 9781450338080.

Ścibior, Adam and Ohad Kammar (2015). "Effects in Bayesian inference". In: *Workshop on Higher-Order Programming with Effects (HOPE)*.

Ścibior, Adam, Ohad Kammar, and Zoubin Ghahramani (2018). "Functional Programming for Modular Bayesian Inference". In: *Proc. ACM Program. Lang.* 2.ICFP.

Ścibior, Adam, 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". In: *arXiv preprint arXiv:1711.03219*.

Shi, Hongjing, Zhisheng Duan, and Guanrong Chen (2008). "An SIS model with infective medium on complex networks". In: *Physica A: Statistical Mechanics and its Applications* 387.8-9, pp. 2133–2144.

Sigal, Jesse (2021). "Automatic differentiation via effects and handlers: An implementation in Frank". In: *arXiv preprint arXiv:2101.08095*.

Snyder, Chris, Thomas Bengtsson, and Mathias Morzfeld (2015). "Performance Bounds for Particle Filters Using the Optimal Proposal". In: *Monthly Weather Review* 143.11, pp. 4750–4761.

Swierstra, Wouter (2008). "Data types à La Carte". In: *J. Funct. Program.* 18.4, pp. 423–436. ISSN: 0956-7968.

Tokuda, Lance and Don Batory (2001). "Evolving object-oriented designs with refactorings". In: *Automated Software Engineering* 8.1, pp. 89–120.

Tolpin, David, 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*. IFL 2016. Leuven, Belgium: Association for Computing Machinery. ISBN: 9781450347679.

Vidal, Germán (2022). "Explanations as Programs in Probabilistic Logic Programming". In: *International Symposium on Functional and Logic Programming*. Springer, pp. 205–223.

Voigtländer, Janis (2008). "Asymptotic improvement of computations over free monads". In: *International Conference on Mathematics of Program Construction*. Springer, pp. 388–403.

Wingate, David, Andreas Stuhlmueller, 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*. Ed. by Geoffrey Gordon, David Dunson, and Miroslav Dudík. Vol. 15. Proceedings of Machine Learning Research. Fort Lauderdale, FL, USA: PMLR, pp. 770–778.

Wingate, David and Theophane Weber (2013). "Automated variational inference in probabilistic programming". In: *arXiv preprint arXiv:1301.1299*.

Wu, Nicolas and Tom Schrijvers (2015). "Fusion for Free Efficient Algebraic Effect Handlers". In: Springer-Verlag Berlin, pp. 302–322.

Yang, Yi, Ana Milanova, and Martin Hirzel (2022). "Complex Python features in the wild". In: *Proceedings of the 19th International Conference on Mining Software Repositories*, pp. 282–293.

Yekutieli, Daniel (2012). "Adjusted Bayesian inference for selected parameters". In: *Journal of the Royal Statistical Society: Series B (Statistical Methodology)* 74.3, pp. 515–541.

Zhang, Cheng, Judith Bütepage, Hedvig Kjellström, and Stephan Mandt (2018). "Advances in variational inference". In: *IEEE transactions on pattern analysis and machine intelligence* 41.8, pp. 2008–2026.

APPENDIX A

Implementation: Elaborated definitions

```
-- | Auxiliary lifting functions
liftCall :: e ∈ es ⇒ e a → MulModel env es a
liftCall op = MulModel (call op)

liftHandler :: Handler e es a b → MulModel env (e:es) a → MulModel env es b
liftHandler hdl (MulModel m) = MulModel (hdl m)


-- | SIR model extended with the Writer effect
trans_sir :: Writer [Population] ∈ es ⇒ TransModel env es TransParams Population
trans_sir (TransParams β γ) sir = do
  sir′ ← (trans_si β >=> trans_ir γ) sir
  liftCall (Tell [sir′])
  return sir′

hmm′_sir :: (Conditionable env "ξ" Int, Conditionables env ["β", "γ", "ρ"] Double)
        ⇒ Int → Population → MulModel env es (Population, [Population])
hmm′_sir t = (liftHandler handleWriter) ∘ hmm_sir t
```

Figure A.1: Section 3.4.3 – Lifting operation calls and handlers (defined in Section 2.3) from Comp to MulModel

```
-- | Top-level Single-Site Metropolis-Hastings over a multimodal model
ssmhWith :: Int
      → MulModel env [EnvRW env, MulDist, Observe, Sample, IO] a
      → Env env
      → IO [(a, Env env)]
ssmhWith n mulmodel env = do
  let model = conditionWith env mulmodel
      τ_0   = Map.empty
  rs :: [((a, Env env), (LPTrace, Trace))] ← ssmh n τ_0 model
  return (map fst rs)
```

Figure A.2: Section 3.4.4 – Lifting ssmh (defined in Section 5.2.2) from Model to MulModel

APPENDIX **B**

FORMAL CALCULUS: THEOREM PROOFS

Inductive hypothesis (IH). Implies that ($\Rightarrow$).

B.1 Proof of Theorem (Determinism)

Suppose $\tilde{\rho}$ and M are well-typed.
If $\tilde{\rho}, M \overset{\mathrm{L}}{\rightsquigarrow} \tilde{\rho}_1, M_1$ and $\tilde{\rho}, M \overset{\mathrm{L}}{\rightsquigarrow} \tilde{\rho}_2, M_2$ then $\tilde{\rho}_1 = \tilde{\rho}_2$ and $M_1 = M_2$.

PROOF OF DETERMINISM. Suppose M is well-typed.

$$\tilde{\rho}, M \overset{\mathrm{L}}{\rightsquigarrow} \tilde{\rho}_1, M_1 \quad \wedge \quad \tilde{\rho}, M \overset{\mathrm{L}}{\rightsquigarrow} \tilde{\rho}_2, M_2$$
$$\Rightarrow \exists \mathcal{E}[N].\ M \sim \mathcal{E}[N] \qquad \text{LIFT}$$

From Unique decomposition, $\mathcal{E}[N]$ is the unique decomposition of M.

Then:

$$\tilde{\rho}, \mathcal{E}[N] \overset{\mathrm{L}}{\rightsquigarrow} \tilde{\rho}_1, \mathcal{E}[N_1] \quad \wedge \quad \tilde{\rho}, \mathcal{E}[N] \overset{\mathrm{L}}{\rightsquigarrow} \tilde{\rho}_2, \mathcal{E}[N_2]$$
$$\Rightarrow \tilde{\rho}, N \rightsquigarrow \tilde{\rho}_1, N_1 \quad \wedge \quad \tilde{\rho}, N \rightsquigarrow \tilde{\rho}_2, N_2 \qquad \text{LIFT}$$

As we only have a single reduction rule ($\rightsquigarrow$) per $\tilde{\rho}$ and redex N in Fig. 4.10, $\tilde{\rho}_1 = \tilde{\rho}_2$ and $N_1 = N_2$.

$$\begin{aligned} \text{Then } M_1 &= \mathcal{E}[N_1] \\ &= \mathcal{E}[N_2] \\ &= M_2 \end{aligned}$$

□

B.1.1 Proof of Lemma (Unique decomposition)

Suppose M is well-typed.
If $M \sim \mathcal{E}[N]$ and $M \sim \mathcal{E}'[N']$, then $\mathcal{E} = \mathcal{E}'$ and $N = N'$.

PROOF OF UNIQUE DECOMPOSITION. Suppose M is well-typed and $M \sim \mathcal{E}[N]$ and $M \sim \mathcal{E}'[N']$. We proceed by induction on the well-typed forms of M, and inversion on $M \sim \mathcal{E}[N]$ and $M \sim \mathcal{E}'[N']$.

Case $M = \texttt{return}\ V$.

The only decomposition of $M \sim \mathcal{E}[N]$ we can consider is where $\mathcal{E} = []$ and $N = M$:

$$\frac{\mathsf{redex}(\texttt{return}\ V) \Rightarrow \bot}{\texttt{return}\ V \sim [\texttt{return}\ V]}$$

but $N = \texttt{return}\ V$ is not a redex, so this decomposition is impossible. A similar case holds for $M \sim \mathcal{E}'[N']$.

Case $M = x\,V$.

The only decomposition of $M \sim \mathcal{E}[N]$ we can consider is where $\mathcal{E} = []$ and $N = M$:

$$\frac{\mathsf{redex}(x\,V) \Rightarrow \bot}{x\,V \sim [x\,V]}$$

but $N = x\,V$ is not a redex, so this decomposition is impossible. A similar case holds for $M \sim \mathcal{E}'[N']$.

Case $M = (\lambda x.\,M')\,V$.

The only decomposition of $M \sim \mathcal{E}[N]$ we can consider is where $\mathcal{E} = []$ and $N = M$:

$$\frac{\mathsf{redex}((\lambda x.\,M')\,V)}{(\lambda x.\,M')\,V \sim [(\lambda x.\,M')\,V]}$$

A similar case holds for $M \sim \mathcal{E}'[N']$ where $\mathcal{E}' = []$ and $N' = M$. By transitivity, $\mathcal{E} = \mathcal{E}'$ and $N = N'$.

Case $M = (\Lambda\alpha.\,M')\,[T]$.

The only decomposition of $M \sim \mathcal{E}[N]$ we can consider is where $\mathcal{E} = []$ and $N = M$:

$$\frac{\mathsf{redex}((\Lambda\alpha.\,M')\,[T])}{(\Lambda\alpha.\,M')\,[T] \sim [(\Lambda\alpha.\,M')\,[T]]}$$

A similar case holds for $M \sim \mathcal{E}'[N']$ where $\mathcal{E}' = []$ and $N' = M$. By transitivity, $\mathcal{E} = \mathcal{E}'$ and $N = N'$.

Case $M = \mathrm{op}\, V$.

The only decomposition of $M \sim \mathcal{E}[N]$ we can consider is where $\mathcal{E} = []$ and $N = M$:

$$\frac{\mathsf{redex}(\mathrm{op}\, V) \Rightarrow \bot}{\mathrm{op}\, V \sim [\mathrm{op}\, V]}$$

but $N = \mathrm{op}\, V$ is not a redex, so this decomposition is impossible. A similar case holds for $M \sim \mathcal{E}'[N']$.

Case $M = \phi\, V$.

The only decomposition of $M \sim \mathcal{E}[N]$ we can consider is where $\mathcal{E} = []$ and $N = M$:

$$\frac{\mathsf{redex}(\phi\, V) \Rightarrow \bot}{\phi\, V \sim [\phi\, V]}$$

but $N = \phi\, V$ is not a redex, so this decomposition is impossible. A similar case holds for $M \sim \mathcal{E}'[N']$.

Case $M = \mathtt{let}\ x \leftarrow \mathtt{return}\ V\ \mathtt{in}\ M'$.

Subcase If $M \sim \mathcal{E}[N]$ is of the form $\mathcal{E} = []$ and $N = M$:

$$\frac{\mathsf{redex}(\mathtt{let}\ x \leftarrow \mathtt{return}\ V\ \mathtt{in}\ M')}{\mathtt{let}\ x \leftarrow \mathtt{return}\ V\ \mathtt{in}\ M' \sim [\mathtt{let}\ x \leftarrow \mathtt{return}\ V\ \mathtt{in}\ M']}$$

and similarly if $M \sim \mathcal{E}'[N']$ where $\mathcal{E}' = []$ and $N' = M$, then by transitivity $\mathcal{E} = \mathcal{E}'$ and $N = N'$

Subcase If $M \sim \mathcal{E}[N]$ is of the form $\mathcal{E} = \mathtt{let}\ x \leftarrow []\ \mathtt{in}\ M'$ and $N = \mathtt{return}\ V$:

$$\frac{\dfrac{\mathsf{redex}(\mathtt{return}\ V) \Rightarrow \bot}{\mathtt{return}\ V \sim [\mathtt{return}\ V]}}{\mathtt{let}\ x \leftarrow \mathtt{return}\ V\ \mathtt{in}\ M' \sim \mathtt{let}\ x \leftarrow [\mathtt{return}\ V]\ \mathtt{in}\ M'}$$

then $\mathtt{return}\ V$ is not a redex, so this decomposition is impossible. A similar case holds for $M \sim \mathcal{E}'[N']$.

Case $M = \mathtt{let}\ x \leftarrow M_1\ \mathtt{in}\ M_2$ where $M_1 \neq \mathtt{return}\ V$.

Subcase If $M \sim \mathcal{E}[N]$ is of the form $\mathcal{E} = []$ and $N = M$:

$$\frac{\mathsf{redex}(\mathtt{let}\ x \leftarrow M_1\ \mathtt{in}\ M_2) \Rightarrow \bot}{\mathtt{let}\ x \leftarrow M_1\ \mathtt{in}\ M_2 \sim [\mathtt{let}\ x \leftarrow M_1\ \mathtt{in}\ M_2]}$$

then $N = \mathtt{let}\ x \leftarrow M_1\ \mathtt{in}\ M_2$ is not a redex as $M_1 \neq \mathtt{return}\ V$, so this decomposition is impossible. A similar case holds for $M \sim \mathcal{E}'[N']$.

Subcase If $M \sim \mathcal{E}[N]$ is of the form where $\mathcal{E} = \mathtt{let}\ x \leftarrow \mathcal{E}_0\ \mathtt{in}\ M_2$:

$$\frac{M_1 \sim \mathcal{E}_0[N]}{\mathtt{let}\ x \leftarrow M_1\ \mathtt{in}\ M_2 \sim \mathtt{let}\ x \leftarrow \mathcal{E}_0[N]\ \mathtt{in}\ M_2}$$

and similarly if $M \sim \mathcal{E}'[N']$ is of the form where $\mathcal{E}' = \mathtt{let}\ x \leftarrow \mathcal{E}'_0\ \mathtt{in}\ M_2$

$$\frac{M_1 \sim \mathcal{E}'_0[N']}{\mathtt{let}\ x \leftarrow M_1\ \mathtt{in}\ M_2 \sim \mathtt{let}\ x \leftarrow \mathcal{E}'_0[N']\ \mathtt{in}\ M_2}$$

By IH on M_1, we have by transitivity that $\mathcal{E}_0 = \mathcal{E}'_0$ and $N = N'$. (IH)

$$\begin{aligned}\text{Then } \mathcal{E} &= \mathtt{let}\ x \leftarrow \mathcal{E}_0\ \mathtt{in}\ M_2\\ &= \mathtt{let}\ x \leftarrow \mathcal{E}'_0\ \mathtt{in}\ M_2\\ &= \mathcal{E}'\end{aligned}$$

Case $M = \mathtt{let}\ x \sim \phi\, V\ \mathtt{in}\ M$

The only decomposition of $M \sim \mathcal{E}[N]$ we can consider is where $\mathcal{E} = []$ and $N = M$

$$\frac{\mathsf{redex}(\mathtt{let}\ x \sim \phi\, V\ \mathtt{in}\ M)}{\mathtt{let}\ x \sim \phi\, V\ \mathtt{in}\ M \sim [\mathtt{let}\ x \sim \phi\, V\ \mathtt{in}\ M]}$$

A similar case holds for $M \sim \mathcal{E}'[N']$ where $\mathcal{E}' = []$ and $N' = M$. By transitivity, $\mathcal{E} = \mathcal{E}'$ and $N = N'$.

Case $M = \texttt{with}\ H\ \texttt{handle}\ (\texttt{return}\ V)$

Subcase If $M \sim \mathcal{E}[N]$ is of the form $\mathcal{E} = []$ and $N = M$:

$$\frac{\mathsf{redex}(\texttt{with}\ H\ \texttt{handle}\ (\texttt{return}\ V))}{\texttt{with}\ H\ \texttt{handle}\ (\texttt{return}\ V) \sim [\texttt{with}\ H\ \texttt{handle}\ (\texttt{return}\ V)]}$$

and similarly if $M \sim \mathcal{E}'[N']$ where $\mathcal{E}' = []$ and $N' = M$, then by transitivity $\mathcal{E} = \mathcal{E}'$ and $N = N'$.

Subcase If $M \sim \mathcal{E}[N]$ is of the form $\mathcal{E} = \texttt{with}\ H\ \texttt{handle}\ []$ and $N = \texttt{return}\ V$:

$$\frac{\dfrac{\mathsf{redex}(\texttt{return}\ V) \Rightarrow \bot}{\texttt{return}\ V \sim [\texttt{return}\ V]}}{\texttt{with}\ H\ \texttt{handle}\ (\texttt{return}\ V) \sim \texttt{with}\ H\ \texttt{handle}\ [\texttt{return}\ V]}$$

then $\texttt{return}\ V$ is not a redex, so this decomposition is impossible. A similar case holds for $M \sim \mathcal{E}'[N']$.

Case $M = \texttt{with}\ H\ \texttt{handle}\ \mathcal{E}_0[\mathsf{op}\ V]$

Subcase If $M \sim \mathcal{E}[N]$ is of the form $\mathcal{E} = []$ and $N = M$:

$$\frac{\mathsf{redex}(\texttt{with}\ H\ \texttt{handle}\ \mathcal{E}_0[\mathsf{op}\ V])}{\texttt{with}\ H\ \texttt{handle}\ \mathcal{E}_0[\mathsf{op}\ V] \sim [\texttt{with}\ H\ \texttt{handle}\ \mathcal{E}_0[\mathsf{op}\ V]]}$$

and similarly if $M \sim \mathcal{E}'[N']$ where $\mathcal{E}' = []$ and $N' = M$, then by transitivity $\mathcal{E} = \mathcal{E}'$ and $N = N'$.

Subcase If $M \sim \mathcal{E}[N]$ is of the form $\mathcal{E} = \texttt{with}\ H\ \texttt{handle}\ []$ and $N = \texttt{return}\ V$:

$$\frac{\dfrac{\mathsf{redex}(\texttt{return}\ V) \Rightarrow \bot}{\texttt{return}\ V \sim [\texttt{return}\ V]}}{\texttt{with}\ H\ \texttt{handle}\ (\texttt{return}\ V) \sim \texttt{with}\ H\ \texttt{handle}\ [\texttt{return}\ V]}$$

then $\texttt{return}\ V$ is not a redex, so this decomposition is impossible. A similar case holds for $M \sim \mathcal{E}'[N']$.

Case $M = \texttt{with}\ H\ \texttt{handle}\ M'$ where $M' \neq \texttt{return}\ V$ and $M' \neq \mathcal{E}_0[\mathsf{op}\ V]$

Subcase If $M \sim \mathcal{E}[N]$ is of the form $\mathcal{E} = []$ and $N = M$:

$$\frac{\mathsf{redex}(\texttt{with}\ H\ \texttt{handle}\ M') \Rightarrow \bot}{\texttt{with}\ H\ \texttt{handle}\ M' \sim [\texttt{with}\ H\ \texttt{handle}\ M']}$$

then $N = \texttt{with}\ H\ \texttt{handle}\ M'$ is not a redex as $M' \neq \texttt{return}\ V$ and $M' \neq \mathcal{E}_0[\mathsf{op}\ V]$, so this decomposition is impossible. A similar case holds for $M \sim \mathcal{E}'[N']$.

Subcase If $M \sim \mathcal{E}[N]$ is of the form where $\mathcal{E} = \texttt{with}\ H\ \texttt{handle}\ \mathcal{E}_0$:

$$\frac{M' \sim \mathcal{E}_0[N]}{\texttt{with}\ H\ \texttt{handle}\ M' \sim \texttt{with}\ H\ \texttt{handle}\ \mathcal{E}_0[N]}$$

and similarly if $M \sim \mathcal{E}'[N']$ is of the form where $\mathcal{E}' = \texttt{with}\ H\ \texttt{handle}\ \mathcal{E}'_0$

$$\frac{M' \sim \mathcal{E}'_0[N']}{\texttt{with}\ H\ \texttt{handle}\ M' \sim \texttt{with}\ H\ \texttt{handle}\ \mathcal{E}'_0[N']}$$

By IH on M', we have by transitivity that $\mathcal{E}_0 = \mathcal{E}'_0$ and $N = N'$. (IH)

$$\begin{aligned}
\text{Then } \mathcal{E} &= \texttt{with}\ H\ \texttt{handle}\ \mathcal{E}_0 \\
&= \texttt{with}\ H\ \texttt{handle}\ \mathcal{E}'_0 \\
&= \mathcal{E}'
\end{aligned}$$

□

B.2 Proof of Theorem (Progress)

Suppose $\varepsilon; \varepsilon \vdash \tilde{\rho}_1 : \Omega$ and $\varepsilon; \varepsilon; \Omega \vdash M_1 : \underline{C}$.
Then M_1 is in canonical form, or $\exists M_2.\ \tilde{\rho}_1, M_1 \overset{\mathrm{L}}{\rightsquigarrow} \tilde{\rho}_2, M_2$.

Proof of Progress. Suppose $\varepsilon; \varepsilon \vdash \tilde{\rho}_1 : \Omega$ and $\varepsilon; \varepsilon; \Omega \vdash M_1 : \underline{C}$. We proceed by induction on the typing derivation for M_1.

Case

$$\frac{\varepsilon; \varepsilon; \Omega \vdash V : A}{\varepsilon; \varepsilon; \Omega \vdash \texttt{return}\ V : A\,!\,R}\ \textsf{return}$$

$\Rightarrow$ $\texttt{return}\ V$ is in canonical form.

Case

application

$$\frac{\varepsilon;\varepsilon;\Omega \vdash V_1 : A \to \underline{C} \qquad \varepsilon;\varepsilon;\Omega' \vdash V_2 : A}{\varepsilon;\varepsilon;\Omega \uplus \Omega' \vdash V_1\,V_2 : \underline{C}}$$

Subcase $V_1 = x$ var

$\Rightarrow$ This is impossible as V_1 is closed.

Subcase $V_1 = \lambda x : A.\, M$ function

$\Rightarrow$ $\tilde{\rho}, (\lambda x : A.\, M)\, V_2 \rightsquigarrow \tilde{\rho}, M[x \mapsto V_2]$ APP

$\Rightarrow$ $\tilde{\rho}, [(\lambda x : A.\, M)\, V_2] \overset{\mathrm{L}}{\rightsquigarrow} \tilde{\rho}, [M[x \mapsto V_2]]$ LIFT

Case

type application

$$\frac{\varepsilon;\varepsilon;\Omega \vdash V : \forall\alpha : K.\, \underline{C} \qquad \varepsilon \vdash T : K}{\varepsilon;\varepsilon;\Omega \vdash V\,[T] : \underline{C}[\alpha \mapsto T]}$$

Subcase $V = x$ var

$\Rightarrow$ This is impossible as V is closed.

Subcase $V = (\Lambda\alpha : K.\, M)$ type abstraction

$\Rightarrow$ $\tilde{\rho}, (\Lambda\alpha : K.\, M)[T] \rightsquigarrow \tilde{\rho}, M[\alpha \mapsto T]$ TY-APP

$\Rightarrow$ $\tilde{\rho}, [(\Lambda\alpha : K.\, M)[T]] \overset{\mathrm{L}}{\rightsquigarrow} \tilde{\rho}, [M[\alpha \mapsto T]]$ LIFT

Case

operation call

$$\frac{\mathrm{op} : A \to B \in E \qquad \varepsilon;\varepsilon;\Omega \vdash V : A}{\varepsilon;\varepsilon;\Omega \vdash \mathrm{op}\ V : B\,!\,(E;R)}$$

$\Rightarrow$ op V is in canonical form $\mathcal{E}[\mathrm{op}\ V]$ where op $\notin$ Handled($\mathcal{E}$), where $\mathcal{E} = [\cdot]$.

Case

distribution call

$$\frac{\phi : A \to B \in \Phi \qquad \varepsilon;\varepsilon;\Omega \vdash V : A}{\varepsilon;\varepsilon;\Omega \vdash \phi\ V : B\,!\,(\mathtt{Dist};R)}$$

$\Rightarrow$ $\tilde{\rho}, \phi\, V \rightsquigarrow \tilde{\rho}, \mathtt{dist}_{\phi}(V, \mathtt{Nothing})$ DIST CALL

$\Rightarrow$ $\tilde{\rho}, [\phi\, V] \overset{\mathrm{L}}{\rightsquigarrow} \tilde{\rho}, [\mathtt{dist}_{\phi}(V, \mathtt{Nothing})]$ LIFT

Case let-bind

$$\frac{\varepsilon;\varepsilon;\Omega \vdash M : A\,!\,R \qquad \varepsilon;\varepsilon\cdot(x:A);\Omega' \vdash N : B\,!\,R}{\varepsilon;\varepsilon;\Omega \uplus \Omega' \vdash \texttt{let}\ x \leftarrow M\ \texttt{in}\ N : B\,!\,R}$$

Subcase $M = \texttt{return}\ V$ return

$\Rightarrow\ \tilde{\rho}, \texttt{let}\ x \leftarrow \texttt{return}\ V\ \texttt{in}\ N \ \rightsquigarrow\ \tilde{\rho}, N[x \mapsto V]$ LET-BIND (RET)

$\Rightarrow\ \tilde{\rho}, [\texttt{let}\ x \leftarrow \texttt{return}\ V\ \texttt{in}\ N] \overset{\mathrm{L}}{\rightsquigarrow} \tilde{\rho}, [N[x \mapsto V]]$ LIFT

Subcase $M = \mathcal{E}'[\mathsf{op}\,V]\ \wedge\ \mathsf{op} \notin \mathsf{Handled}(\mathcal{E}')$

$\Rightarrow$ $\texttt{let}\ x \leftarrow \mathcal{E}'[\mathsf{op}\,V]\ \texttt{in}\ N$ is in canonical form $\mathcal{E}[\mathsf{op}\,V]$ where $\mathcal{E} = \texttt{let}\ x \leftarrow \mathcal{E}'\ \texttt{in}\ N$ and $\mathsf{op} \notin \mathsf{Handled}(\mathcal{E})$

Subcase $M = \mathcal{E}'[\mathsf{op}\,V]\ \wedge\ \mathsf{op} \in \mathsf{Handled}(\mathcal{E}')$

Then $\mathcal{E}$' can be refined into an innermost handle statement that can handle op :

$\Rightarrow \exists \mathcal{E}, \mathcal{E}_0, H, M_0.\ \mathcal{E}'[\mathsf{op}\,V] = \mathcal{E}[\texttt{with}\ H\ \texttt{handle}\ \mathcal{E}_0[\mathsf{op}\,V]]$

where $\{\mathsf{op}\,x_0\,k_0 \to M_0\} \in H \wedge \mathsf{op} \notin \mathsf{Handled}(\mathcal{E}_0)$ Handled

$\Rightarrow \tilde{\rho}, \texttt{with}\ H\ \texttt{handle}\ \mathcal{E}_0[\mathsf{op}\,V]$

$\rightsquigarrow \tilde{\rho}, M_0[x_0 \mapsto V, k_0 \mapsto \lambda y.\,\texttt{with}\ H\ \texttt{handle}\ \mathcal{E}_0[\texttt{return}\ y]]$ HANDLE (OP)

$\Rightarrow \tilde{\rho}, \mathcal{E}[\texttt{with}\ H\ \texttt{handle}\ \mathcal{E}_0[\mathsf{op}\,V]]$

$\overset{\mathrm{L}}{\rightsquigarrow} \tilde{\rho}, \mathcal{E}[M_0[x_0 \mapsto V, k_0 \mapsto \lambda y.\,\texttt{with}\ H\ \texttt{handle}\ \mathcal{E}_0[\texttt{return}\ y]]]$ LIFT

$\Rightarrow\ \tilde{\rho}, \texttt{let}\ x \leftarrow \mathcal{E}[\texttt{with}\ H\ \texttt{handle}\ \mathcal{E}_0[\mathsf{op}\,V]]\ \texttt{in}\ N$

$\overset{\mathrm{L}}{\rightsquigarrow} \tilde{\rho}', \texttt{let}\ x \leftarrow \mathcal{E}[M_0[x_0 \mapsto V, k_0 \mapsto \lambda y.\,\texttt{with}\ H\ \texttt{handle}\ \mathcal{E}_0[\texttt{return}\ y]]]\ \texttt{in}\ N$ LIFT

Subcase $M \neq \texttt{return}\ V$ and $M \neq \mathcal{E}'[\mathsf{op}\,V]$

M is not in canonical form, so:

$\Rightarrow\ \exists M'.\ \tilde{\rho}, M \overset{\mathrm{L}}{\rightsquigarrow} \tilde{\rho}', M'$ (IH)

$\Rightarrow \tilde{\rho}, \mathcal{E}[M_0] \overset{\mathrm{L}}{\rightsquigarrow} \tilde{\rho}', \mathcal{E}[M_0']$ where $M = \mathcal{E}[M_0]\ \wedge\ M' = \mathcal{E}[M_0']\ \wedge\ \tilde{\rho}, M_0 \rightsquigarrow \tilde{\rho}', M_0'$ LIFT

$\Rightarrow\ \tilde{\rho}, \texttt{let}\ x \leftarrow \mathcal{E}[M_0]\ \texttt{in}\ N \overset{\mathrm{L}}{\rightsquigarrow} \tilde{\rho}', \texttt{let}\ x \leftarrow \mathcal{E}[M_0']\ \texttt{in}\ N$ LIFT

Case let-bind (~)

$$\frac{\phi : A \to B \in \Phi \qquad \varepsilon;\varepsilon;\Omega \vdash V : A \qquad \varepsilon;\varepsilon\cdot(x:B);\Omega' \vdash M : C\,!\,(\texttt{Dist};R)}{\varepsilon;\varepsilon;\Omega \uplus \Omega'\cdot(\tilde{x}:B) \vdash \texttt{let}\ \tilde{x} \sim \phi\ V\ \texttt{in}\ M : C\,!\,(\texttt{Dist};R)}$$

The context $\Omega \uplus \Omega' \cdot (\tilde{x} : B)$ implies the model environment has the form $\tilde{\rho} \cdot (\tilde{x} : U)$.

$\Rightarrow\ \tilde{\rho}\cdot(\tilde{x}:U), \texttt{let}\ \tilde{\mathbf{x}} \sim \phi\ V\ \texttt{in}\ M \ \rightsquigarrow\ \tilde{\rho}\cdot(\tilde{x}:U'), \texttt{let}\ x \leftarrow \texttt{dist}_{\phi}\,(V,V')\ \texttt{in}\ M$ LET-BIND (~)

where $(V', U') = U\,!$

$\Rightarrow\ \tilde{\rho}\cdot(\tilde{x}:U), [\texttt{let}\ \tilde{\mathbf{x}} \sim \phi\ V\ \texttt{in}\ M] \overset{\mathrm{L}}{\rightsquigarrow} \tilde{\rho}\cdot(\tilde{x}:U'), [\texttt{let}\ x \leftarrow \texttt{dist}_{\phi}\,(V,V')\ \texttt{in}\ M]$ LIFT

Case
$$\begin{array}{l}\textsf{handle}\\ \dfrac{\varepsilon;\varepsilon \vdash H : A\,!\,(E;R) \Rightarrow^{E} B\,!\,R \qquad \varepsilon;\varepsilon;\Omega \vdash M : A\,!\,(E;R)}{\varepsilon;\varepsilon;\Omega \vdash \texttt{with}\ H\ \textsf{handle}\ M : B\,!\,R}\end{array}$$

Subcase $M = \textsf{return}\ V$ where $\textsf{return}\ x \longrightarrow M' \in H'$

$\Rightarrow\ \tilde{\rho}, \texttt{with}\ H\ \textsf{handle}\ (\textsf{return}\ V) \leadsto \tilde{\rho}, M'[x \mapsto V]$ HANDLE (RET)

Subcase $M = \textsf{return}\ V$ where $\textsf{return}\ x \longrightarrow M' \notin H$

This case is impossible, as handlers without return clauses are syntactically ill-formed.

Subcase $M = \mathcal{E}[\mathrm{op}\,V]$ where $\mathrm{op}\ x\ k \longrightarrow M' \in H \wedge \mathrm{op} \notin \textsf{Handled}(\mathcal{E})$

$\Rightarrow\ \tilde{\rho}, \texttt{with}\ H\ \textsf{handle}\ \mathcal{E}[\mathrm{op}\ V]$

$\qquad \leadsto \tilde{\rho}, M'[x \mapsto V, k \mapsto \lambda y.\,\texttt{with}\ H\ \textsf{handle}\ \mathcal{E}[\textsf{return}\ y]]$ HANDLE (OP)

$\Rightarrow\ \tilde{\rho}, [\texttt{with}\ H\ \textsf{handle}\ \mathcal{E}[\mathrm{op}\ V]]$

$\qquad \overset{\text{L}}{\leadsto} \tilde{\rho}, [M'[x \mapsto V, k \mapsto \lambda y.\,\texttt{with}\ H\ \textsf{handle}\ \mathcal{E}[\textsf{return}\ y]]]$ LIFT

Subcase $M = \mathcal{E}[\mathrm{op}\,V]$ where $\mathrm{op}\ x\ k \longrightarrow M' \notin H \wedge \mathrm{op} \notin \textsf{Handled}(\mathcal{E})$

$\Rightarrow$ $\texttt{with}\ H\ \textsf{handle}\ \mathcal{E}[\mathrm{op}\,V]$ is in canonical form $\mathcal{E}_0[\mathrm{op}\,V]$ where $\mathcal{E}_0 = \texttt{with}\ H\ \textsf{handle}\ \mathcal{E}$ and $\mathrm{op} \notin \textsf{Handled}(\mathcal{E}_0)$

Subcase $M = \mathcal{E}[\mathrm{op}\,V]$ where $\mathrm{op} \in \textsf{Handled}(\mathcal{E})$

Then $\mathcal{E}$ can be refined into an innermost $\textsf{handle}$ statement that can handle op:

$\Rightarrow\ \exists H', \mathcal{E}_0, \mathcal{E}_1.\ \mathcal{E}[\mathrm{op}\,V] = \mathcal{E}_0[\texttt{with}\ H'\ \textsf{handle}\ \mathcal{E}_1[\mathrm{op}\ V]]$ Handled

$\quad \wedge\ \ \mathrm{op}\ x\ k \longrightarrow M' \in H'$

$\quad \wedge\ \ \mathrm{op} \notin \textsf{Handled}(\mathcal{E}_1)$

$\Rightarrow\ \tilde{\rho}, \texttt{with}\ H'\ \textsf{handle}\ \mathcal{E}_1[\mathrm{op}\ V]$

$\qquad \leadsto \tilde{\rho}, M'[x \mapsto V, k \mapsto \lambda y.\,\texttt{with}\ H'\ \textsf{handle}\ \mathcal{E}_1[\textsf{return}\ y]]$ HANDLE (OP)

$\Rightarrow\ \tilde{\rho}, \mathcal{E}_0[\texttt{with}\ H'\ \textsf{handle}\ \mathcal{E}_1[\mathrm{op}\ V]]$

$\qquad \overset{\text{L}}{\leadsto} \tilde{\rho}', \mathcal{E}_0[M'[x \mapsto V, k \mapsto \lambda y.\,\texttt{with}\ H'\ \textsf{handle}\ \mathcal{E}_1[\textsf{return}\ y]]]$ LIFT

$\Rightarrow\ \tilde{\rho}, \texttt{with}\ H\ \textsf{handle}\ \mathcal{E}_0[\texttt{with}\ H'\ \textsf{handle}\ \mathcal{E}_1[\mathrm{op}\ V]]$

$\qquad \overset{\text{L}}{\leadsto} \tilde{\rho}', \texttt{with}\ H\ \textsf{handle}\ \mathcal{E}_0[M'[x \mapsto V, k \mapsto \lambda y.\,\texttt{with}\ H'\ \textsf{handle}\ \mathcal{E}_1[\textsf{return}\ y]]]$ LIFT

Subcase $M \neq \textsf{return}\ V$ and $M \neq \mathcal{E}[\mathrm{op}\,V]$

M is not in canonical form, so:

$\Rightarrow\ \exists M'.\ \tilde{\rho}, M \overset{\text{L}}{\leadsto} \tilde{\rho}', M'$ (IH)

$\Rightarrow \tilde{\rho}, \mathcal{E}[M_0] \overset{\text{L}}{\leadsto} \tilde{\rho}', \mathcal{E}[M_0']$ where $M = \mathcal{E}[M_0]\ \wedge\ M' = \mathcal{E}[M_0']\ \wedge\ \tilde{\rho}, M_0 \leadsto \tilde{\rho}', M_0'$ LIFT

$\Rightarrow\ \tilde{\rho}, \texttt{with}\ H\ \textsf{handle}\ \mathcal{E}[M_0] \overset{\text{L}}{\leadsto} \tilde{\rho}', \texttt{with}\ H\ \textsf{handle}\ \mathcal{E}[M_0']$ LIFT

□

B.3 Proof of Theorem (Type preservation)

Suppose $\varepsilon; \varepsilon; \vdash \tilde{\rho} : \Omega$ and $\varepsilon; \varepsilon; \Omega \vdash M : \underline{C}$.
If $\tilde{\rho}, M \overset{\mathrm{L}}{\rightsquigarrow} \tilde{\rho}', M'$, then $\varepsilon; \varepsilon; \vdash \tilde{\rho}' : \Omega$ and $\varepsilon; \varepsilon; \Omega \vdash M' : \underline{C}$.

Proof of Type preservation. Suppose $\varepsilon; \varepsilon; \vdash \tilde{\rho} : \Omega$ and $\varepsilon; \varepsilon; \Omega \vdash M : \underline{C}$.

$\tilde{\rho}, M \overset{\mathrm{L}}{\rightsquigarrow} \tilde{\rho}', M'$

$\Rightarrow \tilde{\rho}, \mathcal{E}[N] \overset{\mathrm{L}}{\rightsquigarrow} \tilde{\rho}', \mathcal{E}[N']$ where $M = \mathcal{E}[N] \;\wedge\; M' = \mathcal{E}[N'] \;\wedge\; \tilde{\rho}, N \rightsquigarrow \tilde{\rho}', N'$ lift

Suppose $\varepsilon; \varepsilon; \Omega \vdash N : \underline{D}$

By Type preservation of $\rightsquigarrow$, we have that $\varepsilon; \varepsilon; \vdash \tilde{\rho}' : \Omega$ and $\varepsilon; \varepsilon; \Omega \vdash N' : \underline{D}$

By Context invariance, we have that $\varepsilon; \varepsilon; \Omega \vdash \mathcal{E}[N'] : \underline{C}$

□

B.3.1 Proof of Theorem (Type preservation of $\rightsquigarrow$)

Suppose $\varepsilon; \varepsilon; \vdash \tilde{\rho} : \Omega$ and $\varepsilon; \varepsilon; \Omega \vdash M : \underline{C}$.
If $\tilde{\rho}, M \rightsquigarrow \tilde{\rho}', M'$, then $\varepsilon; \varepsilon; \vdash \tilde{\rho}' : \Omega$ and $\varepsilon; \varepsilon; \Omega \vdash M' : \underline{C}$.

Proof of Type preservation of $\rightsquigarrow$. Suppose $\varepsilon; \varepsilon; \vdash \tilde{\rho} : \Omega$ and $\varepsilon; \varepsilon; \Omega \vdash M : \underline{C}$ and $\tilde{\rho}, M \rightsquigarrow \tilde{\rho}', M'$. We proceed by induction on the reduction rules for $\tilde{\rho}, M$:

Case $\tilde{\rho}, (\lambda x : A.\, M)\ V \rightsquigarrow \tilde{\rho}, M[x \mapsto V]$. app

$$\frac{\varepsilon; \varepsilon \cdot (x : A); \Omega \vdash M : \underline{C} \qquad \varepsilon; \varepsilon; \Omega' \vdash V : A}{\varepsilon; \varepsilon; \Omega \uplus \Omega' \vdash (\lambda x : A.\, M)\ V : \underline{C}}\ \text{application}$$

$\Rightarrow \varepsilon; \varepsilon; \Omega \uplus \Omega' \vdash M[x \mapsto V] : \underline{C}$ (Value substitution)

Case $\tilde{\rho}, (\Lambda\alpha : K.\, M)[T] \rightsquigarrow \tilde{\rho}, M[\alpha \mapsto T]$

$$\frac{\varepsilon \cdot (\alpha : K); \varepsilon; \Omega \vdash M : \underline{C} \qquad \varepsilon \vdash T : K}{\varepsilon; \varepsilon; \Omega \vdash (\Lambda\alpha : K.\, M)\ [T] : \underline{C}[\alpha \mapsto T]}\ \text{type application}$$

$\Rightarrow \varepsilon; \varepsilon; \Omega \vdash M[\alpha \mapsto T] : \underline{C}[\alpha \mapsto T]$ (Type substitution)

Case $\quad \tilde{\rho}, \texttt{let}\ x \leftarrow \texttt{return}\ V\ \texttt{in}\ M \rightsquigarrow \tilde{\rho}, M[x \mapsto V]$

$$\dfrac{\dfrac{\varepsilon;\varepsilon;\Omega \vdash V : A}{\varepsilon;\varepsilon;\Omega \vdash \texttt{return}\ V : A\,!\,R}\ \text{return} \qquad \varepsilon;\varepsilon \cdot (x : A);\Omega' \vdash M : B\,!\,R}{\varepsilon;\varepsilon;\Omega \uplus \Omega' \vdash \texttt{let}\ x \leftarrow \texttt{return}\ V\ \texttt{in}\ M : B\,!\,R}\ \text{let-bind}$$

$\Rightarrow \quad \varepsilon;\varepsilon;\Omega \uplus \Omega' \vdash M[x \mapsto V] : B\,!\,R$ (Value substitution)

Case $\quad \tilde{\rho} \cdot (\tilde{x} : U), \texttt{let}\ \tilde{x} \sim \phi\ V\ \texttt{in}\ M \rightsquigarrow \tilde{\rho} \cdot (\tilde{x} : U'), \texttt{let}\ x \leftarrow \texttt{dist}_{\phi}(V, V')\ \texttt{in}\ M$
where $(V', U') = U\,!$

$$\overset{\text{model env (non-empty)}}{\dfrac{\varepsilon;\varepsilon \vdash \tilde{\rho} : \Omega \qquad \varepsilon;\varepsilon;\varepsilon \vdash U : \texttt{List}\ B}{\varepsilon;\varepsilon \vdash \tilde{\rho} \cdot (\tilde{x}, U) : \Omega \cdot (\tilde{x}\!: B)}} \quad \wedge \quad \overset{\text{let-bind (obs)}}{\dfrac{\phi : A \rightarrow B \in \Phi \qquad \varepsilon;\varepsilon;\Omega \vdash V : A \qquad \varepsilon;\varepsilon \cdot (x : B);\Omega' \vdash M : C\,!\,(\texttt{Dist};R)}{\varepsilon;\varepsilon;\Omega \uplus \Omega' \cdot (\tilde{x} : B) \vdash \texttt{let}\ \tilde{x} \sim \phi\ V\ \texttt{in}\ M : C\,!\,(\texttt{Dist};R)}}$$

By Definition (!), we have that $\varepsilon;\varepsilon;\varepsilon \vdash V' : \texttt{Maybe}\ B$ and $\varepsilon;\varepsilon;\varepsilon \vdash U' : \texttt{List}\ B$.

Then for preservation of environments:

$$\dfrac{\varepsilon;\varepsilon \vdash \tilde{\rho} : \Omega \qquad \varepsilon;\varepsilon;\varepsilon \vdash U' : \texttt{List}\ B}{\varepsilon;\varepsilon \vdash \tilde{\rho} \cdot (\tilde{x}, U') : \Omega \cdot (\tilde{x}\!: B)}\ \text{model env (non-empty)}$$

And for preservation of computations:

$$\dfrac{\dfrac{\dfrac{\phi : A \rightarrow B \in \Phi}{\texttt{dist}_{\phi} : A \times \texttt{Maybe}\ B \rightarrow B \in \texttt{Dist}} \qquad \varepsilon;\varepsilon;\Omega \vdash V : A \qquad \varepsilon;\varepsilon;\varepsilon \vdash V' : \texttt{Maybe}\ B}{\varepsilon;\varepsilon;\Omega \vdash \texttt{dist}_{\phi}(V, V') : B\,!\,(\texttt{Dist};R)}\ \text{operation call} \qquad \varepsilon;\varepsilon \cdot (x : B);\Omega' \vdash M : C\,!\,(\texttt{Dist};R)}{\varepsilon;\varepsilon;\Omega \uplus \Omega' \cdot (\tilde{x} : B) \vdash \texttt{let}\ x \leftarrow \texttt{dist}_{\phi}(V, V')\ \texttt{in}\ M : C\,!\,(\texttt{Dist};R)}\ \text{let-bind}$$

Case $\tilde{\rho}, \phi\, V \rightsquigarrow \tilde{\rho},\ \mathtt{dist}_{\phi}(V,\ \mathtt{Nothing})$

$$\dfrac{\phi : A \to B \in \Phi \qquad \varepsilon; \varepsilon; \Omega \vdash V : A}{\varepsilon; \varepsilon; \Omega \vdash \phi\ V : B \,!\, (\mathtt{Dist}; R)}\ \text{distribution call}$$

$$\Rightarrow \quad \dfrac{\dfrac{\phi : A \to B \in \Phi}{\mathtt{dist}_{\phi} : A \times \mathtt{Maybe}\ B \to B \in \mathtt{Dist}} \qquad \varepsilon; \varepsilon; \Omega \vdash V : A \qquad \varepsilon; \varepsilon; \varepsilon \vdash \mathtt{Nothing} : \mathtt{Maybe}\ B}{\varepsilon; \varepsilon; \Omega \vdash \mathtt{dist}_{\phi}(V,\ \mathtt{Nothing}) : B \,!\, (\mathtt{Dist}; R)}\ \text{operation call}$$

Case $\tilde{\rho}, \mathtt{with}\ H\ \mathtt{handle}\ (\mathtt{return}\ V) \rightsquigarrow \tilde{\rho}, M[x \mapsto V]$
where $\mathtt{return}\ x \longrightarrow M \in H$

$$\dfrac{\dfrac{\begin{array}{c}\varepsilon; \varepsilon \cdot (x\!: A); \varepsilon \vdash M : B \,!\, R \\ \left[\varepsilon; \varepsilon \cdot (x_i\!: A_i) \cdot (k_i\!: B_i \to B \,!\, R); \varepsilon \vdash M_i : B \,!\, R\right]_{\forall \mathrm{op}_i : A_i \to B_i \in E}\end{array}}{\varepsilon; \varepsilon \vdash \{\mathtt{return}\ x \to M\} \uplus \{\mathrm{op}_i\ x_i\ k_i \to M_i\} : A \,!\, (E; R) \Rightarrow^{E} B \,!\, R}\ \text{handler} \qquad \dfrac{\varepsilon; \varepsilon; \Omega \vdash V : A}{\varepsilon; \varepsilon; \Omega \vdash \mathtt{return}\ V : A \,!\, (E; R)}\ \text{return}}{\varepsilon; \varepsilon; \Omega \vdash \mathtt{with}\ \{\mathtt{return}\ x \to M\} \uplus \{\mathrm{op}_i\ x_i\ k_i \to M_i\}\ \mathtt{handle}\ (\mathtt{return}\ V) : B \,!\, R}\ \text{handle}$$

$\Rightarrow \ \varepsilon; \varepsilon; \Omega \vdash M[x \mapsto V] : B \,!\, R$ (Value substitution)

Case

$$\tilde{\rho}, \mathtt{with}\ H\ \mathtt{handle}\ \mathcal{E}[\mathsf{op}\ V] \rightsquigarrow \tilde{\rho}, M[x \mapsto V, k \mapsto \lambda y.\,\mathtt{with}\ H\ \mathtt{handle}\ \mathcal{E}[\mathtt{return}\ y]]$$

$$\begin{array}{ll} \text{where} & \mathsf{op}\ x\ k \to M \in H \\ \land & \mathsf{op} \notin \mathsf{Handled}(\mathcal{E}) \end{array}$$

$$\dfrac{\dfrac{\begin{array}{c} E = \{\mathsf{op} : A_{\mathsf{op}} \to B_{\mathsf{op}}\} \uplus E' \\ \varepsilon;\varepsilon \cdot (x{:}\ A_{\mathsf{op}}) \cdot (k{:}\ B_{\mathsf{op}} \to B\,!\,R);\varepsilon \vdash M : B\,!\,R \\ \varepsilon;\varepsilon \vdash H' : A\,!\,(E';R) \Rightarrow^{E'} B\,!\,R \end{array}}{\varepsilon;\varepsilon \vdash \{\mathsf{op}\ x\ k \to M\} \uplus H' : A\,!\,(E;R) \Rightarrow^{E} B\,!\,R}\ \text{handler} \qquad \varepsilon;\varepsilon;\Omega \vdash \mathcal{E}[\mathsf{op}\ V] : A\,!\,(E;R)}{\varepsilon;\varepsilon;\Omega \vdash \mathtt{with}\ \{\mathsf{op}\ x\ k \to M\} \uplus H'\ \mathtt{handle}\ \mathcal{E}[\mathsf{op}\ V] : B\,!\,R}\ \text{handle} \tag{B.1}$$

- Given $\varepsilon;\varepsilon;\Omega \vdash \mathcal{E}[\mathsf{op}\ V]$, there must exist a decomposition $\Omega = \Omega_1 \uplus \Omega_2$ such that $\varepsilon;\varepsilon;\Omega_2 \vdash \mathsf{op}\ V$.

  Suppose any such Ω_1, Ω_2:

$$\varepsilon;\varepsilon;\Omega_1 \uplus \Omega_2 \vdash \mathcal{E}[\mathsf{op}\ V] : A\,!\,(E;R) \tag{B.2}$$

$$\land \quad \dfrac{\dfrac{E = \{\mathsf{op} : A_{\mathsf{op}} \to B_{\mathsf{op}}\} \uplus E'}{\mathsf{op} : A_{\mathsf{op}} \to B_{\mathsf{op}} \in E} \qquad \varepsilon;\varepsilon;\Omega_2 \vdash V : A_{\mathsf{op}}}{\varepsilon;\varepsilon;\Omega_2 \vdash \mathsf{op}\ V : B_{\mathsf{op}}\,!\,(E;R)}\ \text{operation call} \tag{B.3}$$

- Next, we also have that:

$$\dfrac{\dfrac{y : B_{\mathsf{op}} \in \varepsilon \cdot (y : B_{\mathsf{op}})}{\varepsilon;\varepsilon \cdot (y : B_{\mathsf{op}});\varepsilon \vdash y : B_{\mathsf{op}}}\ \text{var}}{\varepsilon;\varepsilon \cdot (y : B_{\mathsf{op}});\varepsilon \vdash \mathtt{return}\ y : B_{\mathsf{op}}\,!\,(E;R)}\ \text{return} \tag{B.4}$$

  By context invariance over $\mathcal{E}[\mathsf{op}\ V]$ from (B.2) where we plug in (B.4) for (B.3), we have:

$$\varepsilon;\varepsilon \cdot (y : B_{\mathsf{op}});\Omega_1 \vdash \mathcal{E}[\mathtt{return}\ y] : A\,!\,(E;R)$$

  (B.2, B.3, B.4, Context invariance)

  We can then derive the type of the continuation as:

$$\dfrac{\dfrac{\varepsilon;\varepsilon \vdash H : A\,!\,(E;R) \Rightarrow^{E} B\,!\,R \qquad \varepsilon;\varepsilon \cdot (y : B_{\mathsf{op}});\Omega_1 \vdash \mathcal{E}[\mathtt{return}\ y] : A\,!\,(E;R)}{\varepsilon;\varepsilon \cdot (y : B_{\mathsf{op}});\Omega_1 \vdash \mathtt{with}\ H\ \mathtt{handle}\ \mathcal{E}[\mathtt{return}\ y] : B\,!\,R}\ \text{handle}}{\varepsilon;\varepsilon;\Omega_1 \vdash \lambda y.\,\mathtt{with}\ H\ \mathtt{handle}\ \mathcal{E}[\mathtt{return}\ y] : B_{\mathsf{op}} \to B\,!\,R}\ \text{function} \tag{B.5}$$

Finally, by substitution of (B.3) and (B.5) into M from (B.1), we have:

$$\varepsilon;\varepsilon \cdot (x{:}\ A_{\mathsf{op}}) \cdot (k{:}\ B_{\mathsf{op}} \to B\,!\,R);\varepsilon \vdash M : B\,!\,R$$

(B.1)

$$\Rightarrow \quad \varepsilon;\varepsilon \cdot (k{:}\ B_{\mathsf{op}} \to B\,!\,R);\Omega_2 \vdash M[x \mapsto V] : B\,!\,R$$

(B.3 Value substitution)

$$\Rightarrow \quad \varepsilon;\varepsilon;\Omega_1 \uplus \Omega_2 \vdash M[x \mapsto V, k \mapsto \lambda y.\,\mathtt{with}\ H\ \mathtt{handle}\ \mathcal{E}[\mathtt{return}\ y]] : B\,!\,R$$

(B.5, Value substitution)

$$\Rightarrow \quad \varepsilon;\varepsilon;\Omega \vdash M[x \mapsto V, k \mapsto \lambda y.\,\mathtt{with}\ H\ \mathtt{handle}\ \mathcal{E}[\mathtt{return}\ y]] : B\,!\,R$$

□

$$
\begin{array}{llll}
\text{Value} & V, U & ::= & \ldots \\
 & & & \mathtt{draw}\ (\phi\ V_1)\ V_2 \qquad\qquad \text{draw} \\[1ex]
\text{Computation} & M, N & ::= & \ldots \\
 & & & \mathtt{match}\ V\ \mathtt{as}\ \{\mathtt{Just}\ x \rightarrow M_1,\ \mathtt{Nothing} \rightarrow M_2\} \quad \text{match (maybe)} \\
 & & & \mathtt{match}\ V\ \mathtt{as}\ \{x :: xs \rightarrow M_1,\ \mathtt{Nil} \rightarrow M_2\} \quad \text{match (list)}
\end{array}
$$

(a) Syntax

$\boxed{\Delta; \Gamma; \Omega \vdash V : A}$

draw

$$\frac{\phi : A \rightarrow B \in \Phi \qquad \Delta; \Gamma; \Omega_1 \vdash V_1 : A \qquad \Delta; \Gamma; \Omega_2 \vdash V_2 : \mathtt{Double} \qquad 0 \leq V_2 \leq 1}{\Delta; \Gamma; \Omega_1 \uplus \Omega_2 \vdash \mathtt{draw}\ (\phi\ V_1)\ V_2 : B}$$

$\boxed{\Delta; \Gamma; \Omega \vdash M : \underline{C}}$

match (maybe)

$$\frac{\begin{array}{c}\Delta; \Gamma; \Omega_1 \vdash V : \mathtt{List}\ A \\ \Delta; \Gamma \cdot (x : A) \cdot (xs : \mathtt{List}\ A); \Omega_2 \vdash M_1 : B \,!\, R \qquad \Delta; \Gamma \cdot (x : A) \cdot (xs : \mathtt{List}\ A); \Omega_2 \vdash M_2 : B \,!\, R\end{array}}{\Delta; \Gamma; \Omega_1 \uplus \Omega_2 \vdash \mathtt{match}\ V\ \mathtt{as}\ \{x :: xs \rightarrow M_1,\ \mathtt{Nil} \rightarrow M_2\} : B \,!\, R}$$

match (list)

$$\frac{\Delta; \Gamma; \Omega_1 \vdash V : \mathtt{Maybe}\ A \qquad \Delta; \Gamma \cdot (x : A); \Omega_2 \vdash M_1 : B \,!\, R \qquad \Delta; \Gamma \cdot (x : A); \Omega_2 \vdash M_2 : B \,!\, R}{\Delta; \Gamma; \Omega_1 \uplus \Omega_2 \vdash \mathtt{match}\ V\ \mathtt{as}\ \{\mathtt{Just}\ x \rightarrow M_1,\ \mathtt{Nothing} \rightarrow M_2\} : B \,!\, R}$$

(b) Type rules

$\boxed{\tilde{\rho}, M \rightsquigarrow \tilde{\rho}', M'}$

$$
\begin{array}{lrl}
\textsc{match (just)} & & \tilde{\rho}, \mathtt{match}\ \mathtt{Just}\ V\ \mathtt{as}\ \{\mathtt{Just}\ x \rightarrow M_1,\ \mathtt{Nothing} \rightarrow M_2\} \\
 & \rightsquigarrow & \tilde{\rho}, M_1[x \mapsto V] \\
\textsc{match (nothing)} & & \tilde{\rho}, \mathtt{match}\ \mathtt{Nothing}\ \mathtt{as}\ \{\mathtt{Just}\ x \rightarrow M_1,\ \mathtt{Nothing} \rightarrow M_2\} \\
 & \rightsquigarrow & \tilde{\rho}, M_2 \\
\textsc{match (cons)} & & \tilde{\rho}, \mathtt{match}\ V_1 :: V_2\ \mathtt{as}\ \{x :: xs \rightarrow M_1,\ \mathtt{Nil} \rightarrow M_2\} \\
 & \rightsquigarrow & \tilde{\rho}, M_1[x \mapsto V_1, xs \mapsto V_2] \\
\textsc{match (nil)} & & \tilde{\rho}, \mathtt{match}\ \mathtt{Nil}\ \mathtt{as}\ \{x :: xs \rightarrow M_1,\ \mathtt{Nil} \rightarrow M_2\} \\
 & \rightsquigarrow & \tilde{\rho}, M_2
\end{array}
$$

(c) Small-step operational semantics

Figure B.1: Extended calculus with `match-as` and `draw`. The semantics for `draw` is omitted, being specific to each possible primitive distribution ϕ.