Anik Roy

A probabilistic programming language in OCaml

Computer Science Tripos - Part II

Christ's College

May 8, 2020

Declaration of Originality

I, Anik Roy of Christ's College, being a candidate for Part II of the Computer Science Tripos, hereby declare that this dissertation and the work described in it are my own work, unaided except as may be specified below, and that the dissertation does not contain material that has already been used to any substantial extent for a comparable purpose.

I, Anik Roy of Christ's College, am content for my dissertation to be made available to the students and staff of the University.

Signed: 

Date: May 8, 2020

Proforma

Candidate Number:	2349E
Project Title:	A Probabilistic Programming Language in OCaml
Examination:	Computer Science Tripos Part II - 2020
Word Count:	11998 [1]
Final Line Count:	4533 [2]
Project Originator:	Dr R. Mortier
Supervisor:	Dr R. Mortier

Original Aims of the Project

The original aim was to design and implement a language in OCaml, which allows the user to specify probabilistic models naturally in code, and perform inference on these models automatically in order to separate the concerns of defining generative models and performing inference. Exact Bayesian inference is intractable in practice, so I needed to develop approximate algorithms. I aimed to allow users to build models from a set of primitive distributions, and choose from a selection of inference algorithms to perform inference. An efficient and expressive way to represent and combine distributions needed to be developed.

Work Completed

I have achieved all the initial requirements. My PPL exists as a shallowly embedded DSL in OCaml, is able to represent a wide variety of models, and is not limited to finite graphical models or discrete distributions. I have implemented several inference procedures, exceeding my initial requirements, and shown that they are correct using statistical tests on simple problems solved analytically. I represent distributions as a GADT which is a monad, allowing distributions to be combined to build up models. I have also included functionality to easily create plots from distributions.

Special Difficulties

None

[1] This word count was computed by `texcount -inc -sum -1 diss.tex` and only includes the main body, excluding code listings

[2] This line count was computed by `cloc` and excludes blank lines and comments

Contents

List of Figures

List of Listings

List of Tables

1 Introduction

This dissertation presents a universal probabilistic programming language, OwlPPL, implemented as a domain specific language embedded within OCaml. I discuss the design decisions involved in developing this system. and evaluate its performance and suitability as a tool to be used in data science.

1.1 Motivation

Creating statistical models and performing inference on these models is key to data science. Such modelling involves formulating a prior belief over some parameters, the generative model ($p(x)$), and a set of conditions ($p(y|x)$) which specify the likelihood of observed data given the parameters. The goal is then to find the posterior, the (inferred) distribution over the parameters given the data we observe ($p(x|y)$). In general, this kind of Bayesian inference is intractable, so we must use methods which approximate the posterior.

A probabilistic programming language (PPL) is a language used to create complex models through composition of simpler models and probability distributions, and allows inference to be performed automatically on these models. The problem of efficient inference is then separated from the specification of the model. Inference code can be written and optimised once, independent of specific models, and probabilistic models can be written by domain experts, without having to worry about hand-writing inference. The inference 'engine' can also implement many different inference algorithms, allowing selection of methods appropriate to the models being used.

PPLs allow us to create these models as programs in code. Generative models are built by taking samples from probability distributions, so PPLs need some way of modelling this non-determinism. Being able to condition the values of variables on data is the other key part of PPLs, since we are interested in the posterior, which is conditional on the data. Without conditioning, we can run a program forwards, which generates samples from the model we write (the prior). By taking into account conditioning, we can infer the distribution of the input parameters based on the data we observe and sample from this distribution.

PPLs can either be standalone languages or be embedded into another language. Embedding a PPL into a pre-existing language allows utilising the full power of the host language, and gives access to operations in the host language without having to implement them independently. This makes it easier to combine models with each other as well as integrate them more easily into other programs written in the host language. Specifically, embedding a domain specific languages (DSL) into OCaml allows us to represent a wide range of models using OCaml's inbuilt syntax and libraries, as well as leverage OCaml features such as an expressive type system or efficient native code generation.

1.2 Related work

There are many examples of PPLs, both as DSLs embedded into other languages (including OCaml) and as standalone compilers. Some early PPLs, such as BUGS [1] or JAGS [2], limited the types of models representable in the language to finite graphical models, where the model could be expressed as a static graph of random variables and their relationships. Restricting the types of possible models can lead to efficient implementations of inference algorithms. Languages such as STAN [3] or Infer.NET [4] exploit this, and do not allow, for example, unbounded recursion when defining models. PPLs which can express models that have an unlimited number of random variables (and so do not compile to a static graph) are known as 'universal' [5], and include Church [6], WebPPL [7] and Anglican [8]. These tends to lead to slower inference procedures due to the need to support a greater range of models. Some PPLs restrict the types of distribution allowed, for example HANSEI [9] and IBAL [10] only allow discrete distributions. A summary of some PPLs is given in Table 1.1.

PPL	**Embedded In**	**Universal**	**Continuous Distributions**	**Year**
BUGS [1]	N/A	✗	✓	1994
IBAL [10]	OCaml	✗	✗	2000
JAGS [2]	N/A	✗	✓	2004
Church [6]	LISP	✓	✓	2008
HANSEI [9]	OCaml	✗	✗	2009
Infer.NET [4]	F#	✗	✓	2011
STAN [3]	N/A	✗	✓	2012
Anglican [8]	Clojure	✓	✓	2014
WebPPL [7]	JavaScript (node)	✓	✓	2018
Pyro [11]	Python	✓	✓	2019
OwlPPL	OCaml	✓	✓	2020

Table 1.1 – *A collection of PPLs*

2 Preparation

This chapter describes the research carried out before starting the project and the professional practices followed throughout the project. I include an overview of the tools and technologies used, give a more detailed account of probabilistic programming and describe of several classes of inference algorithms.

2.1 Starting Point

There do exist PPLs embedded in OCaml, such as IBAL [10] or HANSEI [9]. PPLs have also been embedded in other languages, such as WebPPL [7], Church [6] or Infer.Net [4]. My PPL used some of the ideas introduced by these languages, particularly in implementing efficient inference engines. I needed to research the different approaches taken by these PPLs and decide what form of PPL to implement, especially in deciding the types of model I wanted my PPL to be able to represent and the inference methods I implemented.

I used an existing OCaml numerical computation library (Owl). This library does not contain methods for probabilistic programming in general, but it does contain a large number of statistical functions for common distributions as well as efficient random number generation. Efficient computation of these functions is key to developing a performant PPL.

I had experience with the core SML language, which aided in learning basic OCaml due to similarities in the languages, however I did have to learn the modules system. 1B Foundations of Data Science also gave me a basic understanding of Bayesian inference. I did not have experience with domain specific languages in OCaml, although the 1B compilers course did implement a compiler and interpreter in OCaml.

2.2 Requirements

Before starting to write any code, I made sure to set out the features I aimed to implement for my DSL. The success criteria were:

- **Language Features**: Since I have written an embedded DSL, a user of my PPL should be able to take advantage of all standard OCaml features in their models. I needed to make sure that this is the case, and features such as recursive or higher order functions will work.
- **Available distributions**: I aimed to make sure my PPL has at minimum the Bernoulli and normal distributions available as basic building blocks to build more complex probabilistic programs.
- **Correctness of inference**: I used the PPL developed on example problems to ensure correct results are produced. These results were compared to analytic solutions for simple problems.

- **Available Inference Algorithms**: I aimed to include at least one available inference algorithm. However, since different problems are more or less well suited to different general-purpose inference procedures, I wrote implementations for five separate algorithms.
- **Performance**: This is a quantitative measure, comparing programs written in my PPL to equivalent programs in other PPLs. I used the profiling tools spacetime and GProf to profile my OCaml code. Performing inference needed to be possible within a reasonable amount of time, even though the project does not have a significant focus on performance. I also benchmarked the performance with regards to scalability, so the performance is still reasonable as models are conditioned on more data.

2.3 Tools and Technologies

The main tools I used are listed here:

- OCaml 4.08 - The host language for the PPL
- Dune - Build system for OCaml
- Opam - OCaml package manager
- Alcotest - Unit testing framework
- Quickcheck - Invariant testing library with random input generation
- Bisect_ppx - Code coverage tool
- Spacetime - OCaml-specific memory profiler for OCaml
- Gprof - Generic Linux profiler
- Owl - OCaml scientific computing library
- VSCode (with OCaml extensions) - IDE for OCaml development
- Git with GitHub - version control

Using OCaml 4.08 allows me to use new features of OCaml, in particular the ability to define custom let operators as syntax sugar for monads. I used the dune build system to more easily manage building and testing my code, as well as automatically creating documentation from comments and function signatures in my code. The profilers I used allowed me to work out the causes of performance issues and remedy them, as well as measure the performance of my PPL.

2.3.1 OCaml

I have chosen to use the OCaml language to implement my PPL. There are many features of OCaml which make it suitable for writing a DSL. Using OCaml, I can make sure that expressions in the DSL are well-typed, so that programs don't fail to run. OCaml's algebraic datatypes make it easy to represent probabilistic programs as trees, and pattern matching makes it easy to interpret and transform these trees. The module system also makes sure that certain types can be hidden from the user, which ensures only valid values are created.

2.3.2 GADTs

The main data structure I use to represent distributions is a GADT - a generalised algebraic data type. GADTs are similar to ADTs (sum types), as they have a set of constructors, but

these constructors can have their output types annotated with different return types. This makes GADTs more general than normal ADTs, whose return types are all the same. An example is a GADT to represent expressions, and the corresponding ADT:

GADT

```
type _ expr =
    F   : float -> float expr
  | B   : bool -> bool expr
  | Add : float expr * float expr
          -> float expr
(* correctly doesn't type check *)
Add(F 0.5, B true)
```

ADT

```
type 'a expr =
    F   of float
  | B   of bool
  | Add of float expr * float expr

(* type checks but wont evaluate *)
Add(F 0.5, B true)
```

Listing 2.1 – *Two ways to implement the type for a simple language - the GADT prevents bad expressions being accepted by the type checker*

With this GADT, booleans can't be combined with `Add`, and we can statically check that expressions will not fail to evaluate, which would not be possible with an ordinary ADT.

Type inference for GADTs is undecidable, and so type inference often fails when pattern matching on GADTs, especially for recursive functions. I often need to annotate functions as being explicitly polymorphic using the syntax below.

explicit for all 'a

```
let f: 'a.'a t -> 'a t = fun x -> ...
```

2.3.3 Monads

Monads are a design pattern commonly used in functional programming languages.

The key data structure I use to model probability distributions is a monad. A data type is a monad if it defines two operations, `return` and `bind`, and can be thought of as a type which 'wrap' values. The return function takes a value and returns a monad wrapping that value. The bind function takes a monad and a function, and applies the function to the value wrapped inside the monad, and then re-wraps this value. The type must also satisfy a set of laws, which I omit here [12]. Monads can be used to structure programs in a general way, and allow side effects to be described in types.

Probability Monad It has been shown that probability distributions form a monad, [13, 14], and that they can be used to create distributions composed from other distributions [15]. In this case, `return x` represents a distribution with only one value (x) - known as a Dirac distribution. So `bind d f` is the main operator for composing distributions. Binding distributions together represents taking the output of one distribution (d) and using it in the body of the function (f). This can be thought of as taking a sample from d. However, it is important to note that calling bind will not directly produce a sample, but expose that structure to an interpreter (the inference engine) which can then decide what to do at that point.

Custom let operators OCaml 4.08 allows me to define definitions for custom let operators. This is used to provide syntactic sugar for working with monads, and is similar to do-notation in Haskell. The reference documentation [1] specifies that a monad should provide a module

[1] `http://caml.inria.fr/pub/distrib/ocaml-4.08/ocaml-4.08-refman.html#s%3Abinding-operators`

which implements the (let*) and (and*) operators. The (let*) operator is the standard bind function - it takes the identifier bound to as the first argument to the function in bind. The (and*) operator is the product operation, it takes two monads and returns the monad product of the arguments - it has signature `'a m -> 'b m -> ('a * 'b) m`, where m is the monad type. An example, using the Option monad is given below to show the transformation that takes place. This feature allows the user to not have to use the bind (often aliased by »=) or product functions explicitly, and offers a more intuitive syntax.

New Syntax

```
open Option

let add_options x y =
  (product x y)
    >>= (fun (a,b) ->
      Some (a+b)))
```

Old Syntax

```
open Option

let add_options x y =
  (product x y)
    >>= (fun (a,b) ->
      Some (a+b)))
```

Listing 2.2 – *New let syntax which makes monadic binds easier to express*

2.3.4 Modules

The module system is a key feature of the OCaml language. Every function in OCaml is in a module, by default the name of the file it resides in. Modules can also have signatures, which define what code is visible to a user, and constrain the module. Modules can hide types and implementations to provide a clean API, and are often used to wrap a particular type, for example a list or map. This means that to create or manipulate that type, the user must go through the module's API, ensuring only permitted operations are carried out. This is a feature I've used in designing distribution types. Modules can also be dynamically created from other modules, using functors, which are functions from modules to modules. This technique is used extensively in the Core library, and it allows modules to be customised and extended.

In OCaml, the module language (functors, modules, signatures, etc.) and the core language (functions, values, types, etc.) are considered separate, and values can't contain modules. First class modules provide a way around this constraint, and modules can be used in much the same way as ordinary values. This means a library can define functions to create modules which can then be used by other functions.

2.3.5 Owl

Owl is a scientific computing library written for OCaml [16]. It contains functions for working with a wide variety of probability distributions, e.g. normal, beta, binomial, etc. In particular, it is important to be able to find the probability density function (pdf) of distributions and to efficiently sample from distributions - these are the functions needed to perform inference. The distributions available in Owl form the primitive distributions I support, and are the building blocks of a probabilistic model.

Owl also provides many efficient helper functions, which can be used to calculate statistics over arrays of samples. Another important feature of Owl is the plotting API for which I wrote a wrapper to easily visualise output distributions from my PPL.

2.4 Probabilistic Programming

Probabilistic programming is a programming paradigm where statistical models can be written as code and analysed. Models are a collection of random variables and the relationships between them.

Existing PPLs take two main forms - they can be standalone or embedded in another language. Standalone languages have their own syntax and compiler, so can be fine tuned to the task of inference, but often lack features since they have to be built from scratch. Embedded languages can utilise facilities in their host language, such as type checking, compilers or libraries, as well as allowing programs to be integrated into existing systems easily. However they need to work around the syntax and semantics of the host language.

The other main trade-off made in the design of PPLs is the range of models that can be expressed in the language against the efficiency of inference. Many languages restrict the set of allowed models in order to use more efficient inference algorithms which can take advantage of the restricted structure of models. Universal languages can represent any model, but suffer from less predictable inference procedures since many properties of the model (such as the number of random variables) are not available at compile-time. Overall, the more general models that need to be supported, the less efficient inference is.

There are two main approaches to building PPLs. One approach is graph-based, where a finite graph representing the variables and their relationships is generated from a program, over which efficient inference can take place. This approach is used in languages such as Infer.NET or JAGS, and has the benefit of being able to process high-dimensional data well, since efficient computation graph frameworks can be leveraged - another example is Edward [17], which uses TensorFlow as a backend. This approach restricts the types of models to those that can be represented by the underlying graph.

Another approach, taken by the PPLs such as Anglican or WebPPL, is trace-based. This approach considers execution traces, with a 'trace' being one run of a program, where the intermediate random variables take a particular value. Inference algorithms reason about these traces in order to produce a posterior distribution. A trace-based approach can lead to more models being able to be expressed, since we are not limited by the constraints of a graph. However, inference is often slower, since inference algorithms need to be more general purpose, and often converge slower. I have taken a trace-based approach in OwlPPL in order to allow my language to represent models which uses regular OCaml language constructs.

2.4.1 Bayesian Inference

Inference is the key motivating feature of probabilistic programming, and is a way to find a distribution over some input parameters based on the data we observe. The main feature of Bayesian inference is that we assign every model some prior belief. Often this prior is chosen based on our knowledge of the problem, but the prior can also be uninformative. The goal of Bayesian inference is to calculate the posterior distribution, which can be represented by Bayes' formula,

$$P(\theta \mid x) = \frac{P(x \mid \theta)P(\theta)}{P(x)} \propto P(x \mid \theta)P(\theta)$$

with $P(\theta)$ being the prior, and $P(x \mid \theta)$ being the likelihood model we define - the probability of observing the data given some parameters θ. Both x and θ are often vectors of variables, representing multiple parameters and observations respectively.

Unfortunately, exact Bayesian inference is usually computationally infeasible, especially when the number of random variables we consider is large. The main issue is computing the normalising factor $P(x)$,

$$P(x) = \int_{\Theta} P(x, \theta) \, d\theta$$

This represents marginalising x out of the joint distribution by summing over all possible values of θ. If the state space becomes very large (or infinite), or if θ is a very large vector, computing this exactly is intractable. For some distributions, this integral does not even have a analytic solution.

In the PPL setting, the prior is the generative model we define and conditioning statements define the likelihood model. Running the program forward without inference produces samples from the prior. Running inference on the program produces a new distribution, the posterior. Since the exact distribution usually can't be computed, inference algorithms I implement return distributions which can only be sampled from. Other statistics (e.g., mean, pdf, etc.) can then be estimated by taking a large number of samples.

2.4.2 Inference Algorithms

Inference algorithms are ways to systematically generate samples from posterior distributions given a likelihood function and a prior distribution. In trace-based PPLs, a model consists of latent (internal) variables and observed variables, and a single execution of a model (a program) can be thought of as an assignment to each of these variables, known as an *execution trace*. This can be defined mathematically as below, by Bayes' rule:

$$p(x_{1:N} \mid y_{1:N}) \propto \tilde{p}(y_{1:N}, x_{1:N}) \tag{2.1}$$

Note that the trace may have a different number of variables each time a model is run, due to the fact that we allow general models which allow for unbounded recursion.

In (2.1), p is the posterior distribution of a particular trace x, given the observed variables y. This is proportional to the joint distribution of all the variables ($\tilde{p}$). The aim is then to find the posterior over the latent variables we are interested in (by marginalising out the other variables). We can specify which variables we care about within the program, either as part of the model, or outside it in a query to the model.

In general, there are two classes of inference algorithms - static and dynamic [18] which correspond roughly to graph-based and trace-based PPLs respectively. In static methods, the program is compiled to a static structure (e.g. a Bayesian network), which is analysed for inference to be performed. These methods generally constrain the models that can be represented (often to finite graphical models). Since my PPL aims to be universal, I focused mainly on dynamic methods, which use sampling to run programs and use conditioning statements that occur on these runs to perform inference.

Exact Inference

Exact Inference is the simplest method of calculating the posterior, but is usually computationally intractable. It involves calculating Bayes formula exactly, of which calculating the normalising constant is usually the problem. For discrete posterior distributions it can be thought of as calculating the probability of every possible value of the variable of interest. Since a random variable will be dependent on several others, this involves enumerating every possible combination of these variables and their outcomes.

Rejection Sampling

Since exact inference is too difficult in practice, we usually have to resort to *Monte Carlo* [19] methods which rely on repeated sampling to infer properties of a distribution.

One such method, rejection sampling, is a very simple inference method which uses a 'proposal' distribution which *can* be sampled from. We take samples from the proposal distribution, and either accept or reject them. How likely we are to accept or reject a sample depends on the pdf of this proposal distribution. It can be shown that samples taken using this method converge to the required distribution [20].

Importance Sampling

Importance sampling is another simple method improving on rejection sampling. It also uses a proposal distribution that can be sampled from. We calculate the ratio of the likelihoods between the two distributions to weight samples from the proposal. From doing this repeatedly with multiple samples from the proposal, we can build a posterior represented by a set of weighted samples.

Monte Carlo Markov Chains (MCMC)

MCMC methods involve constructing a Markov chain with a stationary distribution equal to the posterior distribution. A Markov chain is a statistical model consisting of a sequence of events, where the probability of any event depends only on the previous event. The stationary distribution is the distribution over successive states that the chain converges to (if it converges to one).

There exists several algorithms for finding this Markov chain, for example Metropolis-Hastings. Several MCMC algorithms require that we have a function, $f(x)$, which is proportional to the density of the distribution. The function is easy to compute for the posterior, since it is simply the prior multiplied by the posterior - the normalising constant can be ignored since we only need a proportional function.

MCMC algorithms have the same basic structure - to first 'run' the chain for a burn-in period, taking samples and discarding them. Then, running the chain and collecting the states visited as samples. This set of samples is then a set of samples from the posterior, since the posterior should be equal to the stationary distribution. An important trade-off is made in the length of the burn-in period - too long and time is wasted discarding states, but too short and the chain will not converge to the correct distribution.

Sequential Monte Carlo (SMC)

SMC methods are algorithms which are based on using large numbers of weighted samples ('particles') to represent a posterior distribution. SMC methods are also known as particle filters. A particle is a value paired with an unnormalised weight which represents the likelihood of that value in the distribution. These particles are updated when data is observed and re-sampled from in order to converge the set of particles to the posterior.

For a set of weighted particles,

$$\{(x^{[i]}, w^{[i]})\}_{i=1..N}$$

the pdf of this distribution is given by

$$p(x) = \sum_{i=1}^{N} w^{[i]} \delta_{x^{[i]}}(x)$$

where δ is the Dirac distribution.

The simplest SMC algorithms are particle filters [21], which simply resample particles on encountering new data, updating the weights of the particles based on how likely this data is deemed to be. However, many variations exist - the resampling method, updating the weights and the initialisation of particles can all be varied. The common feature of SMC algorithms is that they sequentially create sets of particles which converge to the desired distribution.

Particle Monte Carlo Markov Chain (PMCMC)

SMC methods can also be combined with MCMC methods. These algorithms are known as particle Monte Carlo Markov chain (PMCMC) algorithms, and were first introduced for probabilistic programming in the Anglican language [8]. PMCMC methods are essentially MCMC algorithms which use an SMC algorithm as a proposal distribution.

2.5 Professional Practice

I adopted several best practices in order to ensure the project was successful. This includes performing regular testing, splitting code into separate modules designing signatures first, and ensuring my code follows a consistent style[2].

2.5.1 Testing

The statistical nature of my library makes it difficult to write thorough unit tests for inference and sampling procedures. Ensuring posterior samplers are correct is a difficult problem due to inherent randomness, and the solutions to this I implemented are covered in section 3.8.

Despite this, there is a suite of unit tests for individual deterministic functions. This is especially important for deterministic helper functions which should always work the same way, and unit tests allow regressions to be caught. I also used the `Quickcheck` library to perform some unit tests, which allows me to ensure certain invariants are preserved by automatically testing on many randomly generated inputs. I use the `bisect_ppx` library to produce code coverage reports to ensure I am thoroughly testing code.

[2] `https://opensource.janestreet.com/standards/`

2.5.2 Continuous Integration

Since I will be developing a library, it is important to make sure that any unit tests are run regularly to ensure there are no regressions - no new code affects old behaviour. It is also important to make sure the library will function on different platforms, and that documentation is built (and possibly uploaded) automatically. To achieve this, I will be using GitHub's continuous integration service, 'actions' to make sure code on the master branch always works. The CI can also be used to ensure the library works with older versions of OCaml, and is backwards-compatible.

2.5.3 Licenses

The libraries I use, Owl and Jane Street Core, are both licensed under the MIT license. This allows me to freely open-source my work using a similar license.

3 | Implementation

In this chapter I describe how my PPL is implemented and the design decisions considered. I discuss the core data structures used to represent distributions and how inference was implemented over these. I also describe auxiliary functionality included, such as visualisation functions. Finally, I describe the challenges of testing a probabilistic system and how I overcame them. Documentation for all publicly exposed functions can be found in appendix B.

3.1 Repository Overview

The code for the PPL library is in the `ppl` directory, which contains the core library, unit tests, statistical testing code and some example programs written using the library. A separate evaluation folder contains code to benchmark my PPL's performance compared to other languages.

The structure follows the de facto standard for dune projects. The lib directory contains all the code for the library, bin contains standalone scripts which contain example models, doc contains auto-generated documentation, and test contains unit and hypothesis tests. The ppl.opam file describes my library and it's dependencies, to allow my library to be distributed via opam.

The lib directory contains several modules to separate functionality within my PPL. The entry point is the Ppl module, which exports the other modules so that opening Ppl is sufficient to define models. The inference directory contains inference algorithms in separate modules, which are then re-exported in the main Inference module. The Dist module is the module for user-defined models on which inference is run.

All code is written in OCaml 4.08, with the main dependencies being Jane Street's `Core` and `Owl` [16].

```
ppl
├── bin
│   └── ...
├── doc
│   └── ...
├── lib
│   ├── inference
│   │   └── ...
│   ├── dist.ml
│   ├── empirical.ml
│   ├── helpers.ml
│   ├── inference.ml
│   ├── monad.ml
│   ├── plot.ml
│   ├── ppl.ml
│   └── primitive.ml
├── test
│   ├── unit_tests
│   │   └── ...
│   ├── test_models.ml
│   └── hypothesis_tests.ml
├── dune-project
└── ppl.opam
```

Figure 3.1 – *Directory structure*

3.2 Language Design

I chose to implement my language as a domain specific language (DSL), shallowly embedded into the main OCaml language. Using a shallow embedding means we can use all of the features of OCaml as normal, including branching (if/then/else), loops, references, let bindings, (higher-order) functions, and recursion. This has the benefit of leveraging OCaml

features such as type checking, as well as permitting library code to be included directly within models.

Using a shallow embedding allows for recursively defined models. This can allow non-terminating (and therefore invalid) models to be defined. However, we can write functions which are *stochastic recursive* [22], that is, functions which have a probability of termination that tends to 1 as the number of successive calls tends to infinity. This leads to functions which terminate their recursion non-deterministically. Any model which does not satisfy this will be considered an invalid model - unfortunately as it is not possible to determine whether or not a program will halt, this property cannot be enforced.

An embedded PPL can be thought of as being the same as the host language, except for two extra operators:

- sample - for taking a sample from a distribution, either a primitive distribution or another (sub-)model.
- condition - for conditioning on observations, defines how likely data observed is (also called observe or score in other PPLs).

The problem of designing a PPL is then finding a way to model the nondeterminism in the sample operator and integrate the information from the condition operator to guide inference and the execution traces explored. Most universal PPLs use a feature that enables exploring subcomputations - the different execution traces. This can be done using continuation passing style (CPS) transformations, as in WebPPL and Church [7, 6], or algebraic effects as in Pyro [11]. In order to achieve a similar effect, I model conditional distributions as monads in OwlPPL, as in [23], and realise probabilistic programs as a GADT.

Log Probabilities Calculating probabilities can often lead to underflow, since it is common to multiply many small probabilities together. To avoid this, I use logs of probabilities internally and add logs where I would multiply the original probabilities. This behaviour can be changed by changing the `Prob` module used by the `Dist` module - the documentation for `Prob` is in appendix B.

3.3 Representing Distributions

In order to define my DSL, I use 3 different data structures to represent the different types of distribution I use:

- Input distributions - primitive distributions that are used to build models.
- General probabilistic models - composed primitives conditioned on data.
- Output distributions - empirical distributions built from a set of samples from a posterior.

3.3.1 Primitive Distributions

In PPLs, users build complex models by composing more simple elementary primitive distributions for which we have extra information such as exact equations and the ability to sample directly. These primitive distributions need to have a few operations defined on them, namely `sample, pdf, cdf, ppf` (inverse of cdf) and `support` (the set of values that

a distribution can take). These are all standard properties of distributions, and are used to perform inference.

An extension goal achieved here is to allow users to define their own primitive distributions if they have not already been defined in the library. For example, I have not implemented the Poisson distribution as a primitive distribution, but you can imagine models which need to use the Poisson as a building block. To achieve this, the user simply has to write a function which takes the parameters of the distribution as arguments and return a first class module matching the primitive distribution signature. This technique also allows users to use modules that they may have already defined, and constrain them to the required signature for use in the PPL.

The type of a primitive distribution is

```
type 'a prim_dist = (module PRIM_DIST with type t='a)
```

with the `PRIM_DIST` signature defined as in Listing 3.1.

```
module type PRIM_DIST = sig
  type t

  val sample : unit -> t
  val pdf : t -> float
  val ppf : t -> float
  val cdf : t -> float
end
```

Listing 3.1 – *Signature of the module that primitive distributions must implement*

```
(* full definitions omitted *)
let poisson l = (module struct
  type t = int
  let sample () = ... _
  let pdf k = ... _
  let ppf k = ... _
  let cdf k = ... _
end: PRIM_DIST with type t = int)
```

Listing 3.2 – *Adding a new distribution as a primitive*

An example of this being used to add a new primitive distribution is given in Listing 3.2, for the specific case of the Poisson distribution. The `Poisson` distribution can now be used as other primitives are, e.g. in `observe` statements.

3.3.2 General Probabilistic Models

Statistical models are designed by the user to model a process they are interested in and are given as the input to inference procedures. They are built up from primitive distributions, and should be both composable (in order to build bigger models) and amenable to inference procedures.

Probability Monad

As mentioned in section 2.3.3, monads are a natural way to represent composable probability distributions. They allow the output from one distribution (essentially a sample), to be used as if it was of the type that the distribution is defined over. Essentially, the `bind` operation allows us to 'unwrap' the 'a dist type to allow us to manipulate a value of type 'a. We must then use `return` to 'wrap' the value back into a value of type 'a dist. The type signatures of these functions are below, with `m` being the monad type. Using monads also allows us

```
val bind: 'a m -> ('a -> 'b m) -> 'b m
val return: 'a -> 'a m
```

to define several helper functions which can be used when working with distributions. For example, we can 'lift' operators to the `dist` type, for example allowing us to define adding two distributions over integers or floats using liftM or liftM2. We can also fold lists of distributions using a similar technique.

Using monads also allows the use of the extended let operators introduced in OCaml 4.08. These allow the definition of custom let operators, which mimic do-notation in Haskell. This means that sampling from a distribution (within a model) can be done using the `let*` operator, and the variable that is bound to can be used as if it were a normal value. The one caveat is that the user must remember to `return` at the end of the model with whatever variable(s) they want to find the posterior over. The `and*` operator can also be used when we use several independent distributions in a row. This can make for more efficient sampling (and inference) since more structure is encoded. It is also a common pattern to set up a model by first independently drawing from several distributions, as below.

```
(* two independent draws from standard normals *)
let* x = normal 0. 1.
and* y = normal 0. 1. in
(* ...rest of model  *)
return (x + y)
```

Listing 3.3 – *Use of `and*` for independent draws*

I define my own functor for monads in order to automatically generate several helper functions. This takes a module with the basic monad functions and extends it with helper functions defined in terms of return and bind. The full module documentation for this can be found in appendix B. An example is the liftM2 function, which allows normal operations to be lifted to distributions, e.g. an addition operator for the output for two distributions can be simply created by lifting the normal addition operator, allowing distributions to be naturally 'added'.

```
let ( +~ ) = liftM2 ( +. )
(* the distribution of the sum of 2 independent draws from standard normals *)
let d = (normal 0. 1.) +~ (normal 0. 1.)
```

Listing 3.4 – *Lifting addition to distributions*

However, there are many different underlying data structures which can be used to represent distributions which conform to the definition of a monad. The simplest is a list of pairs representing a set of values and corresponding probabilities, `('a * float) list`. This is a natural way to represent discrete distributions, with return and bind defined as in Listing 3.5. Here, `return` gives us the distribution with just one value, and bind combines a distribution with a function that takes every element from the initial distribution and applies a function that creates a set of new distributions. The new distributions are then flattened into a single list and normalised. This approach has been used to create functional probabilistic languages [24], but has several drawbacks, primarily the fact that it cannot be used to represent continuous distributions, and that inference is not efficient - there is no information from the model

encoded in this representation, such as how random variables are combined or from what distributions they came from.

```
type 'a dist = ('a * float) list

let unduplicate = (* omitted *)_
let normalise = (* omitted *)_
let bind d f =
  let mult_snd p = List.map ~f:(fun (a,x) -> (a,x*.p)) in
  List.concat_map ~f:(fun (x,p) -> mult_snd p (f x)) d
  |> unduplicate |> normalise

let return x = [(x,1.)]
```

Listing 3.5 – *Probability monad as a List*

Another issue is that flattening distributions is inefficient since duplicate values must be combined, and this approach is $O(n^2)$ when using a list since we scan up to the length of the entire list for every element. A better option is to use a map, which is provided in Core, and implemented as a balanced tree, significantly improving the time complexity of combining distributions.

```
type 'a dist = ('a, float) Map.Poly.t

let normalize map = (* omitted *)_

let bind d f =
  fold d ~init:empty ~f:(fun ~key ~data sofar ->
      Map.merge sofar (f key))
      ~combine:(fun ~key -> ( *. ))
  |> normalize_map

(* create a map with a single pair, x:1 *)
let return x = add (empty) x 1.
```

Listing 3.6 – *Probability monad as a map*

Although this is not the final data structure I chose for general probabilistic models, it is the one I used for discrete empirical distributions.

GADT

The structure that I selected to represent general models is a generalised algebraic data type. GADTs are often used to implement interpreters in functional languages, and have been used to represent probabilistic models. The GADT I implement here (and some inference algorithms) is based on (Scibior et al. 2015) [23]. This represents a model in a very general way, and can then be 'interpreted' by a sampler or an inference algorithm. For sampling, I traverse the model, ignoring conditionals to enable forward sampling from the prior.

For inference, I provide some inference functions as transforming conditional distributions to distributions without any conditional statements, allowing sampling to be performed

as normal. Some inference functions are also implemented by generating an empirical distribution that can be sampled from similarly.

Listing 3.7 shows each of the variants. The monad functions are also provided, which construct the corresponding variants in the GADT - `Return` represents a distribution with only one value, and `Bind` contains a distribution and a function, which represents that function being applied to the output from that distribution, and is also bound to `(let*)`. The product function is used for models with independent sub-parts, such as drawing samples from many independently distributed variables, and could be used to parallelise models. The `Independent` variant is also bound to `(and*)`.

Primitive distributions have a variant which takes the primitive distribution type. We can find the exact pdf/cdf of these distributions, unlike the more general `dist` type, which can only be sampled from.

The `Conditional` variant is used to assign scores (likelihoods) to execution traces, and contains a function which takes an element produced by a model and returns a score for the corresponding trace. I define several wrappers over this variant to represent different types of conditioning, outlines in section 3.4.

```
type _ dist =
  | Return : 'a -> 'a dist
  | Bind : 'a dist * ('a -> 'b dist) -> 'b dist
  | Primitive : 'a primitive -> 'a dist
  | Independent : 'a dist * 'b dist -> ('a * 'b) dist
  | Conditional : ('a -> float) * 'a dist -> 'a dist

let return x = Return x
let bind d f = Bind (d, f) (* also used as ( let* )  *)

let product d1 d2 = Independent (d1, d2) (* also used as ( and* )  *)
```

Listing 3.7 – *Representing a probabilistic model using a GADT*

An important feature of this type is that it is polymorphic - this allows distributions to be defined over any type, including arbitary ADTs or even distributions themselves.

3.3.3 Empirical Distributions

The output of Bayesian inference is a probability distribution over the variables we are interested in. Ideally, we would be able to produce an exact posterior distribution, and be able to extract exact relevant statistics. However, approximate inference only allows us to create functions to sample from this posterior. We can define a signature for a type of an empirical distribution that is created from posterior distributions by taking many samples. This can then be used to calculate useful statistics - e.g. mean, variance, pdf, cdf, etc.. The type is abstract to allow different implementations for discrete and continuous distributions.

For discrete distributions, I use a `Core.Map`[1], with the keys being the values that the distribution can take and the values the number of samples with that value. Continuous

[1] `https://ocaml.janestreet.com/ocaml-core/latest/doc/base/Base/Map/index.html`

distributions use a dynamically resizing array - adding each sample is then $O(1)$ amortized, and statistics can be calculated using Owl's functions that operate on arrays. A constraint on continuous distributions of this type is that they are defined over floats, and only represent one dimension.

```
module type Empirical = sig
  type 'a t
  type 'a support = Discrete of 'a array | Continuous

  val from_dist : ?n:int -> 'a dist -> 'a t
  val empty : 'a t
  val add_sample : 'a t -> 'a -> 'a t
  val get_prob : 'a t -> 'a -> float
  val to_pdf : 'a t -> 'a -> float
  val to_cdf : 'a t -> 'a -> float
  val to_list : 'a t -> 'a list
  val support : 'a t -> 'a support
end
```

Listing 3.8 – *Signature for empirical distributions*

The signature in Listing 3.8 is implemented for both continuous and discrete distributions. For the continuous case, I perform binning to approximate the continuous distribution by a discrete one in order to approximate the pdf. The number of bins used is calculated automatically from the number of samples taken.

3.4 Conditioning

The GADT described in section 3.3.2 can be used to describe general models, in particular conditional distributions, thanks to the `Conditional` variant. Without this variant, we can only define prior distributions, but including it means we can incorporate observed data into our models and perform inference.

The condition variant in my GADT is used to assign scores to traces, and takes a function which takes an element and returns a float, a 'score'. This score represents how likely the current trace is, given the value passed to the function. In this way, we can represent observations.

I have also implemented a few helpers to make it easier to condition models. The three main helpers are `condition`, `score` and `observe`, which are all specific cases of the general `Condition` variant.

The `condition` operator is used for hard conditioning, which conditions the model on an observation being true. If true is passed in, then the score assigned is 1, and if false, the score assigned is 0. This score represents how likely it is for the current trace to occur, and different inference algorithms will use this information to produce a distribution over all possible traces. We can use this operator to constrain certain variables or outcomes in a model. For example in Listing 3.9, we roll two dice and observe that the sum is 4 - we can then find the distribution over the first die (which won't include 4,5 or 6 since they are >=4).

This function is mostly useful for discrete models when using equality in this manner, since the probability of observing any given value in a continuous distribution is zero. However, if we are dealing with ranges, then we can use hard conditioning as in Listing 3.10, which constrains the standard normal distribution to be positive.

```
let* dice1 = uniform [1;2;3;4;5;6] in
let* dice2 = uniform [1;2;3;4;5;6] in
condition (dice1+dice2 = 4)
        (return dice1)
```

Listing 3.9 – *Hard conditioning for discrete model*

```
let* x = normal 0. 1. in
condition x > 0.
        (return x)
```

Listing 3.10 – *Hard conditioning for continuous model*

For soft conditioning, for example an observation that we know comes from a certain distribution, there is an `observe` function. This function is essential for continuous distributions, since the probability of observing any one value is 0, making hard conditioning redundant since it will just assign a score of zero to every trace. Instead, we can use the pdf of the distribution to determine how likely that observation is in the model.

The `score` function is similar to the condition operator, except instead of 0, it assigns a particular constant score to the trace. This is generally used in a branching statement, where a constant score will be assigned depending on some (deterministic) condition.

```
let condition b d = Conditional((fun _ -> if b then 1. else 0.), d)
let score s d = Conditional((fun _ -> s),d)
let observe x dst d = Conditional((fun _ -> Primitive.pdf dst x),d)
```

Listing 3.11 – *The definitions of the different conditioning operators*

3.5 Forward Sampling

The simplest operation to define on models is to sample from them. Sampling from conditional distributions requires inference, and is discussed in section 3.6. Here, we run a probabilistic program 'forwards', that is, running a generative model and seeing the outputs without conditioning on observed data.

In PPLs, a complete program is a posterior distribution of a parameter given some observed data, $P(\theta \mid x)$. The generative model, i.e. the program without condition statements, is the prior distribution, $P(\theta)$. The condition statements then define the likelihood model, $P(x \mid \theta)$, the probability of the observations in the current model. So sampling from the prior is the same as sampling normally, but ignoring the conditionals (essentially ignoring the data).

We can also take into account the conditionals, and produce weighted samples, with the weight being the score assigned by each conditional branch, accumulated by multiplying all the scores. This gives us a set of values with corresponding weights which represent how likely those values are. An important property of these weights is that they are not normalised, so we cannot use them to find the posterior directly. I have implemented several variants of functions for finding the prior and sampling, all with the same concept as Listing reflst:sampling.

```
let rec sample : 'a. 'a dist -> 'a = function
  | Return x -> x
  | Bind (d, f) ->
      let y = f (sample d) in
      sample y
  | Primitive d -> Primitive.sample d
  | Independent (d1, d2) -> (sample d1, sample d2)
  | Conditional (_, _) -> raise Undefined

let rec prior_with_score : 'a. 'a dist -> ('a * prob) dist = function
  | Conditional (c, d) ->
      let* x, s = prior_with_score d in
      return (x, s *. c x)
  | Bind (d, f) ->
      let* x, s = prior_with_score d in
      let* y, s1 = prior_with_score (f x) in
      return (y, s *. s1)
  | Primitive d ->
      let* x = Primitive d in
      return (x, Primitive.pdf d x)
  | Return x -> return (x, 1.)
  | Independent (d1, d2) ->
      let* x, s1 = prior_with_score d1
      and* y, s2 = prior_with_score d2 in
      return ((x, y), s1 *. s2)
```

Listing 3.12 – *Sampling functions*

The function for generating a prior does not directly take samples, but manipulates the structure of the dist GADT. For example, in the `Bind` branch, it actually introduces 2 new bind variants (via `let*`) which produces a new distribution lazily. This makes it easier to use the prior within inference algorithms, and allows it to be composed with other distribution modifying functions.

3.6 Implementing Inference

Inference is the key motivation behind probabilistic programming. Up to this section, I have discussed how to represent models but not do anything with them that couldn't be done in a standard language. With inference, we can produce a sampler which will accurately reflect a posterior distribution.

Inference can be thought of as a program transformation [23, 25]. In my ppl, this corresponds to a function of type `'a dist -> 'a dist`. This method allows for the composition of inference algorithms, exemplified in section 3.6.7.

3.6.1 Enumeration (Exact Inference)

Enumeration is the simplest way to perform exact inference on probabilistic programs, and essentially consists of computing the joint distribution over all the random variables in the model. This involves enumerating every execution path in the model, in this case performing a depth first search over the `dist` data structure. For every `bind` (i.e. every `let*`), there is

a distribution (d) and a function from samples to new distributions (f). I call this function on every value in the support of the distribution d, and then enumerate all the possibilities. The final output is a `('a * float) list`, from which duplicates are removed and is then normalised, so that the probabilities sum to one.

```
let rec enumerate : type a. a dist -> float -> (a * float) list =
 fun dist multiplier ->
  if multiplier = 0. then []
  else
    match dist with
    | Bind (d, f) ->
        let c = enumerate d multiplier in
        List.concat_map c ~f:(fun (opt, p) -> enumerate (f opt) p)
    | Conditional (c, d) ->
        let c = enumerate d multiplier in
        List.map c ~f:(fun (x, p) -> (x, p *. c x))
    | Primitive p -> (
      match support p with
      | Discrete xs -> List.map xs ~f:(fun x -> (x, multiplier *. pdf p x))
      | Continuous -> raise Undefined )
    | Return x -> [(x, multiplier)]
    | Independent (d1, d2) ->
        cartesian (enumerate d1 multiplier) (enumerate d2 multiplier)
        |> List.map ~f:(fun ((x1, p1), (x2, p2)) -> ((x1, x2), (p1 *. p2)))
```

Listing 3.13 – *Enumerating all paths through a model*

This method is very naive, and therefore inefficient. Since we essentially take every possible execution trace, we do not exploit structure such as overlapping traces. This can be made slightly more efficient by using algorithms such as belief propagation [26], but they still only work on models made up from discrete distributions (and are not compatible with the way I represent models). Exact inference of this kind only works on models that can be represented as finite networks, and for Bayesian networks is in fact NP-hard [27]. So instead, most of my project focuses on approximate inference.

3.6.2 Rejection Sampling

In my implementation of rejection sampling, I take samples from the prior, with accumulated scores. If the score is above some constant threshold, then the sample is accepted, and rejected otherwise. The specific case of the general rejection sampling algorithm used here sets the proposal distribution as the prior, and we use the scores to approximate the density function of the posterior (Listing 3.14).

This method is naive, since it runs an entire trace even if the first condition dropped the score below the threshold. An optimisation I implemented is to short-circuit this, and reject as soon as the trace goes below the threshold. It is also implemented as a dist transformation, so can again be used with the same sample methods.

This particular function is hard rejection, since samples with a lower score are always rejected. I have also implemented functionality to perform 'soft' rejection. This method instead sets the probability of acceptance being the score attached to the sample.

```
let rejection ?(threshold = 0.) dist =
  (* repeat until a sample is accepted *)
  let rec repeat () =
    let* x, s = prior_with_score dist in
    if Float.(s > threshold) then return (x, s) else repeat () in
  repeat ()
```

Listing 3.14 – *Simplest rejection sampling method*

A problem with rejection sampling is if conditions make most execution traces very unlikely, it will take a very large number of samples to have enough (or any) accepted samples. An example is given in Listing 3.15, where the condition only has a 1% chance of being true. This means that, on average, for every 1000 samples, we will only accept one.

```
let* x = bernoulli 0.001 in
condition (x=0)
(return x)
```

Listing 3.15 – *An example of a model that is very inefficient under rejection sampling*

3.6.3 Likelihood Weighting (Importance Sampling)

Likelihood weighting is an importance sampling method, when the proposal distribution we use is the prior. We want any algorithm we use to be as general as possible, and not need to be tuned using auxiliary distributions chosen by hand. Since for any model we can find the prior distribution easily, it is natural to use this as a proposal distribution here - this can be seen in several of the implementations of inference.

The implementation of likelihood weighting is simple - we simply take a set of samples (with weights) from the prior, remove duplicates and normalise, and use this set of particles as a the categorical distribution representing the posterior.

```
let importance n d =
let particles_dist = sequence @@ List.init n ~f:(fun _ -> prior d) in
let* particles = particles_dist in
categorical particles
```

Listing 3.16 – *Likelihood weighting*

The sequence function is a monad function that takes a list of distributions and fold them together so that they act as a single distribution returning entire lists. This allows the use of `(let*)` to sample a set of particles at once, and use them directly as the distribution.

3.6.4 Metropolis Hastings (MCMC)

Metropolis Hastings is an MCMC algorithm, and so is used to find a Markov chain with the stationary distribution equal to the target distribution, here the posterior. There are many variants of this algorithm, and the one I implemented is the independent metropolis hastings

(IMH) algorithm. I use the prior as a proposal distribution, using scores as an approximation to a density function. The algorithm is outlined below.

- Let π be the target distribution that we want to sample from.
- Let q be the density function of the prior, approximated by the scores.
- Initialise by taking a sample from the prior as the first state in the chain.
- Let x be a sample from the prior.
- Let y be the last state in the chain.
- Calculate the acceptance probability, $\alpha(x, y)$ by (3.1)

$$\alpha(x, y) = \begin{cases} \min\left(\frac{\pi(y)q(x)}{\pi x q(y)}, 1\right) & \pi(x)q(x) > 0 \\ 1 & \pi(x)q(x) = 0 \end{cases} \tag{3.1}$$

- The state x is then accepted with probability $\alpha(x, y)$. If accepted, we use x as the next state, or if rejected, we re-use y as the next state.

This produces a Markov chain with transition probability:

$$p(x, y) = q(y)\alpha(x, y) \qquad y \neq x$$

It is known as 'independent' metropolis hastings (IMH) since subsequent candidate states (x) are independent of previous values of states.

I have implemented IMH as a function transforming distributions (`'a dist -> 'a dist`). This allows it to be composed with other inference algorithms, as well as allowing the standard sample function to be used on the output. To model a Markov chain, I use a `Core.Sequence.t` - which is a data structure for a lazy list. The constructor takes a function that takes a previous state to produce a new state and output a value - analogous to the transition function. In this case, the output is the same as the state.

```
let mh_transform ~burn d =
  let proposal = prior_with_score d in
  let iterate (x, s) =
    let y, r = sample proposal in
    let ratio = if Float.(s = 0.) then 1. else r /. s in
    let accept = sample @@ bernoulli @@ Float.min 1. ratio in
    let next = if accept then (y, r) else (x, s) in
    Yield (return next, next) in
  let seq = Sequence.unfold_step ~init:(sample proposal) ~f:iterate in
  let seq = Sequence.drop_eagerly seq burn in   (* burn initial *)
  let r = ref seq in
  let* _ = return () in  (* to close over the sequence ref *)
  match Sequence.next !r with
  | Some (hd, tl) ->
      let* x, _ = hd in
      r := tl ;
      return x
  | None -> raise Undefined
```

Listing 3.17 – *Metropolis hastings*

One important property of the return distribution is that consecutive sample statements will need to return different values (to simulate running the chain). In order to achieve this, I create some mutable state - the sequence, which will take a step every time sample is called on the output distribution. In order to make sure this sequence is persistent, I use a reference and put it after a bind (let*) statement, incrementing the chain every time the function is called (which is only on sampling). Since the bind statement contains a function, the reference is closed over and is persistent to the output distribution.

3.6.5 Bootstrap Particle Filter (SMC)

Particle Filters are a class of algorithms which use particles to approximate a posterior. This is similar to the technique I used in importance sampling (3.6.3), but the difference here is that the particles are sequentially updated as we observe condition statements (i.e. as we observe data). In fact, an example of an smc algorithm is sequential importance sampling, but here I use an algorithm called the bootstrap filter [21].

The code given in Listing 3.18 transforms a conditional distribution to a new conditional distribution. In order to find the posterior, we simply ignore the conditional by finding the prior after using the smc method.

The GADT is traversed top down, with particles being initialised at a 'leaf' - primitives or returns. From this root, bind functions apply functions to the particles, and conditional statements updates the weights and resamples. The `resample` function takes a set of particles and takes samples from this set with replacement - this is the 'bootstrap' resampling method. The output distribution is conditioned by the total weight of all particles.

Increasing the number of particles finds a more accurate distribution with a finer resolution, but also increases the amount of time and memory required.

3.6.6 Particle Cascade (SMC)

The particle cascade algorithm (also asynchronous sequential Monte-Carlo) is an algorithm introduced in (Paige et al. 2014) [28], that extends the particle filter. It uses a lazily generated infinite set of particles, which allows it to be 'anytime', that is, it can generate more particles without having to start regenerate a large particle set from scratch. It also features a parallelisable re-sample step, although I will not make use of this feature, since I am not using multi-core OCaml.

The main implementation difference is that instead of resampling, a particle 'branching' operation is used, which produces a lazily generated set of particles at each resample step. Each particle produces 0 or more children to be used in the next iteration.

3.6.7 Particle-Independent Metropolis-Hastings (PMCMC)

SMC and MCMC algorithms are two distinct classes of algorithms, but can be combined to produce more efficient inference procedures. A simple example of these algorithms is the particle-independent metropolis-hastings algorithm [29]. This algorithm first uses a particle filter to find an approximation of the posterior, then uses this approximation as a proposal distribution for Metropolis-Hastings.

```
let rec smc: 'a.int -> 'a dist -> 'a samples dist = fun n d ->
  match d with
  (* resample at each piece of evidence/data, *)
  | Conditional(c,d) ->
    let updated = fmap normalise @@
      condition' (List.sum (module Float) ~f:snd) @@
      let* last_particles = smc n d in
      let new_particles =
        List.map
          (* update particles by weight given by condition *)
          ~f:(fun (x,w) -> (x, (c x) *. w))
          last_particles in
      return new_particles
    in
    let ps* = updated in
    resample ps
  (* apply function to each particle, no resampling *)
  | Bind(d,f) ->
    let* particles = smc n d in
    mapM (fun (x,weight) ->
        let* y = f x in
        return (y, weight))
      particles
  (* initialise n particles wih weights from the pdf *)
  | Primitive d ->
    List.init n
      ~f:(fun _ -> (fmap (fun x-> (x, Primitive.pdf d x)) (Primitive d)))
    |> sequence
  (* initialise n particles with the same value and weight *)
  | Return x ->
    List.init n ~f:(fun _ -> return (x,1.))
    |> sequence
```

Listing 3.18 – *Particle Filter*

Due to the fact that my PPL allows composition of inference algorithms, a basic implementation is very simple. However, this composition is only possible since the `smc` function produces a dist with conditionals, which no other inference method does.

```
let pimh k n d = mh k (Smc.smc n d)
```

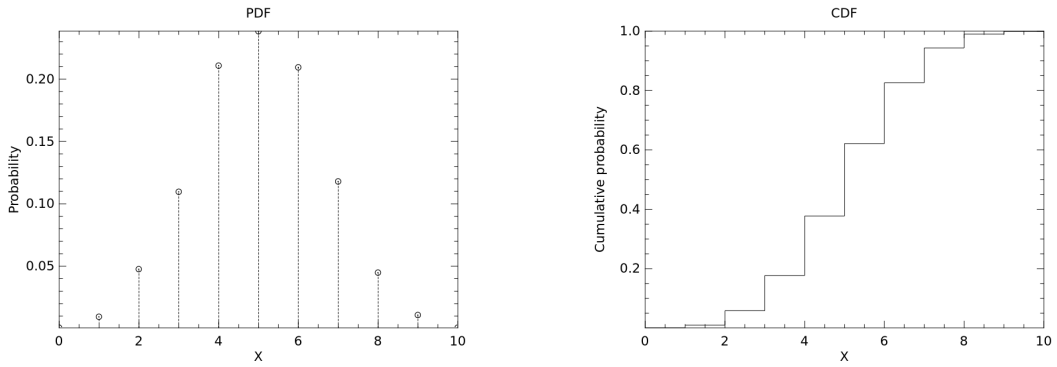
Listing 3.19 – *Particle-Independent Metropolis-Hastings*

3.7 Visualisations

Visualising the output distributions from inference can be done using the `Owl_plplot` module, which allows plotting directly from OCaml, rather than having to interface with other programs manually. I have implemented several functions which simplify visualising distributions created by my PPL.

Empirical distributions are approximated by histograms displayed as bar charts using

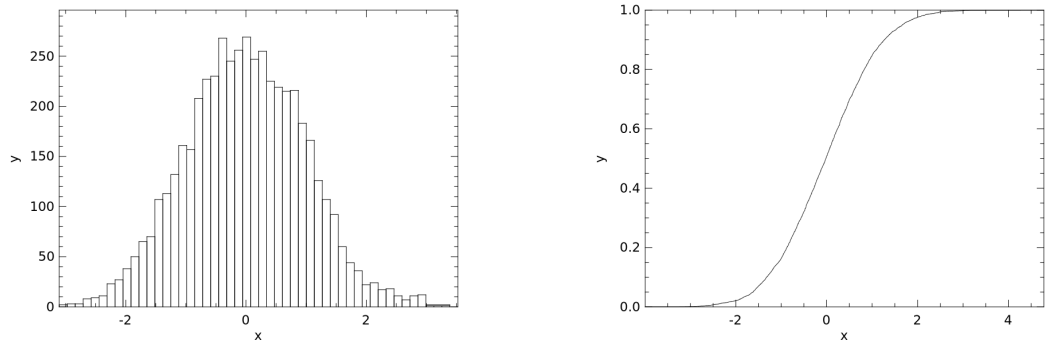
`Owl_plplot`. For discrete distributions, this conversion is simple - each bar is simply the probability of the distribution at each value in the support. This is calculated by drawing N samples, then for each value x_i, finding $\frac{n}{N}$, where n is the number of samples that equal x_i, to find the approximate probability of that value in the distribution, $P(X = x_i)$. Discrete distributions can also have their cdf visualised as a step function.



(a) *PDF* **(b)** *CDF*

Figure 3.2 – *Samples from a binomial distribution visualised,* $n = 10,000$

Continuous distributions are also displayed as histograms, with a set of samples being put into n equal width bins. The height of each bar is the pdf, which is calculated by finding the number of samples in each bin, then dividing by the total number of samples. To display the cdf, we can display the empirical cdf directly as a scatter plot, or join points to draw a step function.

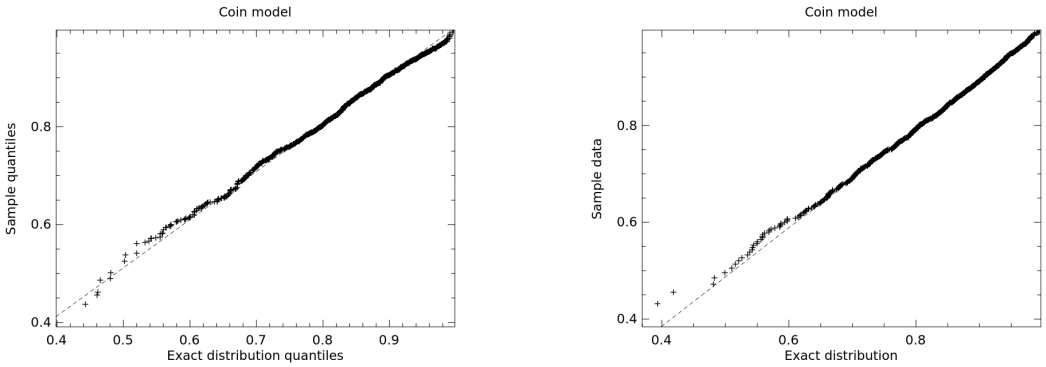


(a) *PDF* **(b)** *CDF*

Figure 3.3 – *Approximate pdf and cdf of samples from a standard normal distribution*

Other important visualisations for continuous distributions are the Q-Q and P-P plots. These provides a way to qualitatively compare distributions. P-P plots plot the cdfs of two distributions against each other, that is, for two cdfs F and G, the points $(F(z), G(z))$ are plotted for some values of z in the range $(-\infty, +\infty)$. Q-Q plots plot the quantiles of both distributions - it uses the inverse of the cdfs (the ppf) to plot the points $(F^{-1}(q), G^{-1}(q))$, where q, the quantile, is in the interval $[0, 1]$. This plots will generally use as many points as the data allows, and calculate the percentile for every data point available. For both plots, if all the points lie on the the line $y = x$, the distributions are identical. These plots are often used to

find the differences between a theoretical expected distribution and the distribution given by the data. This can be used in the PPL context to find whether a distribution given by a model matches what was expected in the theory. Figure 3.4 shows the output of inference for a model that is expected to output a beta distribution (the coin model in section 4.1.2) - the points are close to the expected line, showing successful inference.



(a) *Q-Q plot* **(b)** *P-P plot*

Figure 3.4 – *Plots to compare inferred distributions with the exact solutions*

For primitive continuous distributions, a smooth line can also be drawn since we have a function that can calculate the exact pdf or cdf. This can also be overlaid onto a histogram, to compare two distributions as in Figure 3.5.

```
Open Ppl.Plot

let () =
  coin_inferred
  |> hist_continuous ~fname
  |> add_pdf ~dist:coin_exact
  |> show
```

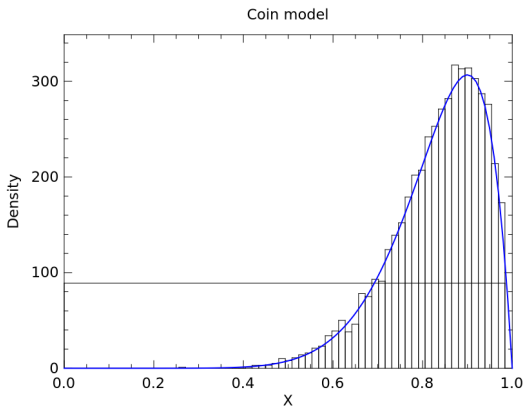


Figure 3.5 – *The approximate and exact pdf of the output of inference for a biased coin model, with code to produce plot*

3.8 Testing

Testing systems which are inherently random can be tricky, as it is difficult to test behaviour that is expected to change from one execution to the next. For a PPL, the most important functions to test are inference procedures, and ensuring the posterior samplers that are returned are correct. The issue is that the sequence of samples generated will change every run.

One approach is to set a fixed random seed and make sure the same sequence of results are produced. The aim of a unit test, however, is to make sure that a desired property does

not change from one version of the code to the next. The desired property here is that the samples fit a distribution, not that the same sequence is generated. For example, for sampling from $X \sim \mathrm{Bernoulli}(0.5)$, we might observe true,false,true,false. However, the sequence of false,true,false,true would also satisfy the distribution (and be correct), but would fail the test. Even with a fixed random seed, a change in code may cause new outputs even though the desired statistical property hasn't changed.

Another approach is to perform a hypothesis test such as Kolmogorov-Smirnov [30], to ensure distributions produced by my library are equal to what is expected. The alternative hypothesis is that the samples do not fit the distribution, and we aim to not accept this. A problem with tests of this kind is that they are expected to fail sometimes. There are two possible errors:

- *Type 1 error* - false positive where we reject a true null hypothesis, equal to the significance level of the test.
- *Type 2 error* - false negative where we fail to reject a false null hypothesis.

In unit testing terms, tests will sometimes fail for functions that work, and sometimes succeed for functions that are subtly broken, so unit tests based on hypothesis tests will be flaky. In fact, since we expect these tests to fail a certain percentage of the time, if they do not sometimes fail there is a problem with our program.

I decided to use hypothesis testing to evaluate my PPL's inference implementations, but did not build them into automated regression testing due to their flakiness. Instead, I used weaker unit tests for inference functions, testing that they could be applied to example models and produce samples without raising any exceptions. The only inference algorithm that could be unit tested thoroughly was the exact inference method, which is deterministic and always produces the correct (exact) posterior.

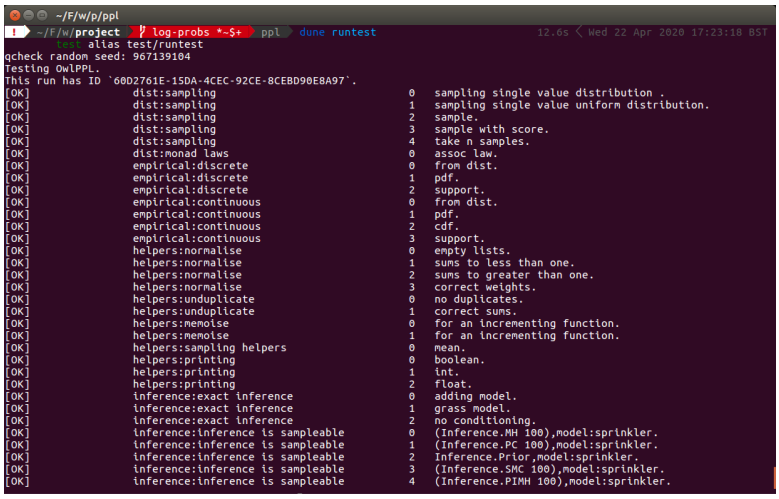
I wrote comprehensive unit tests for the deterministic functionality. The test framework I used, `Alcotest`, checks that outputs of functions match expected values. All the helper functions (e.g. normalise), the types which could be reliably created with the same values (e.g. empirical distributions), and simple distribution properties (e.g. sampling from a Dirac distribution always produces the same value) were tested.

I also used an additional library `Quickcheck`, to test that a specified invariant holds for a function - it uses randomly generated inputs to check a wide range of values. As an example (Listing 3.20), for the normalise function, we expect that the output probabilities always sum to one, no matter the input array - Listing 3.20.

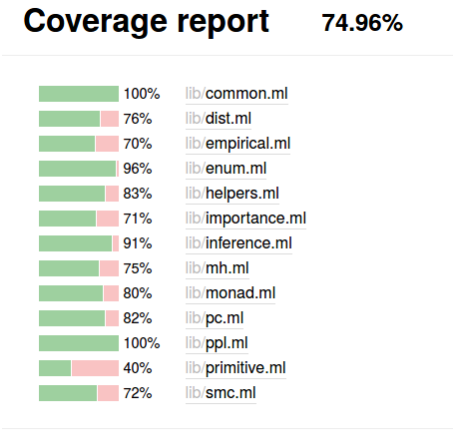
```
let test_normalise_sum_to_1 =
QCheck.Test.make ~count:1000 ~name:"test normalisation"
QCheck.(list (pair int float)) (* type to randomly generate *)
(fun l -> (List.sum ~f:snd (normalise l)) = 1.)
```

Listing 3.20 – *Testing the normalisation function for particles*

A subset of the test output and code coverage are given in Figure 3.6.



(a) *Test report, abbreviated*



(b) *Code coverage report*

Figure 3.6 – *Output from running unit tests*

4 | Evaluation

So far, I have developed a PPL that can be used to define arbitrary probabilistic models and perform Bayesian inference on them. To evaluate the performance of my PPL, I will present some examples to show programs written in my PPL and translated into equivalent programs in other PPLs, and then measure time and memory consumption of inference[1]*. I will also determine the correctness of inference procedures on simple problems by using hypothesis tests to assert posterior samples fit the expected distribution.*

4.1 Examples

To show how my PPL would be used on real problems, I now outline a set of example problems. The first examples here are simple, and have analytic solutions - this is so that I can then test the correctness of applying inference on them. More complex realistic models are also included to test performance. Full derivations of the solutions as well as mathematical descriptions of the models are given in appendix A.

4.1.1 Sprinkler

The sprinkler model is a commonly used example in Bayesian inference due to it's simplicity. It is an example of a *Bayesian network*, and can be visualised as in Figure 4.1. The code in Listing 4.1 shows the model in the diagram encoded as a program. This particular program models the probability of rain given that the grass is wet.

4.1.2 Biased Coin

Modelling a biased coin shows an example of a very simple model with a continuous posterior that can be calculated analytically [31]. The problem is that given a coin, we flip it 10 times and observe 9 heads. We not want to find out whether or not the coin is biased and with what weight (how likely is it to flip a heads). This model uses an uninformative prior, and the posterior over the weight of the coin is $\text{Beta}(10, 2)$ (derivation given in A.2)

The program in my PPL is shown in Listing 4.3, and demonstrates setting up the model, performing inference as well as finding the mean of the posterior. The application is to find the chance of the next coin flip landing heads.

The comparison given in Figure 4.2 shows how the same model is defined in other languages. Both languages use similar constructs, despite differing syntax. This example also shows that my PPL is similar to existing systems, and is not more verbose.

4.1.3 HMM

Hidden Markov models are slightly more involved models, where we have a sequence of hidden states, which emit observed states. There are two distributions involved here, the transition distribution, which defines how likely the next state is given the current state, and

[1] All tests are carried out on a single core of an Intel$^{(R)}$ Core$^{(TM)}$ i5-7200U CPU @ 2.50GHz

```
let sprinkler_model =
  let* cloudy    = bernoulli .8 in
  let* rain      = bernoulli
   (if cloudy then .8 else .1) in
  let* sprinkler = bernoulli
   (if cloudy then .1 else .5) in
  let* wet_grass = bernoulli
    (match rain,sprinkler with
         true,true -> .99
       | true,false -> .9
       | false,true -> .9
       | false,false -> 0.) in
  condition wet_grass
    (return rain)
```

Listing 4.1 – *Sprinkler model in OwlPPL*

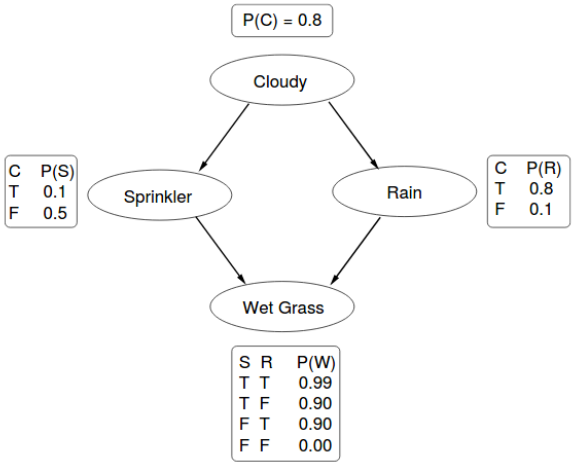


Figure 4.1 – *Sprinkler model as a network*

```
let () =
  exact_inference sprinkler_model
  |> print_exact_bool
(*false: 0.13706 true: 0.86294*)
```

Listing 4.2 – *Output of inference*

```
let coin_model inference_algorithm =
  let coin heads =
    let* theta = continuous_uniform 0. 1. in
    observe heads (binomial 10 theta) (return theta) in
  let posterior_single_coin = infer (coin 9) inference_algorithm in
  sample_mean ~n:10000 posterior_single_coin

(* 0.833 *)
```

Listing 4.3 – *Coin model - getting the mean of the posterior*

```
var coin_model = function (method) {
  var coin = function () {
    var theta = uniform(0.0, 1.0);
    observe(
      Binomial({ n: 10, p: theta }),9
    );
    return theta;
  };
  return Infer(method, coin);
};
```

Listing 4.4 – *WebPPL*

```
(defn coin_model [inference_method]

 (defquery coin
   (let [theta (sample (uniform 0 1))]
     (observe (binomial 10 theta) 9)
     (predict (theta))
   )
 )
 (doquery inference_method coin [])
)
```

Listing 4.5 – *Anglican*

Figure 4.2 – *The coin model in WebPPL (JS) and Anglican (Clojure)*

the emission distribution, which is the distribution over the observed states given the hidden state. The example in Listing 4.6 uses discrete distributions, but any type of distribution can

be used. The exact posterior for simple models can be found using the forward-backward algorithm, detailed in A.3.

```
type 'a hmm_model = {states: 'a list; observations: 'a list}

let transition s = if s then bernoulli 0.7 else bernoulli 0.3
let observe s = if s then bernoulli 0.9 else bernoulli 0.1

let rec hmm n =
  let* prev =
    match n with
    | 1 -> return {states= [true]; observations= []}
    | _ -> hmm (n - 1)
  in
  let* new_state = transition (List.hd_exn prev.states) in
  let* new_obs = observe new_state in
  return
    { states= new_state :: prev.states
    ; observations= new_obs :: prev.observations }

let model =
  let obs = [false; false; false] in
  let* r = hmm 3 in
  condition Stdlib.(r.observations = obs) (return @@ List.rev r.states)
```

Listing 4.6 – *Hidden Markov Model*

4.1.4 Linear Regression

This example shows how to use multiple data points to infer a continuous distribution. This example can be used to infer the parameters of a line through a set of 2-D points. The fold function can be used to condition on many observations easily. The fmap function is used to map outputs from a distribution. Since the linear regression model produces tuples of parameters, we can create individual distributions over either one. The comparison programs in other languages are given in A.4.

```
open Ppl
let linreg_model points =
  let linreg' =
    let* m = normal 0. 2. in
    let* c = normal 0. 2. in
    List.fold
      points
      ~init:(return (m,c))
      ~f:(fun d (x,y) -> observe y (Primitive.(normal (m*x+c) 1.)) d)

let slope = fmap fst (linreg_model points)
let y_intercept = fmap snd (linreg_model points)
```

Listing 4.7 – *Linear Regression*

4.1.5 Infinite Mixture Model - Dirichlet Process

This example demonstrates the common task of clustering a set of data points without knowledge of the number of clusters. This is a model which cannot be expressed in PPLs such as STAN or Infer.Net, since it is a non-parametric Bayesian model - we do not know the number of clusters beforehand. The infinite nature of this model requires the use of unbounded recursion and demonstrates the power of universal PPLs.

The prior for this model is a Dirichlet process [32], which defines a distribution over distributions. This model is a Dirichlet Process mixture model with an infinite number of Gaussians components [33], and the number of clusters is allowed to grow with the dataset size. I use a mixture of Gaussians, meaning the likelihood of a point belonging to each cluster is given by different normal distributions.

```
let dpMixture =
  let rec sample_index residual idx =
    let* keep = bernoulli (residual idx) in
    if keep then return idx else sample_index residual (idx + 1) in
  let clusters =
    let class_dist = sample_index (memo (fun _ -> sample (beta 1. 1.))) 0 in
    let* variances = return @@ memo (fun _ -> 10. /. sample (gamma 1. 10.)) in
    let* means = return @@ memo (fun i -> sample @@ normal 0. (variances i)) in
    return (class_dist, variances, means) in
  let obs = [1.; 1.1; 1.2; 3.1; 3.2; 3.15; 3.24] in
  let init = fmap (fun x -> (x, [])) clusters in
  let score y (_cluster, var, mean) =
    Primitive.pdf Primitive.(normal mean (sqrt var)) y in
  let add_point d y =
    condition'
      (fun x -> score y (List.hd_exn (snd x)))
      (let* clusters, rest = d in
       let class_dist, variances, means = clusters in
       let* cluster = class_dist in
       let point = (cluster, variances cluster, means cluster) in
       return (clusters, point :: rest)) in
  List.fold_left obs ~f:add_point ~init

let cluster_assignments =
  dpMixture
  |> fmap (fun x -> List.rev (List.map ~f:(fun (x, _, _) -> x) (snd x)))
```

Listing 4.8 – *Infinite mixture model in OwlPPL, gets the cluster assigned to each data point*

4.2 Statistical tests

To evaluate the correctness of my PPL, I used statistical tests which measure goodness-of-fit, i.e. how similar two distributions are to each other. I compare the empirical distribution of 10,000 samples from an approximated distribution to an exact distribution which is calculated analytically. Test distributions (e.g. the χ^2 distribution) were calculated using `Owl`, and functions to calculate the test statistics and p-values are given in the `Evaluation` module.

For all tests described below, I set the significance level $\alpha = 0.05$ and use null and alternative hypotheses as follows:

H_0 : The sample data follow the desired distribution
H_1 : The sample data do not follow the desired distribution

4.2.1 Chi-squared

The χ^2 test is a simple goodness-of-fit test which can test discrete distributions. The test statistic is as follows, with each i being a distinct element in the distribution, x_i is the observed number of samples with the value i, and m_i is the expected number of samples for the value i.

$$X^2 = \sum_{i=1}^{k} \frac{(x_i - m_i)^2}{m_i}$$

This test statistic is compared against the critical value (at the significance level) of the chi-squared distribution, with $k - 1$ degrees of freedom , where k is the number of possible values of the distribution.

Results

Inference Method	Sprinkler	HMM
Rejection Sampling	0.927	0.955
Importance Sampling	0.193	0.955
Metropolis Hastings	0.927	0.649
Particle Filter	0.421	0.859
Particle Cascade	0.808	0.859
Particle-Independent Metropolis-Hastings	0.193	0.955

Table 4.1 – *p-values of χ^2 test on different models using different inference procedures*

Table 4.1 shows the results of carrying out the test on all inference procedures for the sprinkler and hidden markov models. All of the values are greater than 0.05, so we do not reject the null hypothesis in each case. Therefore we conclude that, at the 5% significance level, the distributions are not significantly dissimilar.

4.2.2 Kolmogorov-Smirnov

The Kolmogorov-Smirnov test is a non parametric test which is used to compare a set of samples with a distribution - this is the one-sample K-S test. There is also a two-sample K-S test, which compares two sets of samples against each other. I use the one-sample test here to compare samples taken from the inferred posteriors to their exact analytic solutions.

The test statistic is as follows, with $F_n(x)$ being the empirical cumulative distribution of n samples, and $F(x)$ being the exact cumulative distribution.

$$F_n(x) = \frac{1}{n} \sum_{i=1}^{n} I_{[-\infty,x]}(X_i)$$
$$D_n = \sup_x |F_n(x) - F(x)|$$

This test statistic is compared against the critical values of the Kolmogorov distribution, rejecting the null hypothesis if $\sqrt{n}D_n > K_\alpha$, where K_α is the critical value at the significance level α, and n is the number of samples.

Results

Inference Method	coin
Rejection Sampling	0.619
Importance Sampling	0.666
Metropolis Hastings	0.26
Particle Filter	0.666
Particle Cascade	0.946
Particle-Independent Metropolis-Hastings	0.977

Table 4.2 – *p-values of K-S test on different models using different inference procedures*

Table 4.2 shows that for the biased coin model, the p-value obtained from all tests are greater than then 0.05. This means we do not reject H_0 for any inference procedure, so we can be confident (at the 5% significance level) that the inference procedures are correct. We can be more sure of some inference procedures, indicated by higher p-values. These results show that there is no significant difference between the generated posterior and the real solution.

4.3 Convergence of sampling

I also used the KL-divergence metric to determine the (dis)similarity of two distributions. The formula for KL Divergence of discrete distributions P and Q is easy to calculate by (4.1)

$$D_{KL}(P \parallel Q) = \sum_{x \in \mathcal{X}} P(x) \log \left(\frac{P(x)}{Q(x)} \right) \tag{4.1}$$

The continuous version is similar, with p and q now being density functions as in (4.2).

$$D_{KL}(P \parallel Q) = \int_{-\infty}^{\infty} p(x) \log \left(\frac{p(x)}{q(x)} \right) dx \tag{4.2}$$

Since I only have a set of samples from q, this integral can't be calculated exactly - there is no exact density function. However, there are ways to estimate density functions, as in (Perez 2008) [34]. The first step is to approximate the cdf by finding the empirical cdf, P_e (4.3) then linearly interpolating between points to produce a continuous function P_c (4.4). Estimating the derivative of P_c then gives the pdf estimator, $\hat{P}$ (4.5).

$$P_e(x) = \frac{1}{n} \sum_{i=1}^{n} U(x - x_i), \qquad \text{where } U \text{ is the step function} \tag{4.3}$$

$$P_c(x) = \begin{cases} 0 & x < x_0 \\ a_i x + b_i & x_{i-1} \leq x \leq x_i \\ 1 & x_{n+1} \leq x \end{cases}, \qquad a_i \text{ and } b_i \text{ chosen to make } P_c \text{ continuous} \tag{4.4}$$

$$\hat{P}(x) = \frac{P_c(x + \delta) - P_c(x)}{\delta}, \qquad \text{for sufficiently small } \delta \tag{4.5}$$

The final estimator (4.6) is given by using the exact pdf of q and performing Monte Carlo integration.

$$\hat{D}_{KL}(P\|Q) = \frac{1}{n}\sum_{i=1}^{n} \log\left(\frac{\hat{P}(x_i)}{q(x_i)}\right) \tag{4.6}$$

Both the metrics (discrete and continuous) were computed with code written using the `EmpiricalDist` modules.

The idea behind conducting this test is ensuring that the KL divergence decreases as we take more samples from the posterior. This ensures that the solution converges to the correct distribution - a KL divergence of 0 implies the distributions are identical.

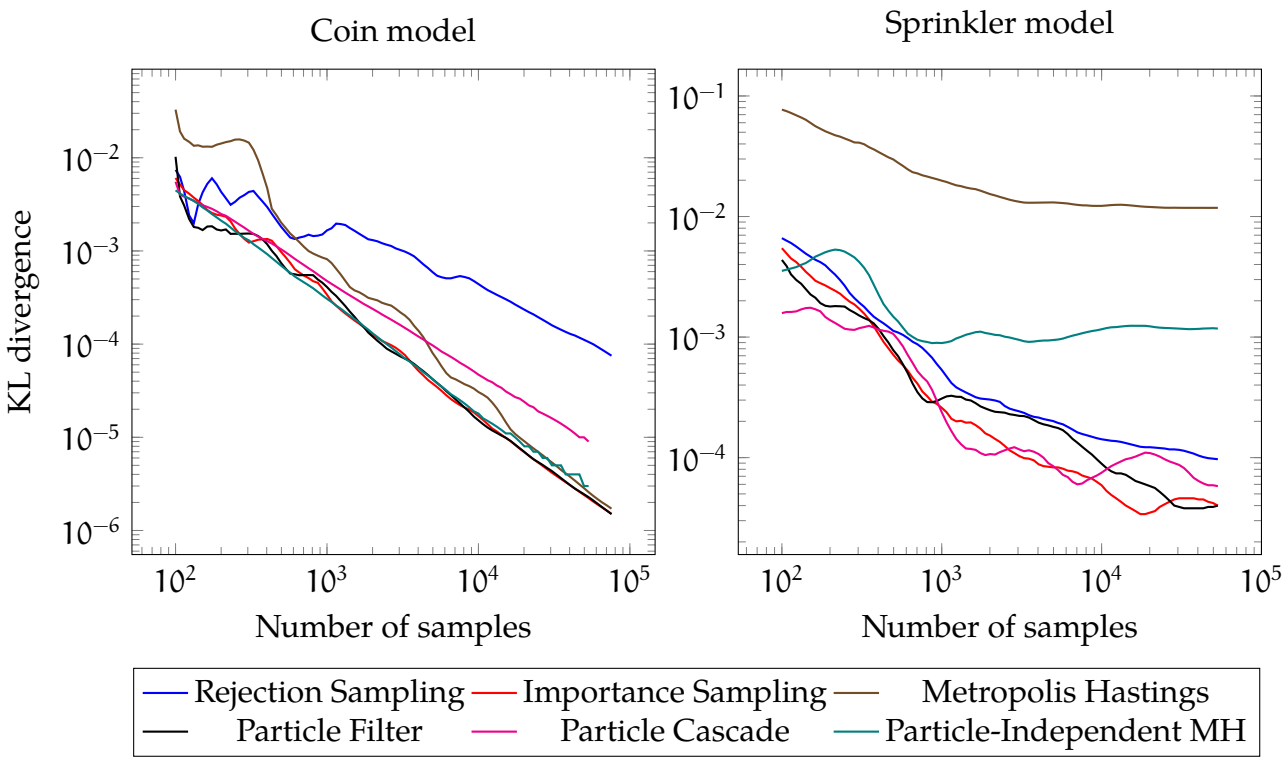


Figure 4.3 – *Plot of KL-divergence with increasing number of samples for different models and inference procedures - average over 20 runs*

Figure 4.3 shows the results of this test. For each inference procedure, we can see that the KL-divergence for each model generally decreases as we take more samples. Rejection sampling is consistently the worst performing inference procedure, with particle based methods such as the particle filter or importance sampling generating more accurate distributions. These plots have all been smoothed by taking a moving average in order to reduce the impact of noise.

4.4 Performance

I evaluated the performance of OwlPPL against Anglican and WebPPL. All of these languages are universal PPLs embedded in different host languages, so are comparable to my PPL. I only compare the inference algorithms for which there are comparable implementations in the other languages.

Figures 4.4 and 4.5 shows how my PPL compares against these languages for a range of models and inference procedures. All the models have been introduced previously, and have been shown to produce correct results in my PPL when using the given inference procedures. I measure both running time[2] and peak memory usage[3]. Here, smc refers to the particle filter implementation.

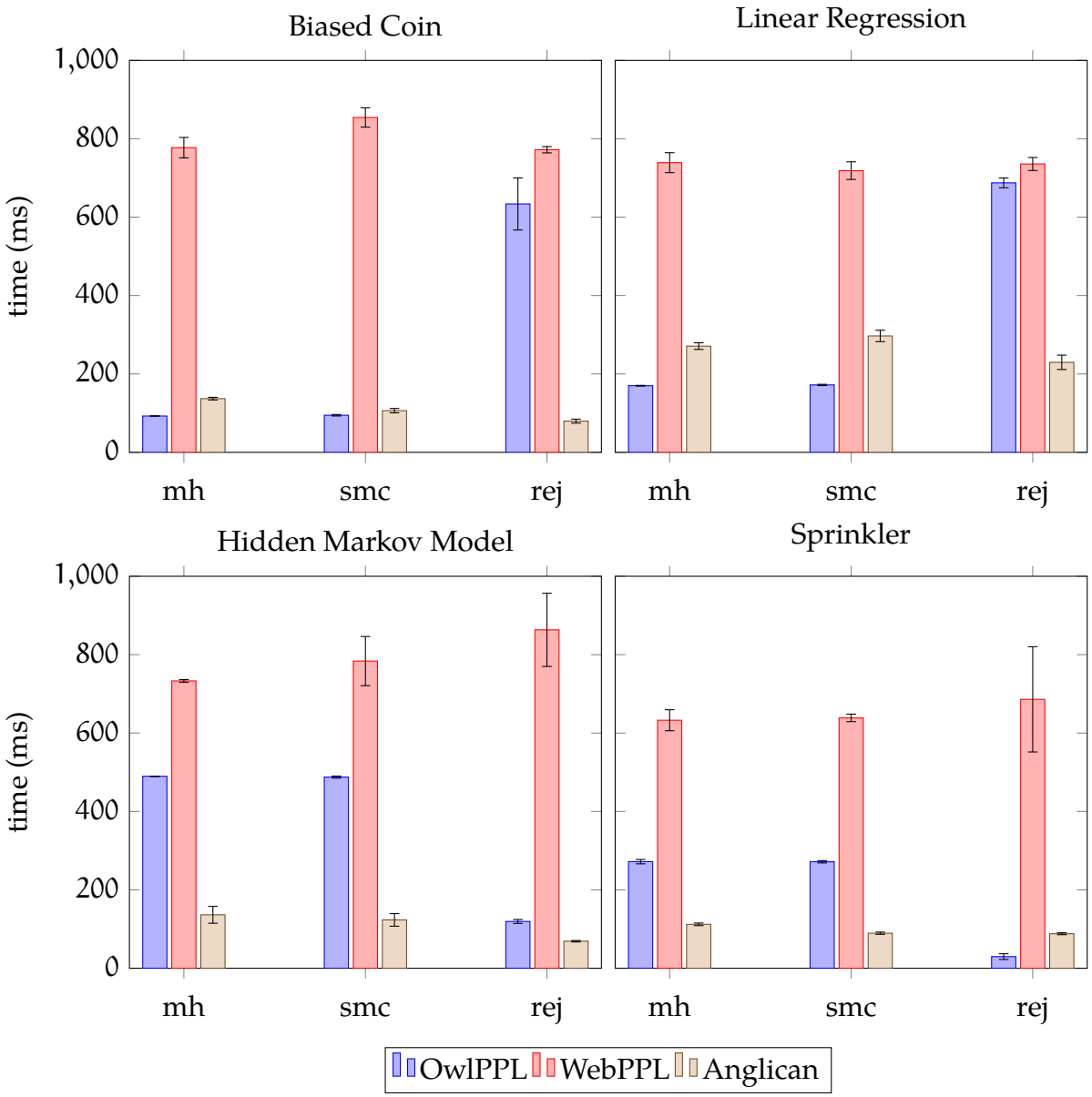


Figure 4.4 – *Time taken for inference against other languages for different models and inference algorithms, taking 10,000 samples from the posterior, averaged over 20 runs. Error bars show the 95% confidence interval*

These graphs show that OwlPPL performs consistently better that WebPPL in both memory and time. There is also less variance in the performance of OwlPPL, except in the case of rejection sampling for the coin model.

OwlPPL slightly outperforms Anglican on the two continuous models for some algorithms (My implementation of rejection sampling performs poorly on continuous models, which

[2]timed using libraries in respective languages
[3]computed using `time -f "%M"`

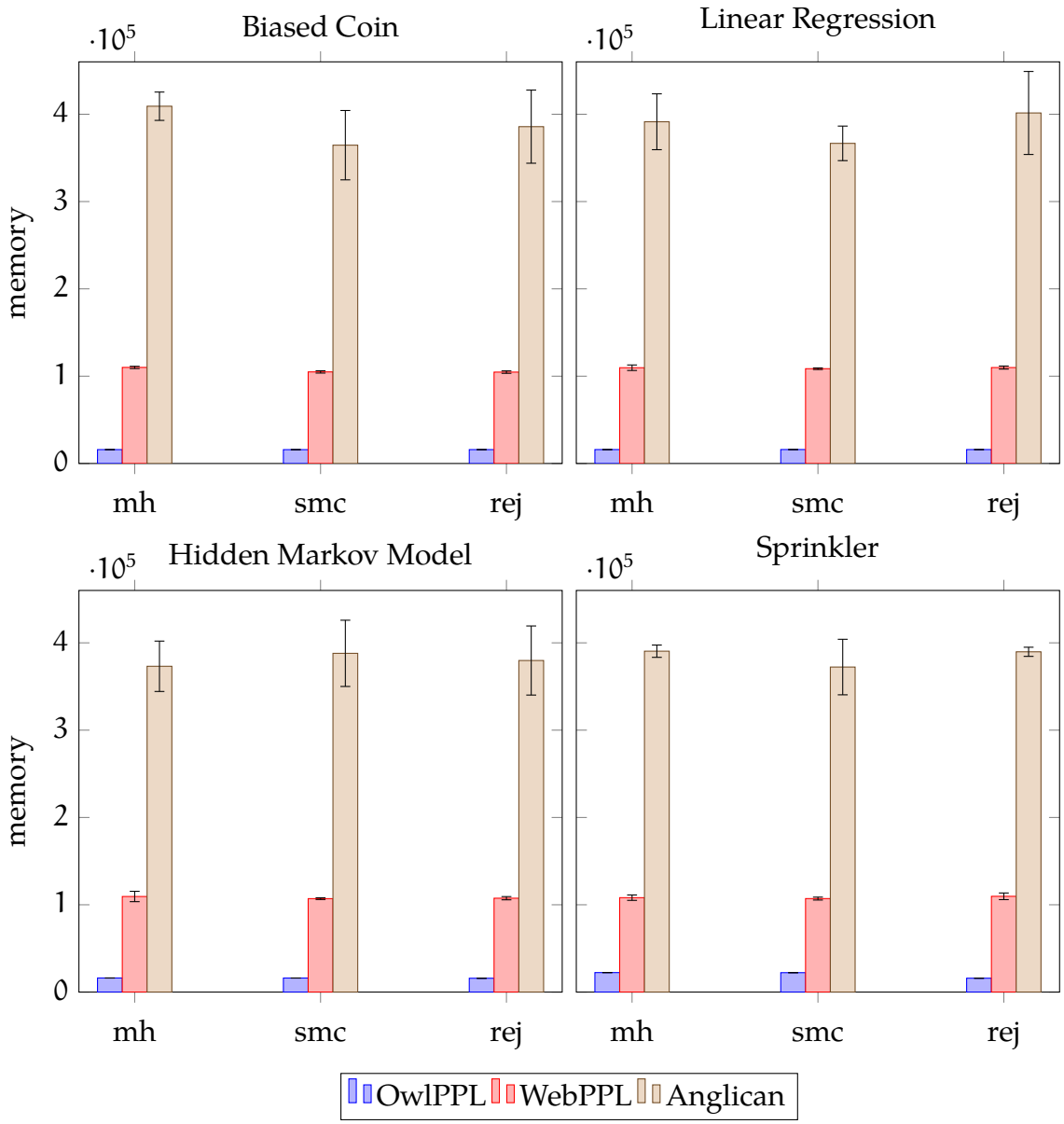


Figure 4.5 – *Memory Usage of my PPL, compared against other languages for different models, all using an MCMC algorithm, taking 10,000 samples from the posterior, averaged over 20 runs. Error bars show the 95% confidence interval*

is expected from my naive implementation), but is generally slower than Anglican for the discrete models. This may be because Anglican uses a more efficient representation for discrete distributions, and my representation may need to be optimised. Anglican is a Clojure library, which runs on the JVM, whereas WebPPL uses NodeJS, a JavaScript interpreter. WebPPL may incur and interpretive overhead, explaining slower running times.

My PPL is significantly better than both Anglican and WebPPL in terms of peak memory usage. All three languages are garbage collected, but OCaml does not run code through a virtual machine, and native binaries are generated. This could explain the lower memory usage, with the overheads of the JVM and the node runtime dominating in those languages.

I also measured the running time of inference with increasing data used. I tested the linear regression model, increasing the length of the array used as input. Models which are

conditioned on more data are expected to give more accurate results so it is desirable for inference to have low complexity.

My results show that running time increases linearly with the size of data. Figure 4.6 shows that all the inference functions run in time linear to the size of the input, but the constant factors vary substantially - for example, Sequential Monte Carlo has a much steeper gradient than Metropolis-Hastings, but both have the same shape. This also shows how some inference procedures take much more time than others, however, this often results in more accurate results. For example, the particle filter is slower than rejection sampling here, but Figure 4.3 shows that the particle filter produces more accurate posterior distributions. I use a log y-axis here due to the wildly varying gradients of the functions - the curves shown map to lines on a linear axis.

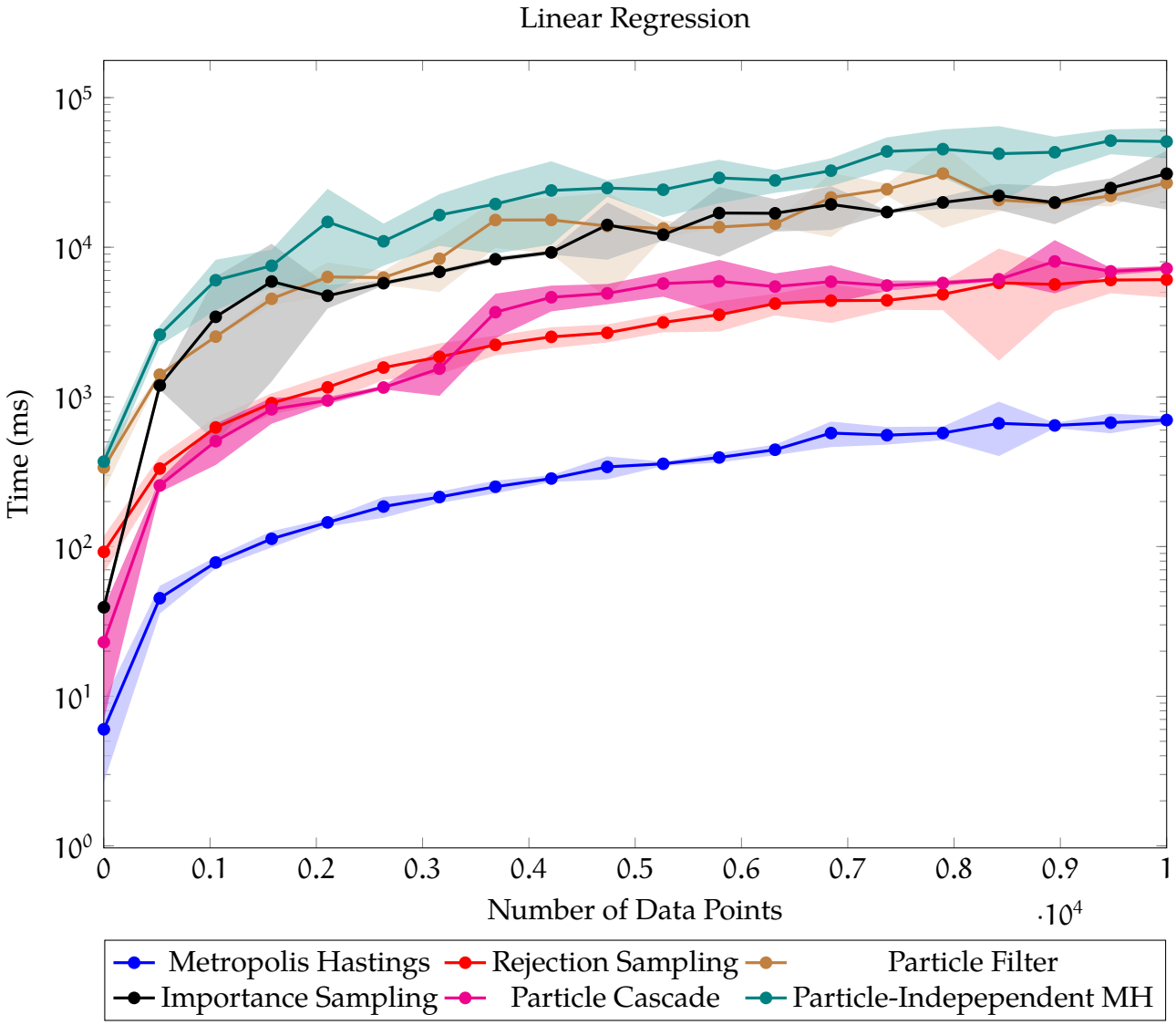


Figure 4.6 – *Time taken for inference as a function of input data length as the mean of 10 runs each taking 1,000 samples from the posterior, shaded areas are the 95% confidence interval ($\pm 2\sigma$)*

5 Conclusion

5.1 Work Completed

In this project, I have designed, developed and tested a probabilistic programming language embedding in OCaml. It has achieved all the core requirements as well as some of the extensions. My PPL can represent a wide variety of models, including infinite models with unbounded recursion, fulfilling the definition of a universal PPL. Standard OCaml features, such as pattern matching or higher order functions, and well as existing deterministic functions can be used in my PPL. Models can then be combined in complex ways, and existing OCaml libraries can be used within models.

I have also performed extensive evaluation of my PPL, showing that the performance is competitive with other universal PPLs. In particular, the memory usage of OwlPPL proved to be significantly lower than other languages, which could make it appropriate for edge computing. In addition, performing hypothesis tests shows that my PPL produces correct results, and my implementations of inference procedures are very unlikely to be faulty, which is the best guarantee that can be given. Programs written in my PPL are also not overly verbose when compared to these languages.

5.2 Further Work

Future work would mainly focus on how to improve inference. For example, there are several algorithms I have implemented that would benefit from the use of multiple cores - which may be possible with the ongoing development of multi-core OCaml. In particular, the Independent variant could benefit greatly from parallelisation.

I could also use more efficient inference algorithm implementations. This may require changing the core data structure or adding more variants to give more information, for example adding variable names in order to keep track of the primitives being used, or allowing a user to specify guide distributions for models to create more specific proposal distributions for MCMC (a technique used by several PPLs, including Pyro and WebPPL).

One of my initial goals was for my PPL to be able to represent as many types of model as possible. This prompted the use of a trace based approach (inspired by Church) rather than using a computation graph. However, there are recent universal PPLs which use dynamic computation graphs to make inference more efficient (e.g. Pyro). Since `Owl` contains a powerful computational graph implementation, this could be a further extension.

5.3 Lessons Learnt

While there are many possible extensions to this project, it successfully achieved all the initial goals, and I have learnt a great deal about OCaml and probabilistic programming, including the mathematics behind tractable Bayesian inference, the difficulties in testing statistical code as well as language design.

[**Word count**: 11998]

Bibliography

[1] Wally R Gilks, Andrew Thomas, and David J Spiegelhalter. A language and program for complex bayesian modelling. *Journal of the Royal Statistical Society: Series D (The Statistician)*, 43(1):169–177, 1994.

[2] Martyn Plummer. Jags: Just another gibbs sampler, 2004.

[3] Bob Carpenter, Andrew Gelman, Matthew D Hoffman, Daniel Lee, Ben Goodrich, Michael Betancourt, Marcus Brubaker, Jiqiang Guo, Peter Li, and Allen Riddell. Stan: A probabilistic programming language. *Journal of statistical software*, 76(1), 2017.

[4] Shen SJ Wang and Matt P Wand. Using infer. net for statistical analyses. *The American Statistician*, 65(2):115–126, 2011.

[5] Johannes Borgström, Ugo Dal Lago, Andrew D Gordon, and Marcin Szymczak. A lambda-calculus foundation for universal probabilistic programming. *ACM SIGPLAN Notices*, 51(9):33–46, 2016.

[6] Noah Goodman, Vikash Mansinghka, Daniel M Roy, Keith Bonawitz, and Joshua B Tenenbaum. Church: a language for generative models. *arXiv preprint arXiv:1206.3255*, 2012.

[7] Claus Möbus. Structure and interpretation of webppl. 2018.

[8] Frank Wood, Jan Willem van de Meent, and Vikash Mansinghka. A new approach to probabilistic programming inference. In *Proceedings of the 17th International conference on Artificial Intelligence and Statistics*, pages 1024–1032, 2014.

[9] Oleg Kiselyov and Chung-Chieh Shan. Embedded probabilistic programming. In *IFIP Working Conference on Domain-Specific Languages*, pages 360–384. Springer, 2009.

[10] Avi Pfeffer. The design and implementation of ibal: A general-purpose probabilistic language. *Applied Sciences*, 01 2000.

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

[12] Philip Wadler. Comprehending monads. In *Proceedings of the 1990 ACM conference on LISP and functional programming*, pages 61–78, 1990.

[13] Michele Giry. A categorical approach to probability theory. In *Categorical aspects of topology and analysis*, pages 68–85. Springer, 1982.

[14] Claire Jones and Gordon Plotkin. A probabilistic powerdomain of evaluations. In *Proceedings. Fourth Annual Symposium on Logic in Computer Science*, pages 186–187, 1989.

[15] Norman Ramsey and Avi Pfeffer. Stochastic lambda calculus and monads of probability distributions. In *Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 154–165, 2002.

[16] Liang Wang. Owl: A general-purpose numerical library in ocaml. *CoRR*, abs/1707.09616, 2017.

[17] Dustin Tran, Matthew D. Hoffman, Rif A. Saurous, Eugene Brevdo, Kevin Murphy, and David M. Blei. Deep probabilistic programming, 2017.

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

[19] Nicholas Metropolis and S. Ulam. The monte carlo method. *Journal of the American Statistical Association*, 44(247):335–341, 1949.

[20] Bernard D Flury. Acceptance–rejection sampling made easy. *SIAM Review*, 32(3):474–476, 1990.

[21] N. J. Gordon, D. J. Salmond, and A. F. M. Smith. Novel approach to nonlinear/non-gaussian bayesian state estimation. *IEE Proceedings F - Radar and Signal Processing*, 140(2):107–113, 1993.

[22] David Siegmund. *Note on a stochastic recursion*, volume Volume 36 of *Lecture Notes–Monograph Series*, pages 547–554. Institute of Mathematical Statistics, Beachwood, OH, 2001.

[23] Adam Ścibior, Zoubin Ghahramani, and Andrew D Gordon. Practical probabilistic programming with monads. In *ACM SIGPLAN Notices*, volume 50, pages 165–176. ACM, 2015.

[24] Martin Erwig and Steve Kollmansberger. Functional pearls: Probabilistic functional programming in haskell. *J. Funct. Program.*, 16(1):21–34, January 2006.

[25] Robert Zinkov and Chung chieh Shan. Composing inference algorithms as program transformations. *ArXiv*, abs/1603.01882, 2016.

[26] Judea Pearl. Reverend bayes on inference engines: A distributed hierarchical approach. In *Proceedings of the Second AAAI Conference on Artificial Intelligence*, AAAI'82, page 133–136. AAAI Press, 1982.

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

[28] Brooks Paige, Frank Wood, Arnaud Doucet, and Yee Whye Teh. Asynchronous anytime sequential monte carlo, 2014.

[29] Christophe Andrieu, Arnaud Doucet, and Roman Holenstein. Particle markov chain monte carlo methods. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, 72(3):269–342, 2010.

[30] Frank J Massey Jr. The kolmogorov-smirnov test for goodness of fit. *Journal of the American statistical Association*, 46(253):68–78, 1951.

[31] Damon Wischik. Foundations of data science, 2019.

[32] Yee Whye Teh. Dirichlet process., 2010.

[33] Carl Edward Rasmussen. The infinite gaussian mixture model. In *Proceedings of the 12th International Conference on Neural Information Processing Systems*, NIPS'99, page 554–560, Cambridge, MA, USA, 1999. MIT Press.

[34] Fernando Pérez-Cruz. Kullback-leibler divergence estimation of continuous distributions. In *2008 IEEE international symposium on information theory*, pages 1666–1670. IEEE, 2008.

[35] Dave Moore and Maria I. Gorinova. Effect handling for composable program transformations in edward2. *CoRR*, abs/1811.06150, 2018.

A | Example Programs

Here I give more detail about the models I used to evaluate OwlPPL, as well as derivations for the analytic solutions used where appropriate. I also list the code in Anglican and WebPPL that was compared against.

A.1 Sprinkler

This example implements a simple Bayesian network conditioned on a single value. It consists of 4 boolean random variables:

C - cloudy R - raining S - sprinkler is on W - grass is wet

Where the grass being wet is dependent on whether or not it is raining or the sprinkler is on, and subsequently the probability of rain and the sprinkler is dependent on whether or not it is cloudy. The network diagram is given in Figure 4.1.

A.1.1 Code

WebPPL

```
var sprinkler_model = function (method) {
  return function () {
    var sprinkler = function () {
      var cloudy = bernoulli({ p: 0.5 });
      var rain = cloudy ?
             bernoulli({ p: 0.8 })
           : bernoulli({ p: 0.2 });
      var sprinkler = cloudy ?
             bernoulli({ p: 0.1 })
           : bernoulli({ p: 0.5 });
      var wet_grass = rain || sprinkler;
      condition(wet_grass);
      return rain;
    };
    return Infer(method, sprinkler);
  };
};
```

Anglican

```
(defquery sprinkler
  (let [
    cloudy (sample (flip 0.5))
    rain
    (cond
      (= cloudy true) (sample (flip 0.8))
      (= cloudy false) (sample (flip 0.2))
    )
    sprinkler
    (cond
      (= cloudy true) (flip 0.1)
      (= cloudy false) (flip 0.5))
    wet-grass
    (cond
      (or (= rain true) (= sprinkler true))
        (flip 1.)
      :else (flip 0.))]
  (observe wet-grass true)
  rain))
```

Listing A.1 – *Sprinkler model in in Anglican and WebPPL*

A.1.2 Exact Posterior

The joint probability function is:

$$P(C, S, R, W) = P(W \mid S, R) \cdot P(S \mid C) \cdot P(R \mid C) \cdot P(C)$$

We require the probability that it is raining given that the grass is wet. This is found by summing over all the values for the extra variables we aren't interested in (A.1). The

C	R	S	W	Probability
F	F	F	T	0.0
F	F	T	T	0.081
F	T	F	T	0.009
F	T	T	T	0.0099
T	F	F	T	0.0
T	F	T	T	0.0144
T	T	F	T	0.5184
T	T	T	T	0.06336

C	R	S	W	Probability
F	F	F	F	0.09
F	F	T	F	0.009
F	T	F	F	0.001
F	T	T	F	0.0001
T	F	F	F	0.144
T	F	T	F	0.0016
T	T	F	F	0.0576
T	T	T	F	0.00064

probability of false can be calculated similarly or by subtracting from 1.

$$\begin{aligned} P(R = T \mid W = T) &= \frac{P(R = T, W = T)}{P(W = T)} = \frac{\sum_{C,S \in \{T,F\}} P(C, S, R = T, W = T)}{\sum_{C,R,S \in \{T,F\}} P(C, S, R, W = T)} \\ &= 0.8629428 \\ P(R = F \mid W = T) &= 1 - 0.8629428 \\ &= 0.1370572 \end{aligned} \tag{A.1}$$

A.2 Biased Coin

This example is of a coin which is flipped n times and lands on heads x times. We require the distribution over the weight of the coin (i.e. show likely it is to land on heads again). The likelihood model (X) is a binomial, and I use an uninformative prior, the uniform.

$$\begin{aligned} \text{Prior: } \Theta &\sim \text{Uniform}(0, 1) \\ \text{Likelihood: } X &\sim \text{Binom}(n, \theta) \end{aligned}$$

A.2.1 Exact Posterior

We then use Bayes' rule to calculate the posterior.

$$\begin{aligned} P(\Theta = \theta \mid X = x) &= \frac{1}{\kappa} P(\Theta = \theta) P(X = x \mid \theta) \\ &= \frac{1}{\kappa} \binom{n}{x} \theta^x (1 - \theta)^{n-x} \\ &= \frac{1}{\kappa'} \theta^x (1 - \theta)^{n-x} \end{aligned}$$

$$\kappa' = \int_{\phi=0}^{1} \phi^x (1 - \phi)^{n-x} \, d\phi$$

By matching parameters, we can see this is the beta distribution - $\text{Beta}(x + 1, n - x + 1)$.

A.3 Hidden Markov Model

The HMM I use is defined as below:

Set of hidden states	Set of observed states	Starting state
{True, False},	{True, False}	True

Transition matrix:

$$T = \begin{pmatrix} 0.7 & 0.3 \\ 0.3 & 0.7 \end{pmatrix} \tag{A.2}$$

Emission matrix:

$$O = \begin{pmatrix} 0.9 & 0.1 \\ 0.1 & 0.9 \end{pmatrix} \tag{A.3}$$

A.3.1 Code Samples

WebPPL

```
var hmm_model = function () {
  var transition = function (s) { return s ? flip(0.7) : flip(0.3); };
  var emission = function (s) { return s ? flip(0.9) : flip(0.1); };
  var hmm = function (n) {
    var prev = n == 1 ? { states: [true], observations: [] } : hmm(n - 1);
    var newState = transition(prev.states[prev.states.length - 1]);
    var newObs = emission(newState);
    return {
      states: prev.states.concat([newState]),
      observations: prev.observations.concat([newObs])};
  };
  var trueObs = [false, false, false];
  var model = function () {
    var r = hmm(3);
    factor(_.isEqual(r.observations, trueObs) ? 0 : -Infinity);
    return r.states;
  };
};
```

Anglican

```
(defquery hmm
  (let [
    observations [false false false]
    init-dist    (categorical {true 1.})
    transition  {true  (flip 0.7) false (flip 0.3)}
    emission    {true  (flip 0.9) false (flip 0.1)}]
    (reduce
      (fn [states obs]
        (let [state (sample (get transition (peek states)))]
          (observe (get emission state) obs)
          (conj states state)))
      [(sample init-dist)]
      observations)))
```

Listing A.2 – *HMM in Anglican and WebPPL*

A.3.2 Exact Posterior

With 3 observations, we can find the posterior for each individual hidden state using the forward backward algorithm. For given observations, $o_{1:T} := o_1, \ldots, o_T$, for the hidden state at each time-step t, $X_t \in \{X_1, \ldots, X_T\}$, we compute the distribution $P(X_t \mid o_{1:T})$ using (A.4).

$$\begin{aligned} \text{Forward: } & P(X_t \mid o_{1:t}) \\ \text{Backward: } & P(o_{t+1:T} \mid X_t) \\ \text{Combined: } & P(X_t \mid o_{1:T}) = P(X_t \mid o_{1:t}, o_{t+1:T}) \propto P(o_{t+1:T} \mid X_t) \cdot P(X_t | o_{1:t}) \end{aligned} \tag{A.4}$$

I use an example with three hidden states. The final posterior over each of the three states (not including the starting state) given the observations `[False,False,False]` is:

State	True	False
1	0.92528736	0.07471264
2	0.68390805	0.31609195
3	0.84482759	0.15517241

A.4 Linear Regression

For a standard linear regression for 2 dimensional data, the setup is

$$y = \beta_1 x_1 + \beta_0 + \epsilon$$

where the βs are the terms to be determined, and the ϵ represents random error. The frequentist approach is to minimise the sum of least squares between the known values and the predicted outputs, to find a single best set of values for the parameters. In Bayesian linear regression, we also use some prior distributions to augment the data. Here, I use (A.7) as the likelihood model, with (A.5) and (A.5) being the priors over the slope and y-intercept.

$$\beta_0 \sim N(0, 1) \tag{A.5}$$

$$\beta_1 \sim N(0, 1) \tag{A.6}$$

$$y \sim N(\beta_0 + \beta_1 x_1 + \epsilon, 1.) \tag{A.7}$$

A.4.1 Code Samples

WebPPL

```
var linreg_model = function () {
  var xs = [0, 1, 2, 3, 4, 5, 6, 7];
  var ys = [0, 2, 4, 6, 8, 10, 12, 14];
  var model = function () {
    var m = gaussian(0, 2);
    var c = gaussian(0, 2);
    var f = function (x) {
      return m * x + c;
    };
    map2(function (x, y) {
      factor(Gaussian({
            mu: f(x),
            sigma: 1,}).score(y));
      }, xs, ys);
    return m;
  };
};
```

Anglican

```
(defquery linreg_model
  (let [m (sample (normal 0. 2.))
        c (sample (normal 0. 2.))
        observations [[0 0] [1 2]
                      [2 4] [3 6]
                      [4 8] [5 10]
                      [6 12] [7 14]]
        ]
    (map
     (fn [[x y]]
       (observe
        (normal (+ c (* m x)) 1.)
        y))
     observations)
    m
  )
)
```

Listing A.3 – *Linear Regression in Anglican and WebPPl*

B | Documentation

This appendix contains a subset of the API provided by my library. I give the main parts of the API that a user would interact with, but omit some helper functions. This documentation is automatically generated by `ODoc`

Up – ppl » Ppl

B.1 Module `Ppl`

A shallowly embedded DSL for Probabilistic Programming

This is a library for universal probabilistic programming using distributions as monads. Includes several approximate inference algorithms. Contains utilities to plot distributions and evaluate correctness of inference.

- Core
- Extra

Core

`module Dist : sig ... end`
Module used for defining probabilistic models

`module Primitive : sig ... end`
Module defining a type for primitive distributions

`module Empirical : sig ... end`
A module for empirical distributions generated from samplers

`module Inference : sig ... end`
Implementation of inference algorithms

Extra

`module Plot : sig ... end`
Plotting utilies

`module Evaluation : sig ... end`
A module for evaluating the correctness of models and inference procedures

`module Helpers : sig ... end`
Utilities for working with distributions

Up – ppl » Ppl » Dist

B.2 Module `Ppl.Dist`

Module used for defining probabilistic models

Contains a type `dist` which is used to represent probabilistic models.

- Condition Operators
- Monad Functions
- Sampling
- Prior Distribution

```
exception Undefined
```

```
module Prob : Ppl__.Sigs.Prob
```
The module used to represent probability, can be switched to use log probs

```
type prob = Prob.t
```
A type for which values need to sum to 1 (not an enforced property)

```
type likelihood = Prob.t
```
A type for which values don't need to sum to 1 (not an enforced property)

```
type 'a samples = ('a * prob) list
```
A set of weighted samples, summing to one

```
type _ dist = private
```

`| Return : 'a -> 'a dist`	distribution with a single value
`| Bind : 'a dist * ('a -> 'b dist) -> 'b dist`	monadic bind
`| Primitive : 'a Primitive.t -> 'a dist`	primitive exact distribution
`| Conditional : ('a -> likelihood) * 'a dist -> 'a dist`	variant that defines likelihood model
`| Independent : 'a dist * 'b dist -> ('a * 'b) dist`	for combining two independent distributions

GADT for representing distributions, private to avoid direct manipulation

Condition Operators

```
val condition' : ('a -> float) -> 'a dist -> 'a dist
```
The most general condition operator

```
val condition : bool -> 'a dist -> 'a dist
```
Hard conditioning

```
val score : float -> 'a dist -> 'a dist
```
Soft conditioning, add a constant score to a trace

```
val observe : 'a -> 'a Primitive.t -> 'b dist -> 'b dist
```
Soft conditioning for observations from a known distribution

Monad Functions

Monad functions

```
type 'a t
```

```
val return : 'a -> 'a t
```

```
val bind : 'a t -> ('a -> 'b t) -> 'b t
```

```
val (>>=) : 'a t -> ('a -> 'b t) -> 'b t
```

```
val let* : 'a t -> ('a -> 'b t) -> 'b t
```

```
val fmap : ('a -> 'b) -> 'a t -> 'b t
```

```
val liftM : ('a -> 'b) -> 'a t -> 'b t
```

```
val liftM2 : ('a -> 'b -> 'c) -> 'a t -> 'b t -> 'c t
```

```
val mapM : ('a -> 'b t) -> 'a list -> 'b list t
```

```
val sequence : 'a t list -> 'a list t
```

```
val and* : 'a dist -> 'b dist -> ('a * 'b) dist
```

Primitives

These functions create `dist` values which correspond to primitive distributions so that they can be used in models.

```
type 'a primitive
```

```
val binomial : int -> float -> int primitive
```

```
val normal : float -> float -> float primitive
```

```
val categorical : ('a * float) list -> 'a primitive
```

```
val discrete_uniform : 'a list -> 'a primitive
```

```
val beta : float -> float -> float primitive
```

```
val gamma : float -> float -> float primitive
```

```
val continuous_uniform : float -> float -> float primitive
```

```
val bernoulli : float -> bool dist
```

Sampling

```
val sample : 'a dist -> 'a
```

```
val sample_n : int -> 'a dist -> 'a array
```

```
val sample_with_score : 'a dist -> 'a * likelihood
```

Prior Distribution

```
val prior' : 'a dist -> 'a dist
```

```
val prior_with_score : 'a dist -> ('a * likelihood) dist
```

```
val support : 'a dist -> 'a list
```

```
module PplOps : Ppl__.Sigs.Ops with type 'a dist := 'a dist
```

Common operators for combining distributions

B.3 Module Ppl.Inference

Implementation of inference algorithms

Inference algorithms to be called on probabilistic models defined using `Dist`. The `infer` method can be used to call all the inference algorithms listed, but the underlying method is also exposed for convenience. The

- Helpers
- Exact Inference
- Importance Sampling
- Rejetion Sampling
- Sequential Monte Carlo
- Metropolis Hastings
- Particle Independent Metropolis Hastings
- Particle Cascade
- Common

```
exception Undefined
```

```
type 'a samples = ('a * Dist.prob) list
```

Helpers

```
val unduplicate : 'a samples -> 'a samples
```

```
val resample : 'a samples -> 'a samples Dist.dist
```

```
val normalise : 'a samples -> 'a samples
```

```
val flatten : ('a samples * Dist.prob) list -> 'a samples
```

Exact Inference

```
val enumerate : 'a Dist.dist -> float -> 'a samples
```

```
val exact_inference : 'a Dist.dist -> 'a Dist.dist
```

Importance Sampling

```
val importance : int -> 'a Dist.dist -> 'a samples Dist.dist
```

Rejetion Sampling

```
type rejection_type =
                                    ______
                                   | Hard
                                   | Soft
                                   ______
```

```
val pp_rejection_type : Stdlib.Format.formatter -> rejection_type ->
unit
```

```
val show_rejection_type : rejection_type -> string
```

```
val rejection : ?n:int -> rejection_type -> 'a Dist.dist -> 'a Dist.dist
```

Sequential Monte Carlo

```
val smcStandard' : int -> 'a Dist.dist -> 'a Dist.dist
```

```
val smcMultiple' : int -> int -> 'a Dist.dist -> 'a Dist.dist
```

Metropolis Hastings

```
val mh' : int -> 'a Dist.dist -> 'a Dist.dist
```

Particle Independent Metropolis Hastings

```
val pimh' : int -> int -> 'a Dist.dist -> 'a Dist.dist
```

Particle Cascade

```
val cascade' : int -> 'a Dist.dist -> 'a Dist.dist
```

Common

```
type infer_strat =
```

```
| MH of int
| SMC of int
| PC of int
| PIMH of int
| Importance of int
| Rejection of int * rejection_type
| Prior
| Enum
| Forward
```

```
val pp_infer_strat : Stdlib.Format.formatter -> infer_strat -> unit
```

```
val infer : 'a Dist.dist -> infer_strat -> 'a Dist.dist
```

```
val infer_sampler : 'a Dist.dist -> infer_strat -> unit -> 'a
```

Up – ppl » Ppl » Empirical

B.4 Module `Ppl.Empirical`

A module for empirical distributions generated from samplers

Contains a signature as well as two implementations, for continuous and discrete distributions respectively

```
module type S = sig ... end
```

```
module Discrete : S
```

```
module ContinuousArr : sig ... end
```

Up – ppl » Ppl » Empirical » S

B.5 Module type `Empirical.S`

```
type 'a t
```

```
val from_dist : ?n:int -> 'a Dist.dist -> 'a t
```
Create a empirical distribution from a distribution object, using n samples to approximate it

```
val empty : 'a t
```
Create an empty distribution

```
val add_sample : 'a t -> 'a -> 'a t
```
Add another sample to the distribution

```
val get_num : 'a t -> 'a -> int
```
Get the numer of samples with the value

```
val get_prob : 'a t -> 'a -> float
```
Get the probability of a particular value

```
val to_pdf : 'a t -> 'a -> float
```
Create a pdf function

```
val print_map : (module Core.Pretty_printer.S with type t = 'a) -> 'a t ->
  unit
```
print the entire distribution

```
val to_arr : 'a t -> ('a * int) array
```
Get array of samples

```
val support : 'a t -> 'a list
```
Get the set of values for the distribution

Up – ppl » Ppl » Primitive » PRIM_DIST

B.6 Module type `Primitive.PRIM_DIST`

The signature for new primitive distributions

```
type t
```

```
val sample : unit -> t
```

```
val pdf : t -> float
```

```
val cdf : t -> float
```

```
val ppf : t -> float
```

```
val support : t support
```

Up – ppl » Ppl__Sigs » Prob

B.7 Module type `Ppl__Sigs.Prob`

This is an signature for types representing probabilities - it overloads mathematical operators which are commonly used in probability calculations. At the moment, there are two implementations - ordinary float probabilities and log probabilities. Both are backed by floats, but use of the LogProb module can lead to a reduced risk of underflow.

```
type t
```

```
val to_float : t -> float
```

```
val of_float : float -> t
```

```
val (=) : t -> t -> bool
```

```
val (*.) : t -> t -> t
```

```
val (/.) : t -> t -> t
```

```
val (+) : t -> t -> t
```

```
val min : t -> t -> t
```

```
val one : t
```

```
val zero : t
```

C | Project Proposal

Computer Science Tripos – Part II – Project Proposal

A probabilistic programming language in OCaml

2349E

Project Originator: Dr. R. Mortier

Project Supervisor: Dr R. Mortier

Director of Studies: (name removed)

Project Overseers: Dr J. A. Crowcroft & Dr T. Sauerwald

Introduction

A probabilistic programming language (PPL) is a framework in which one can create statistical models and have inference run on them automatically. A PPL can take the form of its own language (i.e. a separate DSL), or be embedded within an existing language (such as OCaml). The ability to write probabilistic programs within OCaml would allow us to leverage the benefits of OCaml, such as expressiveness, a strong type system, and memory safety. The use of a numerical computation library, Owl, will allow us to perform inference in a performant way.

PPLs work well when working with *generative* models, meaning the model describes how some data is generated. This means the model can be run 'forward' to generate outputs based on the model. The more interesting application, however, is to run it 'backwards' in order to infer a distribution for the model.

The power of a probabilistic programming language comes in being able to describe models using the programming language - in my case, probabilistic models would be described in OCaml code, with basic distributions being able to be combined using functions and operators as in OCaml code.

Starting Point

There do exist PPLs for OCaml, such as IBAL [10], as well as PPLs for other languages, such as WebPPL - JS [7], Church - LISP [6] or Infer.Net - F# [4] to name a few. My PPL can draw on some of the ideas introduced by these languages, particularly in implementing efficient inference engines.

I will be using an existing OCaml numerical computation library (Owl). This library does not contain methods for probabilistic programming in general, although it does contain modules which will help in the implementation of an inference engine such as efficient random number generation and lazy evaluation.

I have experience with the core SML language, which will aid in learning basic OCaml due to similarities in the languages, however I will still have to learn the modules system. 1B Foundations of Data Science also gives me an understanding of basic statistics and Bayesian inference. I do not have experience with domain specific languages in OCaml, although the 1B compilers course did implement a compiler in OCaml.

Substance and Structure of the Project

I will be building a PPL in OCaml, essentially writing a domain specific language. There are 2 main components to the system, namely the modelling API (language design) and the inference engine.

Modelling

The modelling API is used to represent a statistical model. For example, in mathematical notation, a random variable representing a coin flip may be represented as $X \sim N(0, 1)$, but in a PPL we need to represent this as code. An example would be

```
Variable<double> x = Variable.GaussianFromMeanAndVariance(0, 1)
```

in the Infer.Net language. In OCaml, there will be many different options for representing distributions, and a choice will need to be made about whether to create a separate domain specific language (DSL) or whether to embed the language in OCaml as a library.

I will also need to make sure the design of the modelling language is suitable for the implementation of the inference engine. For example, WebPPL [7] uses a continuation passing style transformation, recording continuations when probabilistic functions are called in order to build an execution trace. This then allows the engine to perform inference. A similar approach could be applied here. There are many alternative approaches to build execution traces, such as algebraic effects [35] or monads [23], and one such method will need to be chosen. However I decide to implement this, I will need to ensure that features of OCaml can be used appropriately.

Inference Engine

There are many different options for a possible inference engine. A decision also need to be made about whether to use a trace-based model (as mentioned) or a graph based model (such as Edward, where a computational graph is generated). This decision will need to be made before implementing the inference engine since it will affect the modelling language.

In both these cases, I will need to decide how to convert from a program into a data structure that allows inference to be performed, and then actually carry it out. Ideally, the inference algorithm used is separated from the definition of the models, so that different algorithms can be chosen, or new algorithms added in the future.

Evaluation

The PPL developed here will be compared to existing PPLs - for example, IBAL (written in OCaml), comparing performance for programs describing the same models. I will also use the PPL developed on example problems in isolation to ensure it can be used correctly and delivers correct results. An example would be to use it on an established dataset (e.g. the stop-and-search dataset used in 1B Foundations of Data Science) to attempt to fit a model.

I will also want to quantify exactly what kind of problems need to be supported by my PPL and make sure these kind of programs can be run. I will also support a minimum number of standard distributions, e.g. bernoulli, normal, geometric, etc. or enable users to define custom distributions.

Success Criteria

The project will succeed if a usable probabilistic programming language is created. Usable is defined by the following:

- **Language Features**: I will aim to support some subset of language features, such as 'if' statements (to allow models to be conditional), operators and functions. Some of these features may only be available by special added keywords (e.g. a custom 'if' function).

- **Available distributions**: I will aim to make sure my PPL has at minimum the bernoulli and normal distributions available as basic building blocks to build more complex probabilistic programs.
- **Correctness of inference**: I will use the PPL developed on sample problems mentioned before to ensure correct results are produced. This would be determined by comparing to results produced in other PPLs. I will aim to include at least one inference algorithm.
- **Performance**: This is a quantitative measure, comparing programs written in my PPL to equivalent programs in other PPLs. I can use the spacetime program to profile my OCaml code. Performing inference should be possible within a reasonable amount of time, even though the project does not have a significant focus on performance. I will also benchmark the performance with regards to scalability, i.e. ensure the performance is still reasonable as traces/graphs get larger.

Extensions

There are several extensions which could be considered, time permitting:

1. There could be more options for the inference engine, i.e. implementing more than one inference algorithm. Different algorithms are suited to different inference tasks, so this would be a worthwhile extension
2. Optimisations could be considered to ensure the performance of inference was better than other comparable languages - especially looking into making use of multicore systems.
3. I could add more distributions, as well as the ability to create custom distributions
4. Include the ability to visualise results using the plotting module in owl.
5. Include the ability to visualise the model in which inference is being performed (e.g. the factor graph)

Schedule

Planned starting date is 28/10/19, the Monday after handing in project proposal. Work is broken up into roughly 2 week sections.

Michaelmas Term

- **Weeks 3-4 (28/10/19 – 10/11/19)**
  Set up IDE and local environment - installing Owl and practicing using it. Read the first 10 chapters of Real World OCaml, available online. Read papers on past PPLs and implementations, both ocaml or otherwise. Set up project repository and directory structure.
  Milestone: learn OCaml basics, set up project
- **Weeks 5-6 (11/10/19 – 24/11/19)**
  Design a basic modelling API and write module/function signatures. Decide how to implement this (e.g. DSL vs library). Specify what language features I will include as a baseline. Research inference algorithms and make sure they will fit into the modelling

API designed.
Milestone: specification of which language features and inference algorithms will be implemented

- **Weeks 7-8 (25/10/19 – 08/12/19)**
  Begin to implement the modelling API, and allow running a model 'forward', i.e. generating samples.
  Milestone: A basic working DSL

Christmas Holidays

- **Weeks 1-2 (09/12/19 – 22/12/19)**
  Begin to implement a basic inference algorithm (such as MCMC) allowing programs to be run 'backwards' to infer parameters.
- **Weeks 3-4 (23/12/19 – 05/01/20)** [*Christmas Break*]
- **Weeks 5-6 (06/01/20 – 19/01/19)**
  Aim to finish the main bulk of implementation and get a baseline system working by the end of this week and consider extensions if finished early. Begin writing up progress report.
  Milestone: finish baseline implementation

Lent Term

- **Weeks 1-2 (20/01/19 – 02/02/20)**
  Finish progress report and implementation as well as any extensions, time permitting. Use the PPL developed on example problems in order to evaluate it, comparing against problems in other languages.
  Milestone: Progress report deadline (31/01/19)
- **Weeks 3-4 (03/02/20 – 16/02/20)**
  Prepare for the presentation, begin planning the dissertation, particularly the structure and the content I need to write for each section. Begin writing, starting with the first sections (i.e. introduction and preparation).
  Milestone: Progress report presentations (06/02/20), finish introduction and preparation
- **Weeks 5-6 (17/02/20 – 01/02/20)**
  Finish writing up the bulk of the implementation section.
  Milestone: Finish implementation section
- **Weeks 7-8 (02/03/20 – 15/03/20)**
  Complete first draft of dissertation, finish the evaluation and conclusion sections and complete any unfinished tasks.
  Milestone: Finish first draft

Easter Holidays

- **Weeks 1-6 (16/03/20 – 26/04/20)**
  Improve dissertation based on supervisor feedback

Easter Term

- **Weeks 1-2 (27/04/20 – 07/04/20)**
  Finalise disseration after proof reading and hand in.
  Milestone: Electronic Submission deadline (08/04/20)

Resources Required

Hardware I intend to use my personal laptop for the main development and subsequent write up (HP Pavilion 15, 8GB RAM, i5-8265U CPU, running Ubuntu and Windows dual booted).

Software The required software includes the ocaml compiler, with a build system (dune) and a package manager (opam). I will also use the IDE VSCode with an OCaml extension, as well as git for version control and latex for the write up.

Backups For backups, I will use GitHub to host my git repository remotely, pushing frequently. I will also backup weekly to a USB stick in case of failures. The software I require is available on MCS machines, so I'll be able to continue work in the event of a hardware failure with my laptop.