Anik Roy

A probabilistic programming language in Ocaml

Computer Science Tripos - Part II

Christ's College

April 13, 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 [signature]

Date: April 13, 2020

Proforma

Candidate Number:	Anik Roy
College:	Christ's College
Project Title:	A Probabilistic Programming Language in Ocaml
Examination:	Computer Science Tripos Part II
Word Count:	11245 [1]
Final Line Count:	[2]
Project Originator:	Dr R. Mortier
Supervisor:	Dr R. Mortier

Original Aims of the Project

The original aim of was to design and implement a language in OCaml, which allows the user to specify probabilistic models in code, and perform inference on these models automatically. Since Bayesian inference is intractable in practice, approximate algorithms need to be developed to achieve this. 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. The motivation is to separate the concerns of defining generative models and performing inference.

Work Completed

I have successfully implemented the core of my project, completing 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.

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

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

Special Difficulties

None

Contents

List of Figures

List of Listings

Chapter 1

Introduction

1.1 Motivation

Creating statistical models and performing inference on these models is an important part of data science. A probabilistic programming languages (PPL) is a language used to create models, and allow inference to be performed on these models automatically. This allows the problem of efficient inference to be abstracted away from the specification of the model, and means inference code does not have to be hand-written for every model, making the task of designing models easier. Probabilistic models can then be written by domain experts, without having to worry about the task of inference. The inference 'engine' can also implement many different inference algorithms, which will each be more or less well-suited to different types of models. The core idea is that we have a prior belief over some parameters, ($p(x)$, the generative model), and a set of conditions ($p(y|x)$) which specify the likelihood of observed data given those parameters. What we are interested in is the posterior, the (inferred) distribution over the parameters given the data we observe ($p(y|x)$). In general, this kind of Bayesian inference is intractable, so we must use methods which approximate the posterior.

PPLs usually allow us to create these models as programs. However, since generative models are built up by sampling from probability distributions, PPLs need some way of modelling this non-determinism. Being able to condition programs 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 essentially means generating samples using the model we write. However, when we include a condition statement, we can infer the distribution of the input parameters based on the data we observe.

PPLs can either be standalone languages or be embedded into another language. Embedding a PPL into a pre-existing language allows us to utilise the full power of the 'host' language, and gives us access to operations in the host language without having to implement them separately. 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 DSL into OCaml allows us to represent a wide range of models using OCaml's inbuilt syntax, and leverage OCaml features such as an expressive type system or efficient native code generation.

1.2 Related works

There are many examples of PPLs, even as domain specific languages (DSLs) in OCaml. Some PPLs choose to limit the models that can be expressed in them in order to make inference more efficient - for example HANSEI in OCaml can only represent discrete distributions [9]. Others, such as STAN or infer.NET (F#) can only operate on finite graphical models, and do not allow unbounded recursion when defining models. Some, such as Church or WebPPL, can express more general models, and are therefore known as 'universal' [] but often make the trade-off of less predictable or slower inference [].

Chapter 2

Preparation

In this chapter, I will discuss the research done before starting the project, and some of the design decisions made based on this. In particular, common patterns in OCaml (and functional programming in general), influenced the final DSL, as well as the design of other similar probabilistic programming systems. I also give a general description on several classes of inference algorithms.

2.1 Starting Point

There do exist PPLs for OCaml, such as IBAL [16] or HANSEI [9], as well as PPLs for other languages, such as WebPPL - JavaScript[12], Church - LISP[5] or Infer.Net - F#[24] 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 need 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 will want my PPL to be able to represent and the inference methods I implement.

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 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 main goal was to produce a usable language, which was defined by the following criteria:

- **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 the deterministic parts of their models. I need 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 results produced in other PPLs as well as comparisons 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 should be possible within a reasonable amount of time, even though the project does not have a significant focus on performance.

2.3 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 (Jane Street)[1].

2.3.1 Testing

Testing systems which are linked to randomness can be quite tricky, as it is difficult to test behaviour that is expected to change from one execution to the next. 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. Even with a fixed random seed, a change in code may cause new outputs even though the fundamental statistical property desired hasn't changed. Another approach is to perform some statistical test such as kolmogorov-smirnov [10], to ensure distributions produced by my library are equal to what is expected. A problem with tests of this kind is that they are expected to fail sometimes, meaning unit tests will provide limited utility due to their inherent flakiness with random programs. 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.

As such, most of the unit tests I wrote are fairly simple and only catch very basic bugs. Unit tests were possible to write for the exact inference procedure when working with discrete distributions, since we always expect the same output distribution. However, all the other inference algorithms

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

are approximate, so tests suffer from the problems described above. I was still able to carry out a full evaluation, performing hypothesis tests such as the kolmogorov-smirnov or chi-squared tests.

However, there are several auxiliary functions that can be tested effectively using conventional methods, so I was able to test them as normal. I also used the `Quickcheck` library to perform tests, which allows me to ensure certain invariants are preserved by functions by testing on large amounts of randomly generated inputs. I also use the `bisect_ppx` library to produce code coverage reports so that I can ensure I am thoroughly testing code.

2.3.2 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.

2.3.3 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.

2.4 Tools and Technologies

The main tools I used are listed here:

- Ocaml 4.08 - The language I wrote the main PPL library in
- Dune - Build system for OCaml
- Opam - OCaml package manager
- Alcotest - Unit testing framework
- Quickcheck - Random testing library
- Bisect_ppx - Code coverage tool
- Spacetime - Memory profiler for OCaml
- Gprof - A Linux profiler
- Owl - Scientific computing library in OCaml
- 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. The dune build system also allows me 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.

2.5 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 also 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 are hidden from the user, which can ensure only valid structures are created by the user.

2.5.1 GADTs

The main data structure I use to represent distributions is GADT - a generalised algebraic data type. GADTs are similar to ADTs (sum types), in that they have a set of constructors, but the main difference is that these constructors can have their output types annotated. The fact that the return types of constructors can vary is what makes GADTs more general than normal ADTs, whose return types are all the same. In OCaml, the syntax for this is `type \_ t = Constructor : type' -> type' t ...`. Since we can now know which constructors produce which concrete types of t (e.g. an `int t` or a `float t`, etc.), functions can be defined to only accept certain constructors, and we can make sure the whole structure is well typed.

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

2.5.2 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 datatypes 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 rewraps this value. The type must also satisfy a set of laws, which I omit here[22]. Monads can be used to structure programs in a general way, and allow side effects to be described in a pure way.

Probability Monad

It has been shown that probability distributions form a monad, [4] [8], and that they can be used to create distributions composed from other distributions [17]. In this case, `return x` represents a distribution with only one value, x (technically 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). It can be thought of as taking a sample from d, however, it is important to note that calling bind does not directly produce a sample, but exposes that structure to an interpreter (here 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 [2] outlines this, and a monad should provide a module which implements the (let*) and (and*) operators. The (let*) operator is the standard bind function - here 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 pair - 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 syntax allows the user to not have to use the bind (or the alias `>>=`) explicitly, and offers a more intuitive syntax.

```
open Option
(* new syntax *)
let add_options x y =
  let* a = x in
  let* b = y in
  x+y
(* transforms to *)
let add_options x y =
  x >>= (fun a ->
      y >>= (fun b ->
          a+b))
```

2.5.3 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. This allows modules to hide types and implementations to provide a clean API. Modules 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

[2] `http://caml.inria.fr/pub/distrib/ocaml-4.08/ocaml-4.08-refman.html`

must go through the modules 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 allows modules to be used in much the same way as ordinary values. This allows us to simplify the creation of modules from the users point of view, and means a library can define functions to create modules which can then be used by other functions.

2.6 Owl

Owl is a scientific computing library written for OCaml [23]. Importantly for my PPL, it contains functions for working with a wide variety of statistical functions. In particular, it contains functionality relating to many common distributions, e.g. normal, beta, binomial, etc. Since my language will allow the user to combine these basic distributions into larger models, I will need to use these functions to allow sampling from and the ability to perform inference on models. In particular, it is important to be able to find the probability density function (pdf) and cumulative density function (cdf) of these distributions as well as sample from them. Another important feature of owl is it's plotting functionality which enabled me to write functions which visualise output distributions directly from OCaml.

2.7 Approaches to probabilistic programming

Existing PPLs take several different forms, both as standalone and embedded languages. The main trade-off that is made in the design of PPLs is the number of models that can be expressed in the language compared to how efficient inference is. One approach is graph-based, where a *factor graph* is generated from the program, over which efficient inference can take place. This approach can be seen in languages such as infer.NET or JAGS, and has the benefit of very fast inference, particularly since efficient computation graph frameworks can be leveraged - an example is Edward [21], which uses TensorFlow as a backend. However, these languages usually cannot represent infinite models or unbounded recursion. Another approach, taken by the likes of WebPPL or Anglican, is trace-based. This approach considers execution traces, with a 'trace' being one run of a program, with all the intermediate variables taking a particular value. Inference algorithms can reason about these traces in order to produce a posterior distribution, as will be seen in the next section. A trace-based approach often leads to clearer programs, since we are not working with a computation graph. It also leads to more models being able to be expressed , since we are not limited by the constraints of a graph. These languages are often referred to as being 'universal'. However, inference is often slower, particularly when dealing with data in high dimensions, since inference algorithms need to be more general purpose, and often converge slower.

2.8 Bayesian Inference

Inference is the key motivating feature of probabilistic programming, and is a way to infer a distribution over the 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,

$$P(A \mid B) = \frac{P(B \mid A)P(A)}{P(B)}$$

with $P(A)$ being the prior, and $P(B \mid A)$ being the likelihood model we define. In the PPL setting, the prior is the generative model we define, with condition statements defining the likelihood. Running inference on the program then produces a sampler for the posterior.

This formula holds for the continuous case too, using probability density functions (f_X),

$$f_{X|Y=y}(x) = \frac{f_{Y|X=x}(y)f_X(x)}{f_Y(y)}$$

Unfortunately, exact Bayesian inference is usually computationally infeasible, especially when the number of random variables we consider is large. If we have 50 variables which can take one of two values, then we have to sum over 2^{50} values. There do exist some algorithms which operate on Bayesian networks (DAGs which represent random variables and their interdependence), and reduce the number of calculations needed. These include static methods such as *Variable Elimination* or *Message Passing* [15]. Alternatively, in the continuous case the formula is

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

with

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

The normalizing constant here is an integral that often does not have an analytic solution, and so must be approximated. Another issue is that directly sampling from a distribution requires that we also invert this formula (in order to use inversion sampling).

2.9 Inference Algorithms

Inference algorithms are ways to systematically generate samples from posterior distributions given a likelihood function and a prior distribution. In probabilistic programming, a model consists of latent 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:

$$p(x_{1:N}|y_{1:N}) \propto \tilde{p}(y_{1:N}, x_{1:N})$$

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.

Here, 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 ($\hat{p}$). The aim is then to find the posterior over the latent variables we are interested in (by marginalising 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[6]. In static methods, the program is compiled to a particular model (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 focus instead on dynamic methods, which use sampling to run programs and use conditioning statements that occur on these runs to perform inference.

2.9.1 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. To sample directly, we would also need to find the inverse cumulative distribution to be able to use the inversion sampling method.

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 finding every possible combination of these variables and their outcomes.

2.9.2 Rejection Sampling

Since exact inference is too difficult in practice, we usually have to resort to *Monte Carlo* [11] methods.

One such method, rejection sampling, is a very simple inference method which uses another distribution which *can* be sampled from. We take samples from this 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 [].

2.9.3 Importance Sampling

Importance sampling is another simple method, improving on rejection sampling, that can be used to sample from a target distribution using another distribution, known as the proposal distribution. We then 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.

2.9.4 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.

There exists several algorithms for finding this Markov chain, for example metropolis-hastings. This algorithm requires 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.

2.9.5 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,

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

the pdf of the posterior is represented by

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

where δ is the Dirac distribution

The simplest SMC algorithms are particle filters[7], 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 main feature of SMC algorithms is that they sequentially create sets of particles which converge to the desired distribution.

SMC methods can also be combined with MCMC methods to create new algorithms. These algorithms are known as particle Monte Carlo Markov chain (PMCMC) algorithms, and were first introduced for probabilistic programming in the Anglican language [27].

Chapter 3

Implementation

In this chapter I describe how the DSL is implemented. Documentation for all publicly exposed functions can be found in appendix B.

3.1 Repository Overview

The majority of my code is in the `ppl` directory, which contains the core library, unit tests, statistical testing code and some example programs written using the library.

I have implemented my ppl as a library residing in the `lib` subdirectory, contained within the module `Ppl`. Opening this module in a script allows a user to use my library. Ppl includes several submodules, most importantly the `Dist` and `Inference` modules, which contains the code for representing, creating and combining distributions, and performing inference, respectively. These functions can be found in separate files in the inference folder, and are all exported in the inference module. The other modules exported in Ppl are plot, prim_dist, and samples.

`Prim_dist` contains code for using primitive distributions - as well as types which a user can provide to define their own distributions. It also defines functions which can be used by the user to create new primitive distributions.

The `Plot` module contains helper functions which wrap around Owl_plplot, allowing users to easily create visualisations from distributions defined in my ppl. This includes histograms

The `bin` directory contains several example programs to show how the ppl can be used to express many different models.

The `test` directory contains basic unit tests as well as hypothesis tests to check the correctness of inference.

The `evaluation` directory contains code to compare my ppl to both hand-written inference procedures, as well as equivalent programs in other PPLs. There are several directories, which each correspond to a particular problem/model.

All code is written in OCaml 4.08, with the main dependencies being Jane Street's `Core` and `Owl` [23].

3.2 Language Design

I chose to implement my language as a domain specific language (DSL) shallowly embedded into the main OCaml language. This allows models built in the ppl to be easily composed with other standard OCaml programs.

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 importantly, recursion. This can allow us to define models that do not terminate and are therefore invalid. However, we can write functions which are *stochastic recursive* [20], 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.

I use a set of primitive distributions which can be combined (using arbitrarily complex deterministic OCaml code) to produce new more complex distribution. For example, one can take the sum of two discrete uniform distributions to simulate the addition of two dice rolls.

PPLs in general are similar to normal programming languages, but need to support two extra operations - `sample` and `condition` [6]. The sample operator allows models to be built up .The condition operator has a few variants, which are covered in section 3.4.

3.3 Representing Distributions

In order to define my DSL, I needed data structures to represent probabilistic models as well as distributions that are used as both input and output. Input distributions are the primitive distributions that are used to build models and constrain observations. We have more information about these distributions, such as how to sample from them exactly and exact pdf functions, and this information is used in inference. The output of inference is usually a distribution which can be sampled from to obtain the posterior, but we generally want to also collate statistics about the posterior distribution. Since we only have an approximate sampler, we can output empirical distributions from which many statistics, e.g. mean, variance, cdf etc., can be approximated.

3.3.1 Probability Monad

As mentioned before, monads are a natural way to represent 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 signature of bind is `'a m -> ('a -> 'b m) -> 'b m`, and return is `'a -> 'a m`, with m being the monad type.

Using monads also allows us 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 we have more information about the structure of the program. It is also a common pattern to set up a model by first independently drawing from several distributions. An example of let* and and* is given below.

```
let* x = normal 0. 1.
and* y = normal 0. 1. in
return (x + y)
```

I define my own functor for monads in order to define several helper functions. This functor takes a module wrapping any monad type and extends it with helper functions. The full module documentation for this can be found in appendix B (`Monad.Make_extended`). An example if 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.

```
let ( +~ ) = liftM2 ( +. )
(* the distribution created by adding two normals *)
let d = (normal 0. 1.) +~ (normal 0. 1.)
```

However, there are many different underlying data structures which can be used to represent distributions. The simplest is a list of pairs representing a set of values and corresponding probabilities, `('a * float) list`. This is a very convenient and natural way to represent discrete distributions, with return and bind defined as in listing 1. 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' and normalised. This approach has been used to create functional probabilistic languages [3], 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.

A major problem with this approach is that in flattening distributions, we must make sure that duplicated values are combined, and this approach is $O(n^2)$ when using a list since we must scan up to the length of the entire list for every element. A better option is to use a map, which is provided in the Jane Street Core library, and implemented as a balanced tree, significantly improving the time complexity of combining distributions.

```
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 (f x) p) d
  |> unduplicate |> normalise

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

Listing 1: Simple Probability Monad

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

let normalize map = (* omitted *) _

let bind d f =
  let new_map =  mapi d ~f:(fun ~key ~data -> data *. f key) in
  normalize new_map

let return x =
  let m = empty in
  add m x 1.
```

Listing 2: Simple probability monad using a map

Although this is not the final data structure I chose for general probabilistic models, it is the one I used for discrete distributions. I also used a `Map` for the data structure that produced approximations to discrete posterior distributions.

3.3.2 GADT

The structure that I landed on 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)[19]. The GADT 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. For inference, I provide some inference functions as transforming the 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. Primitive distributions also have a special variant (which takes a different `primitive` type).

We can find the exact pdf/cdf of these distributions, unlike the `dist` type, which can only be sampled from. Listing 3 shows each of the variants. The monad functions are also provided, which construct the corresponding variant 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.

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

let return x = Return x
let bind d f = Bind (d,f)
```

Listing 3: 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. Usually, this type is floats or integers, but polymorphism allows us to define distributions over anything, including distributions over distributions - a feature used in certain models such as a Dirichlet process.

3.3.3 Primitive Distributions

In PPLs, users build complex models by composing more simple elementary primitive distributions (ERP) [25]. These primitive distributions need to have a few operations defined on them, namely `sample, pdf, cdf` and `support`.

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. A concrete use case - 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 method 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 4.

An example of this being used to add a new primitive distribution is given in listing 5, for the specific case of the Poisson distribution. The `Poisson` distribution can be used as other primitives are, e.g. in `observe` statements.

```
module type PRIM_DIST = sig
  type t
  val sample: unit -> t
  val pdf: t -> float
  val cdf: t -> float
  val support: t support
end
```

Listing 4: Signature of the module that primitive distributions must implement

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

Listing 5: Adding a new distribution as a primitive

3.3.4 Empirical Distributions

The output of Bayesian inference is a probability distribution over the variable we are concerned with. Ideally, we would be able to produce an exact posterior distribution, and be able to extract exact statistics from it. However, approximate inference only allows us to create functions to sample from this posterior. Since my inference procedures all produce an object of type `'a dist`, we can define a signature for a type of an empirical distribution. The type is abstract since there are different implementations for discrete and continuous distributions. For discrete distributions, we use a `Core.Map`[1], whereas continuous distributions use a `Core.Set`[2]. Creating values with these types required passing a first class module representing the type of the keys - this is so that an appropriate compare function can be used in the internal tree data structure.

I also provide modules which are backed by the polymorphic versions of these data structures (`Core.Map.Poly`). These types don't require the use of first class modules to create objects, since the keys are compared using the polymorphic compare function. While this makes using the module simpler to use (no need to pass the first class module), it also makes them more inefficient due to the use of polymorphic comparison.

The signature given in listing 6 is implemented for both continuous and discrete distributions. In order to create the continuous case, I perform binning to approximate the continuous distribution by a discrete one - exemplified by figure 3.1. The number of bins used is calculated automatically from the range and shape of the samples drawn.

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

```
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 mean: 'a t -> 'a
  val support : 'a t -> 'a support
end
```

Listing 6: Signature for empirical distributions

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.

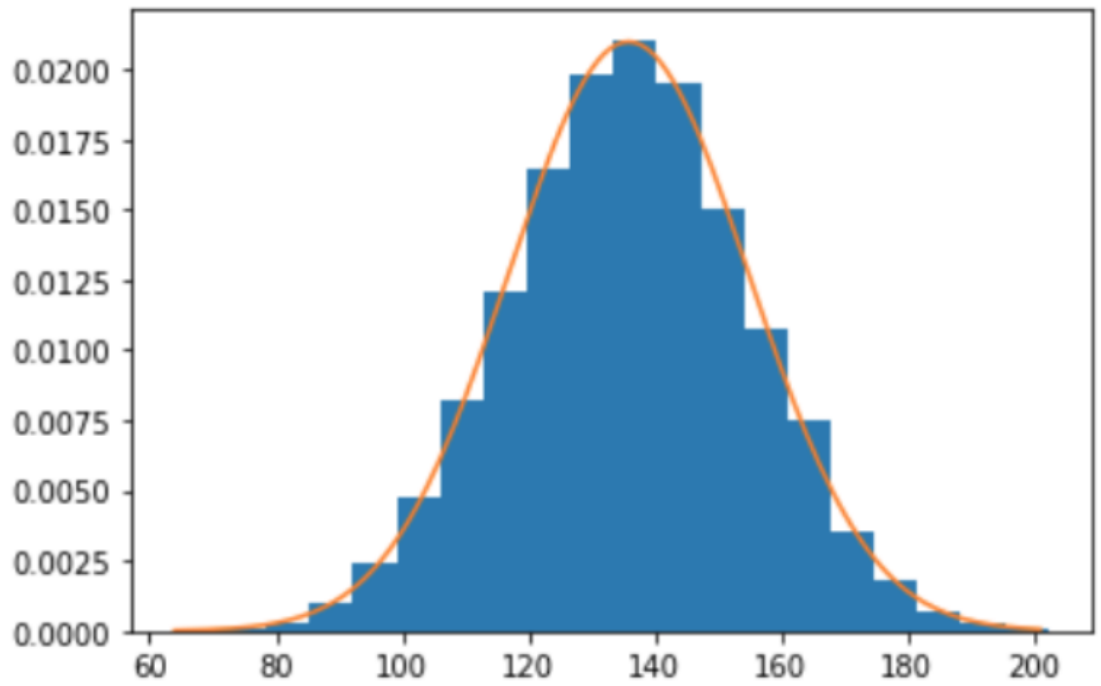


Figure 3.1: An example of approximating a continuous distribution using discrete bins

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 functions. 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 the below model, 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, the sum).

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

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 the model below, which constrains the standard normal distribution to be positive.

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

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 since it will just reject every trace. Instead, we can use the pdf function 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 (any float) to the trace. This is generally used in a branch, 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 7: 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 required 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 can be thought of as a posterior, $P(\theta \mid x)$, the distribution of a parameter given some observed data. The generative model, i.e. the program without condition statements, can be thought of as the prior distribution, $P(\theta)$. The condition statements then define the likelihood model, that is, $P(x \mid \theta)$, the probability of the observations in the current model (the prior). So finding the prior is the same as disregarding the conditionals (essentially ignoring the data). Sampling is only difficult in the presence of conditionals (as this requires inference), so this allows us to sample from the prior using the same sample function defined before. We can also transform any `'a dist` into a different `'a dist` that is the prior by ignoring conditional statements.

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 this to find the posterior directly. I have implemented several variants of functions for finding the prior and sampling, all with the same concept as below.

```
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
  | 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.)
```

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 Inference

Inference is the key motivation behind probabilistic programming. Up to this section, we 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 [19] [28]. 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

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.

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 [15], 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 exact inference for Bayesian networks is in fact NP-hard[2]. So instead, most of my project focuses on approximate inference.

3.6.2 Rejection Sampling

In my implementation of rejection sampling, we take samples from the prior, with accumulated scores. If the score is above some constant threshold, then we accept the sample, and if not, we reject the sample. The specific case of the general rejection sampling algorithm we use here sets the proposal distribution as the prior, and we use the scores to approximate the density function of the posterior. The implementation is shown in listing 9.

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. This does slightly increase the time taken for small models, and so is not the default. It is also implemented as a dist transformation, so can again be used with the same sample methods.

```
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)]
```

Listing 8: Enumerating all paths through a model

```
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 9: Simplest rejection sampling method

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.

A problem with rejection sampling is if conditions make most execution traces very unlikely. This means it will take a very large number of samples to have enough (or any) accepted samples. An example is given in listing 10, 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.

3.6.3 Likelihood Weighting

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

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

Listing 10: An example of a model that is very inefficient under rejection sampling

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 11: 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

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 implement here 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 since subsequent candidate states (x) are independent on previous values of states.

I have implemented IMH as a function transforming `'a dists` (`'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`, which is a data structure for a lazy list. The creator function uses 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.

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.

```
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
  match Sequence.next (!r) with
    Some (hd,tl) ->
    let* x,_ = hd in
    r:=tl; return x
  | None -> raise Undefined
```

Listing 12: Metropolis hastings

3.6.5 Bootstrap Particle Filter

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 a particle filtering algorithm is sequential importance sampling, but here I use an algorithm called the bootstrap filter[7].

The code given in listing 13 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, but also increases the amount of time and memory required.

3.6.6 Particle Cascade

The particle cascade algorithm (also Asynchronous Sequential Monte-Carlo) is an algorithm that extends the particle filter, introduced in (Paige et al. 2014)[14]. 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

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 (known as PMCMC), is the particle-independent metropolis-hastings algorithm[1]. This algorithm first uses a particle filter to find an approximation of the posterior, then uses this approximation as a prior distribution for metropolis-hastings.

Due to the fact that my PPL allows composition of inference algorithms, a basic implementation is very simple.

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

```
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 13: Particle Filter

However, there are flaws in this implementation, since any sampler produced is slow and can be made more efficient. In addition, the reason this works is because the `smc` function produces a dist with conditionals, which no other inference method does.

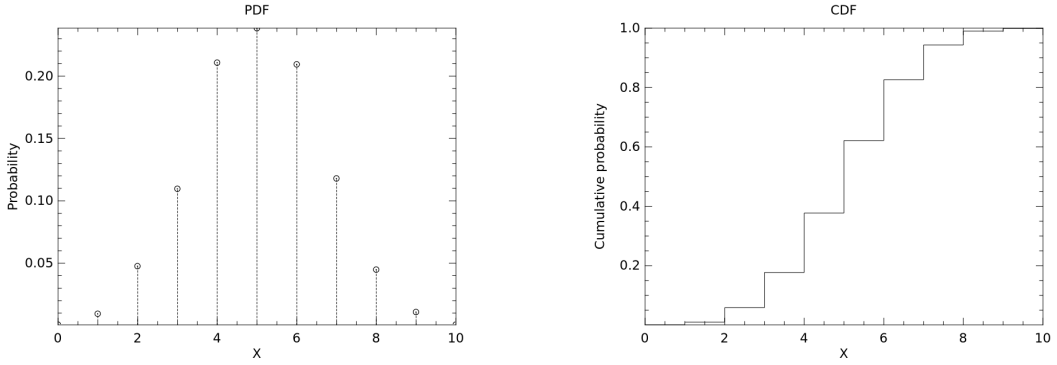
3.7 Visualisations

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 helper 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 pmf (probability mass function) 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, $P(X = x_i)$.

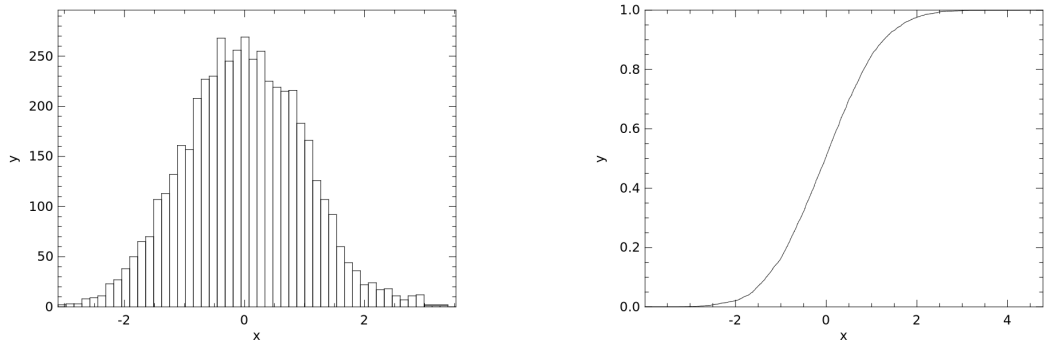
Where there is an ordering on the type, discrete distributions can also have their cdf visualised. The cdf of a discrete distribution is a step function. The ordering is given by a first class module representing the type. There are cases where there is no natural ordering - for example a distribution over an arbitrary ADT, so this also allows a user to custom define the ordering. Continuous distributions are also displayed as histograms, with a set of samples being put into



(a) PDF (b) CDF

Figure 3.2: Samples from a binomial distribution visualised

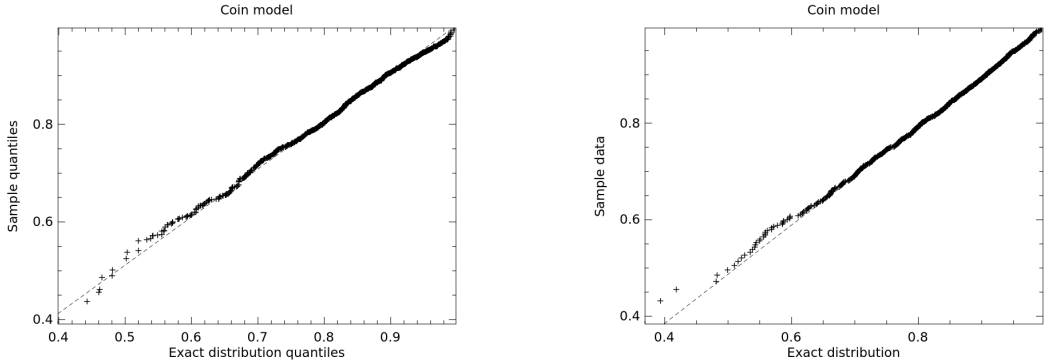
n equal width bins. The height of each bar is the the pdf (continuous analogue to the pmf), which is calculated by finding the number of samples in each bin, then dividing by the total number of samples. To display the cdf, we can display the empirical cdf directly as a scatter plot, or join points to 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. Q-Q plots plot the quantiles of two distributions against each other as a scatter plot - if all the points lie on the the line $y = x$, the distributions are identical. 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 $(-\infty, +\infty)$. Both these plots are often used to find the differences between some theoretical expected distribution and the distribution given by some 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 distributions (the coin model in section 4.1.2) - the points are close to the expected line, visualising a successful inference procedure.



(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 again compare two distributions. Figure 3.6 shows an exact beta distribution overlaid onto samples taken from a beta distribution.

```
Open Ppl.Plot

let coin_compare () =
  hist_dist_continuous ~fname coin_i
  |> add_pdf ~dist:coin_exact
  |> show
```

Listing 14: Code to produce plot

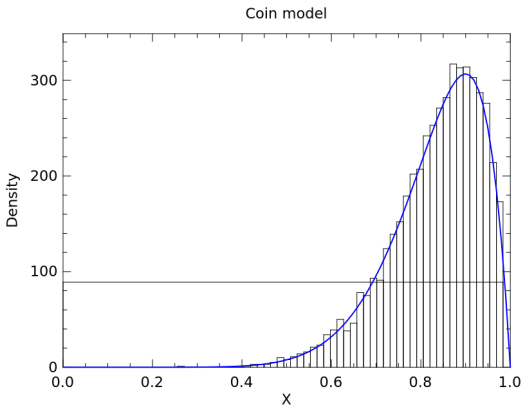


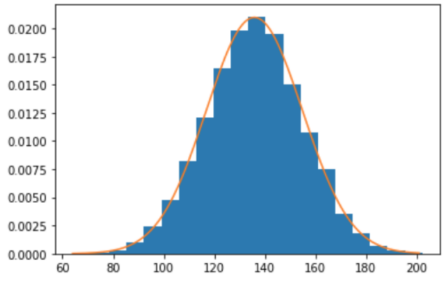
Figure 3.5: Output

Figure 3.6: The approximate and exact pdf of the output of inference for a biased coin model

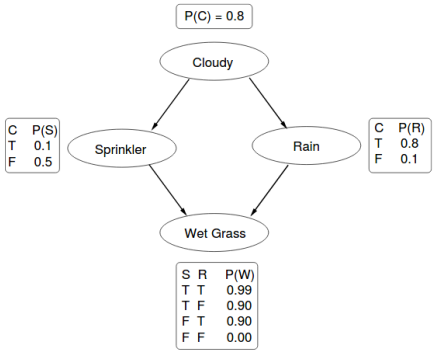
3.8 Testing

Automated testing of PPLs is difficult for reasons outlines in the preparation. However, I still wrote basic unit tests. The test framework I used, Alcotest, lets me check that outputs of functions match expected values. This allows me to test all deterministic helper functions, e.g. normalise. It also lets me test the exact inference method, and very simple distributions and models which only contain a single value.

The test output and code coverage output can be seen in figure 3.7.



(a) Test report



(b) Code coverage report

Figure 3.7: Output from running unit tests

Chapter 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 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 by using hypothesis tests which use drawn samples to determine whether two distributions are the same or not. For all these tests, I will use simple models with analytic solutions to compare to.

4.1 Examples

To show how my PPL would be used on real problems, I now outline a set of example problems. Several examples here will be simple, and have analytic solutions - this is so that I can then test the correctness of applying inference on them. 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 15 shows the model in the diagram encoded as a program. This particular program can be used (by applying an inference function) to find 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[26]. The model is of a coin that is tossed n times to give x heads. We do not know if the coin is biased or not, and would like to find out the bias, p of

[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 0.8 in
  let* rain      = bernoulli (if cloudy then 0.8 else 0.1) in
  let* sprinkler = bernoulli (if cloudy then 0.1 else 0.5) in
  let wet_grass = bernoulli
      (match rain,sprinkler with
         true,true -> 0.99
       | true,false -> 0.9
       | false,true -> 0.9
       | false,false -> 0.
      ) in
  condition wet_grass
    (return rain)

let () = print_exact_bool (exact_inference sprinkler_model)
(*
false: 0.137057
true: 0.862943
*)
```

Listing 15: Sprinkler model

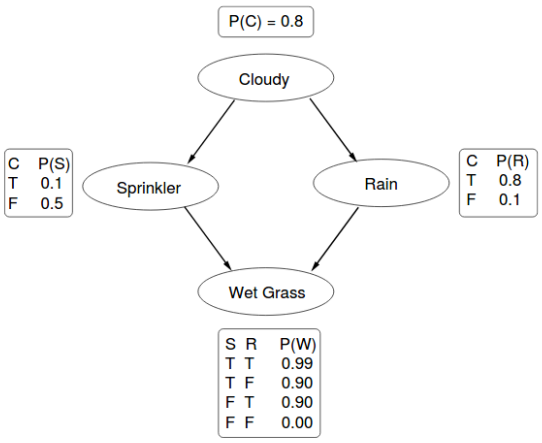


Figure 4.1: Bayesian Network example

the coin, where p is the probability of heads, with $p = 0.5$ being an unbiased coin. To find the posterior, we use an uninformative prior (Θ), the uniform. This results in the posterior, the beta distribution, specifically Beta$(x + 1, n - x + 1)$.

The program in my PPL is shown in listing 16, 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. This example uses $n = 10$ and $x = 9$, so the mean produced is roughly 0.83, the mean of Beta$(10, 2)$.

The comparison given in Figure 4.2 shows how the same model is defined in other languages.

```
let coin_mean 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 16: Coin model

```
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 17: WebPPL

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

Listing 18: Anglican

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

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 the emission distribution, which is the distribution over the observed states given the hidden state. The example in Listing 19 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.

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

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

Listing 19: Hidden Markov Model

function can be used to condition on many observations easily. The fmap function is used to map outputs from a distribution. Since the linreg model produces tuples of parameters, we can create individual distributions over either one.

```
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 20: Linear Regression

4.1.5 Mixture Model

This example demonstrates a model which cannot be expressed in some PPLs such as STAN or Infer.Net, since it is a non-parametric Bayesian model. This is a Dirichlet Process mixture model with an infinite number of Gaussians[18]. It is used for the common task of clustering a set of data points without knowledge of the number of clusters. This means the number of clusters is allowed to grow with the dataset size. We use a mixture of Gaussians, meaning the likelihood of a point belonging to each cluster is given by different normal distributions. The full code for this model, along with comparisons to other languages is given in appendix A.5.

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 statistic distributions (e.g. the χ^2 distribution) were calculated using `Owl`, and empirical distributions generated using the `EmpiricalDist` modules.

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 exact distribution
H_1 : The sample data do not follow the exact distribution

4.2.1 Chi-squared

The χ^2 test is a simple goodness-of-fit test which can test whether or not a given discrete distribution

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 the degrees of freedom being $k - 1$, where k is the number of possible values of the distribution.

Results

Table 4.1 shows the results of carrying out the test on several inference procedures for different discrete models. None of the values are below 0.05, so we cannot reject the null hypothesis, so we conclude that, at the 5% significance level, the distributions are not significantly different.

	rejection	importance	metropolis hastings	particle filter
sprinkler	1.	0.999527	1.	0.999863

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

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

Table 4.2 shows that for all the continuous models considered, the p-value obtained from all tests are greater than then 0.05. This means we do not reject H_0 for any model/inference procedure combination, so can be confident (at the 5% significance level) that the inference procedures are correct. This shows that the generated posterior is not significantly different from the real solution.

	rejection	importance	metropolis hastings	particle filter
coin	0.895793892046	0.243719262886	0.391381549312	0.544000635464

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

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

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

The continuous version is similar, with p and q now being density functions:

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

Since we cannot compute this integral exactly (we only have the exact density function for one of the distributions), I put the set of samples into discrete bins to approximate $q(x)$. I then used Monte Carlo integration to compute the integral. 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 shows the results of this. For each inference procedure, we can see that the KL-divergence for each model decreases as we take more samples. In the case of the sprinkler model, there is a rise near the end, but this can be attributed to noise since the KL-divergence is actually lowest for that model, implying that inference performed best for it.

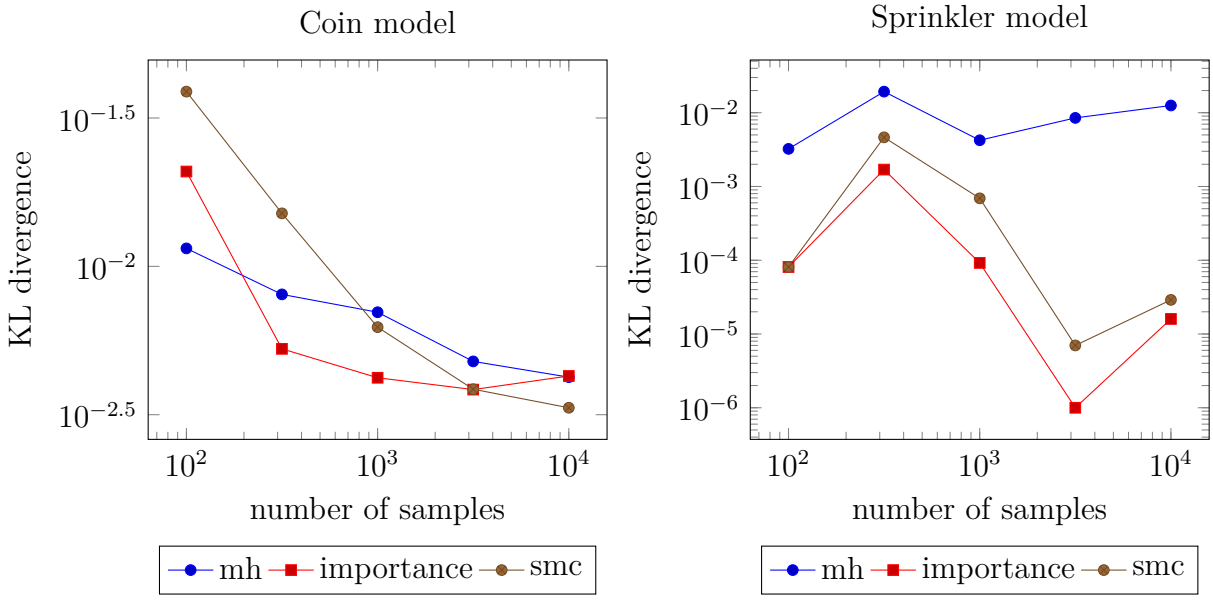


Figure 4.3: Plot of KL-divergence with increasing number of samples for different models and inference procedures

4.4 Performance

I evaluated the performance of my ppl against Anglican, WebPPL, and Pyro. All of these languages are universal PPLs embedded in different host languages, so are comparable to my PPL.

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 and peak memory usage.

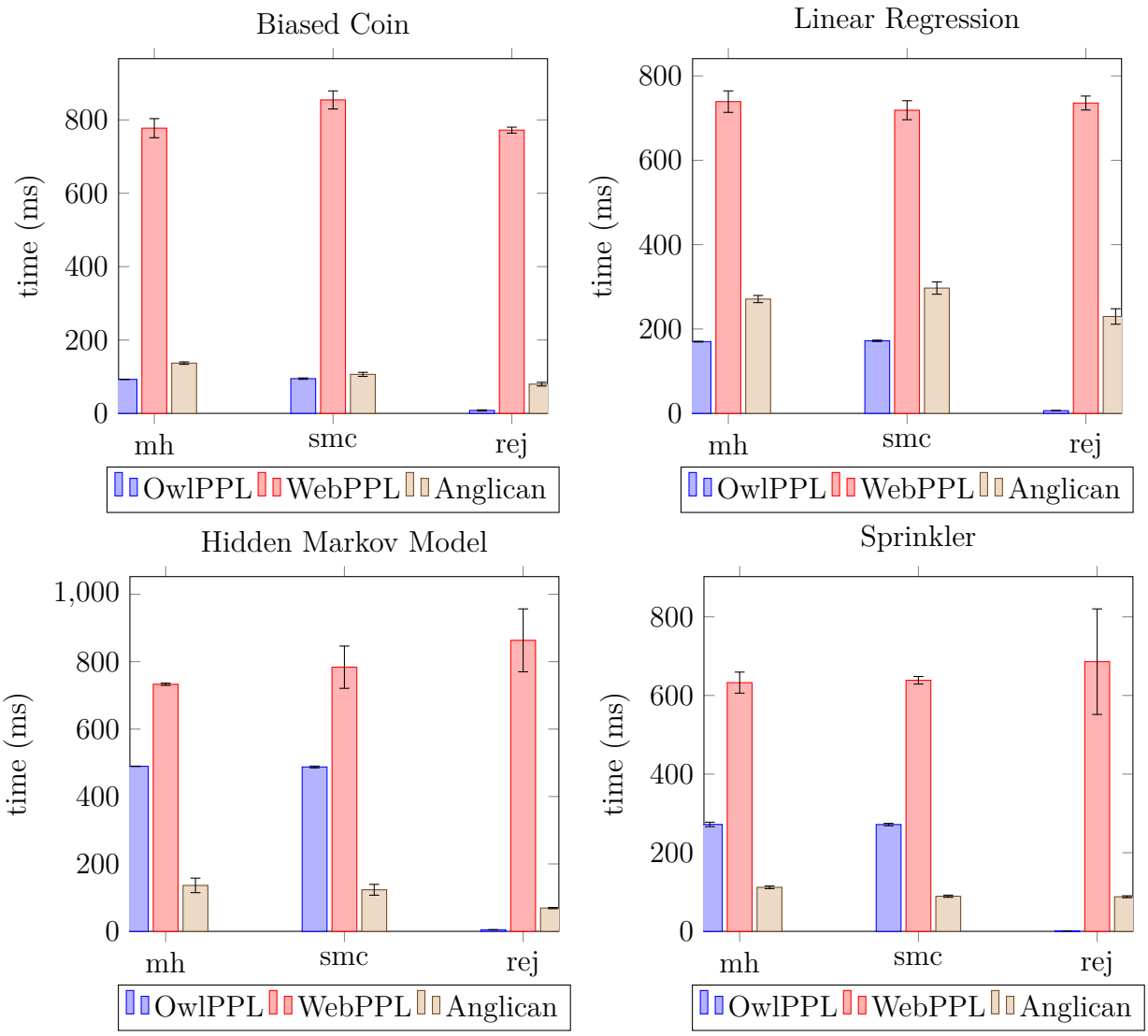


Figure 4.4: Performance of my ppl against other languages for different models and inference algorithms, taking 10,000 samples from the posterior, averaged over 20 runs. Error bars show the 95% confidence interval

These graphs show that my PPL performs reasonably well compared to webPPL in both memory and time. It is less efficient for some models, but not excessively so. In particular, my implementation of a particle filter performs relatively poorly. This is possibly because when using a large number of particles, a large number of small memory allocations are made by OCaml, which introduced overhead both to my program and the garbage collector. The languages I compare with are both also garbage collected, but may be more optimised for this use case. Anglican is a Clojure library, which runs on the JVM, whereas WebPPL is run using nodejs, a JavaScript interpreter. It is possible there is an interpretive overhead with webPPL, explaining slower running times - however based on my results, it is not significant.

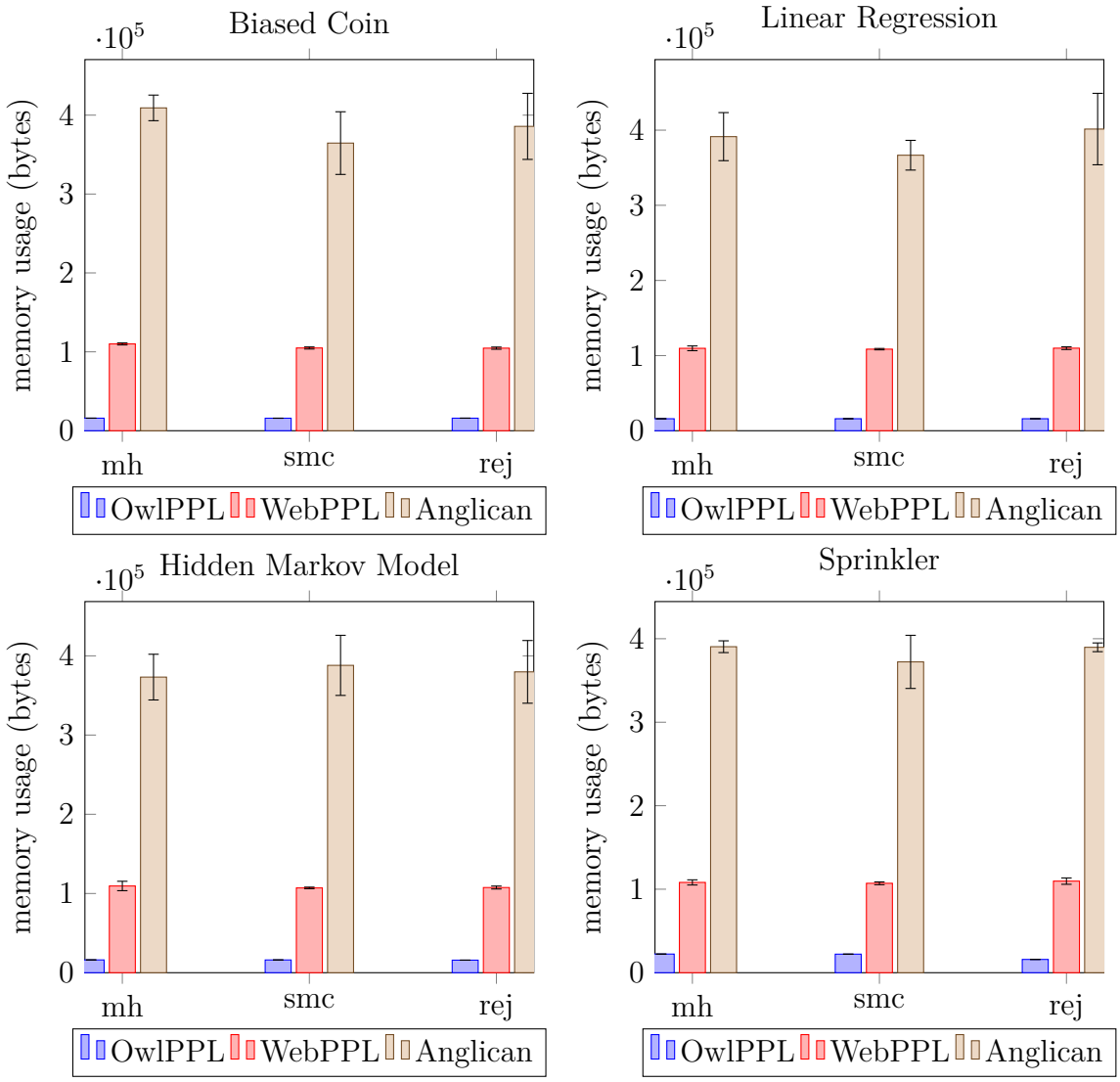


Figure 4.5: Memory Usage of my ppl, compared against other languages for different models, all using an MCMC algorithm, taking 10,000 samples from the posterior, averaged over 20 runs. Error bars show the 95% confidence interval

Chapter 5

Conclusion

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. The performance is reasonable when compared to similar systems (other universal PPLs). Code written in my PPL is also not overly verbose compared to these languages.

Future work would mainly focus on how to improve inference. 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.

Other, more efficient inference algorithms may also be able to be written. 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 to create more specific proposal distributions for MCMC.

One of my initial goals was for my PPL to be able to represent as many distributions (models) as possible. This is why I used a trace based approach inspired by Church rather than using a computation graph. However, there are 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. I was not able to complete one extension due to the fact that I had not chosen this approach - the ability to visualise the model itself (e.g. as a network). If I used a graph approach, this would likely be possible.

My visualisations, while basic, allow producing graphs of posterior distributions and exact distributions. Extensions include visualising higher-dimensional data.

While there are many possible extensions to this project, it successfully achieved all the initial goals.

Bibliography

[1] 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.

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

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

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

[5] 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.

[6] 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.

[7] 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.

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

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

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

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

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

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

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

[15] 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.

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

[17] 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.

[18] 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.

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

[20] 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.

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

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

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

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

[25] David Wingate, Andreas Stuhlmueller, and Noah Goodman. Lightweight implementations of probabilistic programming languages via transformational compilation. In Geoffrey Gordon, David Dunson, and Miroslav Dudík, editors, *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics*, volume 15 of *Proceedings of Machine Learning Research*, pages 770–778, Fort Lauderdale, FL, USA, 11–13 Apr 2011. PMLR.

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

[27] 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.

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

Appendix A

Example Programs

A.1 Sprinkler

A.2 Coin

The likelihood model (X) is a binomial.

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

We then use Bayes' rule to calculate the posterior.

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

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

This is the beta distribution - Beta()

A.3 Linear Regression

A.4 Hidden Markov Model

A.5 Dirichlet Process

Appendix B

Full Documentation

Appendix C

Project Proposal

Computer Science Tripos – Part II – Project Proposal

A probabilistic programming language in OCaml

A. Roy, Christ's College

April 13, 2020

Project Originator: Dr. R. Mortier

Project Supervisor: Dr R. Mortier

Director of Studies: Dr R. Mortier

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 [16], as well as PPLs for other languages, such as WebPPL - JS[12], Church - LISP[5] or Infer.Net - F#[24] 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 specfic 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[12] 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[13] or monads[19], 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.