Anik Roy

A probabilistic programming language in Ocaml

Computer Science Tripos - Part II

Christ's College

March 3, 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: March 3, 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:	4925 [1]
Final Line Count:	1252 [2]
Project Originator:	Dr R. Mortier
Supervisor:	Dr R. Mortier

Original Aims of the Project

Work Completed

Special Difficulties

None

[1] This word count was computed by `texcount -inc -sum -1 diss.tex`

[2] This line count was computed by `cloc (git ls-files)` and excludes blank lines and comments

Contents

List of Figures

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. 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, ($p(x)$) and a generative model ($p(y|x)$) which specifies the likelihood of 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 'generative models' as programs. Writing such a program requires us to be able to sample from distributions. 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 operator, 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 some other 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. Embedding a DSL into OCaml allows us to represent a wide range of models using

OCaml's inbuilt functions and operators, and lets us 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 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 [4]. Others, such as STAN or infer.NET (F#) can only operate on finite models (factor graphs), and do not allow unbounded recursion when defining models. Some, such as Church or Pyro, 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.

2.1 Starting Point

There do exist PPLs for OCaml, such as IBAL [4], as well as PPLs for other languages, such as WebPPL - JS[5], Church - LISP[3] or Infer.Net - F#[10] 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 do 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 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. I also benchmarked the performance with regards to scalability, i.e. made sure the performance is still reasonable as models are conditioned on more data.

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 guide (Jane Street style[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.

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

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 [], 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 fail they show 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 are approximate, so tests suffer from the problems described above.

Despite this, I will still be able to carry out a full evaluation, performing hypothesis tests such as the kolmogorov-smirnov or chi-squared tests.

2.3.2 Licenses

The major libraries I use, Owl and Jane Street Core, are both licensed under the MIT 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
- Spacetime - Memory profiler for OCaml
- Landmarks - Profiler for OCaml
- 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.

2.4.1 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 the code on the master branch always works. The CI can also be used to ensure the library works with older versions of OCaml, and is backwards-compatible.

2.5 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, 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*, 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.

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` []. The sample function is for sampling from some distribution - either a primitive distribution or a user defined distribution.

2.6 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.

2.7 Owl

Owl is a scientific computing library written for OCaml [9]. Importantly for my PPL, it contains functions for working with multi-dimensional arrays, as well as a wide variety of statistical functions. In particular, it contains functionality relating to many common distributions, efs.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 these 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 let me visualise output distributions, as well as visualising performance statistics, all directly from OCaml.

2.8 Probability Monad

2.8.1 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. This allows me to compose distributions easily, by using the output from one distribution in another using the bind operator. In OCaml, I can also use the new (let*) operators to make this more ergonomic, and make code look more natural (as opposed to using the `>>=` operator everywhere).

2.9 Approaches to probabilistic programming

Existing PPLs take several different forms, both as standalone and embedded languages. The main tradeoff 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 [], which uses tensorflow. 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. However inference is often slower, particularly when dealing with data in high dimensions, since the algorithms need to be more general purpose, and often converge slower.

2.10 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 (which is a program in the PPL setting).

This formula hold 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. For example, consider a set of discrete random variables X_i, with p the joint probability distribution of all X_i. The marginal distribution of any individual X_i is:

$$P(X_i = x_i) = \Sigma$$

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 methods such as *Variable Elimination* or *Message Passing*.

Alternatively, in the continuous case the formula

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

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.

2.11 Inference Algorithms

Inference algorithms are ways to systematically generate samples from posterior distributions given a generative model and a prior. 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:

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

Here, p is the posterior distribution of a particular trace x, given the observed variables y The aim then is to find the posterior over the latent variables we are interested in, which we can specify within the program, either as part of the model, or outside it in a query to the model.

Show how this relates to a program here or in implementation?

2.11.1 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.

2.11.2 Rejection Sampling

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

One such method, rejection sampling, is a very simple inference method which takes uses another distribution which can be sampled from.

2.11.3 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.

2.11.4 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. This 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.

2.11.5 Sequential Monte Carlo (SMC)

SMC methods are algorithms which are based on using large numbers of weighted samples ('particles') to represent a posterior distribution. These particles are updated and resampled from in order to converge the set of particles to the posterior.

Chapter 3

Implementation

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. The build system dune makes

I have implemented my ppl as a library (in the `lib` subdirectory), contained within the module `Ppl`. This contains several submodules, most importantly the `Dist` module, which contains the code for representing, creating and combining distributions. The Primitives module contains an type for representing primitive distributions. This type is not abstract, in order to allow users to create their own distributions outside of the ones provided in the module.

The Plot module contains helper functions which wrap around Owl$_p lplot, allowing users to easily create visualisations from distributions defined in my ppl$.

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, for example the `hmm` folder for an example of a hidden markov model.

All code is written in OCaml 4.08, with the main dependencies being Jane Street's `Core` and `Owl` [9].

3.2 Representing Distributions

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.

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 type signature of bind is `'a m -> ('a -> 'b m) -> 'b m`, and return is `'a -> 'a m`, with m being the monad type.

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 3.1. 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 [2], 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.

```
1  open Core
2
3  type 'a dist = ('a * float) list
4
5  let unduplicate = ...
6
7  let bind d f =
8    let mult_snd p = List.map ~f:(fun (a,x) -> (a,x*.p)) in
9    List.concat_map ~f:(fun (x,p) -> mult_snd (f x) p) d
10
11 let return x = [(x,1.)]
```

Listing 3.1: Simple Probability Monad

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 polymorphic map, which is provided in the Jane Street Core library, and

implemented as a balanced tree, significantly improving the time complexity of combining distributions.

```
open Core

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

let bind d f =


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

Listing 3.2: 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 GADT

The structure that I landed on to represent general models is a generalised algebraic data type. GADTs have been used to represent probabilistic models [8] and are widely used to implement interpreters in functional languages. 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 type arguments, making sure that programs are well-typed and rejecting invalid programs. The GADT represents a model, 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 functions which transform the conditional distributions to distributions without any conditional statements, allowing sampling to be performed as normal. Primitive distributions also have a special variant (which takes a different `primitive` type), since we can find exact the exact pdf/cdf of these distributions, unlike the `dist` type, which can only be sampled from. The implementation can be seen in listing 3.3. The monad functions are also provided, which just construct the corresponding variant in the GADT.

```
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 3.3: Representing a probabilistic model using a GADT

3.4 Conditioning

The condition variant is used to assign scores to traces, and takes a function which takes an element and returns a float, a 'score', which represents how likely that element is. 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 0, and if false, the score assigned is -infinity.

For soft conditioning, for example an observation that we know comes from a certain distribution, there is an `observe` function. This function is essential for continuous distributions, since the probability of observing any one value is 0, making hard conditioning redundant.

The `score` function is similar to the condition operator, except instead of 0, it assigns a particular score (any float) to the trace. An example is

3.5 Primitive Distributions

In PPLs, users build complex models by composing more simple elementary primitive distributions (ERPs) [11]. These primitive distributions need to have a few operations defined on them, namely `sample, pdf, cdf` and `support`.

3.6 Forward Sampling

In order to perform inference, we will often need to find a prior distribution. For a posterior, $P(\theta \mid x)$, the model written by the user is $P(x \mid \theta)$, and the prior is $P(\theta)$, so finding the prior is the same as disregarding the conditionals (essentially ignoring the data). Since sampling is only difficult in the presence of conditionals, this allows us to sample from the prior using essentially the same sample function. 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).

3.7 Inference

Inference is the key motivation behind probabilistic programming.

Inference can be thought of as a program transformation [8] [12]. 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 3.7.7.

3.7.1 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`, which needs to have duplicates removed and then be normalised.

```
let rec enumerate: type a.a dist -> score -> ((a * score) list)
  = fun d multiplier ->
    if Float.(multiplier = 0.) then []
    else
      match d 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 ch = enumerate d multiplier in
        List.map ch ~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 3.4: Enumerating all paths through a model

This method is very naive, and therefore inefficient. Since we essentially take every possible execution trace, we do not exploit structure such as overlapping traces. This can be made slightly more efficient by using algorithms such as belief propagation [7], but they still only work on models made up from discrete distributions. Exact inference of this kind only works on models known as bayesian networks, and exact inference for bayesian networks is in fact NP-hard[1]. So, instead, most of my project focuses on approximate inference.

3.7.2 Rejection Sampling

3.7.3 Importance Sampling

3.7.4 Metropolis Hastings

3.7.5 Particle Filter

3.7.6 Particle Cascade

3.7.7 Particle-Independent Metropolis-Hastings

3.8 Examples

3.8.1 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 3.1

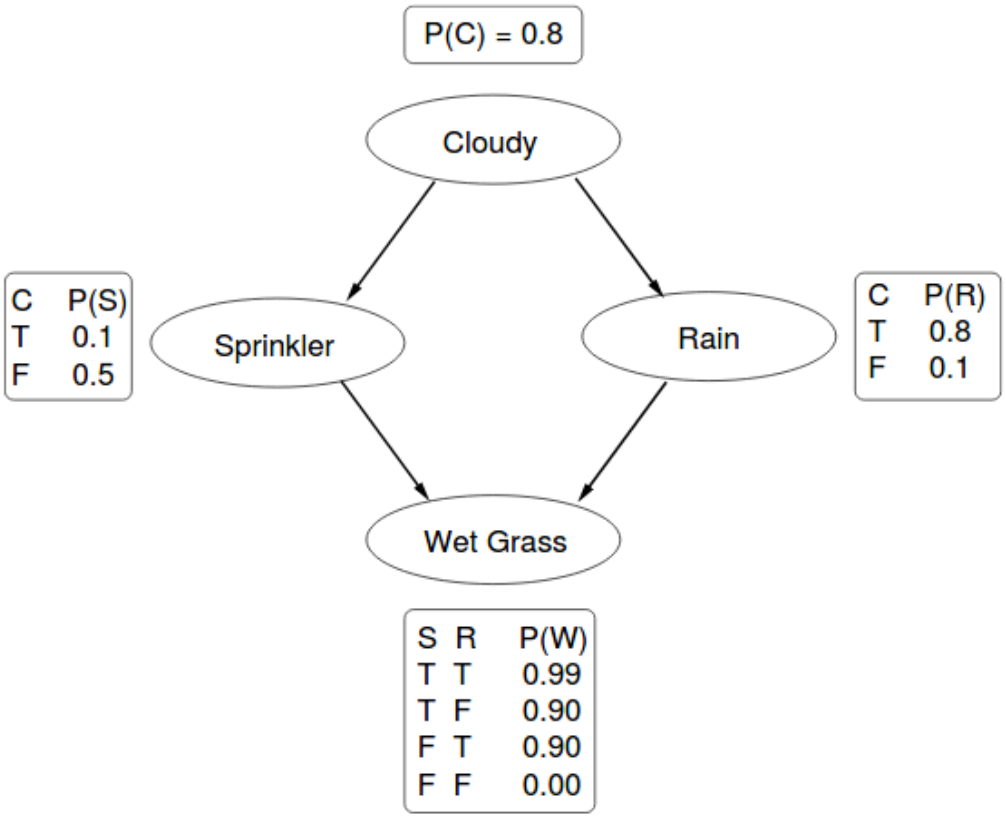

Figure 3.1: Bayesian Network example

3.8.2 Biased Coin

3.8.3 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 emmission distribution, which is the

3.8.4 Linear Regression

3.8.5 Dirichlet process

3.9 Statistical tests

I had to implement a number of statistical tests in order to test the correctness of the output distribution. These tests are known as goodness-of-fit tests, and

3.10 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.

Chapter 4

Evaluation

4.1 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.

For all tests described below, I set the 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

Setting the significance level, $\alpha = 0.01$ for all tests

4.1.1 Kolmogorov-smirnov

4.1.2

Chapter 5

Conclusion

Bibliography

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

[2] Martin Erwig and Steve Kollmansberger. Functional pearls: Probabilistic functional programming in haskell. *J. Funct. Program.*, 16(1):2134, January 2006.

[3] 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.

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

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

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

[7] Judea Pearl. Reverend bayes on inference engines: A distributed hierarchical approach. In *Proceedings of the Second AAAI Conference on Artificial Intelligence*, AAAI82, page 133136. AAAI Press, 1982.

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

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

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

[11] David Wingate, Andreas Stuhlmueller, and Noah Goodman. Lightweight implementations of probabilistic programming languages via transformational compilation. In Geoffrey Gordon, David Dunson, and Miroslav Dudk, 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.

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

Appendix A

Project Proposal

Computer Science Tripos – Part II – Project Proposal

A probabilistic programming language in OCaml

A. Roy, Christ's College

March 3, 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 [4], as well as PPLs for other languages, such as WebPPL - JS[5], Church - LISP[3] or Infer.Net - F#[10] 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[5] 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[6] or monads[8], 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.