Anik Roy

A probabilistic programming language in Ocaml

Diploma in Computer Science

Christ's College

February 10, 2020

Proforma

Name:	**Anik Roy**
College:	**Christ's College**
Project Title:	**A probabilistic programming language in Ocaml**
Examination:	**Computer Science Tripos Part II**
Word Count:	–[1] **(well less than the 12000 limit)**
Final Line Count:	–[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 `detex diss.tex | tr -cd '0-9A-Za-z \n' | wc -w`

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

Declaration

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.

Signed [signature]

Date [date]

Contents

1 Introduction **1**
1.1 Motivation 1
1.2 Related works 2

2 Preparation **3**
2.1 Motivation 3
2.2 Related works 4

3 Implementation **5**

4 Evaluation **7**

5 Conclusion **9**

Bibliography **11**

A Project Proposal **13**

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.

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 it's type system or efficient native code generation.

1.2 Related works

There are many examples of PPLs, even in OCaml. Some PPLs choose to limit the models that can be expressed in them in order to make inference more efficient, e.g. HANSEI in OCaml can only represent discrete distributions [2]. Others, 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

2.1 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.

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 it's type system or efficient native code generation.

2.2 Related works

There are many examples of PPLs, even in OCaml. Some PPLs choose to limit the models that can be expressed in them in order to make inference more efficient, e.g. HANSEI in OCaml can only represent discrete distributions [2]. Others, 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 3

Implementation

Chapter 4

Evaluation

Chapter 5

Conclusion

Bibliography

[1] 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.

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

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

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

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

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

Appendix A

Project Proposal

Computer Science Tripos – Part II – Project Proposal

A probabilistic programming language in OCaml

A. Roy, Christ's College

February 10, 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 [2], as well as PPLs for other languages, such as WebPPL - JS[3], Church - LISP[1] or Infer.Net - F#[6] 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[3] 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[4] or monads[5], 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 VS-Code 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.