Automatic code rewriting in probabilistic programming

Internship report by
Diane Gallois-Wong

supervised by
Hongseok Yang

at the
Department of Computer Science
of the University of Oxford

First year of Master’s Degree
at ENS Paris

March - July 2015
Introduction

Probabilistic programming languages provide a powerful and flexible support for probabilistic modelling and reasoning. They allow to describe sophisticated probabilistic models easily and intuitively, and to answer various queries about these models using built-in inference algorithms. Model description takes the form of code, which means it inherits a lot of conveniences from classic programming languages such as expressiveness, flexibility, use of libraries, verification techniques. Also, the user does not have to implement an inference algorithm for his or her model, which is usually highly nontrivial and requires a lot of efforts and expertise in machine learning. Instead, she or he can reason about the model using built-in generic inference algorithms of these languages. Most probabilistic programming languages support Bayesian statistics where one is interested in computing so called posterior probability distributions.

However the current probabilistic programming languages still need a lot of improvements before they are as universal as they are meant to be, and able to compete with common alternatives where the user develops both a model and a model-specific inference algorithm. There are example programs on which the simulation-based inference algorithms of these languages do not converge on such programs, and they generate samples that are not properly distributed according to the intended distribution.

A most interesting example is Latent Dirichlet Allocation (LDA), a widely used generative topic model. Topic models attempt to classify a collection of documents based only on the words they contain, by determining the main themes that run through them, and identifying which of these themes appear in each document. The basic LDA model is straightforward to express in a probabilistic programming language. However, usual inference algorithms work poorly on this short LDA code: the results are too much influenced by sheer randomness, and not enough by the documents given as input.

There is a well-known alternative model of LDA, said to be collapsed, which is much more friendly to inference algorithms. The code of this collapsed LDA in a probabilistic programming language will allow for rather good performance, but it is quite long and cumbersome to write, which goes against the purpose of probabilistic programming languages. We would like to get the performance of the collapsed model even when the user writes code for the basic LDA model. The way the inference algorithms work makes them unlikely to be adapted directly to perform better on this basic model. A better solution is automatic source code transformation.

During my internship I worked on such a transformation. More precisely, I used tools from programming languages and developed a method for automatically transforming and optimizing probabilistic programs, which includes the checking of necessary conditions for such transformation. The method works on some naive implementations of LDA, as well as other programs with similar properties. It is restricted to code using a few selected directives, which are enough to implement LDA. It produces code on which the inference algorithms have significantly better performance: their results are closer to what they should be ideally. I describe the transformation formally using a small lambda calculus. The code issued should be equivalent to the initial code. I would have liked to show that they are semantically equal, but defining a semantics for probabilistic programming languages is really difficult, one of the current challenges in the field. I discuss restrictions on programs which may be transformed: some are unavoidable, but others I have set to ease code rewriting or reasoning about code equivalence. The transformation is based on a property of Dirichlet and Categorical distributions, called conjugate prior. It could be adapted to handle other distributions with similar properties. Other potential improvements include relaxing...
the constraints that I impose on programs to be transformed, and more sophisticated memory management in transformed code which could lead to better performance but would require a more complex analysis of the initial code.

All my work is done with Anglican [11] in mind. Anglican is a particular probabilistic programming language built on top of the functional language Clojure, and is developed by researchers in Oxford where I did my internship. However, my results may be adapted to some other similar languages, such as Venture [5] and Church [3], with minor changes. In Clojure I implemented my method for transforming source code written in a small sub-language of Anglican\(^1\). The method was tested on LDA examples, whose results highlight the quality of the output of the transformed code compared with the initial one.

Contents

1 Overview 3

2 Preliminaries 4
   2.1 Bayesian statistics ................................. 4
   2.2 The probabilistic programming language Anglican .................. 5
   2.3 Inference algorithms ................................ 7
   2.4 Latent Dirichlet Allocation (LDA) .......................... 8

3 Collapsing LDA: interest and justification 10
   3.1 Highly expressive latent variables: a challenge for existing inference algorithms . . 10
   3.2 Dirichlet Process: the core of collapsed LDA ..................... 11
   3.3 Conjugate prior: justification of collapsing ..................... 11

4 Automatic code transformation: technical decisions 12
   4.1 Necessary conditions on variables to be marginalised .............. 13
   4.2 Simulating side effects with a state and modular functions ........ 14

5 Automatic code transformation: formalism 14
   5.1 Lambda calculus ........................................ 14
   5.2 \(\alpha\)-normal form .................................... 15
   5.3 Identifying potential variables to marginalise ................... 15
   5.4 Final decision on variables to marginalise: type and effect system .................... 16
   5.5 Code rewriting ........................................ 18

6 Potential improvements to the automatic transformation 19

7 Experimental performance of naive and collapsed implementations 20

Conclusion 22

\(^1\)My implementation can be found at https://bitbucket.org/dianegalloiswong/anglican-collapse-lda
1 Overview

In the probabilistic programming language Anglican, model description is based on objects representing mathematical distributions (2.2). A random value can be drawn according to a distribution using the primitive `sample`. `observe` enforces the assumption that a given value has been drawn from a given distribution. Built-in inference algorithms take all of them into account to generate relevant samples (2.3).

In the most simple implementation of the topic model LDA (2.4), a pattern appears several times: a value sampled from a Dirichlet distribution is used to build a Discrete distribution on which a lot of `sample` or `observe` are called. This is what makes the inference algorithms work poorly on this naive implementation of LDA (3.1). The variable containing the value sampled from a Dirichlet distribution is called a *highly expressive latent variable*. An alternative way to describe LDA uses Dirichlet Processes (3.2): each highly expressive latent variable disappears, but instead, some information representing the Discrete distribution which used to be built on it has to be updated every time `sample` or `observe` is called on it. This is the collapsed version of LDA, and the inference algorithms give far better results on corresponding code. The highly expressive latent variables which disappear in the collapsed version are said to be *marginalised*.

Below are examples of naive LDA code and corresponding collapsed code, using some Anglican primitives but not its syntax.

```
θ := (sample (Dirichlet α))

v := copy of α

d := (Discrete θ)

...  
x₀ := (sample (Discrete v))

...  

x₀ := (sample d)

v := (update v x₀)

...

(observe d x₁)

(observe (Discrete v) x₁)

v := (update v x₁)
```

The collapsed code is more complicated to write, especially when there are a lot of `sample` or `observe` in functions and other complex structures, updates of `v` need to be passed around carefully.

During my internship, I have been working on an automatic code transformation from naive code into collapsed code. This allows the user to write naive code, which is easier, and get the performance of collapsed code.

I describe the transformation formally using a small lambda calculus (5.1), with just enough directives to implement the naive version of LDA.

First of all, I put it under $a$-normal form (5.2), which means that I introduce a lot of intermediate variables so that each assignment contains at most one evaluation: for example something like

```
θ := (sample (Dirichlet α))
```

would become

```
x_{intermediate} := (Dirichlet α); θ := (sample x_{intermediate}).
```

The next step consists of identifying every highly expressive latent variable which can be marginalised. I impose strong necessary conditions on them, which will make the code rewriting easier, since there will be less different cases to handle (4.1). For example, I don’t want marginalised variables to be used otherwise than as argument to a Discrete distribution, as this case is directly handled by a Dirichlet Process but the others are not. I check the conditions in two steps: I first identify potentially marginalisable variables, which fulfil some of the conditions, during a quick and easy program analysis (5.3), then I use a type and effect system (5.4) to gather more information on these and determine whether they fulfil all the conditions.
The final step consists of the actual rewriting of the code (5.5). As we said before, the Dirichlet Process uses information which have to be updated every time they are used, which means they need to be passed around carefully. The easiest would be to use side effects, but they are not very efficient in Anglican. Instead, we simulate side effects using a state, which consists of a hashtable containing the information for all Dirichlet Processes (4.2). If we ensure that the state is passed everywhere in the program, then keys to the state are similar to references. We also define modular functions, considered as primitives in the transformed program, which are able to handle both usual values and keys to the state (when the initial argument has been marginalised). This modularity lightens considerably the code rewriting step, as it allows for a more systematic transformation.

2 Preliminaries

2.1 Bayesian statistics

Bayesian statistics describe states of belief by assigning probabilities to diverse hypothesis. Most importantly, they can be updated after observation of new evidence.

For example, let us assume that I am given a coin which is biased to have probabilities 2/3 and 1/3 to give a result or the other, but I don’t know whether the 2/3 is for heads or tails, I cannot tell just looking at it. I have no reason to believe one or the other more, so I think both to be equally likely. I toss the coin and get heads. I update my belief using this observation: I think it more likely that the coin is biased to give heads more often, but not extremely more likely that the other way. I toss it two more times and get two more heads: now I think it very likely that heads is favoured.

Update of belief following observation is formalised by Bayes’ theorem.

**Bayes’ Theorem.** Let $H$ and $X$ be two events.

$$ p(H|X) = \frac{p(X|H) p(H)}{p(X)} $$

This is really easy to prove by considering two writings of $p(X \cap H)$. Yet this most basic equality is the basement of important probabilistic reasoning.

$H$ is the hypothesis that we want to study. The initial belief is represented by $p(H)$ called the prior probability. $X$ is an observation. The updated belief is given by $p(H|X)$ called the posterior probability. $p(X|H)$, called the likelihood function, describes how well the hypothesis can explain the observation. Bayes’ theorem is often summarised as

**Posterior probability $\propto$ Likelihood $\times$ Prior probability**

Indeed, we often want to compare the relevance of different hypothesis, in which case we do not need to compute $p(X)$ which is the same for all of them; considerations of proportionality are sufficient.

Let us formalise the example above. We note $H_h$ the event that the coin has probability 2/3 to give heads, $H_t$ that 2/3 is the probability to give tails. Initial belief: $p(H_h) = p(H_t) = 1/2$. We note $X_h$ the event that we toss the coin and get heads. Update on this observation:
\[ p(H_h|X_h) \propto 2/3 \times 1/2, \quad p(H_t|X_h) \propto 1/3 \times 1/2. \] Moreover these the two latter probabilities sum to 1, so \( p(H_h|X_h) = 2/3, \quad p(H_t|X_h) = 1/3. \) As expected, after observing \( X_h, \) \( H_h \) is more likely than \( H_t \) but the difference is not extreme. Note that we didn’t need to compute \( p(X_h) \). Now consider the observation \( X_{3h} \) of tossing the coin 3 times and getting only heads. \( p(H_h|X_{3h}) \propto (2/3)^3 \times 1/2, \quad p(H_t|X_{3h}) \propto (1/3)^3 \times 1/2 \) so \( p(H_h|X_{3h}) = 8/9, \quad p(H_t|X_{3h}) = 1/9. \) As claimed, \( H_h \) is far more likely.

When reasoning about the relevance of a hypothesis given an observation, called here the posterior probability, it is quite natural to consider the likelihood: hypothesis which give a high probability to the observed event should of course be given an edge. What people often forget is weighting it by the prior probability. An event which is a priori extremely unlikely but explains the observation very well is still less likely than one which has a high prior probability and a medium likelihood.

For example, I am given a bag with 10 balls which are either black and white. I know it has been constructed by tossing 10 times a balanced coin (probability \( 1/2 \) to get either result), with a colour associated to each result, and selecting the corresponding balls. I draw 5 times a ball from the bag (putting them immediately back in), and I get only black balls. The hypothesis that the bag contains 10 black balls has the maximum possible likelihood of 1, but due to the construction process of the bag, its prior probability is only \( 1/2^{10} \). The hypothesis that the bag contains 5 balls of each colour has the mediocre likelihood of \( 1/2^5 \), and a prior of \( \binom{5}{10}/2^{10} \), so it is proportional to \( 63/8 \times 1/2^{10} \), so it is almost 8 times more likely than the previous hypothesis.

A good example of application of Bayes’ theorem (on which many people would intuitively tend to forget the prior probability) is given in [14].

2.2 The probabilistic programming language Anglican


Clojure is a general purpose functional, dynamically-typed programming language that compiles just-in-time to the Java Virtual Machine (JVM). It is a dialect of Lisp, inheriting prefix notation and parenthesis surrounding the function called as well as the arguments. It offers very interesting meta-programming support, through built-in manipulation and evaluation of abstract syntax.

A recurrent Clojure form which may need an explanation is the \texttt{let}: it takes a vector of bindings, in which each variable to bind is followed by the value it should take, and any number of expressions, which are all evaluated using the defined bindings, and the value of the last is returned. The value to associate to a variable may use bindings already defined even in the same vector.

Anglican is integrated inside Clojure. This means that inside Clojure code, some Anglican code can be introduced using a special form. In Anglican code, the user has access to new primitives allowing him to describe probability models.

The key elements of Anglican are \textit{distributions}, representing what their name suggests. They are objects destined to be called by two primitives: \texttt{(sample \textit{d})} returns a value randomly chosen according to the distribution \textit{d}, and \texttt{(observe \textit{d} \textit{x})} enforces the assumption that the value \textit{x} has been sampled from \textit{d}. Distributions can be defined by writing methods for \texttt{sample} and \texttt{observe}.
(let [more-heads (sample (flip 0.5))]
  coin (flip (if more-heads (/ 2 3) (/ 1 3)))
  (observe coin true)
  (observe coin true)
  (observe coin true)
  (predict more-heads))

Colours: Clojure special form, Clojure primitive, Anglican primitive.

Figure 1: Example of Anglican code: implementation of an example of 2.1

However, Anglican primitives already handle most of the common ones. For example (flip p) returns a boolean Bernoulli distribution: (sample (flip p)) gives true with probability p, false with probability 1-p.

(observe d x), where d is a distribution and x a value which may be sampled from d, returns the log-probability to get x when sampling from d. More importantly, it conditions the inference with the observation that distribution d has given the value x. The next example will show the influence this has on the inference. This allows to study posterior probabilities in Bayesian models.

predict is the output primitive. It is used to indicate which variables should appear in the output. If an Anglican code contains no predict, it can be run normally but there is no way to see any result.

When an Anglican code is run, it returns a Clojure data structure containing values for every variable appearing in a predict. These values come from samples of coherent executions of the program, where the outcome of every random choice has been selected. Built-in inference algorithms select such executions, so that they are well distributed, as explained below.

We call trace of execution all the information about a given possible execution of code with all non-deterministic choices, which in Anglican are the uses of sample. So in our case, a trace of execution is characterised by the list of the values taken by the random variables, which may affect the control flow, what variables are defined, and so on. It is not affected by the calls to observe. To each trace of execution, we can associate a joint probability, which depends on how likely the results of the sample were to be obtained, and on the calls to observe. This defines a joint distribution over the traces. Ideally, the variables in predict issued come from traces of execution selected according to this joint distribution. (In fact, no known algorithm does this both precisely and quickly on any kind of Anglican program.) For example, if a query contains (let [x (sample (flip 0.5))] (predict x)) executed in any case and x is not conditioned by any observe, about half of the issued traces of execution will associate the value true to x, the other half false. This is not the case anymore if x gets involved, even indirectly, in an observe, as in the following example.

Example

Let us recall the first example given about Bayesian probabilities, and write an Anglican program representing it (figure 1).

I have a biased coin which gives a result or the other with probabilities 2/3 and 1/3 respectively, but I don’t know whether heads or tails has the 2/3 probability. Initially I believe both situations to
be equally probable. Then I toss the coin 3 times and always get heads. Based on the observation I update my belief: now I think the probability that the coin is biased toward giving more heads to be 8/9 (cf 2.1).

To describe this model in Anglican, I define a random variable `more-heads`, which should be `true` if the coin has probability 2/3 to give heads, `false` otherwise, by sampling from a `flip` distribution with argument 1/2, which gives both booleans with equal probability. I can now define my biased coin as another `flip` distribution, with argument either 2/3 or 1/3 depending `more-heads`. This represents the coin, `true` meaning heads and `false` tails. Sampling from it would be the same as tossing the coin. However, in order to implement the example, what I want to indicate is that I have tossed it and gotten a specific result. This is what `observe` is for. I have tossed the coin 3 times and gotten as many heads, so as many times, I write that the distribution `coin` gives `true`. I finally use `predict` to indicate that I want to get the sequence of the values associated to `more-heads` (either `true` or `false`) in each trace of execution generated by the inference. Given the definition of this random variable, one could expect to see roughly as many of each. But because of the `observe` statements, since `coin` is more likely to actually give `true` when `more-heads` is `true`, we get `true` really more often than `false`. When considering a lot of samples, we do get `true` about 8 out of 9 times (88,796 times in 100,000 samples), as calculated in 2.1.

### 2.3 Inference algorithms

As seen precedently, to a trace of executions, which details all the results of random choices, is associated a joint probability, such that there is a joint distribution over all possible traces. Executing a program in Anglican is supposed to give samplings of executions according to this joint distribution. This is done thanks to built-in inference algorithms. Anglican proposes several of them, which can be selected with a tag. The first algorithm described below is easy to understand: I present it because it shows well what we want to achieve, but it is not usable in practice, as I will explain. The second one is actually proposed by Anglican, and one of the most often selected.

A naive way to simulate the target posterior distribution is as follows. Sample all variables using the distributions directly given in the code (in the previous example, sample `more-heads` from `(flip 0.5)`). When `(observe d x)` is encountered, draw a sample from d, and keep going only if it is equal to x, otherwise reject the whole execution and start a new one. In the example, an execution is accepted only if three times, we sample from `coin` and actually get `true`. If d is a continuous distribution, this is not doable because there is almost no chance to sample exactly x, instead decide on an acceptance protocol based on the value taken by the probability density function in x; this function can be computed for any distribution explicitly definable in the language.

This guarantees that accepted executions are likely to give sense to the observations described in the program. The overall distribution over the traces of execution obtained this way is really the target one. However this is not doable in practice, even with the precaution we mentioned about continuous distributions. When the model implemented gets complex, it is far too long to get an accepted execution, and reasonably impossible to get a lot of them to study the distribution; yet drawing one sample from a distribution, even if it is exactly the right distribution, is useless.
Lightweight Metropolis-Hastings (LMH) [10] is available in Anglican and one of the most often selected. Its principle is to build a Markov chain of traces of execution. It starts by computing an execution without taking the `observe` into account; the associated trace becomes the first current trace of execution. Then it chooses randomly a point where a random choice is recorded in the current trace, with more probability to choose points where the result appearing in the trace had a lower probability to be drawn. From this point on, it makes new draws for all random choices (maybe following a new control flow depending on the results), thus generating a new trace of execution which shares a prefix with the previous one. Then the respective joint probabilities of the two traces are compared. If the one of the newest trace is higher, the algorithm switches to it as current trace. If not, it still has a probability to switch on the new one, depending on the difference between the two joint probabilities, in order not to stay stuck. Otherwise the current trace execution is kept. The algorithm repeats this as many times as required by the user, providing at each step the values in the current trace of the variables in `predict`. This describes a Markov chain over the possible executions, with the correct distribution. However, whether the distribution represented by a realistic number of iterations is close to this correct distribution depends on the program: it is usually quite good, but on some treacherous ones such as naive LDA, it is too much affected by random noise.

2.4 Latent Dirichlet Allocation (LDA)

Topic models are about identifying themes, called topics, which appear frequently in a large collection of documents, and for each document indicate which of these topics are present, based on the words composing the documents. They can be used to organize, search or understand a large amount of data.

LDA [1] is a very simple and intuitive generative topic model. The idea is that a topic is characterised by a few recurring words, and each document is a mixture of a few topics, the words of which form most of the document. Therefore a collection of documents can be generated as follows. The vocabulary is a fixed collection of available words, for example all English words. The number of topics can also be fixed, or it can be randomly chosen beforehand.

- Generate the topics: for each of them, randomly choose a distribution over the worlds of the vocabulary.
- For each document:
  - randomly choose a distribution over topics
  - for each word of the document:
    * randomly choose a topic according to the distribution just above
    * randomly choose a word according to the distribution associated to the selected topic

Documents’ distributions over topics and topics’ distributions over words are categorical distributions: simply a choice over a finite number of outcomes, the probability of each outcome being specified. From now on we call them discrete distributions, a usually less precise appellation, but here it is the only kind of discrete distributions we will be considering, and it corresponds to the name we are using in Anglican.

In the generative model described, we need a way to randomly generate such discrete distributions, which can be characterised by a vector of the probability of each outcome. We use Dirichlet distributions. They are characterised by a vector of prior belief, and range over the probability
vectors (components summing to 1) of the same length, thus also over the discrete distributions with a number of categories equal to the length of the vector.

We can now write the generative model using probabilistic notations. \( \sim \) means sample from a distribution. \( \text{Dirichlet}(\alpha) \) is the Dirichlet distribution characterised by \( \alpha \); sampling from it gives a probability vector of the same length as \( \alpha \). \( \text{Discrete}(p) \), where \( p \) is a probability vector of length \( n \), is the corresponding discrete distribution over integers from 0 to \( n - 1 \).

Capital letters are fixed integers. Vocabulary: \( \{0, 1, ..., V - 1\} \). Number of topics: \( K \). We want to generate \( D \) documents of \( N \) words each, which is the same as generating words \( w_{d,n} \) from the vocabulary with \( 0 \leq d < D \), \( 0 \leq n < N \). We note \( \varphi_d \) the probability vector representing the distribution over topics chosen for document \( d \), \( \theta_k \) the one representing the distribution over words for each topic, \( 0 \leq k < K \). \( z_{d,n} \) is the topic selected for the word \( w_{d,n} \). \( \alpha \) and \( \beta \) are fixed vector of length \( K \) and \( V \) respectively, with all components equal, and low especially for \( \beta \): this way the \( \theta_k \) will typically be very sparse, with most of the weight on a few words only.

\[
\begin{align*}
\theta_k & \sim \text{Dirichlet}(\beta) \\
\varphi_d & \sim \text{Dirichlet}(\alpha) \\
z_{d,n} & \sim \text{Discrete}(\varphi_d) \\
w_{d,n} & \sim \text{Discrete}(\theta_{z_{d,n}})
\end{align*}
\]

As a topic model, the usual goal when applying LDA is, given documents, finding out the topics and which topics are involved in each document, which means: given the \( w_{d,n} \), determining the most likely \( \theta_k \) and \( \varphi_d \). Like all generative models, it can become a conditional model using Bayesian statistics. The problem becomes computing the posterior distributions of the \( \theta_k \) and \( \varphi_d \), called latent variables, conditioned on the observation of the \( w_{d,n} \). \( K \) the number of topics, \( \alpha \) and \( \beta \) the arguments to Dirichlet distributions are hyperparameters; we could try to optimise them as well by also considering probability distributions on them, making them hyperpriors, but to make things easier we consider them to be fixed values given as inputs like the documents. In practice, we will write probabilistic programs which return the \( z_{d,n} \) instead of the \( \theta_k \) and \( \varphi_d \), the latter being easy to estimate based on the \( z_{d,n} \) and \( w_{d,n} \) (for \( \varphi_d \) look at the repartition of the \( z_{d,n} \) when \( n \) varies, for \( \theta_k \) consider all \( w_{d,n} \) such that \( z_{d,n} = k \)).

This is easy and short to implement in Anglican (figure 2). Here we have \( D = 2 \) documents of \( N = 4 \) words each, \( K = 3 \) topics, \( V = 5 \) words in the vocabulary. Of course such code can be written more properly using functions on sequences such as \texttt{map} \( \), allowing the parameters above to be much bigger, but it gets difficult to read in an unfamiliar language so I prefer writing the example this way.

This is the naive implementation of LDA on which the inference algorithms will give bad results; we will explain why in the next section.
3 Collapsing LDA: interest and justification

3.1 Highly expressive latent variables: a challenge for existing inference algorithms

The inference algorithms of Anglican give poor results on the naive implementation of LDA above: the predictions depend more on random noise than on the input documents. This is due to the $\theta_k$ and $\varphi_d$ being highly expressive latent variables: the quality of the inference on the whole program depends critically on getting these variables right. Let us show their impact on the two examples of inference algorithms given in 2.3.

Highly expressive latent variables make the naive algorithm first described in 2.3 even less usable on naive LDA than other programs. Indeed, an execution would need to sample, just by chance, realistic values for all of them, to have any chance at all to be accepted by every subsequent $\text{observe}$. Therefore obtaining results would take even longer, even less realistic time.

LMH also needs to guess right while sampling highly expressive latent variables. It is still better than the previous algorithm because when it starts having reasonable values for some of them, it has a possibility to keep them in the unchanged trace prefix, instead of having to get all of them right at once. However, if the first defined variables are inadequate, it needs to sample them anew.
at some point, which means also sampling anew all variables which are defined later. So it really
needs a lot of steps to start having reasonable values for the very first variables, then the next
ones, and so on. In practice, this sometimes happens in a reasonable time, and sometimes not, cf.
graphical results in 7.

The main thing to remember is that such highly expressive latent variables, which are defined as
just one sampling in the beginning of the program but conditioned by a lot of \texttt{observe}, are highly
challenging for inference, and no known algorithm can handle them both quickly and accurately.

### 3.2 Dirichlet Process: the core of collapsed LDA

Dirichlet Process is an essential component of the collapsed version of LDA. It consists of starting
with a real vector of length \( n \) and repeatedly sample an integer in \( \{0, 1, ..., n - 1\} \) according to
the discrete distribution described by the current vector (the probability of getting an integer is
the corresponding component divided by the sum of the components), then update the vector by
adding 1 to the component corresponding to the integer obtained. This way the more an integer
has been sampled, the more probable it is to sample it again. The great interest of Dirichlet Process
is that is it equivalent to sampling a probability vector from the Dirichlet distribution described
by the initial vector, then repeatedly sampling from the discrete distribution given by the vector
obtained. In other words, given the initial real vector \( \alpha \), the two following models are equivalent,
in the sense that the joint probability \( p(x_0, x_1, ..., x_{k-1}) \) is the same in both.

\[
\begin{align*}
\theta & \sim \text{Dirichlet}(\alpha) \\
x_0 & \sim \text{Discrete}(\theta) \\
x_1 & \sim \text{Discrete}(\theta) \\
... \\
x_{k-1} & \sim \text{Discrete}(\theta)
\end{align*}
\]

\[
\begin{align*}
v & := \text{copy of } \alpha \\
x_0 & \sim \text{Discrete}(v) \quad v[x_0] := v[x_0] + 1 \\
x_1 & \sim \text{Discrete}(v) \quad v[x_1] := v[x_1] + 1 \\
... \\
x_{k-1} & \sim \text{Discrete}(v) \quad v[x_{k-1}] := v[x_{k-1}] + 1
\end{align*}
\]

This is most interesting for implementing LDA in an inference-friendlier way. Indeed, the
problematic highly expressive latent variables \( \varphi_d \) and \( \theta_k \) have a role similar to \( \theta \) in the model
above, thus can be replaced with as many Dirichlet Processes. They are said to be \textit{marginalised}:
they don’t appear explicitly in the code anymore, but their sense is indirectly maintained through a
succession of small updates. This is very good for inference algorithms, which are better at several
simple guesses than a single complex one.

\textbf{Note} - With the variable names above, if we want to observe a value \( x \) from the distribution
\texttt{Discrete(\theta)} in Anglican but \( \theta \) has been marginalised, we need both to observe \( x \) from \texttt{Discrete(\alpha)}
with the current value of \( \alpha \), and then to update \( \alpha \) by incrementing the component corresponding
to \( x \).

### 3.3 Conjugate prior: justification of collapsing

The equivalence between the models described above comes from the following property about
Dirichlet and discrete distributions.

\textbf{Theorem.} These two models give the same joint probability \( p(\theta, x) \)
\[ \theta \sim \text{Dirichlet}(\alpha) \quad x \sim \text{Discrete}(\alpha) \]
\[ x_0 \sim \text{Discrete}(\theta) \quad \theta \sim \text{Dirichlet}(f(\alpha, x)) \]

where \( f(\alpha, x) \) is \( \alpha \) where the component with index \( x \) has been incremented.

If we consider \( \theta \) to be the latent variable and \( x \) the observation, then \( \text{Dirichlet}(\alpha) \) is the prior distribution, \( \text{Dirichlet}(f(\alpha, x)) \) the posterior distribution and \( \text{Discrete}(\theta) \) the likelihood (the posterior is conditioned on \( x \) and the likelihood on \( \theta \)). Since the prior and the posterior are in the same family (both Dirichlet distributions), we say that they are conjugate, and that Dirichlet distribution is a conjugate prior for a Discrete likelihood. These names exist because a lot of other distributions have this kind of property. The fact that the upper right distribution is also a Discrete distribution, the same family as the likelihood, is not relevant.

This explains the equivalence between the two models presented in 3.2: we can transform the first one into the second one progressively as shown below. During the first step we only change the definitions of \( \theta \) and \( x_0 \) and the conjugate prior property assures that the joint probability \( p(\theta, x_0) \) is conserved, so \( p(\theta, x_0, x_1, \ldots, x_{k-1}) \) is conserved as well. We do this for each \( x_i \) in turn. \( p(\theta, x_0, x_1, \ldots, x_{k-1}) \) never changes, in particular \( p(x_0, x_1, \ldots, x_{k-1}) \) is also the same in the first and last models.

\[
\begin{align*}
\theta & \sim \text{Dirichlet}(\alpha) & x_0 & \sim \text{Discrete}(\alpha) & x_0 & \sim \text{Discrete}(\alpha) & x_0 & \sim \text{Discrete}(\alpha) \\
x_0 & \sim \text{Discrete}(\theta) & \theta & \sim \text{Dirichlet}(f(\alpha, x_0)) & \alpha & := f(\alpha, x_0) & \alpha & := f(\alpha, x_0) \\
x_1 & \sim \text{Discrete}(\theta) & x_1 & \sim \text{Discrete}(\theta) & x_1 & \sim \text{Discrete}(\alpha) & x_1 & \sim \text{Discrete}(\alpha) \\
& \ldots & & & & \alpha & := f(\alpha, x_1) & \alpha & := f(\alpha, x_1) \\
x_{k-1} & \sim \text{Discrete}(\theta) & x_{k-1} & \sim \text{Discrete}(\theta) & \theta & \sim \text{Dirichlet}(\alpha) & \theta & \sim \text{Dirichlet}(\alpha) \\
& \ldots & & & & & & \\
x_{k-1} & \sim \text{Discrete}(\theta) & & & & & & \\
& \alpha & := f(\alpha, x_{k-1}) & & & & & \\
& \theta & \sim \text{Dirichlet}(\alpha) & & & & &
\end{align*}
\]

This collapsing transformation can therefore be applied to other distributions as well, so long as they are a conjugate prior. I have focused on automatising it for Dirichlet with Discrete likelihood only. Yet in 7, I give an example of hand-written naive and collapsed implementations in Anglican for other distributions: Normal conjugate prior with Normal likelihood.

## 4 Automatic code transformation: technical decisions

As explained in 3.2, the collapsing of LDA consists of marginalising highly expressive latent variables, by replacing them with Dirichlet Processes. In terms of Anglican code, for each such variable, let us say it is called \texttt{theta}, here is what we want to do. \texttt{theta} should be defined as a sampling from a (\texttt{dirichlet alpha}) distribution. We suppress this definition, but instead we define \texttt{DP-vector} initialised as \texttt{alpha}. We replace any occurrence of \texttt{(sample (discrete theta))} by \texttt{(sample (discrete DP-vector))} and an update of \texttt{DP-vector}: incrementation of the component corresponding to the sampled value. We replace all \texttt{(observe (discrete theta)x)} with \texttt{(observe (discrete DP-vector)x)} and update of \texttt{DP-vector}: incrementation of the component corresponding to \( x \). This should also be done for expressions such as \texttt{(sample d)} when \( d \) is equal to \texttt{(discrete theta)}. Doing this automatically requires some precautions and technical decisions.
I discuss some of them below, before exposing my formal definition of the automatic transformation in the next sections.

4.1 Necessary conditions on variables to be marginalised

The first step of the transformation consists of determining which variables should be marginalised. Not any variable can be replaced with a Dirichlet Process. For example, it needs to be sampled from a Dirichlet distribution. Moreover, we may not want to marginalise any variable that could technically be. Indeed, this implies changing potentially numerous pieces of code where it may appear, which can become really complex depending on how the variable is defined and used. Imposing more necessary conditions on variables to be marginalised allows to make simpler and more systematic modifications of the code without breaking it.

A necessary condition imposed on variables to be marginalised is that it is only used as argument to `discrete`. Indeed, a marginalised variable `theta` does not exist anymore in the collapsed code, that was the goal of the transformation. The Dirichlet Process provides a way to replace sampling and observation of Discrete distributions with argument `theta`. However, if `theta` appears outside of a Dirichlet distribution, for example if we want to do some operation on its components (recall it represents a vector), or even if we simply want to use `predict` on it, we cannot do it in the collapsed program because we do not have its value. Actually we could do something like sampling it just when we need its value, using the DP-vector which we keep updating. It would look like an intermediate step of the justification of collapsing in 3.3. However this would make the transformation more complex without bringing so much. Moreover once some operations have been done using the actual value, we cannot marginalise the variable again, and it may still be a too much expressive latent variable. That’s why I have decided not to handle such cases at all.

Another necessary condition to marginalise `theta` is that when it is defined, we know the value of the argument `alpha` of the Dirichlet distribution from which it is sampled. Indeed, we need to initialise DP-vector to `alpha`. Not knowing `alpha` can happen: if a function defines and returns the Dirichlet distribution with an argument `alpha` local to the function, and if `theta` is defined outside of the function, then `alpha` is not accessible anymore. I have decided not to marginalise the variable in such a case. Once again, it could be done, for example by wrapping distributions in objects that recall the argument, but it would lead to more complex code for a feature which is not so important.

For example, in the following code, `theta0` can be marginalised; `theta1` cannot because `alpha1` is not known anymore outside of the function; `x2` cannot since it is not sampled from a Dirichlet distribution; `theta3` cannot because it is used outside of `discrete` in the last line.

```plaintext
(let [alpha0 [1 1]
  d0 (dirichlet alpha0)
  theta0 (sample d0)
  f (fn [] (let [alpha1 [2 3 4]] (dirichlet alpha1)))
  d1 (f)
  theta1 (sample d1)
  d2 (discrete theta0)
  x2 (sample d2)
  theta3 (sample d0)]
 (nth theta3 0))
```
4.2 Simulating side effects with a state and modular functions

For every marginalised variable, we need a Dirichlet Process vector which is used and updated every time the corresponding Discrete distribution is used. This information needs to be accessible anywhere such a distribution may be used, inside functions for example. An easy way to do this would be to put it in a reference which is passed around at the same points where the marginalised variable is, and update it with side effects. However Clojure and Anglican do not support side effects well: they make inference slower. Without side effects, we need to make functions return sufficient information about eventual updates happening inside it. What’s more, we want them to behave well both with marginalised and not marginalised variables. Indeed, which type of variables appear cannot be determined statically since it may depend on random choices.

The technique we use to handle this is rather natural. We pass a state everywhere in the program, allowing to simulate side effects. The state is a hash-table, called map in Clojure, containing all the Dirichlet Process vectors. It is both an argument and a returned value of every function.

To make as few changes as possible in the code, marginalised variables are left in the transformed code, only the value that they contain is changed: their new value is the key in the state to the vector of the corresponding Dirichlet Process. This way, the key is naturally passed wherever it may be needed, inside functions for example.

We define new functions $S$-discrete, $S$-sample and $S$-observe, capable of handling both usual variables and distributions, and marginalised ones and Dirichlet Processes. $S$-discrete $\theta$ does the same as discrete $\theta$ if $\theta$ is not marginalised; if it is, which means $\theta$ contains a key to information in the state, it returns an special object carrying this key. If $d$ is a usual distribution, then $S$-sample $d$ $S$ and $S$-observe $d$ $x$ $S$ return $[r]$ where $r$ is what sample $d$ or observe $d$ $x$ would return, and $S$ is unchanged. If $d$ is the special object carrying a key to the state, the corresponding vector from the state is used to generate $r$ which is still the kind of result that sample of observe would give, and in the returned $S$ it has been updated.

The modularity of these new functions allows us to replace any occurrence of discrete, sample, observe in the initial code with the corresponding one, without having to know whether their argument is affected by marginalisation.

5 Automatic code transformation: formalism

5.1 Lambda calculus

In order to formalise the transformation, I introduce a lambda calculus corresponding to a small subset of Anglican. Since Dirichlet and Discrete distributions take vectors as arguments, I need some vector handling: nth $v$ $i$ is the element of $v$ with index $i$, and conj $v$ $x$ the vector obtained by adding $x$ at the end of the vector $v$. $x$ represents a variable, $c$ a primitive.

$$e ::= x \mid \lambda x . e \mid e_1 \ e_2 \mid c$$
$$\mid \text{let } x = e_1 \text{ in } e_2$$
$$\mid [e_1; \ldots ; e_n] \mid \text{nth } e_1 \ e_2 \mid \text{conj } e_1 \ e_2$$
$$\mid \text{dirichlet } e \mid \text{discrete } e$$
$$\mid \text{sample } e \mid \text{observe } e_1 \ e_2 \mid \text{predict } e$$

From now on, we suppose given an expression from this calculus called program, which is the source code that we want to transform. For example it would be the translation of the body of an
Anglican query.

We assume it to be correct code, meaning that the corresponding Anglican code can be executed without any error. We also assume that there is no function inside a list.

### 5.2 *a*-normal form

In order to make reasoning and code rewriting easier, we put the code under *a*-normal form: expressions contain only variables, except `let . in .` and `λ...`. This means that intermediate variables are introduced so that most expressions represent only one evaluation. My notion of *a*-normal form is a little unusual with the kinds of expressions.

\[
a ::= x \mid c \mid \text{nth } x_1 \ x_2 \mid \text{conj } x_1 \ x_2 \\
dirichlet \ x \mid \text{discrete } x \\
sample \ x \mid \text{observe } x_1 \ x_2 \mid \text{predict } x \\
λx.e' \mid x_1 \ x_2
\]

\[
e' ::= x \mid \text{let } x = a \ \text{in } e'
\]

Note: an expression is of form `e'` if and only if it can be written as `let x_1 = a_1 \ \text{in } \ldots \ \text{let } \ x_n = a_n \ \text{in } y`.

**Theorem.** Any expression of form `e` may be transformed to an equivalent one of form `e'`.

**Proof.** By induction on the structure of `e`. Assuming that eventual sub-expressions `e_1` and `e_2` may be transformed to `e'` forms, which we write respectively `let \ x_1^1 = a_1^1 \ \text{in } \ldots \ \text{let } x_n^1 = a_n^1 \ \text{in } y^1` and `let \ x_1^2 = a_1^2 \ \text{in } \ldots \ \text{let } x_m^2 = a_m^2 \ \text{in } y^2`:

- `x` (of form `e`) becomes `x` (of form `e'`)
- `c` becomes `let \ x = c \ \text{in } x`
- if `**` is `dirichlet`, `discrete`, `sample` or `predict`, then `** \ e_1` becomes `let \ x_1^1 = a_1^1 \ \text{in } \ldots \ \text{let } x_n^1 = a_n^1 \ \text{in } y = \ ** \ y^1 \ \text{in } y`.
- if `**` is `nth`, `conj`, `observe` or nothing (function application), then `** \ e_1 \ e_2` becomes `let \ x_1^1 = a_1^1 \ \text{in } \ldots \ \text{let } x_n^1 = a_n^1 \ \text{in } \text{let } x_1^2 = a_1^2 \ \text{in } \ldots \ \text{let } x_m^2 = a_m^2 \ \text{in } y = \ ** \ y^1 \ y^2 \ \text{in } y`.
- the case `[e_1; \ldots; e_n]` is similar to `** \ e_1 \ e_2` with more expressions
- `λx.e_1` becomes `let \ f = λx.(\text{let } x_1^1 = a_1^1 \ \text{in } \ldots \ \text{let } x_n^1 = a_n^1 \ \text{in } y^1) \ \text{in } f`
- `let \ x = e_1 \ \text{in } e_2` becomes `let \ x_1^1 = a_1^1 \ \text{in } \ldots \ \text{let } x_n^1 = a_n^1 \ \text{in } \text{let } x = y^1 \ \text{in } \text{let } x_1^2 = a_1^2 \ \text{in } \ldots \ \text{let } x_m^2 = a_m^2 \ \text{in } y^2`.

\[\square\]

From now on, we assume program to be in *a*-normal form, more precisely an expression of type `e'`. We forget about the former `e` structure, and rename the current `e'` to `e` to have lighter notations.

### 5.3 Identifying potential variables to marginalise

We make sure that every variable name is unique, by renaming variables if needed. This makes this step easier.
We want to identify variables which could be marginalised. As a first step, we mark variables which satisfy the necessary conditions given in 4.1, except the one about not being used outside of \texttt{discrete}. This condition will be checked later, using more powerful program analysis tools. The rest of them are easy to handle with really basic static analysis.

We add a new expression of type $a$ to the calculus: \texttt{sample-dirichlet $\pi_i$ $x$}, where $x$ is a variable, and $i$ a unique index which will also allow to refer to this point in the program. This expression is used to highlight the definition of a variable which is a candidate to being marginalised because it satisfies the conditions checked here.

Thanks to $a$-normal form, in any expression \texttt{dirichlet $\alpha$} and \texttt{sample $d$}, $\alpha$ and $d$ are variables. We track variables defined as Dirichlet distributions, noting the argument. We also track the scope of every variable. For every occurrence of \texttt{sample $d$}, if $d$ is recorded as \texttt{dirichlet $\alpha$}, and if $\alpha$ is still assigned at this point of the program, we replace \texttt{sample $d$} with \texttt{sample-dirichlet $\pi_i$ $\alpha$}, where $i$ is a new unique index. As explained in 4.1, if the variable defined by this expression is actually marginalised, $\alpha$ will be useful to initialise the Dirichlet Process vector.

For example, the piece of code on the left is transformed to the one on the right. \texttt{sample $d_1$} cannot be replaced because $d_1$ represents a Dirichlet distribution with argument $\alpha$ which is not assigned anymore outside of the function.

```
let $\alpha_0 = \ldots$ in
let $f = \lambda x. (\text{let } \alpha = \ldots \text{ in }
\text{let } d = \text{dirichlet } \alpha \text{ in }
\text{d}) \text{ in }
\text{let } d_0 = \text{dirichlet } \alpha_0 \text{ in }
\text{let } \theta_0 = \text{sample } d_0 \text{ in }
\text{let } x_1 = 0 \text{ in }
\text{let } d_1 = f \ x_1 \text{ in }
\text{let } \theta_1 = \text{sample } d_1 \text{ in }
\ldots
```

```
let $\alpha_0 = \ldots$ in
let $f = \lambda x. (\text{let } \alpha = \ldots \text{ in }
\text{let } d = \text{dirichlet } \alpha \text{ in }
\text{d}) \text{ in }
\text{let } d_0 = \text{dirichlet } \alpha_0 \text{ in }
\text{let } \theta_0 = \text{sample-dirichlet } \pi_0 \alpha_0 \text{ in }
\text{let } x_1 = 0 \text{ in }
\text{let } d_1 = f \ x_1 \text{ in }
\text{let } \theta_1 = \text{sample } d_1 \text{ in }
\ldots
```

### 5.4 Final decision on variables to marginalise:

**type and effect system**

The variables of which the definition has been marked in the previous step fulfil every necessary condition to be marginalised, except maybe the one about not being used outside of \texttt{discrete}. This step is about checking this condition.

A \textit{type and effect system} is an extension of a type system. To an expression $e$, provided an environment $\Gamma$ which maps variables to types, are associated a type $\tau$ and also an effect $\varphi$, noted $\Gamma \vdash e : \tau \& \varphi$. As usual, the type tells the form of the value that the expression would be evaluated to. The effect tells what happens during the evaluation. For example, it is sometimes used to track which side effects happen at different places in a program. Cf [7].

Here effects are used to report when a variable which is candidate to marginalisation is used outside of \texttt{discrete}, thus should not be marginalised. The difficulty is that we want to track even indirect use: in this example of code, $\theta_0$ should not be marginalised because it is indirectly used as second argument to \texttt{observe}. We do this thanks to types: they are used to tell if an expression may evaluate to the value of a candidate variable or contain it inside a list. More concretely, below are the definitions of types and effects used here.
Type: $\tau = \rho \mid \tau_1 \rightsquigarrow \tau_2$, with $\rho$ = set of $\kappa$, with $\kappa = \pi_i \mid D\pi_i \mid [\kappa]$. Effect: $\varphi = \text{set of } \pi_i$.

A type can be a set of $\kappa$, in which $\pi_i$ means that the value of an expression with this type may be equal to a candidate variable defined at the program point identified with $\pi_i$: $D\pi_i$ means that it may be equal to a Discrete distribution of which the argument is equal to such a candidate variable; $[\kappa]$ means the value of the expression may be a list containing what $\kappa$ describes. In the example above, if $\theta_0$ has type $\{\pi_0\}$, then the type of $l$ would be a set containing $[\pi_0]$ so $x$ would contain $\pi_0$ so we would know that $\pi_0$ appears at second argument of observe. The effect of an expression is the set of all $\pi_i$ that may be used outside of discrete inside the expression. In the same example, the effect of the whole contains $\pi_0$. Types of form $\tau_1 \rightsquigarrow \tau_2$ are for functions, and mean that if the function is called with an argument of type $\tau_1$, then the function call has type $\tau_2$ and effect $\varphi$.

Following are the rules of our type and effect system. $x : \rho \vdash e : \tau \& \varphi$ means that $\Gamma \vdash e : \tau \& \varphi$ for any $\Gamma$ that maps $x$ to the type consisting of the set $\rho$.

$$
\begin{align*}
x : \tau &\vdash x : \tau \& \emptyset \\
x_1 : \rho_1, \ldots, x_n : \rho_n &\vdash [x_1; \ldots; x_n] : \bigcup_{i=1}^n \left( \bigcup_{\kappa_i \in \rho_i} [\kappa_i] \right) \& \emptyset \\
x_1 : \rho_1, x_2 : \rho_2 &\vdash \text{nth } x_1 x_2 : \bigcup_{\kappa_i \in \rho_1} \kappa \& \bigcup_{\pi_i \in \rho_1 \cup \rho_2} \pi_i \\
x_1 : \rho_1, x_2 : \rho_2 &\vdash \text{conj } x_1 x_2 : \rho_1 \cup \left( \bigcup_{\kappa_i \in \rho_2} [\kappa_i] \right) \& \bigcup_{\pi_i \in \rho_1 \cup \rho_2} \pi_i \\
x : \rho &\vdash \text{dirichlet } x : \emptyset \& \bigcup_{\pi_i \in \rho} \pi_i \\
x : \rho &\vdash \text{discrete } x : \bigcup_{\pi_i \in \rho} (D\pi_i) \& \emptyset \\
x : \rho &\vdash \text{sample-dirichlet } \pi_i, x : \{\pi_i\} \& \left( \bigcup_{\pi_j \in \rho} \pi_j \right) \\
x : \rho &\vdash \text{sample } x : \emptyset \& \emptyset \\
x_1 : \rho_1, x_2 : \rho_2 &\vdash \text{observe } x_1 x_2 : \emptyset \& \left( \bigcup_{\pi_i \in \rho_2} \pi_i \right) \\
x : \rho &\vdash \text{predict } x : \emptyset \& \bigcup_{\pi_i \in \rho_2} \pi_i \\
\Gamma, x : \tau_1 &\vdash e : \tau_2 \& \varphi \\
\Gamma \vdash \lambda x.e : \tau_1 \rightsquigarrow \tau_2 \& \emptyset \\
\Gamma \vdash a : \tau_1 \& \varphi_1 \\
\Gamma, x : \tau_1 &\vdash e : \tau_2 \& \varphi_2 \\
\Gamma &\vdash \text{let } x = a \text{ in } e : \tau_2 \& \varphi_1 \cup \varphi_2
\end{align*}
$$

The type and effect associated to an expression given an environment are not unique: they describe what may happen, so a lot of lenient types and effects are correct. We try to be precise while keeping the system reasonably simple. Unsurprisingly, we are still doing a lot of over-approximation. For example, we do not keep track of position within a list: if a value is put in a list, we consider that anything which is later taken from the list may be equal to it.

The functions may have many relevant types when the type of the argument varies. Using these rules, we need to compute a different type for any call to the same function with a different type of argument. It is the main point that affects the complexity of this system, making it potentially exponential in the number of nested functions. Yet this number is generally low. Moreover in my implementation I decided to recall the types obtained for each type of argument, which are usually not so diverse, so that they do not need to be calculated again. All this makes computing this system quick for almost any realistic program. This could probably be improved using polymorphic types,
but I did not make it a priority since it works well enough like this.

The variables that we actually want to marginalise are exactly the ones that are defined as sample-dirichlet \( \pi_i \) such that \( \pi_i \) is not in the effect \( \varphi \) of program (the code that we want to transform), computed with an initial environment where any free variable is associated an empty set. Indeed, thanks to the rule about let merging the effects of the sub-expressions, for any candidate variable potentially used outside of discrete anywhere in program, the corresponding \( \pi_i \) is present in the effect \( \varphi \) of program.

We may want to apply the transformation only to a part of some code, what we call program being included in a larger one. In this case, to marginalise a variable represented by \( \pi_i \), we also want neither \( \pi_i \) nor \( D\pi_i \), or \( [\pi_i] \) or any of them inside any number of [], to be in the type \( \tau \) of program: indeed, we do not want program to return a marginalised variable or distribution built on it or a list containing them, because the transformation makes them usable only in some proper ways, and we could not control what external code would do with them.

This step determines the definitive list of variables to marginalise. For former candidate variables which we finally decide not to marginalise, we reverse the replacement made in 5.3: we put back the original expression with sample instead of sample-dirichlet. Now, variables defined using sample-dirichlet are exactly the ones which should be marginalised.

### 5.5 Code rewriting

This step is easy to define, especially thanks to the modular functions S-discrete, S-sample and S-observe introduced in 4.2, which work correctly independently on the arguments being affected by marginalisation. The state which allows to simulate side effects, also introduced in 4.2, is passed through the program using always the same variable \( S \) (if a variable with this name already exists, rename it before doing this step).

Transformed code does not follow the definition of the lambda calculus anymore, although it has the same kind of structure. let \( [x_1;x_2] = a \) in \( e \) means that if \( a \) is evaluated to a vector with two components, then \( e \) should be evaluated with \( x_1 \) and \( x_2 \) respectively assigned to these components. This mechanism, widely used in Clojure, is called destructuration.

We define the transformation of an expression inductively, noting \( A \) and \( E \) the respective images of eventual sub-expressions \( a \) and \( e \).

Expressions of type \( a \):  
$$
\begin{array}{ll}
\text{discrete } x & \mapsto S\text{-discrete } x \\
\text{sample } x & \mapsto S\text{-sample } x \ S \\
\text{observe } x_1 x_2 & \mapsto S\text{-observe } x_1 x_2 \ S \\
\text{sample-dirichlet } \pi_i \ x & \mapsto \text{marginalise } \pi_i \ x \ S \\
\lambda x.e & \mapsto \lambda x.\lambda S.E \\
x_1 x_2 & \mapsto x_1 x_2 \ S \\
a & \mapsto a & \text{if } a \text{ has any other form: } x, c, [x_1;\ldots;x_n], \text{nth } x_1 x_2, \text{conj } x_1 x_2, \\
\text{dirichlet } x, \text{predict } x
\end{array}
$$

Expressions of type \( e \):  
$$
\begin{array}{ll}
x & \mapsto [x;S]
\end{array}
$$
let \( x = a \) in \( e \) \( \mapsto \) let \( \[x; S]\) = \( A \) in \( E \) if \( a \) has the form \textbf{sample} \( x \),

\[
\text{observe } x_1, x_2, \textbf{sample-dirichlet} \pi_1, x \text{ or } x_1, x_2
\]

let \( x = a \) in \( e \) \( \mapsto \) let \( x = A \) in \( E \) if \( a \) has another form

Let \( P \) be the image of \textit{program} by this transformation. The final collapsed code is:

\[
\text{let } S = \text{emptymap in let } \[x; S\] = P \text{ in } x
\]

Figure 3 gives the code output by the transformation applied to the naive LDA implementation of figure 2, page 10. It is hard to read, but efficient!

```
(let [S {}] [r S] (let [w00 0 w01 0 w02 1 w03 2 w10 4 w11 3 w12 1 w13 4 x21108 1 x21109 1 x21110 1 alpha [x21108 x21109 x21110] x21111 0.1 x21112 0.1 x21113 0.1 x21114 0.1 x21115 0.1 beta [x21111 x21112 x21113 x21114 x21115] x21116 (dirichlet alpha) [theta0 S] (marginalise :pi21147 alpha S) x21117 (dirichlet alpha) [theta1 S] (marginalise :pi21148 alpha S) x21121 (dirichlet beta) [x21118 S] (marginalise :pi21149 beta S) x21122 (dirichlet beta) [x21119 S] (marginalise :pi21150 beta S) x21123 (dirichlet beta) [x21120 S] (marginalise :pi21151 beta S) phis [x21118 x21119 x21120] f (fn [theta S] (let [x21129 (fn [w S] (let [x21126 (S-discrete theta) [z S] (S-sample x21126 S) x21128 (nth phis z) x21127 (S-discrete x21128) [z S]]) [x21129 S]]) [x21134 S] (f theta0 S) [x21130 S] (x21134 w00 S) [x21135 S] (f theta0 S) [x21131 S] [x21130 S] x21135 w01 S) [x21136 S] (f theta0 S) [x21132 S] (x21135 w02 S) [x21137 S] (f theta0 S) [x21133 S] (x21137 w03 S) zs0 [x21130 x21131 x21132 x21133] [x21142 S] (f theta1 S) [x21138 S] (x21142 S) [x21143 S] [x21139 S] [x21143 w11 S] [x21144 S] (f theta1 S) [x21140 S] (x21144 w12 S) [x21145 S] (f theta1 S) [x21141 S] (x21145 w13 S) zs1 [x21138 x21139 x21140 x21141] zs [zs0 zs1] x21146 (predict zs) [x21146 S]]) r)
```

Figure 3: Output of the transformation applied to the naive LDA implementation of figure 2

6 Potential improvements to the automatic transformation

- As mentioned in 4.1, the necessary conditions on variables to be marginalised could be weakened, making the transformation more complex. We discussed the possibility to marginalise a variable which is used outside of \textit{discrete}, sampling it when it is needed using the corresponding Dirichlet Process vector. This would be easy enough to do if it happens at the end of the program. It becomes more complex when there are \textit{observe} to process after it, because once a sampled value has been used, the variable cannot be marginalised again. In some cases we could swap pieces of code, so that the actual value is used last. However this is not always possible, for example if there are different behaviours depending on the value. We could still use this sort of workaround to be able to marginalise variables under weaker necessary conditions.

- The use of a state passed everywhere makes the transformation considerably easier, but may pose problems in some cases. As we said in 4.2, we use it to simulate side effects because the native ones of Anglican can have bad impact on inference. In some kinds of programs, our method has the same kind of negative effect.
It is possible to extend the type and effect system to track where each vector of the state may be used. This would allow to wipe away vectors when they will not be used anymore, and more importantly to pass only a part of the state to a function when we know that the rest cannot be needed. In the same spirit, we could split the information into several states, each of them having its own scope. If the splitting is relevant, this could be a great improvement. This has similarities to a field of research called *region inference* [9], which may suggest useful techniques to do so.

Another interesting lead consists of using a local state inside a function when it defines a marginalised variable and does not return anything allowing to use it later, even indirectly. The developers of Anglican are highly interested in this kind of improvement. However this requires a lot more work before it allows to make transformed programs notably better.

- The justification of collapsing using only the property of conjugate prior in 3.3 suggests that such a transformation can be applied to other distributions with the same property. I have written by hand examples of naive and collapsed code for Normal distributions. A transformation handling them could be defined like the one about Dirichlet and Discrete, with only a few changes. A more interesting problem would be to define a single transformation handling a lot of different distributions.

7 Experimental performance of naive and collapsed implementations

I have implemented², in Clojure, the transformation for a small subset of Anglican corresponding to the lambda calculus defined in 5.1. I have run it on various naive implementations of LDA.

In order to highlight to performance of the inference algorithms on the code issued by the transformation compared to the initial code, I use a small, simplistic and very contrasted corpus of documents. The vocabulary consists of only $V = 3$ words: \{0, 1, 2\}. There are $D = 20$ documents of $N = 10$ words each, and $K = 2$ topics. I generate a corpus, which I will later use as input to LDA implementations, with imposed topics $\theta_0 = [0.5; 0; 0.5]$ and $\theta_1 = [0; 0.5; 0.5]$: one topic consists only of words 0 and 2 in equal proportions, the other only of 1 and 2. I also impose that some documents contain only topic 0 ($\varphi_d = [1; 0]$), some only topic 1 ($\varphi_d = [0; 1]$), and all the others half of each topic ($\varphi_d = [0.5; 0.5]$). This defines a very contrasted corpus, for which the only fitting topics are the initially chosen ones.

I run Anglican on both a naive implementation of LDA and the collapsed code obtained by applying my transformation to it, with this corpus as input. I select the inference algorithm LMH ([10], quick overview in 2.3) and require 100,000 samplings. For each run, I plot all the sampled estimations of the distributions over words characterising each topic: $\theta_0$ and $\theta_1$. They are vectors $[p_0; p_1; p_2]$ where $p_i$ is the proportion of the word $i$. The $x$-axis indicates the number of samples already computed, and the $y$-axis the proportions $p_0$, $p_1$, $p_2$.

²My implementation can be found at https://bitbucket.org/dianegalloiswong/anglican-collapse-lda
As we already said, the corpus is chosen so that the only fitting pair of topics is $[0.5; 0; 0.5]$ and $[0; 0.5; 0.5]$ (in any order).

On the naive implementation, with 100,000 sampling, the correct topics are sometimes discovered (A) and sometimes not (B). I have run it several times, and it regularly gives something good which looks like A, but also often something totally different and incorrect. For example, the topics suggested in B do not explain the presence of documents with about as many 1 as 2 but not any 0: they cannot rely strongly on the second topic because then they would contain at least a few 0, so they have to contain almost only the first topic, but then they should have a lot more of 2 than of 1.

On the transformed code which is a collapsed implementation of LDA, in a little less than ten runs, I have always obtained the right topics like in C.

The main characteristic of a result is the convergence speed. On such a model, where there is a single good answer, the inference algorithms should always converge to it given enough time. The right thing to say about B is that this convergence takes more than 100,000 samples. In A it takes a little less than 40,000, and in C it takes about 10,000 samples. Even when the inference algorithm converges on naive code in less than 100,000 samples, it does it generally slower than on transformed code. This confirms that the program transformation allows to obtain inference-friendlier code.

**Adaptation to another distribution with the conjugate prior property: Normal distribution**

As explained in 3.3, the main fact that makes the transformation possible is the conjugate prior property of Dirichlet distribution with a Discrete likelihood. Other distributions have a similar property. For example, Normal distribution with known variance and latent mean is a conjugate prior with a Normal likelihood [6]. This would allow to define a similar program transformation, marginalising some variables defined as a sample from a Normal distribution. I have not defined the transformation, however I give an example showing what it would look like. I have written a small code where a variable $\mu$ is sampled from a Normal distribution with mean 0 and variance 1, then another Normal distribution is defined with mean $\mu$ and variance 1, from which we observe the values 2 and 3, then sample many values. I have also written by hand a corresponding collapsed
code: instead of sampling $\mu$, information (representing current belief of mean and variance of the distribution which used to be defined with mean $\mu$) are passed around to be used and updated in each `observe` and `sample`. The theoretical posterior Normal distribution after observing 2 and 3 has mean 1.67 and variance 1.33. For the initial code, the mean of the sampled values varies between -1 and 2 in a few runs. For the collapsed code, it is almost always between 1 and 2. So, even with Normal distributions, we obtain better results with collapsed code.

Conclusion

I have defined and implemented a program transformation thanks to which, in some special cases, a user can write simple code and obtain quality results which would otherwise require a more complex implementation. It is functional, and I have shown through experimental results that it indeed produces code which is more efficient than the input. Yet a lot of improvements can be thought of, in order to make it more general or obtain code which performs even better.

Using programming language tools to analyse and transform probabilistic programming code is an efficient method which has already given good results. In particular, [4] and [13] use a form of static or dynamic program slicing to improve the existing inference algorithms, and [8] and [2] use a variant of weakest-precondition calculation and data-flow analysis to help the inference algorithms or perform inference directly. All these approaches focus on low-level optimisations, which affect syntax or programming structures. My work differs from them in that with my transformation, the input and output codes describe equivalent but inherently different probabilistic models: it is a high-level model transformation. An interesting future direction is to explore whether my results can be applied to more advanced models such as sequence momoisers [12].

My supervisor Hongseok Yang is also currently working on a transformation of source code before it is given to inference algorithms so that they are more efficient. However, he uses a more general approach than mine, to be able to optimise a greater variety of programs. He will integrate my work into his more general transformation, which will eventually be added to the probabilistic programming language Anglican itself.

References


