

probKanren: A Simple Probabilistic extension for microKanren

Robert Zinkov¹, William E. Byrd²

¹*Dept. of Engineering Science, University of Oxford, 25 Banbury Rd, Oxford, UK*

²*Hugh Kaul Precision Medicine Institute, University of Alabama at Birmingham, 705 20th Street S., Birmingham, AL 35233, United States of America*

Abstract

Probabilistic programming can be conceptually seen as generalisation of logic programming where instead of just returning a set of answers to a given query, we also return a probability distribution over those answers. But many contemporary probabilistic logic programming languages implementations are not simple extensions of existing logic programming languages but instead involve their own unique implementations. Here we introduce `PROBKANREN`, a simple extension to `microKanren` that transforms it into a probabilistic programming language without needing to make any modifications to the underlying logic language's search. We use several illustrative examples from the probabilistic programming and program synthesis literature to demonstrate the practicality of the approach.

Keywords

Probabilistic Logic Programming, miniKanren, Probabilistic Programming, Sequential Monte Carlo

1. Introduction

Conceptually, logic programming provides a way to model non-determinism. This is accomplished by maintaining a set of answers that satisfy a set of logical constraints. A natural generalisation to this domain is adding a notion of uncertainty to this set of answers by associating with them a probability distribution.

But the conceptual simplicity of this sort of generalisation is not reflected in the complexity of many existing probabilistic logic programming systems. They often involve implementing sophisticated inference algorithms and the underlying systems are not just implemented on top of existing logic programming systems.

We believe that a conceptually simple extension deserves a conceptually simple implementation to go along with it. We thus contribute a simple way to extend `MICROKANREN`, a small logic programming domain-specific language such that it becomes a probabilistic programming language.

1.1. Illustrated Example

To help explain how to use `PROBKANREN` we introduce the following example:


PLP 2021

✉ zinkov@robots.ox.ac.uk (R. Zinkov); webyrd@uab.edu (W. E. Byrd)

🌐 <https://zinkov.com/> (R. Zinkov); <http://webyrd.net/> (W. E. Byrd)



© 2021 Copyright for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

 CEUR Workshop Proceedings (CEUR-WS.org)

```
(run 1000 (q)
  (conj
    (normal 0 3 q)
    (normal q 2 4)))
```

In the above program, we have a probabilistic model for the Gaussian unknown mean problem. We observe a value 4 from the $\mathcal{N}(q, 2)$ distribution, and that q has a prior distribution of $\mathcal{N}(0, 3)$. This `PROBKANREN` program draws 1000 samples from a conditional normal distribution that represents the posterior probability distribution associated with q .

2. Related Work

There is a rich history of extending logic programming formalisms to support probabilistic inference. Early systems like `PRISM`[1] and `ProbLog`[2] allowed associating discrete distributions with facts. Early versions of `ProbLog` were also built on top of `Prolog`, matching one of the goals of our work. Later work[3] introduces distribution clauses so that some continuous distributions like Gaussians can be represented. Other work extended these methods further while focusing on efficient exact inference algorithms like weighted model integration[4, 5]. Our work is most similar to [6], except that while they combine their forward reasoning with an importance sampler we use a particle cascade instead which can be more sample efficient.

3. Background

3.1. `microKanren`

`MICROKANREN`[7, 8] is a pure logic programming language embedded in Scheme. The language consists of a set of terms, a set of goal primitives, and two run forms that turn goal queries into a set of answers. The goal primitives consist of a fresh form for introducing logic variables, a unification primitive `==`, a conjunction combinator `conj`, a disjunction combinator `disj`, and ways to define and apply relations. Further forms are shown in detail in Figure 1.

Goal expressions represent a possibly backtracking search procedure. These goals all take as input some state and output a stream of possible output states. We run these goals by starting with an initial state that holds an empty substitution dictionary, and passing it into the top-level goal expression which then passes it recursively down to sub-expressions. Encountering `fresh` extends the lexical environment with a binding between a lexical variable and a logic variable; `conj` will apply the first goal to the state passed in, and then apply the second goal to the states associated with each resulting stream from the first goal and finally unify the results; `disj` will apply both goals to the same input state and concatenate the resulting streams; and `==` unifies its arguments in the context of the current state, discarding any streams which fail to unify.

To handle goal expressions that might diverge, the streams are expanded in an interleaving[9] fashion, giving each branch a chance to produce answers. This interleaving search is sound and complete[10].

```

<prog> ::= (run <number> ((<id>) <goal-expr>))

<goal-expr> ::= (disj <goal-expr> <goal-expr>)
| (conj <goal-expr> <goal-expr>)
| (fresh ((<id>) <goal-expr>))
| (== <term-expr> <term-expr>)
| (letrec-rel (((<id> (<id> ...) <goal-expr>) ...)
  <goal-expr>))
| (call-rel <lexical-var-ref> <term-expr> ...)
| (prim-rel-call <lexical-var-ref> <term-expr> ...)
| (delay <goal-expr>)

<term-expr> ::= (quote <datum>)
| <lexical-var-ref>
| (cons <term-expr> <term-expr>)
| <term>

<term> ::= <number>
| #f | #t
| <symbol>
| (<term> . <term>)
| <logic-var>

```

Figure 1: Grammar for MICROKANREN

3.2. Grammar and Definitions

To create PROBKANREN, we extend this grammar with distribution clauses such as `normal` and `bern`. Distribution clauses take as arguments the parameters of the distribution and a last argument representing a draw from that distribution. For example, `(normal 0 1 x)` means `x` represents a draw from the $\mathcal{N}(0, 1)$ distribution.

These are just another type of goal expression. We do not need to add a notion of probabilistic variables to the language grammar, as they can be treated as logic variables constrained in a particular way. The semantics of the language though does change from an answer set to a probability distribution on that answer set.

We follow the semantics of [11], this is a denotational semantics where each language form is associated with a measurable function, and an operational semantics of a sampler.

3.3. Probabilistic Programming

Probabilistic Programming Languages [12, 13] are a family of domain-specific languages for posing and efficiently solving probabilistic modelling problems. At their core, all have a way to sample and observe data under a probability distribution or Markov kernel.

There are many inference algorithms for probabilistic programming languages, but methods based on likelihood-weighting and Sequential Monte Carlo algorithms are the simplest to implement.

3.4. Sequential Monte Carlo

Sequential Monte Carlo[14](SMC) is an efficient online way to sample from probabilistic models, which is especially suited for state-space domains. If we imagine our probabilistic programs as straight-line programs with no control-flow we can imagine numbering every sample function f_1, f_2, \dots, f_n and every observe function g_1, g_2, \dots, g_n then our probability density over our latent variables $x_{0:T}$ and observed data $y_{0:T}$ can be defined as:

$$p(x_{0:T}, y_{0:T}) = p(x_0) \prod_{t=1}^T f_t(x_t | x_{t-1}) \prod_{t=0}^T g_t(y_t | x_t) \quad (1)$$

3.5. Sequential Importance Sampling

The simplest way to sample from our target distribution $p(x_{0:T} | y_{0:T})$ is to sample from a sequence of intermediary distributions $p(x_{0:t} | y_{0:t})$ where t goes from 1 to T . We do this by sampling a population of N particles $\{x_t^{(1)}, \dots, x_t^{(i)}, \dots, x_t^{(N)}\}$ from a proposal distribution $q(x_t | x_{0:t-1}, y_{0:t})$, which when combined with a set of importance weights $\{w_t^{(1)}, \dots, w_t^{(i)}, \dots, w_t^{(N)}\}$ lets us approximate each intermediary distribution.

Thanks to the recurrence relation:

$$p(x_{0:t} | y_{0:t}) = p(x_{0:t-1} | y_{0:t-1}) \frac{f_t(x_t | x_{t-1}) g_t(y_t | x_t)}{p(y_t | y_{0:t-1})} \quad (2)$$

we know in T rounds we compute the desired distribution as follows:

Initialise with:

$$x_0^{(i)} \sim p(x_0) \quad (3)$$

$$w_0^{(i)} = 1/N \quad (4)$$

Then for t from 1 to T

$$x_t^{(i)} \sim q(x_t | x_{0:t-1}^{(i)}) \quad (5)$$

$$w_t^{(i)} \propto w_{t-1}^{(i)} \frac{f_t(x_t^{(i)} | x_{t-1}^{(i)}) g_t(y_t | x_t^{(i)})}{q(x_t^{(i)} | x_{0:t-1}^{(i)})} \quad (6)$$

We get the best results if $q(x_t | x_{0:t-1})$ is equal to $p(x_t | x_{0:t-1}, y_{0:t})$.

3.6. Sequential Importance Resampling

Unfortunately, over time SIS is likely to lead to most of the importance weight in a small fraction of the particles, with the rest of the particles having negligible weight. This is usually called *weight degeneracy*. We address this sample efficiency issue in the standard way by replicating the particles with high weight and dropping the ones with low weight. This is called a resampling

step, and it happens after each round. Resampling effectively removes particles with low weight and duplicates particles with higher weight by sampling with replacement our existing particles.

$$\begin{aligned}
 a_t^{(i)} &\sim r(w_{t-1}^{(1:N)}) \\
 x_t^{(i)} &\sim q(x_t \mid x_{0:t-1}^{(i)}) \\
 w_t^{(i)} &\propto w_{t-1}^{(i)} \frac{f(x_t^{(i)} \mid x_{t-1}^{(i)})g(y_t \mid x_t^{(i)})}{q(x_t^{(i)} \mid x_{0:t-1}^{(i)})}
 \end{aligned}$$

For each particle i , resampling tells us which of the existing particles a_i will be its ancestor. We set $r(w) = \text{Categorical}(w)$, which is called multinomial resampling but there are other methods as well[15].

3.7. Particle Cascade

Unfortunately, SMC as defined here requires having access to all the particles at every step in the process. This conflicts with the functional nature of our implementation. Instead we make use of Particle Cascades [16], removing this barrier allowing every particle to be resampled asynchronously with the associated weights being relative to a global running average.

4. Proposed Method

We propose to extend MICROKANREN by turning the search into an SMC sampler. We accomplish this by augmenting each of the search streams with a set of particles. These particles represent the empirical distribution of that stream. Each particle has associated with it a substitution of all the logic and random variables as well as a weight that is proportional to the likelihood of the substitution.

We follow [3] and place the following restrictions on our distribution clauses and the random variables they specify.

Firstly, the arguments of distribution clauses must be grounded. Secondly, a random variable cannot unify with any arithmetic expression.

An initial set of particles is created from the probabilistic program when it is first run. When encountering primitives such as `normal` we first check if all the parameter terms are grounded and then if the last argument is fresh, we sample a value for it for each particle and then add this sampled value along with the associated logic variable to the substitution associated with that particle. If all the terms are ground, we treat the primitive like an observation statement and multiply the weights of each substitution with the likelihood of the observation.

As conjunctions (`conj`) are encountered, all particles in all output streams from the first goal become part of the initial states that are each passed to the second goal.

As disjunctions (`disj`) are encountered, we split evenly the number of particles allocated to each stream. Whenever we encounter a unification primitive, we run a resampling step. This helps to prune particles with low weight and replicate particles with high weight.

As an optimisation, we may create more particles during resampling based on a globally stored counter of the effective sample size of all particles across all streams.

This extension does not modify the search and streams are managed exactly as in `MICROKANREN`. An additional advantage, due to the `microKanren` search being complete, we are guaranteed to recover the true posterior as all paths of the search space will eventually be explored given enough particles.

5. Experiments

We validate that `PROBKANREN` is at least as expressive as other probabilistic logic programming languages by implementing the Friends who Smoke model.

Friends who Smoke is a probabilistic logic program which models the social nature of who smokes cigarettes. The model predicts that people who are friends with people who smoke are more likely to smoke. We replicate the example on https://dtai.cs.kuleuven.be/problog/tutorial/basic/05_smokers.html using 2000 particles and get an empirical distribution that seems to match up with the discrete distribution returned from `ProbLog`.

6. Conclusions

We made a simple-to-implement extension to `MICROKANREN` that lets us support probabilistic inference on both discrete and continuous distributions. The approach does not require modifying the underlying search algorithm or touching any of the backtracking code and comes with a theoretical guarantee that if the underlying search is complete then the probabilistic extension will recover the true posterior given enough particles.

References

- [1] T. Sato, Y. Kameya, Prism: a language for symbolic-statistical modeling, in: *IJCAI*, volume 97, Citeseer, 1997, pp. 1330–1339.
- [2] L. De Raedt, A. Kimmig, H. Toivonen, Problog: A probabilistic prolog and its application in link discovery., in: *IJCAI*, volume 7, Hyderabad, 2007, pp. 2462–2467.
- [3] B. Gutmann, M. Jaeger, L. De Raedt, Extending problog with continuous distributions, in: *International Conference on Inductive Logic Programming*, Springer, 2010, pp. 76–91.
- [4] M. A. Islam, C. Ramakrishnan, I. Ramakrishnan, Inference in probabilistic logic programs with continuous random variables, *Theory and Practice of Logic Programming* 12 (2012) 505–523.
- [5] V. Belle, A. Passerini, G. Van den Broeck, Probabilistic inference in hybrid domains by weighted model integration, in: *Twenty-Fourth International Joint Conference on Artificial Intelligence*, 2015.
- [6] B. Gutmann, I. Thon, A. Kimmig, M. Bruynooghe, L. De Raedt, The magic of logical inference in probabilistic programming, *Theory and Practice of Logic Programming* 11 (2011) 663–680.

- [7] J. Hemann, D. P. Friedman, W. E. Byrd, M. Might, A small embedding of logic programming with a simple complete search, in: Proceedings of the 12th Symposium on Dynamic Languages, DLS 2016, Association for Computing Machinery, 2016, p. 96–107. URL: <https://doi.org/10.1145/2989225.2989230>. doi:10.1145/2989225.2989230.
- [8] D. P. Friedman, W. E. Byrd, O. Kiselyov, J. Hemann, The Reasoned Schemer, 2nd ed., MIT Press, 2018.
- [9] O. Kiselyov, C.-c. Shan, D. P. Friedman, A. Sabry, Backtracking, interleaving, and terminating monad transformers: (functional pearl), ACM SIGPLAN Notices 40 (2005) 192–203.
- [10] D. Rozplochas, A. Vyatkin, D. Boulytchev, Certified semantics for minikanren, in: Proceedings of the 2019 miniKanren and Relational Programming Workshop, 2019, pp. 80–98.
- [11] S. Staton, H. Yang, F. Wood, C. Heunen, O. Kammar, Semantics for probabilistic programming: Higher-order functions, continuous distributions, and soft constraints, in: Proceedings of the 31st Annual ACM/IEEE Symposium on Logic in Computer Science, Association for Computing Machinery, New York, NY, USA, 2016, p. 525–534. URL: <https://doi.org/10.1145/2933575.2935313>. doi:10.1145/2933575.2935313.
- [12] F. Wood, J. W. Meent, V. Mansinghka, A new approach to probabilistic programming inference, in: Artificial Intelligence and Statistics, PMLR, 2014, pp. 1024–1032.
- [13] J. van de Meent, B. Paige, H. Yang, F. Wood, An introduction to probabilistic programming, CoRR abs/1809.10756 (2018).
- [14] N. Chopin, O. Papaspiliopoulos, et al., An Introduction to Sequential Monte Carlo, volume 4, Springer, 2020.
- [15] R. Douc, O. Cappé, Comparison of resampling schemes for particle filtering, in: ISPA 2005. Proceedings of the 4th International Symposium on Image and Signal Processing and Analysis, 2005., IEEE, 2005, pp. 64–69.
- [16] B. Paige, F. D. Wood, A. Doucet, Y. W. Teh, Asynchronous anytime sequential monte carlo, in: Advances in Neural Information Processing Systems, 2014, pp. 3410–3418.