Proposal for a Bachelor’s or Master’s thesis

Implementing an efficient Haskell library for probabilistic functions

Supervisors: Dr. Andreas Lochbihler
Professor: Prof. David Basin
Issue date:

Prerequisites

- Good skills in functional programming (e.g., from the FMFP course)

Summary

The goal of this project is to design and implement a Haskell library for probability distributions. Probability distributions are written as Haskell programs using API functions of the library. The library supports evaluating queries (sampling, probabilities of events, expectation, . . . ) over probability distributions. Distributions should be represented symbolically. The project should investigate whether query evaluation can leverage the symbolic representation to improve efficiency.

Background

Probabilistic choices in algorithms are typically implemented by sampling a value from a source of randomness (e.g., a random number generator). For example, the following Haskell program rolls two dice and returns the sum of their values.

```haskell
sumOfTwoDice = do
  x <- randomRIO (1, 6)
  y <- randomRIO (1, 6)
  return (x + y)
```

As the sampling is hard-coded into the program using randomRIO, this program cannot answer queries such as “What is the probability of the outcome 7?”. We can only approximate such a probability by statistical means, namely by running the program many times and observing the outputs. For example, suppose we roll a pair of dice a 100 times and we observe 15
times the sum 7. Then, we estimate the probability as $\frac{15}{100}$, but the exact probability is $\frac{1}{6}$.

Therefore, Haskell libraries have been developed that support computing with probability distributions [1, 3]. For example, the following Haskell program uses the Probabilistic Functional Programming library (PFP) [1].

```haskell
sumOfTwoDice = do
    x <- uniform [1..6]
    y <- uniform [1..6]
    return (x + y)
```

The PFP library can answer probability queries. In the example, `(== 7) ?? sumOfTwoDice` computes the desired probability. Internally, PFP represents probability distributions as a list of elementary events with their probability. That is, a single die roll is represented as the list `[(1, 0.167), (2, 0.167), (3, 0.167), (4, 0.167), (5, 0.167), (6, 0.167)]`. Thus, rolling two dice yields a list of 36 entries and three dice need 216 entries.

In our example, even if we are only interested in sampling one outcome, PFP computes the whole list of 36 entries. This can be improved, because the above program contains more structure, which is lost in PFP’s internal representation. Namely, it suffices to sample one die first and then the other, each of which only requires to look at 6 entries at most, i.e., 12 entries in total. In this work, a symbolic representation of programs shall be implemented and evaluated to retain the structure of the program.

For a start, the (generalised) algebraic datatype `Prob a` below can represent probability distributions symbolically. It is inspired by the probabilistic language of Ramsey and Pfef-

```haskell
data Prob a where
    Certainly :: a -> Prob a
    Choose :: Rational -> Prob a -> Prob a -> Prob a
    Weighted :: Prob a -> (a -> Prob b) -> Prob b
    Independent :: Prob a -> Prob b -> Prob (a, b)
```

A value of type `Prob a` represents a probabilistic experiment with outcomes of type `a`. In that view, `Certainly` represents a deterministic experiment and `Choose p l r` models the experiment that runs `l` with probability `p` and `r` with probability `1-p`. `Weighted p f` combines a family `f` of probability distributions weighted according to `p`. `Independent p q` runs the two experiments `p` and `q` independently and returns their results.

The different queries on probability distributions can then be implemented as functions that pattern-match on the constructors. For example, `sample p g` samples a value according to the distribution `p` using a random number generator `g`. Other queries like the probability of events, the support of the distribution, the expectation and variance can be implemented similarly [3].

```haskell
sample :: RandomGen g => Prob a -> g -> (a, g)
sample (Certainly a) g = (a, g)
sample (Choose p l r) g
    | numerator p <= n = sample l g
    | otherwise = sample r g
```

1 Monadic sequencing as used by Ramsey and Pfeffer (and PFP and other languages) cannot express independence, but applicative functors can. Therefore, `Prop a` also contains a constructor for independence.
The sum of two dice rolls can now be expressed as follows, but this is not as readable as before. Moreover, it is not obvious that this implementation actually runs faster.

die = Choose (1/2)  
    (Choose (1/3) (Certainly 1) (Choose (1/2) (Certainly 2) (Certainly 3)))  
    (Choose (1/3) (Certainly 4) (Choose (1/2) (Certainly 5) (Certainly 6)))

sumOfTwoDice = Weighted (Independent die die) ((x,y) -> Certainly (x+y))

Objectives

This project aims at designing and evaluating a Haskell library for probabilistic functions that represents probability distributions symbolically. To that end, one first must determine a set of primitive symbolic representations (such as Certainly, Choose, Weighted and Independent) and implement the query operations for them. The primitives should be chosen such that they support evaluating queries efficiently. Then, a library of higher-level operations such as monadic sequencing, uniform distributions, etc. should be expressed in terms of the primitives.

Finally, it should be evaluated how the symbolic representations impact the efficiency of the queries by comparing the new library with existing library implementations on standard examples. Criteria for efficiency can be the execution time of query evaluation (for one query or for multiple queries on one distribution) and the amount of randomness needed for the evaluation (for sampling).

Tasks

This project can be subdivided into the following tasks:

1. Read the relevant literature on probabilistic functional languages (e.g., [1, 2, 3, 4] and the references therein).
2. Study how independence can be exploited in evaluating queries and how independence can be expressed in existing languages.
3. Choose a set of primitives for symbolically representing probability distributions. Implement common probabilistic operations as library functions on top of them.
4. Implement common query operations such as sampling, probability of events, expectation and support.
5. Compare the evaluation times for queries in the symbolic representation and in other Haskell libraries. Test cases may be taken from the existing libraries or newly designed.
6. (optional) Evaluate how symbolic sampling affects the use of randomness in comparison with existing libraries.

7. (optional) Implement a query that computes the variance of a random variable over a distribution and evaluate the efficiency.

8. (optional) Compare the efficiency of sampling with hard-coded implementations of sampling as in the motivating example.

9. Write the final report and prepare the presentation.

Deliverables

The following deliverables are due at the end of the project:

**Final report** The final report should consist of an introduction; an overview over the existing libraries; one or more sections describing the symbolic representations and the motivation for the choice, the implementation, and the evaluation; and a conclusion. The report may be written in English or German.

**Haskell code and test data** Complete Haskell development that runs with a recent version of GHC and all evaluation data that is produced during the project.

**Presentation** At the end of the project, a presentation of 20 minutes must be given during an InfSec group seminar. It should give an overview and discuss the most important highlights of the work.

References


