Embedded Probabilistic Programming

Oleg Kiselyov
FNMOC
oleg@pobox.com

Chung-chieh Shan
Rutgers University
ccshan@rutgers.edu

Abstract

We propose to represent a probability distribution as a program in a *general-purpose* programming language rather than a special language built from scratch. This approach makes it easier for the probabilistic-reasoning and programming-language communities to share their work. To demonstrate that this representation is simple and efficient, we implement inference by variable elimination and importance sampling using the concepts of reflection, memoization, and continuations.

The *embedded* approach to probabilistic programming is to build a library that extends a general-purpose *host* language with probabilistic constructs. Seen from within the host language, these embedded constructs are ordinary functions with side effects. For example, the two most important constructs are `dist`, a function that maps a list of probability-value pairs to a randomly chosen value, and `fail`, a function that takes no input and never returns because it observes an impossible event. The following program, written in the general-purpose programming language OCaml, expresses the distribution over a pair of fair coin flips, conditional on at least one of them being `true`.

```
(1)  let coin_flip () = dist [ (0.5, V true); (0.5, V false) ] in
     let x = coin_flip () in
     let y = coin_flip () in
     if x || y then (x,y) else fail ()
```

The variables `x` and `y` above are host-language variables that represent random variables.

The embedded approach has several advantages over building a probabilistic language from scratch.

- It is easy to learn and implement, because host-language features are reused such as compilers, I/O libraries, database interfaces, timing facilities, data structures, and type checking.
- It can compile stochastic models to machine code. Deterministic parts of the models run as fast as in the host language, without the overhead imposed by non-embedded interpreters.
- It treats probability distributions as an abstract container data-type. This treatment makes it natural to describe distributions over distributions, which are useful for modeling multiagent interactions with imperfect information.

For the library not to just perform naïve rejection sampling, the host language needs certain features other than random-number generation [2]. We use the *delimited control operators* `shift` and `reset` in the host language OCaml [5].

Embedded probabilistic programming essentially builds a library of weighted nondeterminism that turns the host-language compiler into a probabilistic-language compiler, so any improvement in the host-language implementation results in more efficient inference. To illustrate how inference algorithms correspond to library implementation strategies and host-language concepts, we present two implementations: one performing exact inference by enumeration and variable elimination [1], and one performing approximate inference by importance sampling and *evidence pushing* [6]. The implementations are short (totaling 200 lines), expressive, and available online along with tests: `http://okmij.org/ftp/kakuritu/README.dr`.

Both implementations use delimited control operators to convert a stochastic program into a search tree of random choices. Each node in the tree is labeled with the probability mass remaining at that point for its descendants. Each leaf is either an observation failure or a successful outcome. This tree can be lazily explored in various ways; in other words, the inference algorithm and the stochastic program are coroutines, like an operating system and a user process.

Exact inference The simplest inference method is to traverse the tree (say in depth-first order), enumerate the successful outcomes at the leaves, and gather them into a probability table. This *reification* map from stochastic programs to probability tables performs exact inference by enumeration, so we call it `exact_reify`. For example, the OCaml program `exact_reify (fun () -> (1))`, in which '`fun () ->`' makes a function that takes no argument, computes the probability table below.

(2)
```
[(0.25, V (true,true)); (0.25, V (true,false)); (0.25, V (false,true))]
```

The functions `exact_reify` and `dist` are inverses of each other, in that a stochastic program E denotes the same distribution as `dist (exact_reify (fun () -> E))`, and a probability table T is same as `exact_reify (fun () -> dist T)`. These equivalences are a special case of the general inverse relationship between reification and *reflection* [4]. We use this relationship to achieve variable elimination with an explicit order: Whenever we have a stochastic function `f`, we can replace it by the stochastic function

(3)
```
let bucket = memo (fun x -> exact_reify (fun () -> f x)) in
fun x -> dist (bucket x)
```

where `memo` memoizes a function. This latter stochastic function can then be invoked multiple times in different branches of the search tree, without recomputing joins each time.

Approximate inference To perform importance sampling, we traverse the same search tree differently: we explore a few levels at its top in a breadth-first or most-probable-first manner. The traversal weeds out shallow observation failures and yields a list of successful outcomes found as well as open branches yet to be explored. Each successful outcome can be immediately registered as a sample, weighted by its probability. If there are any open branches, we randomly choose one, then traverse it as just described after discarding the other branches. The probability of choosing a branch is proportional to its probability mass. We also remember to scale the importance of all further samples by the total probability mass of the open branches among which we chose one.

Pfeffer [6] introduces this importance sampling technique as *evidence pushing*, because it reduces nondeterminism in the search by pushing observed evidence towards their random cause in a stochastic program. We achieve evidence pushing by accessing the host language's call stack. This access is in fact part of the initial conversion from stochastic programs to search trees using delimited control operators. In other words, evidence pushing is a form of *continuation passing* [3].

References

[1] Dechter, Rina. 1998. Bucket elimination: A unifying framework for probabilistic inference. In *Learning and inference in graphical models*, ed. Michael I. Jordan. Dordrecht: Kluwer. Paperback: *Learning in Graphical Models*, MIT Press.

[2] Filinski, Andrzej. 1994. Representing monads. In *POPL '94: Conference record of the annual ACM symposium on principles of programming languages*, 446–457. New York: ACM Press.

[3] Fischer, Michael J. 1972. Lambda-calculus schemata. In *Proceedings of ACM conference on proving assertions about programs*, vol. 7(1) of *ACM SIGPLAN Notices*, 104–109. New York: ACM Press. Also ACM SIGACT News 14.

[4] Friedman, Daniel P., and Mitchell Wand. 1984. Reification: Reflection without metaphysics. In *Proceedings of the 1984 ACM symposium on Lisp and functional programming*, 348–355. New York: ACM Press.

[5] Kiselyov, Oleg. 2006. Native delimited continuations in (byte-code) OCaml. `http://okmij.org/ftp/Computation/Continuations.html#caml-shift`.

[6] Pfeffer, Avi. 2007. A general importance sampling algorithm for probabilistic programs. Tech. Rep. TR-12-07, Harvard University.