Probabilistic programming using first-class stores and first-class continuations

Oleg Kiselyov
FNMOC
oleg@pobox.com

Chung-chieh Shan
Rutgers University
ccshan@cs.rutgers.edu

Abstract

Probabilistic inference is a popular way to deal with uncertainty in many areas of science and engineering. Following the declarative approach of expressing *probabilistic models* and *inference algorithms* as separate, reusable modules, we built a *probabilistic programming language* as an OCaml library and applied it to several problems. We describe this embedded domain-specific language using aircraft tracking as an example. We focus on our use of *first-class stores* and first-class continuations to perform faster inference (with *local laziness* and *full-speed determinism*) and express more models (with *stochastic memoization* and *nested inference*).

Description

Uncertainty is a pressing concern in many areas of science and engineering, such as computational linguistics, biology, and economics. Probabilistic inference is a popular way to deal with uncertainty. Conceptually, a program for probabilistic inference combines a *probabilistic model* with an *inference algorithm*, just as a search program combines a search space with a search strategy. For example, it is typical to compute the expected value of a random variable over a conditional distribution. That is analogous to querying a logic program and tallying the solutions.

To make programs for probabilistic inference easier to develop and maintain, we and many other practitioners want to write them declaratively—that is, by expressing models and inference as separate, reusable modules. To this end, we built a *probabilistic programming language* HANSEI as an OCaml library and applied it to several problems (Kiselyov and Shan 2009a,b). Figure 1 shows a simple piece of code that uses our library. In this talk, we describe this embedded domain-specific language. We focus on how we implemented it using facilities that OCaml should provide. In particular, we explain for the first time our crucial use of *first-class stores* (Morrisett 1993) and first-class continuations to implement laziness efficiently while models express *nested* nondeterminism.

As a running example, we use Milch et al.'s model of radar blips for aircraft tracking (2007). In this model, a bitmap radar screen monitors a region in the air with an unknown number of planes that move and turn randomly. At each time step, each plane causes a blip on the screen with a certain probability. Due to limited resolution, several planes may result in a single blip. Blips may also be caused by noise. The first problem is to estimate the number of planes from blips and their absence in consecutive radar screenshots. A further step is to identify the planes as they appear and disappear. For example, when our modular combination of model and inference is presented with the 3 consecutive observations depicted in Figure 2, it estimates that there are 3 planes with probability 0.987.

All inference algorithms rely on representing and exploring multiple hypotheses. We represent hypotheses as first-class continuations and manage their exploration by reifying the state of the probabilistic model into a tree of possible execution paths (Filinski 1994). This technique improves performance because deterministic parts of the model run at the full speed of compiled code. It also improves expressivity by letting the model invoke existing libraries, including inference itself. It is by nesting inference in the model in this way that we implemented the *particle filter* that computed the probability estimates in Figure 2. Nested inference also allows one agent to reason about another's reasoning, for instance to plan a flying formation likely to produce a misleading radar screen.

To stay tractable, inference must refine one hypothesis into multiple hypotheses for separate consideration not when a random choice is made by the model but when it is observed with a query. For example, the aircraft tracking program should not enumerate all possible sets of plane locations right away, but rather determine the locations gradually as it iterates over radar-screen observations. This crucial optimization amounts to lazy evaluation, but we cannot use OCaml's (or ML's or Haskell's) built-in lazy evaluation or mutable state, because the random choices memoized within one hypothesis must not pollute the inference about other hypotheses. Thus, we need to associate each execution path with a *first-class store* (Morrisett 1993) in order to combine laziness and nondeterminism (Fischer et al. 2009). A first-class store is a region of mutable objects that can be captured and restored together. Using first-class stores, we can also perform *stochastic memoization* in models to express *nonparametric* distributions succinctly.

We implement first-class stores as immutable maps, and associate one with each delimited continuation. In the presence of nested inference, accessing a memo cell requires traversing the chain of currently active delimited continuations to find the first-class store that contains the cell. We found this simulation of delimited dynamic binding (Kiselyov et al. 2006) to be more efficient in practice than the previous translation from dynamic bindings to control delimiters. Probabilistic programming thus provides newfound motivation for general-purpose programming languages, especially their garbage collectors, to support first-class stores.

References

Filinski, Andrzej. 1994. Representing monads. In *POPL*, 446–457.

Fischer, Sebastian, Oleg Kiselyov, and Chung-chieh Shan. 2009. Purely functional lazy non-deterministic programming. In *ICFP*, 11–22.

Kiselyov, Oleg, and Chung-chieh Shan. 2009a. Embedded probabilistic programming. In *Domain-specific languages*, 360–384. LNCS 5658.

———. 2009b. Monolingual probabilistic programming using generalized coroutines. In *Uncertainty in artificial intelligence*, 285–292.

——— and Amr Sabry. 2006. Delimited dynamic binding. In *ICFP*, 26–37.

Milch, Brian, Bhaskara Marthi, Stuart Russell, David Sontag, Daniel L. Ong, and Andrey Kolobov. 2007. BLOG: Probabilistic models with unknown objects. In *Introduction to statistical relational learning*, ed. Lise Getoor and Ben Taskar, chap. 13, 373–398. Cambridge: MIT Press.

Morrisett, J. Gregory. 1993. Refining first-class stores. In *Proceedings of the ACM SIGPLAN workshop on state in programming languages*.

```
type gender = Female | Male
normalize (exact_reify (fun () ->
  let kid = memo (fun n -> dist [(0.5, Female); (0.5, Male)]) in
  if kid 1 = Male && kid 2 = Male then fail () else kid 1))
```

Figure 1. Using our OCaml library to solve one interpretation of the following puzzle: I have exactly two kids; at least one of them is a girl; what is the probability that my older kid is a girl? (The answer is $2/3$.) Our library provides the functions `normalize` to normalize a probability table, `exact_reify` to perform exact inference, `memo` to memoize a function, `dist` to make a random choice, and `fail` to reject a hypothesis inconsistent with observation.

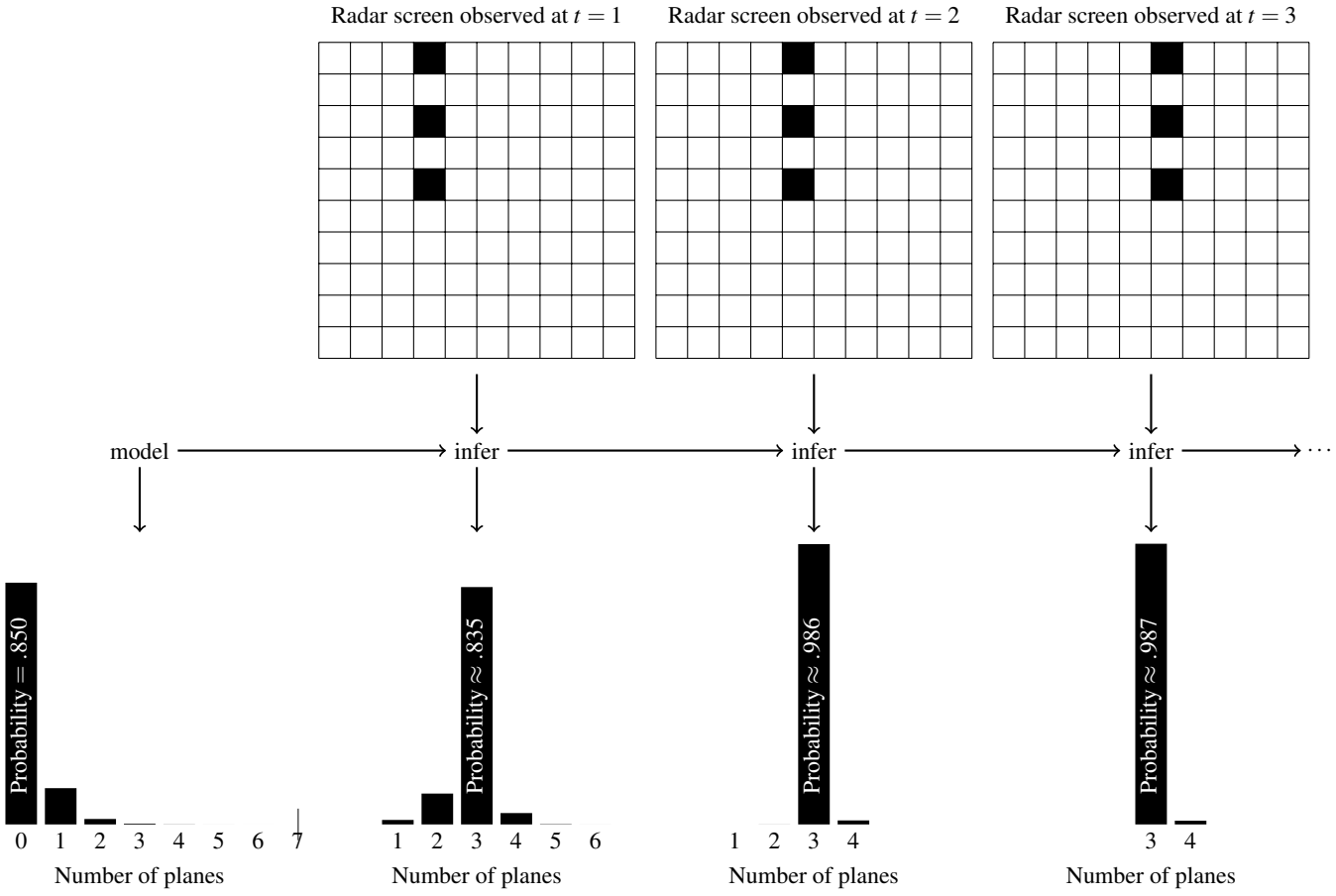

Figure 2. A sample run of aircraft tracking using radar blips. At the top are consecutively observed radar screens. At the bottom are consecutive distributions of the number of planes, first stipulated by the model then gradually computed by our approximate inference.