Strongly typed tracing of probabilistic programs

Adam Ścibior
University of Cambridge and MPI Tübingen
ams240@cam.ac.uk

Michael Thomas
Independent Researcher
mthomas180@gmail.com

Introduction

A commonly encountered concept in probabilistic programming literature is that of a trace, which is a record of all the random variables sampled during the program execution. Traces serve two roles, namely providing observed values for some of the variables on which to perform conditioning and facilitating implementations of certain inference algorithms. Typical examples of such inference algorithms are importance sampling, where a guide program with a trace matching the original program is provided as a proposal distribution, and MCMC where a part of the trace is updated at each iteration.

In this work we propose a method for constructing traces in PPLs in a statically typed fashion using lenses. Our approach statically ensures absence of certain bugs, such as misspelling a name of a random variable, and can be used to statically enforce certain properties of the program, such as the restriction of having a fixed number of random variables of fixed types, used in Stan programs. We focus on a situation where the user provides explicit names for the random variables. This is in contrast to the situation where the random variables are not explicitly given unique names and these have to be assigned automatically using a heuristic, such as in the work of Wingate et al. [6].

How traces are constructed

The trace is a record of all the random variables sampled in the program execution. PPLs extending dynamic languages are not concerned with statically determining any properties of the trace and they typically represent it as a dictionary mapping from strings representing variable names to records describing distributions and values sampled from them. We will use Pyro [1] as an example of such a language although many others use similar approaches to tracing. In statically typed PPLs this construction is unsatisfactory, as we would like to be able to statically verify that the right variables of the right types are recorded in the right places in the trace.

Stand-alone PPLs can force the users to specify programs in a way that makes it easy to determine what should be included in the trace. For example Stan [2] requires that the user declares all the random variables for the whole program in seperate `data` and `parameters` blocks. This approach works well in Stan's setting but it is less suitable for PPLs that extend existing languages. Dalibard [3] uses a similar approach in a PPL based on C++ but it places significant constraints on how the program can be structured.

Lenses

Lenses are a well-established functional programming tool for extracting and updating values in immutable data structures. For example, consider the following record.

```
type person = {name : string; surname : string}
```

A lens for accessing the field `name` would consist of a getter and a setter.

```
type ('r,'v) lens =
  { get : 'r -> 'v; set : 'v -> 'r -> 'r }

let name_ : (person,string) lens = {
  get = (fun p -> p.name);
  set = (fun n p -> { p with name = n })
}
```

By convention lenses associated with a record field have the same name followed by an underscore. Lenses compose naturally as functions allowing direct access to fields in nested records.

Strongly typed tracing

We propose to use strongly typed records instead of dictionaries for tracing in order to take advantage of static typing. To illustrate this idea we introduce a small PPL called Roc extending OCaml where models have type `(trace, a) model`, where `trace` is the type of the trace and `a` is the type of the return value. The `model` type conforms to the standard monadic interface with respect to `a`.

In Roc the trace is a model-specific data type that holds values for all the random variables. It can be any data structure, although typically it is a flat record. The only important requirement is that it comes with a lens for every random variable in the trace, which recovers a record associated with the particular variable.

The trace is a collection of special records, one for each variable. These records are of type `'a obs`, which is a special algebraic data type for storing information about the random variable.

```
type 'a obs =
  | Empty
  | Observed of 'a
  | Sampled  of 'a dist * 'a
```

`obs` encodes the information we have about the specific random variable. It can either be no information, a value that was observed in the model, or the information that the random variable was previously sampled, obtaining a given

```
def model():

  rain = pyro.sample('rain',
          dist.Bernoulli(0.2))
  if rain.item() == 1.0:
    p = 0.7
  else:
    p = 0.1
  wet = pyro.sample('wet',
          dist.Bernoulli(p))

  return rain
```

(a) Pyro

```
data {
  int<lower=0,upper=1> wet;
}
parameters {
  int<lower=0,upper=1> rain
}
model {
  real<lower=0,upper=1> p;
  rain ~ bernoulli(0.2)
  p = rain ? 0.7 : 0.1
  wet ~ bernoulli(p)
}
```

(b) Stan

```
type trace = {
  rain : bool obs;
  wet  : bool obs;
}

let model () : (trace, bool) model =
  sampleAs rain_ (bernoulli 0.2)
    >>= fun rain ->
  let p = if rain then 0.7 else 0.1 in
  sampleAs wet_ (bernoulli p)
    >>= fun wet ->
  return rain
```

(c) Roc

Figure 1. The classic sprinkler model implemented in several PPLs. Pyro does not define a model-specific trace data structure, instead using a generic one with dynamically chosen fields, eschewing static guarantees. In Stan the user needs to define all the random variables and their types in separate blocks and the contents of these blocks defines the trace. Our PPL Roc uses a user-defined model-specific trace data structure and employs lenses to establish correspondence between the model and the trace.

value from a given distribution. The last possibility is useful for various inference algorithms.

The random variables are associated with the trace through a special function `sampleAs` which performs sampling from a given distribution and additionally records the sampling result in the trace.

```
let sampleAs {get;set} d : ('trace, 'a) model =
  getTrace >>= fun tr ->
  match get tr with
  | Empty -> sampleAndRecord d set
  | Observed y ->
      factor (pdf d y) >>= fun _ ->
      return y
  | Sampled (y, q) ->
      factor (pdf d y / pdf q y) >>= fun _ ->
      return y
```

If the trace contains `Empty` for the given variable then we sample its value from the prior and record it as `Sampled`. If it is `Observed` then we include its likelihood as a factor in the model and return the oberved value. Finally, if the random variable was already `Sampled` then we use the previously obtained value and introduce a factor equal to the ratio of densities of the current distribution and the one that was used for sampling the recorded value. Note that this is only meant to happen if the value was previously sampled from a proposal distribution in a guide program. It is an error to sample the same random variable twice within the model.

In the example shown in Figure 1 the names of lenses correspond to the names of program variables. This is by no means necessary and the two can be completely different. However, they often will be the same so we would find ourselves unnecessarily typing the name twice. This boilerplate can be avoided with a simple macro, similarly to what Turing [4] does with untyped traces.

Extensions

The design shown above allows us to statically impose Stan-like restrictions on PPLs extending statically typed functional languages. However, we can easily extend our approach to add more features. For example, Stan programs do not compose easily, since the `parameters` block is global to the whole program. Fortunately, it is easy to define nested trace data structures that hold smaller traces. We can then access the smaller traces through suitably defined lenses.

The restriction that the numbers and types of random variables in the program are known statically can be very useful for inference but it is sometimes too restrictive for modelling. We can partially relax this requirement while retaining as much of the static guarantees as possible. For example, we define a data constructor `multiple`, similar to `obs`, which holds values for some statically unknown number of random variables. This can be used to type traces for programs where a submodel is executed a certain number of times, that number itself being a random variable.

Future work

Our design provides certain static guarantees, but there are additional properties of probabilistic programs that we would like to ensure. In particular, we might want to enforce that each random variable is sampled at most once or exactly once. This restriction is not enforced in Stan, and violating it results in counterintuitive behaviour which Hur et al. [5] labelled as incorrect, even though it matches the semantics of Stan.

Another direction is reflecting more of a program's structure in the trace. In particular Pyro encodes certain information about conditional independence which is useful for performing inference.

References

[1] Eli Bingham, Jonathan P. Chen, Martin Jankowiak, Fritz Obermeyer, Neeraj Pradhan, Theofanis Karaletsos, Rohit Singh, Paul Szerlip, Paul Horsfall, and Noah D. Goodman. 2018. Pyro: Deep Universal Probabilistic Programming. https://arxiv.org/pdf/1810.09538.pdf. (2018).

[2] Bob Carpenter, Andrew Gelman, Matthew D. Hoffman, Daniel Lee, Ben Goodrich, Michael Betancourt, Marcus Brubaker, Jiqiang Guo, Peter Li, and Allen Riddell. 2017. Stan: A probabilistic programming language. *Journal of Statistical Software* 76 (2017).

[3] Valentin Dalibard. 2017. *A framework to build bespoke auto-tuners with structured Bayesian optimisation.* Ph.D. Dissertation. University of Cambridge.

[4] Hong Ge, Kai Xu, and Zoubin Ghahramani. 2018. Turing: a language for flexible probabilistic inference. In *AISTATS*.

[5] Chung-Kil Hur, Aditya Nori, Sriram K. Rajamani, and Selva Samuel. 2015. A Provably Correct Sampler for Probabilistic Programs. In *FSTTCS*.

[6] David Wingate, Andreas Stuhlmüller, and Noah Goodman. 2011. Lightweight Implementations of Probabilistic Programming Languages Via Transformational Compilation. In *AISTATS*.