Embedded Probabilistic Programming

Delimited continuations, OCaml

Oleg Kiselyov[1] Chung-chieh Shan[2]

[1]FNMOC

[2]Rutgers University

DSL 2009

Introduction

Will to represent probability distributions so that

- humans an develop and understand them easily
- computers can perform inference and sampling efficiently

It helps to embed a language of probability distributions in a host language.

Problem

In the embedding setting, the linguistic mismatch degrades efficiency, concision and maintainability of deterministic parts of a model.

Example

Random integers are distinct from regular integers and cannot be added using the addition operation of the host language

Building a standalone language for probability distributions can eliminate the notational overhead, but this language cannot rely on the host language and its infrastructure.

Solution

Combine the advantages of embedded and standalone probabilistic languages $\longrightarrow$ embedding in a very shallow way.
Here, the host language is OCaml.

- We can express probabilistic models using OCaml's built-in operations, control constructs, data structures
- We can use OCaml's type system to discover mistakes earlier
- We can use OCaml's bytecode compiler to perform inference faster

Example

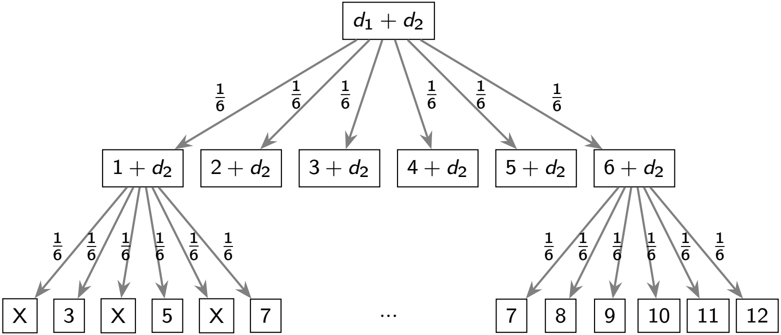

Figure: Tree for sum of two dice given one is even

Module signature

```
type prob = float
module type ProbSig = sig
  type 'a pm
  type ('a,'b) arr
  val n    : int -> int pm
  val dist : (prob * 'a) list -> 'a pm
  val sum  : int pm -> int pm -> int pm
  ...
  val is_null : int pm -> bool pm
  val dis  : bool pm -> bool pm -> bool pm
  val if_  : bool pm -> (unit -> 'a pm) -> (unit -> 'a pm) ->
  val lam  : ('a pm -> 'b pm) -> ('a,'b) arr pm
  val app  : ('a,'b) arr pm -> ('a pm -> 'b pm)
end
```

The model

```
module Dice(S: ProbSig) = struct
  open S
    let is_even e = is_null (modulo e (n 2))

    let let_ e f = app (lam f) e

    let dice_model () =
      let p = 1./.6. in
      let_ (dist [(p, 1); (p, 2); (p, 3); (p, 4); (p, 5); (p,
      let_ (dist [(p, 1); (p, 2); (p, 3); (p, 4); (p, 5); (p,
      let_ (sum die1 die2) (fun sum_dice ->
      if_ (dis (is_even die1) (is_even die2))
   (fun () -> sum_dice) (fun () -> dist []))))
end
```

First approach: monadic

```
type 'a vc = V of 'a | C of (unit -> 'a pV)
and  'a pV = (prob * 'a vc) list

let pv_unit (x : 'a) : 'a pV = [(1.0, V x)]
let rec pv_bind (m : 'a pV) (f : 'a -> 'b pV) : 'b pV =
  List.map (function
      | (p, V x) -> (p, C (fun () -> f x))
      | (p, C t) -> (p, C (fun () -> pv_bind (t ()) f)))
    m
```

First approach: monadic

```
module SearchTree = struct
  type 'a pm = 'a pV
  type ('a,'b) arr = 'a -> 'b pV

  let n = pv_unit
  let dist ch = List.map (fun (p,v) -> (p, V v)) ch
  let sum e1 e2 = pv_bind e1 (fun v1 ->
                    pv_bind e2 (fun v2 -> pv_unit (v1 + v2)))
  ...
  let dis e1 e2 = pv_bind e1 (fun v1 ->
                    if v1 then (pv_unit true) else e2)
  let if_ b e1 e2 = pv_bind b (fun t ->
                      if t then e1 () else e2 ())
  let lam e = pv_unit (fun x -> e (pv_unit x))
  let app e1 e2 = pv_bind e1 (pv_bind e2)
end
```

Second approach: CPS

```
module CPS = struct
  type 'a pm = ('a -> int pV) -> int pV
  type ('a,'b) arr = 'a -> ('b -> int pV) -> int pV

  let n x = fun k -> k x
  let dist ch = fun k ->
    List.map (function (p,v) -> (p, C (fun () -> k v))) ch
  let sum e1 e2 = fun k ->
    e1 (fun v1 -> e2 (fun v2 -> k (v1 + v2)))
    ...
  let if_ et e1 e2 = fun k -> et (fun t ->
                                  if t then e1 () k else e2 ()
  let lam e = fun k -> k (fun x -> e (fun k -> k x))
  let app e1 e2 = fun k -> e1 (fun f -> e2 (fun x -> f x k))
  let reify0 m = m pv_unit
end
```

Final: direct style with implicit continuation

```
   reset (fun () -> 41 + shift (fun k -> k 2))
-> reset (fun () -> (fun k -> k 2)
          (fun x -> reset (fun () -> 41 + x)))
-> reset (fun () -> reset (fun () -> 41 + 2))
-> 43
```

Final: direct style with implicit continuation

```
module Direct = struct
  type 'a pm = 'a
  type ('a,'b) arr = 'a -> 'b

  let n x = x
  let dist ch = shift (fun k ->
                  List.map (function (p,v)
                  -> (p, C (fun () -> k v))) ch)

  let sum e1 e2 = e1 + e2
  ...
  let dis e1 e2 = e1 || e2
  let if_ et e1 e2 = if et then e1 () else e2 ()
  let lam e = e
  let app e1 e2 = e1 e2
  let reify0 m = reset (fun () -> pv_unit (m ()))
end
```

Alterntive syntax for direct style

```
open Direct
let dice_model () =
  let p = 1./.6. in
  let die1 = dist [(p, 1); (p, 2); (p, 3);
                   (p, 4); (p, 5); (p, 6)] in
  let die2 = dist [(p, 1); (p, 2); (p, 3);
                   (p, 4); (p, 5); (p, 6)] in
  if (die1 mod 2 = 0) || (die2 mod 2 = 0) then die1 + die2
  else dist []
```