```
let nearly_one = 1.0 -. 1e-7;;          (* For robust comparison with 1.0 *)

(* Explore but do not flatten the tree:
   perform exact inference to the given depth
   We still pick out all the produced answers and note the failures. *)
let shallow_explore maxdepth (choices : 'a pV) : 'a pV =
  let add_answer pcontrib v mp = PMap.insert_with (+.) v pcontrib mp in
  let rec loop pc depth ans acc = function
    | [] -> (ans,acc)
    | (p,V v)::rest -> loop pc depth (add_answer (p *. pc) v ans) acc rest
    | c::rest when depth >= maxdepth -> loop pc depth ans (c::acc) rest
    | (p,C t)::rest ->
          let (ans,ch) = loop (pc *. p) (succ depth) ans [] (t ()) in
          let ptotal = List.fold_left (fun pa (p,_) -> pa +. p) 0.0 ch in
          let acc =
            if ptotal = 0.0 then acc
            else if ptotal < nearly_one then
             (p *. ptotal, let ch = List.map (fun (p,x) -> (p /. ptotal,x)) ch
                                 in C (fun () -> ch))::acc
            else (p, C (fun () -> ch))::acc in
          loop pc depth ans acc rest
  in
  let (ans,susp) = loop 1.0 0 PMap.empty [] choices
  in PMap.foldi (fun v p a -> (p,V v)::a) ans susp;;


(* Sample a distribution with a look-ahead exploration *)
(* A single sample can give us more than one data point: if one of
   the choices is a definite value, we note it right away, with
   its weight. The rest of the choices will be re-scaled automatically.
*)
(* Given a sampler, a function 'seed->'seed, run it a certain number
   of times and return the resulting seed and the number of runs
*)
type sample_runner =
        {sample_runner : 'seed. 'seed -> ('seed -> 'seed) -> 'seed * int};;

let sample_dist (selector : 'a pV selector) (sample_runner : sample_runner)
    ch : 'a pV =
  let look_ahead pcontrib (ans,acc) = function (* explore the branch a bit *)
    | (p,V v) -> (PMap.insert_with (+.) v (p *. pcontrib) ans, acc)
    | (p,C t) -> begin
        match t () with
        | [] -> (ans,acc)
        | [(p1,V v)] ->
            (PMap.insert_with (+.) v (p *. p1 *. pcontrib) ans, acc)
        | ch ->
            let ptotal = List.fold_left (fun pa (p,_) -> pa +. p) 0.0 ch in
            (ans,
              if ptotal < nearly_one then
                (p *. ptotal, List.map (fun (p,x) -> (p /. ptotal,x)) ch)::acc
              else (p, ch)::acc)
    end in
  let rec loop pcontrib ans = function
    | [(p,V v)]  -> PMap.insert_with (+.) v (p *. pcontrib) ans
    | []         -> ans
    | [(p,C th)] -> loop (p *. pcontrib) ans (th ())
    | ch ->                     (* choosing one thread randomly *)
        begin
        match List.fold_left (look_ahead pcontrib) (ans,[]) ch with
        | (ans,[]) -> ans
        | (ans,cch) ->
        let (ptotal,th) = selector cch in
        loop (pcontrib *. ptotal) ans th end in
  let toploop pcontrib ans cch =       (* cch are already pre-explored *)
    let (ptotal,th) = selector cch in
    loop (pcontrib *. ptotal) ans th in
  let driver pcontrib vals cch =
    let (ans,nsamples) =
```



```
      sample_runner.sample_runner PMap.empty
        (fun ans -> toploop pcontrib ans cch) in
    let ns = float_of_int nsamples in
    let ans = PMap.foldi
                 (fun v p ans ->
                   PMap.insert_with (+.) v (ns *. p) ans) vals ans in
    printf "sample_importance: done %d worlds\n" nsamples;
    PMap.foldi (fun v p a -> (p /. ns,V v)::a) ans [] in
  let rec make_threads pcontrib ans ch =  (* pre-explore initial threads *)
    match List.fold_left (look_ahead pcontrib) (ans,[]) ch with
    | (ans,[]) -> (* pre-exploration solved the problem *)
        PMap.foldi (fun v p a -> (p,V v)::a) ans []
    | (ans,[(p,ch)]) -> (* only one choice, make more *)
        make_threads (pcontrib *. p) ans ch
          (* List.rev is for literal compatibility with an earlier version *)
    | (ans,cch) -> driver pcontrib ans (List.rev cch)
  in
  make_threads 1.0 PMap.empty ch
;;


type pReq = Done |  Choice of (unit -> pReq) cdist

let pp = new_prompt ();;

(* We often use mutable variables as 'communication channel', to appease
   the type-checker. The variable stores the 'option' value --
   most of the time, None. One function writes a Some x value,
   and another function almost immediately reads the value -- exactly
   once. The protocol of using such a variable is a sequence of
   one write almost immediately followed by one read.
   We use the following helpers to access our 'communication channel'.
*)
let from_option = function Some x -> x | None -> failwith "fromoption";;

let read_answer r = let v = from_option !r in r := None; v (* for safety *)

let reify0 (thunk : unit -> 'a) : 'a pV =
  let answer = ref None in
  let rec interp = function
    | Done -> [(1.0, V (read_answer answer))] (* deterministic value *)
    | Choice ch -> List.map (fun (p,th) -> (p, C (fun () -> interp (th ()))))
                 ch
  in
  let mem = !thread_local in
  let v = push_prompt pp (fun () ->
         thread_local := Memory.newm; answer := Some (thunk ()); Done) in
  thread_local := mem;
  interp v;;


(* Two basic library functions for probabilistic programming *)

(* We make it appear as if capturing the continuation also captures
   the dynamic environment, thread_local. Shift implicitly
   wraps the captured continuation into a prompt; and so we
   add mem_prompt: whenever control is delimited, so should be 'memory'.
*)
let dist (choices : 'a dist) : 'a  =
  let curr_mem = !thread_local in
  shift0 pp (fun k ->
    Choice
       (List.map (fun (p,v) ->
         (p, (fun () ->
                  let mem = !thread_local in
                  let () = thread_local := curr_mem in
                  let v = k v in
                  thread_local := mem; v))) choices))
;;
```

```
let fail () = abort pp (Choice []);;

(* The ‘inverse’ of reify: reflect a search tree into a program
   denoting the same distribution.
*)

let reflect (tree : ’a pV) : ’a  =
  let curr_mem = !thread_local in
  let rec make_choices k pv =
    Choice (List.map (
            function (p,V x) -> (p, fun () -> k x)
                  |  (p,C x) -> (p, fun () -> make_choices k (x ()))) pv) in
  shift0 pp (fun k ->
    make_choices (fun x ->
      let mem = !thread_local in
      let () = thread_local := curr_mem in
      let v = k x in
      thread_local := mem; v)
     tree);;

(* Selectors, used by approximate inference procedures *)

(* Random selection from a list of choices, using system randomness *)
let random_selector randomseed : ’a selector =
  let () = Random.init randomseed in
  let rec selection r ptotal pcum = function
      | [] -> failwith "Choice selection: can’t happen"
      | ((p,th)::rest) ->
          let pcum = pcum +. p in
          if r < pcum then (ptotal,th)
          else selection r ptotal pcum rest
  in fun choices ->
    let ptotal = List.fold_left (fun pa (p,_) -> pa +. p) 0.0 choices in
    let r = Random.float ptotal in     (* 0<=r<ptotal *)
    selection r ptotal 0.0 choices
;;

(* A selector from a list of choices relying on the non−determinism
   supported by the parent reifier.
*)
let dist_selector ch =
  let ptotal = List.fold_left (fun pa (p,_) -> pa +. p) 0.0 ch in
  (ptotal, dist (List.map (fun (p,v) -> (p /. ptotal, v)) ch))
;;


let sample_importance selector nsamples (thunk : unit -> ’a) : ’a pV =
  sample_dist selector
    {sample_runner =
     fun z th ->
      let rec loop z = function 0 -> (z,nsamples) | n -> loop (th z) (pred n)
      in loop z nsamples}
    (shallow_explore 3 (reify0 thunk));;
```