# Embedded Probabilistic Programming*

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

[1] FNMOC
oleg@pobox.com
[2] Rutgers University
ccshan@rutgers.edu

**Abstract.** Two general techniques for implementing a domain-specific language (DSL) with less overhead are the *finally-tagless* embedding of object programs and the *direct-style* representation of side effects. We use these techniques to build a DSL for *probabilistic programming*, for expressing countable probabilistic models and performing exact inference and importance sampling on them. Our language is embedded as an ordinary OCaml library and represents probability distributions as ordinary OCaml programs. We use delimited continuations to reify probabilistic programs as lazy search trees, which inference algorithms may traverse without imposing any interpretive overhead on deterministic parts of a model. We thus take advantage of the existing OCaml implementation to achieve competitive performance and ease of use. Inference algorithms can easily be embedded in probabilistic programs themselves.

## 1 Introduction

In many fields of science and engineering, including artificial intelligence (AI), cryptography, economics, and biology, *probabilistic* (or *stochastic*) models are a popular way to account for and deal with uncertainty. The uncertainty may be inherent in the domain being modeled, induced by our imperfect knowledge and observation of the domain, or introduced to simplify a problem that is in principle deterministic. For example, all three kinds of uncertainty enter models in our area of interest, natural-language dialogue: many words and phrases are ambiguous, so interlocutors do not know exactly each other's communicative intentions, and it may be expedient to lump possible intentions into a finite number of equivalence classes.

Whichever kind of uncertainty they are used to model, probabilities are real numbers that can be regarded as weights on nondeterministic choices. A canonical example of a probabilistic model is that of a lawn whose grass may be wet because it rained, because the sprinkler was on, or for some other reason. Mathematically speaking, we have three Boolean variables `rain`, `sprinkler`, and `grass_is_wet`, and a probability distribution defined by

* Thanks to Olivier Danvy, Avi Pfeffer, and the anonymous reviewers. Thanks also to Aarhus University for hosting the second author in the fall of 2008.

W.M. Taha (Ed.): DSL 2009, LNCS 5658, pp. 360–384, 2009.

$$\Pr(\texttt{rain}) = 0.3,$$
$$\Pr(\texttt{sprinkler}) = 0.5,$$
$$\Pr(\texttt{grass\_is\_wet} \mid \texttt{rain} = r \wedge \texttt{sprinkler} = s) = 1 - n(0.1, r) \cdot n(0.2, s) \cdot 0.9.$$

We use the 'noisy not' function $n$ above to express possibilities such as a light rain that dries immediately and a sprinkler with low water pressure:

$$n(p, \text{true}) = p, \qquad n(p, \text{false}) = 1.$$

By the definition of conditional probability ($\Pr(A|B) = \Pr(A \wedge B)/\Pr(B)$), the product of these distributions is the *joint* distribution of all three variables. Unlike this simple example, many probabilistic models in production use contain deterministic parts (laws of motion, for example) alongside random parts [14, 32].

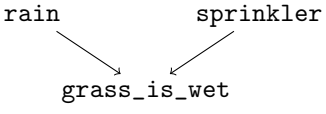
Given a probability distribution such as the lawn model just defined, we want to perform *inference*, which means to compute the probability distribution (called a *marginal* distribution) over some random variables in the model that we care about. For example, we might want to compute $\Pr(\texttt{grass\_is\_wet})$. We might also want to *sample* from the distribution, which means to generate a possible assignment of values to random variables such that each assignment's probability of being generated is equal to its probability in the model. Sampling from a distribution is a popular way to conduct approximate inference: we can approximate $\Pr(\texttt{grass\_is\_wet})$ by sampling the random variable `grass_is_wet` many times and calculating the proportion of times that it comes out true.

We often want to perform inference or sampling not on our probabilistic model as is but conditional on some observed evidence. For example, having observed that the grass is wet, we may want to compute $\Pr(\texttt{rain} \mid \texttt{grass\_is\_wet})$, either exactly or approximately. Such inference lets us *explain* observed evidence and *learn* from them, especially if, as Bayesians, we treat probabilities themselves (such as $\Pr(\texttt{rain})$) as random variables [37].

### 1.1 Languages for Probabilistic Models

Like many AI researchers, we aim to represent probability distributions so that humans can develop and understand them easily and computers can perform inference and sampling on them efficiently. In other words, we aim to build a domain-specific language for probability distributions. Many such languages have been described in the literature [18, 36]. They are instances of declarative programming, in that they separate the description of models from the computations on them.

A fundamental and popular way to represent a probability distribution is as a *graphical model* (specifically a *Bayes net*), a directed acyclic graph whose nodes represent random variables and edges indicate (typically causal) dependencies.



Each node $X$ is annotated with the conditional probability distribution of its value given the values of its parents $\vec{Y}$. The notion of dependencies is such that $\Pr(X|\vec{Z}) = \Pr(X|\vec{Y})$ if $Z$ is a superset of $Y$; informally, each node depends directly on its parents only.

Humans find graphical models intuitive because they describe sampling procedures: to pick a value for a node $X$, first pick values for its parents $\vec{Y}$, then refer to the conditional probability distribution $\Pr(X|\vec{Y})$ for the node $X$. The dependency structure that graphical models express also gives rise to efficient inference algorithms. However, as graphical models scale up to real problems involving thousands or more of random variables and dependencies, they become unwieldy in the raw.

It helps to *embed* [25, 31] a language of probability distributions in a host language such as MATLAB [35] or Haskell [8]. The embedded language is more commonly called a *toolkit* for probabilistic reasoning, because it is structured as a library of data types of graphical models and random variables along with inference procedures that operate on them. Similarly, distributions can be represented and reasoned about in a logic using formulas that talk about probabilities [22]. In the embedded setting, graphical models can be constructed using programs written in the host language, so repeated patterns (such as random variables that recur for each time step) can be factored out and represented compactly. However, a toolkit defines its own data types; for example, random integers are distinct from regular integers and cannot be added using the addition operation of the host language. This linguistic mismatch typically degrades the efficiency, concision, and maintainability of deterministic parts of a model. In particular, it is difficult to include the toolkit itself in a model, as is useful for agents to reason about each other.

The notational overhead caused by linguistic mismatch can be eliminated by building a standalone language for probability distributions. Such a language typically adds constructs for probabilistic choice and evidence observation to a general-purpose language, be it of a functional [20, 30, 38, 40, 42], logical [2, 7, 28, 34, 41, 43], or relational [16, 17, 32, 45, 46] flavor. For example, the conditional distribution $\Pr(\texttt{rain} \mid \texttt{grass\_is\_wet})$ can be expressed as follows for inference in IBAL [38].

```
rain         = dist [ 0.3 : true, 0.7 : false ]
sprinkler    = dist [ 0.5 : true, 0.5 : false ]
grass_is_wet = dist [ 0.9 : true, 0.1 : false ] & rain
             | dist [ 0.8 : true, 0.2 : false ] & sprinkler
             | dist [ 0.1 : true, 0.9 : false ]
observe grass_is_wet in rain
```

The construct `dist` expresses probabilistic choice, and the construct `observe` expresses evidence observation (here, that `grass_is_wet` is true). Because the entire language is probabilistic, the same operation can be used to, for example, add random integers as to add regular integers. It is easy to express deterministic parts of a model, simply by not using probabilistic constructs in part of the

code. These notational conveniences make programs in a standalone probabilistic language easier for domain experts to develop and understand.

The main drawback of a standalone probabilistic language is that it cannot rely on the infrastructure of an existing host language. To start with, the vast majority of standalone probabilistic languages are implemented as interpreters rather than compilers (with the notable exceptions of AutoBayes [13] and HBC [5]). In contrast, an embedded probabilistic language can piggyback on its host language's compilers to remove some interpretive overhead. We are also not aware of any probabilistic language implemented in itself, so again the language cannot be included in a model for multi-agent reasoning. In general, a standalone language takes more effort to implement than an embedded one, if only due to the need to implement 'boring bits' such as string processing, I/O, timing facilities, and database interfaces. A foreign function interface to a mainstream language can help fill the void of libraries, but glue code imposes its own overhead.

### 1.2 Our Approach: A Very Shallow Embedding

We combine the advantages of embedded and standalone probabilistic languages by embedding a language of probabilistic distributions into a general-purpose language in a *very shallow* way. That is, probabilistic functions are embedded as host-language functions, calls to them are embedded as host function calls, and random integers can be added using the addition operation of the host language. This approach lets us take advantage of the existing infrastructure of the host language while also reducing notational and interpretive overhead.

For example, the conditional distribution Pr(`rain` | `grass_is_wet`) can be expressed as the following host-language program. We use the host language's abstraction facility to factor out the notion of a coin `flip` that yields `true` with probability `p`.

```
let flip p = dist [(p, true); (1.-.p, false)]

let grass_model () =
  let rain = flip 0.3 and sprinkler = flip 0.5 in
  let grass_is_wet = flip 0.9 && rain
                  || flip 0.8 && sprinkler
                  || flip 0.1 in
  if grass_is_wet then rain else fail ()
```

Our host language is OCaml: the code above is just an OCaml program that defines two ordinary functions `flip` (of type `float -> bool`) and `grass_model` (of type `unit -> bool`). The definition of `grass_model` binds the ordinary Boolean variables `rain`, `sprinkler`, and `grass_is_wet` using the functions `dist` (for making a probabilistic choice) and `fail` (for observing impossible evidence). These functions are provided by a library we built without modifying or duplicating the implementation of OCaml itself.[1] Accordingly, we can express probabilistic models using OCaml's built-in operations (such as `&&`), control constructs

[1] In contrast, IBAL happens to be written in OCaml, but it cannot perform inference on OCaml programs such as this one or itself.

(such as `if`), data structures, module language, and wealth of libraries, including our implementations of probabilistic inference, which are useful for multi-agent reasoning. We can use OCaml's type system to discover mistakes earlier, and OCaml's bytecode compiler to perform inference faster.

On one hand, because our approach represents probability distributions as ordinary programs, it is easier for programmers to learn, especially those already familiar with the host language. On the other hand, there is some linguistic mismatch between evaluating a probability distribution and evaluating a call-by-value program, so some thunks are necessary. These occurrences of '`fun () ->`' in the code are only slightly bothersome, and we could easily write a Camlp4 preprocessor to eliminate them. They would be unnecessary if we were to express distributions in Haskell rather than OCaml, as monadic computations.

To continue the example above briefly, we can use two other library functions `reify` and `normalize` to evaluate the model at OCaml's top-level prompt '`#`':

```
# normalize (reify None grass_model);;
reify: done; 14 accepted 9 rejected 0 left
bool pV = PV [(0.468471442720369724, V true);
              (0.53152855727963022, V false)]
```

Whereas `dist` takes a list of weighted possibilities as input and makes a random choice, `reify` takes a probabilistic program (a thunk) as input and computes its list of weighted possibilities (return values). The `normalize` function then rescales the weights to sum up to 1. The result above says that the conditional probability that it rained, given that the grass is wet, is 47%. If we do not like this result, we can define and evaluate a different model interactively. Alternatively, we can compile the model by placing the code above for `grass_model` in a file along with the following driver.

```
let PV choices = normalize (reify None grass_model) in
let [(p, _)] = List.filter (function (_,V x) -> x) choices in
Printf.printf "The probability it rained is %g" p
```

### 1.3 Contributions

This paper describes our design of a practical DSL for probabilistic programming. We achieve a very shallow embedding of the language, by a novel combination of two existing techniques for eliminating notational and interpretive overhead in embedded languages.

1. The *direct-style* representation of side effects [11, 12] using *delimited control operators* [4, 10] lets us reify a probabilistic program as a weighted search tree without the run-time overhead of stepping through deterministic code.
2. The *finally-tagless* embedding of object programs [1] using *combinator functions* [33, 47] allows multiple inference algorithms to operate on the same model without the run-time overhead of tagging values with their types.

We apply these general techniques to probabilistic programming and demonstrate how they make embedded DSL programs faster to run and easier to write. We then show how to implement interpreters for our language that perform exact and approximate inference efficiently. More specifically:

1. We implement *variable elimination* [6], an existing algorithm for exact inference.
2. We develop *importance sampling* [15, 44] with look-ahead, a new, general algorithm for approximate inference.
3. We support *lazy evaluation* in probabilistic programs.
4. Users of our library are free to implement their own inference algorithms.

Our implementation, with many examples, benchmarks, and tests, is available online at `http://okmij.org/ftp/kakuritu/README.dr`. It is only a few hundred lines long, yet its speed and accuracy on available benchmarks [27] are competitive with other, state-of-the-art implementations of probabilistic languages such as IBAL [39]. Our work thus establishes that these general techniques for very shallow embeddings are effective in the case of probabilistic programming and compatible with inference algorithms as advanced as importance sampling. The latter compatibility is surprising because a very shallow embedding prevents inference procedures from inspecting the source code of probabilistic programs.

Having reviewed related work above, we structure the rest of this paper as follows. Section 2 develops our very shallow embedding of a probabilistic DSL in OCaml in several steps. We begin with a finally tagless embedding, which is statically typed without tagging overhead, but whose straightforward implementation is verbose to use and does not run deterministic code at full speed. Following Filinski's [11] representation of monads, we address these shortcomings by converting to CPS and then to direct style.

The remainder of the paper shows how to implement efficient inference algorithms by reifying a stochastic computation as a lazy search tree. Section 3 describes the use of reification for exact inference and for stepping through a stochastic model. Section 4 describes our algorithm for importance sampling with look-ahead. Section 5 introduces lazy evaluation in the embedded program. Finally, we describe our performance testing and conclude.

## 2 Embedding Stochastic Programs

In this section, we develop our very shallow embedding of a probabilistic DSL in OCaml in three steps. First, in §2.1, we define the probabilistic DSL in *tagless final* form: as a signature that lists the operations and their types. Second, in §2.2, we describe a basic implementation of the signature, using the probabilistic monad realized as a lazy search tree. The third step is to eliminate the notational overhead of writing deterministic code and the interpretive overhead of running it. In §2.3, we show how to implement the same signature in continuation-passing style (CPS). We then introduce the delimited control operators *shift* and *reset* in §2.4, so as to present the ultimate embedding of our language in §2.5.

### 2.1 Tagless Final Probabilistic Language

To begin, we define the language, that is, introduce its operations and their types. We define the syntax of the language as the following OCaml signature `ProbSig`, so that each implementation of the language is a module that matches the signature [1]. The implementations below make the language call-by-value.

```
type prob = float

module type ProbSig = sig
 type 'a pm
 type ('a,'b) arr

 val b    : bool -> bool pm
 val dist : (prob * 'a) list -> 'a pm
 val neg  : bool pm -> bool pm
 val con  : bool pm -> bool pm -> bool pm
 val dis  : bool pm -> bool pm -> bool pm
 val if_  ; bool pm -> (unit -> 'a pm) -> (unit -> 'a pm) -> 'a pm
 val lam  : ('a pm -> 'b pm) -> ('a,'b) arr pm
 val app  : ('a,'b) arr pm -> ('a pm -> 'b pm)
end
```

Our ultimate embedding in §2.5 is very shallow in the sense that it identifies our probabilistic DSL with the host language OCaml. For exposition, however, we start with Erwig and Kollmansberger's toolkit approach [8] and keep the DSL distinct from OCaml.

For simplicity, the signature `ProbSig` includes only the base type of booleans and the operations necessary for the lawn example. Following the tagless-final approach [1], a boolean expression in the DSL, which denotes a distribution over booleans, is represented as an OCaml expression of the type `bool pm`. Here `pm` is an abstract type constructor, whose name is short for 'probability monad' [8, 19, 42]. Different implementations of `ProbSig` define `pm` in their own ways, giving rise to different interpreters of the same DSL. As an example, the 'true' literal in the DSL is represented in OCaml as `b true`, of the abstract type `bool pm`. The expression denotes the distribution over 'true' with probability 1.

Besides booleans, our language has functions. A function expression in the DSL, say of type `bool->bool`, is represented as an OCaml expression of the type `(bool,bool) arr pm`. Here `arr` is a binary type constructor, again abstract. We use higher-order abstract syntax to represent binding. For example, we represent the curried binary function on booleans $\lambda x.\, \lambda y.\, x \wedge \neg y$ in the DSL as the OCaml expression `lam (fun x -> lam (fun y -> con x (neg y)))`, whose inferred type is `(bool, (bool,bool) arr) arr pm`. To apply this function to the literal 'false', we write `app (lam (fun x -> ...)) (b false)`.

This DSL is probabilistic only due to the presence of the `dist` construct, which takes a list of values and their probabilities. The `dist` expression then denotes a stochastic computation with the given discrete distribution.

A probabilistic program is a functor that takes a `ProbSig` module as argument. For example, we can write the lawn example as follows:

```
module Lawn(S: ProbSig) = struct
  open S

  let flip p = dist [(p, true); (1.-.p, false)]

  let let_ e f = app (lam f) e

  let grass_model () =
  let_ (flip 0.3) (fun rain ->
  let_ (flip 0.5) (fun sprinkler ->
  let_ (dis (con (flip 0.9) rain)
            (dis (con (flip 0.8) sprinkler)
                 (flip 0.1))) (fun grass_is_wet ->
  if_ grass_is_wet (fun () -> rain) (fun () -> dist []))))
end
```

For convenience, we introduce the `let_` 'operator' above.

This example shows a significant notational overhead. For example, boolean stochastic variables have the type `bool pm` rather than just `bool`, so we cannot pass a random boolean to OCaml functions accepting ordinary booleans, and we need to use `con` rather than the built-in conjunction operator `&&`. Also, we cannot use OCaml's `let` and have to emulate it by `let_`. The notational overhead is as severe as if our embedded DSL were just a toolkit. By the end of the section, we shall eliminate most of the overhead except a few stray `fun () ->`.

Despite its verbosity, our embedding already takes advantage of the type system of the host language: our DSL is typed, and we do not need our own type checker—the OCaml type checker infers the types of stochastic expressions just as it does for all other expressions. For example, the expression `app (b true) (b false)` is flagged as a type error because `bool` is not an arrow type.

### 2.2 Search-Tree Monad

The first implementation of our language uses the probability monad that represents a stochastic computation as a lazy search tree. That is, our first `ProbSig` module instantiates `pm` by the type constructor `pV` defined below.

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

Each node in a tree is a weighted list of branches. The empty list denotes failure, and a singleton list `[(p, V v)]` denotes a deterministic successful outcome `v` with the probability mass `p`. A branch of the form `V v` is a leaf node that describes a possible successful outcome, whereas a branch of the form `C thunk` is not yet explored.

The intended meaning of a search tree of type `'a pV` is a discrete probability distribution over values of type `'a`. Of course, several distinct trees may represent the same distribution; for example, the following trees all represent the same distribution.

```
[(0.6, V true); (0.4, V false)]

[(0.4, V true); (0.4, V false); (0.2, V true)]

[(0.4, V true); (1.0, C (fun () ->
                         [(0.4, V false); (0.2, V true)]))]

[(0.4, V true); (0.8, C (fun () ->
                         [(0.5, V false); (0.25, V true)]))]

[(0.4, V true); (0.8, C (fun () ->
                         [(0.5, V false); (0.25, V true);
                          (0.25, C (fun () -> []))]))]
```

The first tree is just a probability table; it is the smallest, consisting only of a root node. We call such trees flat or fully explored, because they represent a distribution that has been evaluated or inferred completely. The last three trees contain unexplored branches, signified by the `C` variant. In particular, the last example includes a failure node `[]`, which becomes manifest when the branch `C (fun () -> [])` is explored.

We consider trees that represent the same distribution equivalent. Constructively, we introduce the following normalization, or, *exploration* procedure for search trees. The procedure traverses the tree depth-first, forces thunks of branches not yet explored to reveal their content, and accumulates all found values along with their weights in the probability table. While exploring a branch `(p, C t)`, we scale the probabilities of the found values by the factor `p`. The resulting probability table is returned as a flat search tree. The procedure uses a map `PMap` from values to probabilities to collate the values encountered during exploration. The procedure below is more general in that it explores the tree up to the depth specified by the first argument; giving the value `None` as the first argument requests complete normalization. Applying `explore None` to all these trees returns the same result, so they are all equivalent.

```
let explore (maxdepth : int option) (choices : 'a pV) : 'a pV =
  let rec loop p depth down choices ((ans,susp) as answers) =
    match choices with
    | [] -> answers
    | (pt, V v)::rest ->
        loop p depth down rest
      (PMap.insert_with (+.) v (pt*.p) ans, susp)
    | (pt, C t)::rest when down ->
    let down' =
          match maxdepth with Some x -> depth<x | None -> true
    in loop p depth down rest
         (loop (pt*.p) (depth+1) down' (t ()) answers)
    | (pt, c)::rest ->
        loop p depth down rest (ans, (pt*.p,c)::susp) in
  let (ans,susp) = loop 1.0 0 true choices (PMap.empty,[])
  in PMap.foldi (fun v p a -> (p, V v)::a) ans susp
```

The last two lines prepend the collated values to the list of unexplored branches.

The type constructor `pV`, equipped with the operations `pv_unit` and `pv_bind` below, forms a monad [48]:

```
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
```

Monad laws are satisfied only modulo tree equivalence. For example, for values `x` of type `'a` and `f` of type `'a -> 'b pV`, the tree resulting from OCaml evaluating `pv_bind (pv_unit x) f` is equivalent but not identical to the result of `f x`. More precisely, if the trees are finite, then exploring them fully (by applying `explore None`) yields identical results. For infinite trees, which too can be represented by the type `'a pV`, characterizing equivalence is trickier [49].

We use this search-tree monad to define the `SearchTree` module below, our first implementation of the `ProbSig` signature.

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

 let b = pv_unit
 let dist ch = List.map (fun (p,v) -> (p, V v)) ch
 let neg e = pv_bind e (fun x -> pv_unit (not x))
 let con e1 e2 = pv_bind e1 (fun v1 ->
                  if v1 then e2 else (pv_unit false))
 let dis e1 e2 = pv_bind e1 (fun v1 ->
                  if v1 then (pv_unit true) else e2)
 let if_ et e1 e2 = pv_bind et (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
```

This implementation is the standard monadic embedding of a call-by-value language. For example, `neg` is the straightforward 'lifting' of boolean negation so to accept and produce values of the monadic type `bool pV`. The implementation of `con` shows short-cut evaluation: if the first argument evaluates to false, we produce the result 'false' without evaluating the second argument. The implementations of `dis` and `if_` follow a similar pattern. The only interesting part is the implementation of `dist`, which merely converts its argument (specifying the distribution) to a flat search tree. We can use this implementation to run the lawn example, by instantiating the `Lawn` functor with the module `SearchTree`:

```
module SST = Lawn(SearchTree)
```

The resulting module `SST` contains a field `grass_model`, of the inferred type `bool SearchTree.pm`, or `bool pV`:

```
let sst1 = SST.grass_model ()
↪ val sst1 : bool SearchTree.pm = [(1., C <fun>)]
```

That tree is obviously unexplored. Exploring it to depths 3, 9, and 11 gives

```
let sste3 = explore (Some 3) sst1
↪ [(0.15, C <fun>); (0.15, C <fun>);
    (0.35, C <fun>); (0.35, C <fun>)]

let sste9 = explore (Some 9) sst1
↪ [(0.27, V true); (0.012, C <fun>);
    (0.0003, C <fun>); (0.0027, C <fun>);
    (0.0012, C <fun>); (0.0108, C <fun>); ...]

let sste11 = explore (Some 11) sst1
↪ [(0.2838, V true); (0.322, V false)]
```

Only when we explore the tree to depth 11 do we finally obtain the exact distribution denoted by the lawn model: an unnormalized distribution of `rain`. After normalization, it becomes what we have seen in §1.2.

The depth of this search tree reveals the run-time overhead imposed by this implementation. Because booleans and functions in our DSL do not share the types of ordinary booleans and functions in OCaml, we cannot invoke a stochastic function as an ordinary function, and we cannot pass stochastic booleans to OCaml boolean functions. We cannot even use OCaml's boolean constants `true` and `false` as they are; rather, we have to lift them to `b true` and `b false`, search trees of type `bool pV`. Computing with such lifted values is not only more verbose but also less efficient: in the `SearchTree` module, all operations on trees involve `pv_bind`, which always increases the depth of the tree by one.

This example thus illustrates that the notational overhead in our representation of stochastic programs corresponds to an interpretive overhead in the search-tree implementation. When a sequence of deterministic operations is applied to a stochastic expression, as in `neg (neg rain)`, each step incurs its own overhead of `pv_bind` and `pv_unit` traversing and constructing lists and variant data structures. We next obviate the pattern matching performed by `pv_bind`.

### 2.3 Embedding in Continuation-Passing Style

We eliminate the run-time and notational overhead in this and the next two sections.

We still use search trees to represent stochastic expressions, but we no longer implement the stochastic language using the operations `pv_unit` and `pv_bind` in the search-tree monad. We use the continuation monad instead: we represent a stochastic expression of type `'a` not as a tree of type `'a pV` but in CPS, as a function from the continuation type `'a -> bool pV` to the *answer type*

`bool pV`. This move is same as Hughes's [26] and Hinze's [24] move from term implementations to context-passing implementations to reduce pattern-matching and improve efficiency. For this section, it is enough to fix the answer type at `bool pV` rather than make it `'w pV` for some generic `'w`.

Our new implementation of `ProbSig` is as follows. Most of this code is trivial; for example, the implementation of `app` is the standard CPS transform of call-by-value application.

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

 let b x = fun k -> k x
 let dist ch = fun k ->
     List.map (function (p,v) -> (p, C (fun () -> k v))) ch
 let neg e = fun k -> e (fun v -> k (not v))
 let con e1 e2 = fun k -> e1 (fun v1 ->
                  if v1 then e2 k else b false k)
 let dis e1 e2 = fun k -> e1 (fun v1 ->
                  if v1 then (b true k) else e2 k)
 let if_ et e1 e2 = fun k -> et (fun t ->
                     if t then e1 () k else e2 () k)
 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
```

The most interesting case is again `dist`: the expression `dist ch` (where `ch` is a list of choices) accepts a continuation `k` as usual but does not return the result of invoking `k`. Rather, `dist ch` expresses nondeterminism by immediately returning the search tree resulting from feeding each choice in `ch` to `k`. That is, `dist ch k` builds a collection of branches, each of which, upon exploration, returns the result of the rest of the computation using a particular choice in `ch`. This definition of `dist` is the same as applying `pv_bind` to the previous definition `SearchTree.dist`.

We can run our lawn example using this new implementation of `ProbSig`. We pass the module `CPS` to the `Lawn` functor.

```
module SCP = Lawn(CPS)
```

The resulting module `SCP` contains a field `grass_model` with the type `unit -> (bool -> bool pV) -> bool pV`. Forcing this thunk gives not a tree anymore but a higher-order function accepting continuations of type `bool -> bool pV`. In other words, `grass_model ()` is a CPS computation. To convert this result to a search tree so that we can compute the distribution, we have to *reify* it by feeding it the initial continuation `pv_unit`. We do so by applying `reify0`.

```
let scp1 = CPS.reify0 (SCP.grass_model ())
↪ [(0.3, C <fun>); (0.7, C <fun>)]
```

The result `scp1` is indeed a search tree. It still has a few unexplored branches; exploring the resulting tree to the depths 1 and 4 gives

```
let scpe1 = explore (Some 1) scp1
↪ [(0.135, C <fun>); (0.015, C <fun>); (0.135, C <fun>);
    (0.015, C <fun>); (0.315, C <fun>); (0.035, C <fun>);
    (0.315, C <fun>); (0.035, C <fun>)]

let scpe4 = explore (Some 4) scp1
↪ [(0.2838, V true); (0.322, V false)]
```

Exploring this tree to depth 4 already flattens it, so it is not nearly as deep as the tree `sst1` computed in §2.2. This comparison hints that we have removed some overhead in the `SearchTree` implementation of the language.

We can see why the new implementation of `ProbSig` incurs less overhead by considering the example `neg (neg rain)` above. Although we still cannot negate the random boolean `rain` directly using OCaml's built-in `not`, at least the definition of `neg` no longer uses `pv_unit` and `pv_bind`, so the inner call to `neg` no longer builds a tree only to be traversed and discarded right away by the outer call. Rather, each boolean is passed to the rest of the computation as is, without building any tree until `reify0` constructs a leaf node or `dist` constructs a choice node. This CPS implementation is thus more efficient than the search-tree implementation, yet equivalent according to the monad laws.

Some overhead remains. Stochastic boolean values remain distinct from ordinary boolean values, so we still have to write `b true` to use even a boolean constant in the language, and applying `b` at run time is not free. The per-application overhead, albeit reduced, is still present due to the need to explicitly invoke continuations. We next eliminate all this overhead, *mechanically*.

### 2.4 Delimited Control

To eliminate the notational and run-time overhead of the embedding of our stochastic DSL, we will use delimited control operators `shift` and `reset`. This section briefly introduces these operators.

The implementation of `CPS.dist` in §2.3 pointed out that a stochastic expression may be regarded as one that can return multiple times, like a `fork` expression in C. If `fork` were available in OCaml, we would use it to implement `dist`. Ordinary OCaml functions, which execute deterministically in a single 'thread', could then be used as they are within a stochastic computation.

The library of delimited continuations for OCaml [29] does provide the analogue of `fork`. To be more precise, the library provides two functions `reset` and `shift` [3, 4].[2] The function `reset`, of type `(unit -> 'a) -> 'a`, is analogous to `try` in OCaml. It takes a thunk, evaluates it, and returns its value—unless a

[2] The library actually implements so-called multi-prompt delimited continuations, similar to `cupto` [21]. We embed stochastic computations using only a single prompt, which we store in a global variable and elide for clarity. The accompanying source file `probM.ml` shows the actual implementation.

control effect occurs during the evaluation. Thus, `reset (fun () -> 41 + 2)` returns 43. Control effects are raised by the function `shift`, which generalizes raising an exception. In particular, `reset (fun () -> 41 + shift (fun k -> 2))` returns the value 2: one may regard `shift (fun k -> 2)` to throw the integer 2 as an exception, caught by the closest dynamically enclosing `reset`.

In the example `reset (fun () -> 41 + shift (fun k -> 2))`, `shift` is applied in the context of the addition to `41` enclosed in `reset`. That context can be represented as a function `fun x -> reset (fun () -> 41 + x)`. Informally, the application of `shift` is replaced by `x` bound outside of `reset`. Therefore, `reset (fun () -> 41 + shift (fun k -> 2))` evaluates as

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

and the result is 2, as we saw. The following examples proceed likewise.

```
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
```

and

```
reset (fun () -> 41 + shift (fun k -> k (k 2)))
↪ reset (fun () -> (fun k -> k (k 2))
                   (fun x -> reset (fun () -> 41 + x)))
↪ reset (fun () -> (fun x -> reset (fun () -> 41 + x))
                   (reset (fun () -> 41 + 2)))
↪ 84
```

The last example shows that the context of `shift`, captured as a function, can be applied more than once. In other words, the `shift` expression can return more than once. In this regard, `shift` behaves like `fork`, splitting the computation into two threads and returning different values in them. That is exactly the behavior we need to implement `dist` with no overhead on deterministic code.

### 2.5 Direct Embedding of the Probabilistic DSL

We can finally implement our embedded DSL without overhead on deterministic code. Following Filinski [11], we do so by mechanically converting the `CPS` implementation to direct style.

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

 let b x = x
 let dist ch = shift (fun k ->
```

```
    List.map (function (p,v) -> (p, C (fun () -> k v))) ch)
  let neg e = not e
  let con 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
```

The transformation is straightforward, especially for expressions that use their continuations normally. The transformation makes the continuation argument `k` in the `CPS` implementation implicit, and so `neg` becomes just `not`. For the same reason, `app`, `lam`, and `b` become the identity function. The only interesting cases are `dist` and `reify0`. As explained in §2.3, the definition of `CPS.dist` is special: `CPS.dist ch` does not return the result of applying a continuation, but rather produces a search tree directly. The implementation of `dist` above expresses exactly the same operation in direct style, but the continuation argument `k` is implicit, so we access it using the control operator `shift`. Similarly, whereas `CPS.reify0` supplies the initial continuation `pv_unit` as an explicit argument, the implementation of `reify0` above uses `reset` to supply it implicitly.

We can run the lawn example as before, passing the `Direct` implementation to the functor `Lawn`. We obtain the same result as in §2.3:

```
module SDI = Lawn(Direct)
let sdi1 = Direct.reify0 (SDI.grass_model)
↪ [(0.3, C <fun>); (0.7, C <fun>)]

let sdie4 = explore (Some 4) sdi1
↪ [(0.2838, V true); (0.322, V false)]
```

The type of a stochastic computation that returns results of type `'a` is `'a pm`, which in the `Direct` implementation is just `'a`. Thus, a random boolean is just an ordinary boolean, which can be passed to ordinary boolean functions such as `not`. We can apply all ordinary OCaml functions, even compiled third-party libraries, to a stochastic value without lifting or changing them at all. Furthermore, since our direct embedding no longer distinguishes stochastic expressions from other (effectful) expressions in OCaml, we can dispense with the `ProbSig` structure. We can program probabilistic models directly in OCaml, using all of its facilities including `let` and `let rec` forms. The notational overhead has vanished (save a few stray thunks). This is how we wrote the lawn example in §1.2, directly in OCaml. The function `reify` mentioned there is the composition of `explore` and `reify0`. Applying `reify None` to the model performs exact inference, presenting the distribution of the model as a flat search tree.

One may remark that the thunk in the definition of `reify0` represents a certain overhead. Indeed, the result of an expression, even if it is completely deterministic, is represented as a search tree, and this boxing is overhead. However,

this overhead is incurred each time a complete probabilistic model is reified, not each time the model performs a deterministic operation. Unlike earlier implementations, we can apply deterministic functions to stochastic values with no overhead. We pay the price of embedding stochastic computations only when calling the two stochastic operators `dist` and `reify0`. The deterministic parts of a stochastic program run at full speed, as if no randomness were present and no inference were in progress. We have achieved our goal to eliminate run-time *and* notational overhead for deterministic parts of probabilistic models.

## 3 Accelerating Inference Using Reification and Reflection

We have seen that exact inference can be performed on a stochastic expression by reifying it into a lazy search tree (using `reify0`) then fully exploring the tree (using `explore None`). In this section, we show that `reify0` and `explore` are useful not only for producing the final inference result; these functions can help achieve the result faster. In other words, these functions let users of our library write optimized inference procedures. In particular, we implement the optimized exact-inference procedure known as *variable elimination* [6].

Throughout the rest of this paper, we use the very shallow embedding described in §2.5. We first introduce a function that generalizes `dist` there:

```
let reflect (choices : 'a pV) : 'a =
  let rec make_choices k pv =
    List.map (function
              | (p, V x) -> (p, fun () -> k x)
              | (p, C x) -> (p, fun () -> make_choices k (x ())))
             pv
  in shift (fun k -> make_choices k choices)
```

Whereas `dist` takes as argument a weighted list of possibilities, `reflect` receives weighted possibilities in the form of a search tree. The only difference between a list and a tree is that a tree may not be flat and may even contain unexplored branches—hence the additional case alternative `(p, C x) -> ...` above. Both `dist` and `reflect` express a random choice according to their argument.

We find `reflect` useful because it is an inverse of `reify0`, in the following sense. The type of `reflect` is `'a pV -> 'a`, and the type of `reify0` is `(unit -> 'a) -> 'a pV`. Informally, `reflect` turns a search tree into a stochastic expression, whereas `reify0` turns a stochastic expression (encapsulated in a thunk) into a search tree. For example, our lawn model in §1.2 defines a thunk `grass_model` of type `unit -> bool`; the stochastic expression `grass_model ()` produces a random boolean. If we pass `grass_model` to the function `reify0`, we obtain

```
let gm_tree = reify0 grass_model
↪ val gm_tree : bool pV = [(0.3, C <fun>); (0.7, C <fun>)]
```

The result `gm_tree` reifies the stochastic expression `grass_model ()` in the form of a tree. We can explore `gm_tree` to get a shallower search tree describing the same distribution:

```
let gm_tree1 = explore (Some 4) gm_tree
↪ val gm_tree1 : bool pV = [(0.2838, V true); (0.322, V false)]
```

Applying the function `reflect` to the trees `gm_tree` and `gm_tree1` produces two stochastic expressions that express the same distribution as `grass_model ()`, the stochastic expression we started with. Using `reify0`, we can turn these stochastic expressions into equivalent search trees.

These observations on `reify0` and `reflect` hold in general: for each search tree `pv` and stochastic expression `e`,

```
reify0 (fun () -> reflect pv)  ≡ pv
reflect (reify0 (fun () -> e)) ∼ e
```

where ≡ means that two search trees are equivalent and ∼ means that two stochastic expressions are equivalent in that they express the same distribution. Thus, `reify0` and `reflect` form a *reification-reflection* pair [11].

As we just observed, any terminating stochastic expression is equivalent to reifying it then reflecting the resulting search tree. In this sense, reification-reflection is a sound program transformation, preserving program equivalence. It turns out that inference on the transformed program often takes less time and space. For example, if `e` is the expression `dist [(0.5, e1); (0.5, e2)]` where `e1` and `e2` are complex deterministic expressions, then reifying `e` gives the search tree `[(0.5, V v1); (0.5, V v2)]` where `v1` and `v2` are the results of `e1` and `e2`. Therefore, reflecting the search tree gives a stochastic expression that, albeit equivalent to `e`, can be evaluated faster because `e1` and `e2` no longer need to be evaluated. Reification followed by reflection can thus speed up inference—especially if the result of reification is (partially) flattened before reflection, because flattening reduces the tree by removing shallow failures and collating repeated values. For example, `reflect gm_tree1` uses the result of exact inference on the lawn model, so it runs faster than the original `grass_model ()`.

This optimization of reification followed by (partial) flattening and reflection gives rise to the inference technique known as variable elimination [6]. Its benefit is demonstrated by the following example, computing the XOR of $n$ coin tosses.

```
let flips_xor n () =
  let rec loop n =
    if n = 1 then flip 0.5
       else flip 0.5 <> loop (n-1)
  in loop n
```

The search tree for `flips_xor 10` is a full binary tree of depth 10. The exploration of this tree for exact inference has to enumerate all of its 1024 leaves. Many parts of the computation are shared: each `flip 0.5` forks the search tree into two branches that make the same recursive call `loop (n-1)`. Variable elimination is thus much more efficient than brute-force exact inference:

```
let flips_xor' n () =
  let rec loop n =
```

```
    if n = 1 then flip 0.5
       else let r = reflect (explore None
                      (reify0 (fun () -> loop (n-1))))
            in flip 0.5 <> r
in loop n
```

Exact inference now explores 9 trees of depth 2, or a total of $9 \times 4 + 2 = 38$ leaves. As is usual of dynamic programming, the optimization turns an exponential-time algorithm into a linear-time one, by sharing the computation for `loop (n-1)` across the two possible results of `flip 0.5` in `loop n`.

More generally, whenever we have a stochastic function `f`, we can replace it by the stochastic function

```
let bucket = memo (fun x -> explore None (reify0 (fun () -> f x)))
in fun x -> reflect (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 repeatedly applying `f` to the same argument. This technique lets us handle benchmarks in Jaeger et al.'s collection [27] in at most a few seconds each. The memo table is also called a *bucket*, so variable elimination is also called *bucket elimination* [6]. It is similar to the nested-loop optimization of database join queries.

## 4 Importance Sampling with Look-Ahead

Because we can reify a probabilistic program as a lazy search tree, we (like any other user of our library) can implement inference algorithms as operations that explore the tree without slowing down the deterministic parts of the model. We have described exact inference algorithms (brute-force enumeration and variable elimination) above. In this section we implement approximate inference.

Recall that reifying a stochastic computation yields a lazy search tree, a data structure of the type `'a pV`. Exact inference traverses the whole tree. For many probabilistic models in the real world, however, the tree is too large to traverse completely, even infinite. Instead, we can sample from the distribution expressed by the probabilistic program, by tracing a path down the tree to a leaf. Whenever we arrive at a choice node, we select a branch at random, according to its weight. By repeating this process many times and computing the frequencies of those sampled values that are not failures, we can approximate the distribution.

This naive, or *rejection*, sampling method accurately approximates the probabilities of likely values. However, it poorly approximates the tail of the distribution, and it takes too long if most sampled paths end up in failure. Alas, a low success probability is quite common. An illustrative if contrived example is computing the `and` of $n$ tosses of a fair coin by a drunk who often loses the coin.

```
let drunk_coin () =
  let toss = flip 0.5 in
  let lost = flip 0.9 in
  if lost then fail () else toss
```

```
let rec dcoin_and = function
  | 1 -> drunk_coin ()
  | n -> drunk_coin () && dcoin_and (n-1)
```

The function `drunk_coin` models the toss of a fair coin by a drunk; with the probability 0.9 the coin ends up in a gutter. The example is simple enough that the exact distribution of `and` of 10 tosses can be easily computed, either analytically or by complete exploration of the search tree: `[(9.7656e-14, V true); (0.05263, V false)]`. Rejection sampling returns `[(0.01, V false)]` for 100 samples and `[(0.052, V false)]` for 10,000 samples. The probability for the drunk coin to come up heads ten times in a row is too low to be noticed by the rejection sampling. If we assert such an observation and wish to estimate its probability, the rejection sampling of `fun () -> dcoin_and 10 || fail ()` offers little help: even with 10,000 attempts, all samples are unsuccessful.

*Importance sampling* [15, 44] is an approximate inference strategy that improves upon rejection sampling by assigning weights to each sample. For example, because assigning `true` to the variable `lost` leads to failure, the sampler should not pick a value for `lost` randomly; instead, it should force `lost` to be `false`, but scale the weight of the resulting sample by 0.1 to compensate for this artificially induced success.

Pfeffer [39] introduced an importance-sampling algorithm to perform approximate inference for the probabilistic language IBAL. The main technique is called *evidence pushing* because it reduces nondeterminism in the search by pushing observed evidence towards their random cause in a stochastic program. Evidence pushing in the IBAL interpreter requires very sophisticated data-flow analysis on probabilistic programs, which our inference procedures cannot perform because they cannot inspect the source code of probabilistic programs.

Nevertheless, we can perform importance sampling on the lazy search tree that results from reifying a probabilistic program. The key is to explore the tree shallowly before committing to any random choice among its branches. Whenever this look-ahead comes across a branch that ends in failure, that branch is eliminated from the available choices. Similarly, whenever the look-ahead comes across a branch that ends in success, the successful result is registered as a sample, and that branch is again eliminated from the available choices. After the look-ahead, the sampler randomly selects among the remaining choices and repeats the process. 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. Since the look-ahead notices not only shallow failures but also shallow successes, one trace down the tree may yield more than one sample.

We have implemented importance sampling with look-ahead as described above. As one might expect, it is much more accurate than rejection sampling on programs like `dcoin_and`. For example, using our algorithm to sample 5000 times from `dcoin_and 10` estimates the distribution as `[(8e-14, V true); (0.0526, V false)]`; we now obtain quite an accurate estimate of the probability of the coin to come up heads ten times in a row. Recall that

rejection sampling failed to estimate this probability even with twice as many samples.

## 5 Lazy Evaluation

Because events in the world usually take place in a different order from when they are observed, values in a probabilistic program are usually produced in a different order from when they are observed. Just as in logic programming, this mismatch makes the search tree much larger, because the part of the search tree between making a random choice and observing its consequence is duplicated. To reduce this mismatch, we turn to lazy evaluation.

The following contrived example illustrates the mismatch that motivates lazy evaluation. We define the producer `flips p n` to be the random result of flipping `n` independent coins, each `true` with probability `p`. We also define the observer `trues n xs` to check that a list of `n` booleans consists entirely of `true`.

```
let rec flips p = function
    | 0 -> []
    | n -> let x  = flip p in
           let xs = flips p (n - 1) in
           x :: xs

let rec trues n xs = match (n, xs) with
    | (0, [])      -> true
    | (n, (x::xs)) -> x && trues (n - 1) xs
```

The program `if trues 20 (flips 0.5 20) then () else fail ()` expresses the observation that a sequence of 20 fair-coin flips comes out all `true`. It is easy to see mathematically that the probability of this observation is $2^{-20}$, but the inference algorithms described above need to explore millions of tree nodes to compute this probability as nonzero, because the entire list of 20 booleans is produced before any element of the list is observed. It is especially common for this kind of dissociation between production and observation to arise naturally in probabilistic models of perception; for example, to parse a given sentence is to observe that a randomly produced sentence is equal to the given sentence.

To reduce the delay between the production and observation of random values, we can revise the probabilistic program using lazy evaluation. We provide a library function `letlazy`, which adds memoization to a thunk. For example, `letlazy (fun () -> flip p)` below creates a thunk that evaluates `flip p` the first time it is invoked, then returns the same boolean thereafter. The data constructors `LNil` and `LCons` are lazy analogues of the list constructors `[]` and `::`.

```
let rec flips p = function
    | 0 -> LNil
    | n -> let x  = letlazy (fun () -> flip p) in
           let xs = letlazy (fun () -> flips p (n - 1)) in
           LCons (x, xs)
```

```
let rec trues n xs = match (n, xs) with
    | (0, LNil)         -> true
    | (n, LCons (x,xs)) -> x () && trues (n - 1) (xs ())
```

To be sure, `letlazy` does not memoize the lazy search tree that results from reifying a probabilistic program. That would make the tree stay longer in memory but not have any fewer nodes. What `letlazy` memoizes is an object value within the probabilistic program. Importance sampling then requires only one sample to compute the probability $2^{-20}$ exactly.

It is incorrect to implement `letlazy` using OCaml's built-in laziness, because the latter uses the mutable state of the underlying hardware, which does not 'snap back' when the inference procedure backtracks to an earlier point in the execution of the probabilistic program. Instead, we implement `letlazy` by using or emulating thread-local storage [23].

For equational reasoning, it is useful for the expression `letlazy t` (where `t` is a thunk) to denote the same distribution as `let v = t () in fun () -> v`. If the body of `t` involves observing evidence (for example, if `t` is `fail`), then this equivalence may not hold because, if the thunk returned by `letlazy t` is never invoked, then neither is `t`. To maintain the equivalence, we can change `letlazy` to always invoke `t` at least once. This variation on lazy evaluation is called *delayed evaluation* [39].

## 6 Performance Testing

The examples above are simple and contrived to explain inference algorithms. We have also expressed and performed inference on a model of realistic complexity, namely Pfeffer's [39] music model. This model describes the transformation of melodic motives in classical music as a combination of transposition, inversion, deletion and insertion operations. The number of operations and their sequence are random, as is the effect of transposition on each note. Given a transformed melody, we wish to obtain the conditional distribution of original motives that could have given rise to the transformed melody.

Exact inference is infeasible for this model because the search tree is very wide and deep: 40% of the nodes have 8 or more children, and the depth of the tree reaches 10. Rejection sampling is also useless because the overall probability of matching the given transformed melody is quite low—on the order of $10^{-7}$. The importance sampling procedure described in Section 4, with the crucial help of lazy evaluation described in Section 5, estimated the conditional probabilities with roughly the same accuracy as Pfeffer's state-of-the-art implementation of evidence pushing for his IBAL language.

For concreteness, we describe a typical instance of this test. The original melody (#1) and transformed melody (#1) are two different motives taken from a movement in an early piano sonata by Beethoven. On a 2GHz Pentium 4 computer, we ran 100 trials in which Pfeffer's importance sampler had 30 seconds per trial (following his original testing), 100 trials in which our sampler had 30

seconds per trial, and another 100 trials in which our sampler had 90 seconds per trial.

Given 30 seconds, Pfeffer's sampler runs about 70,000 times; it estimates the evidence probability to be $\exp(-14.6)$ on average, with the standard deviation $\exp(-15.1)$. Also given 30 seconds, our sampler runs about 10,000 times; it estimates the evidence probability to be $\exp(-13.7)$ on average, with the standard deviation $\exp(-13.8)$. When given 90 seconds, our sampler runs about 30,000 times; it estimates the evidence probability to be $\exp(-13.6)$ on average, with the standard deviation $\exp(-14.4)$. For both samplers, the histogram of estimated probabilities that result from repeated trials is skewed to the right; the skewness is roughly the same for the two samplers on each inference problem.

## 7 Conclusions

We have presented a *very shallow* embedding of a DSL for probabilistic modeling. This embedding exemplifies the following advantages of very shallow embeddings of DSLs in general.

**Negligible notational overhead.** We represent probabilistic programs as ordinary OCaml programs, so random integers can be added using ordinary `+`, random variables can be bound using ordinary `let`, and random processes can be defined using ordinary $\lambda$-abstraction.

**Type safety.** OCaml infers type signatures and detects type errors in stochastic expressions just as in all other OCaml expressions. Our embedding adds no type tags to run-time values.

**No run-time overhead for deterministic code.** Deterministic constants and operations used in a probabilistic model, including those in third-party compiled libraries, run at full speed as if our probabilistic library were not present. (This lack of overhead is similar to how non-privileged client code runs under VMWare as if the hypervisor were not present [9].)

**Ease of learning and maintenance.** We use the unmodified OCaml system as is. Probabilistic programs and inference procedures can use all facilities of the host language, including I/O, libraries, and a module system.

Our implementation is based on reifying probabilistic models as lazy search trees. This reification allows us to evaluate the same model using multiple inference algorithms, including algorithms such as variable elimination and importance sampling that were thought to require a more deeply embedded probabilistic language. We can also step through the random choices made by a model by tracing through its search tree.

The size and complexity of the models we have run are in the ballpark of state-of-the art systems such as IBAL [39]. We plan to exercise this system by implementing larger probabilistic models of real-world phenomena, in particular the nested inference that agents conduct about each other.

## References


[1] Carette, J., Kiselyov, O., Shan, C.-c.: Finally tagless, partially evaluated: Tagless staged interpreters for simpler typed languages. Journal of Functional Programming (2008) (in press)

[2] Cussens, J.: Logic-based formalisms for statistical relational learning. In: [18], ch. 9, pp. 269–290 (2007)

[3] Danvy, O., Filinski, A.: A functional abstraction of typed contexts. Tech. Rep. 89/12, DIKU, University of Copenhagen, Denmark (1989), `http://www.daimi.au.dk/~danvy/Papers/fatc.ps.gz`

[4] Danvy, O., Filinski, A.: Abstracting control. In: Proceedings of the 1990 ACM conference on Lisp and functional programming, pp. 151–160. ACM Press, New York (1990)

[5] Daumé III, H.: Hierarchical Bayes compiler (2007), `http://www.cs.utah.edu/~hal/HBC/`

[6] Dechter, R.: Bucket elimination: A unifying framework for probabilistic inference. In: Jordan, M.I. (ed.) Learning and inference in graphical models. Kluwer, Dordrecht (1998); Paperback: Learning in Graphical Models. MIT Press

[7] Domingos, P., Richardson, M.: Markov logic: A unifying framework for statistical relational learning. In: [18], ch. 12, pp. 339–371 (2007)

[8] Erwig, M., Kollmansberger, S.: Probabilistic functional programming in Haskell. Journal of Functional Programming 16(1), 21–34 (2006)

[9] Feigin, B., Mycroft, A.: Jones optimality and hardware virtualization: A report on work in progress. In: Glück, R., de Moor, O. (eds.) Proceedings of the 2008 ACM SIGPLAN symposium on partial evaluation and semantics-based program manipulation, pp. 169–175. ACM Press, New York (2008)

[10] Felleisen, M., Friedman, D.P., Duba, B.F., Merrill, J.: Beyond continuations. Tech. Rep. 216, Computer Science Department, Indiana University (1987)

[11] Filinski, A.: Representing monads. In: POPL 1994: Conference record of the annual ACM symposium on principles of programming languages, pp. 446–457. ACM Press, New York (1994)

[12] Filinski, A.: Representing layered monads. In: POPL 1999: Conference record of the annual ACM symposium on principles of programming languages, pp. 175–188. ACM Press, New York (1999)

[13] Fischer, B., Schumann, J.: AutoBayes: A system for generating data analysis programs from statistical models. Journal of Functional Programming 13(3), 483–508 (2003)

[14] Forbes, J., Huang, T., Kanazawa, K., Russell, S.J.: The BATmobile: Towards a Bayesian automated taxi. In: Proceedings of the 14th international joint conference on artificial intelligence, pp. 1878–1885. Morgan Kaufmann, San Francisco (1995)

[15] Fung, R., Chang, K.-C.: Weighing and integrating evidence for stochastic simulation in Bayesian networks. In: Henrion, M., Shachter, R.D., Kanal, L.N., Lemmer, J.F. (eds.) Proceedings of the 5th conference on uncertainty in artificial intelligence (1989), pp. 209–220. North-Holland, Amsterdam (1990)

[16] Getoor, L., Friedman, N., Koller, D., Pfeffer, A.: Learning probabilistic relational models. In: Džeroski, S., Lavrač, N. (eds.) Relational data mining, ch. 13, pp. 307–335. Springer, Berlin (2001)

[17] Getoor, L., Friedman, N., Koller, D., Pfeffer, A., Taskar, B.: Probabilistic relational models. In: [18], ch. 5, pp. 129–174 (2007)

[18] Getoor, L., Taskar, B. (eds.): Introduction to statistical relational learning. MIT Press, Cambridge (2007)
[19] Giry, M.: A categorical approach to probability theory. In: Banaschewski, B. (ed.) Categorical aspects of topology and analysis: Proceedings of an international conference held at Carleton University, Ottawa. Lecture Notes in Mathematics, vol. 915, pp. 68–85. Springer, Berlin (1982)
[20] Goodman, N.D., Mansinghka, V.K., Roy, D., Bonawitz, K., Tenenbaum, J.B.: Church: A language for generative models. In: McAllester, D.A., Myllymäki, P. (eds.) 2008: Proceedings of the 24th conference in uncertainty in artificial intelligence, pp. 220–229. AUAI Press (2008)
[21] Gunter, C.A., Rémy, D., Riecke, J.G.: A generalization of exceptions and control in ML-like languages. In: Peyton Jones, S.L. (ed.) Functional programming languages and computer architecture: 7th conference, pp. 12–23. ACM Press, NewYork (1995)
[22] Halpern, J.Y.: An analysis of first-order logics of probability. Artificial Intelligence 46, 311–350 (1990)
[23] Haynes, C.T.: Logic continuations. Journal of Logic Programming 4(2), 157–176 (1987)
[24] Hinze, R.: Deriving backtracking monad transformers. In: ICFP 2000: Proceedings of the ACM international conference on functional programming. ACM SIGPLAN Notices, vol. 35(9), pp. 186–197. ACM Press, New York (2000)
[25] Hudak, P.: Building domain-specific embedded languages. ACM Computing Surveys 28(4es), 196 (1996)
[26] Hughes, J.: The design of a pretty-printing library. In: Jeuring, J., Meijer, E. (eds.) AFP 1995. LNCS, vol. 925, pp. 53–96. Springer, Heidelberg (1995)
[27] Jaeger, M., Lidman, P., Mateo, J.L.: Comparative evaluation of probabilistic logic languages and systems: A work-in-progress report. In: Proceedings of mining and learning with graphs (2007), `http://www.cs.aau.dk/~jaeger/plsystems/`
[28] Kersting, K., De Raedt, L.: Bayesian logic programming: Theory and tool. In: [18], ch. 10, pp. 291–321 (2007)
[29] Kiselyov, O.: Native delimited continuations in (byte-code) OCaml (2006), `http://okmij.org/ftp/Computation/Continuations.html#caml-shift`
[30] Koller, D., McAllester, D.A., Pfeffer, A.: Effective Bayesian inference for stochastic programs. In: AAAI 1997: Proceedings of the 14th national conference on artificial intelligence. The American Association for Artificial Intelligence, pp. 740–747. AAAI Press, Menlo Park (1997)
[31] Landin, P.J.: The next 700 programming languages. Communications of the ACM 9(3), 157–166 (1966)
[32] Milch, B., Marthi, B., Russell, S., Sontag, D., Ong, D.L., Kolobov, A.: BLOG: Probabilistic models with unknown objects. In: [18], ch. 13, pp. 373–398 (2007)
[33] Mogensen, Æ.T.: Self-applicable online partial evaluation of the pure lambda calculus. In: Proceedings of the 1995 ACM SIGPLAN symposium on partial evaluation and semantics-based program manipulation, pp. 39–44. ACM Press, New York (1995)
[34] Muggleton, S., Pahlavi, N.: Stochastic logic programs: A tutorial. In: [18], ch. 11, pp. 323–338 (2007)
[35] Murphy, K.: Bayes Net Toolbox for Matlab (2007), `http://www.cs.ubc.ca/~murphyk/Software/BNT/bnt.html`
[36] Murphy, K.: Software for graphical models: A review. International Society for Bayesian Analysis Bulletin 14(4), 13–15 (2007)

[37] Pearl, J.: Probabilistic reasoning in intelligent systems: Networks of plausible inference. Morgan Kaufmann, San Francisco (1988) (Revised 2nd printing, 1998)
[38] Pfeffer, A.: The design and implementation of IBAL: A general-purpose probabilistic language. In: [18], ch. 14, pp. 399–432 (2007)
[39] Pfeffer, A.: A general importance sampling algorithm for probabilistic programs. Tech. Rep. TR-12-07, Harvard University (2007)
[40] Phillips, A., Cardelli, L.: Efficient, correct simulation of biological processes in the stochastic pi-calculus. In: Calder, M., Gilmore, S. (eds.) CMSB 2007. LNCS (LNBI), vol. 4695, pp. 184–199. Springer, Heidelberg (2007)
[41] Poole, D., Mackworth, A.: Artificial intelligence: Foundations of computational agents. Cambridge University Press, Cambridge (2009)
[42] Ramsey, N., Pfeffer, A.: Stochastic lambda calculus and monads of probability distributions. In: POPL 2002: Conference record of the annual ACM symposium on principles of programming languages, pp. 154–165. ACM Press, New York (2002)
[43] Sato, T.: A glimpse of symbolic-statistical modeling by PRISM. Journal of Intelligent Information Systems 31(2), 161–176 (2008)
[44] Shachter, R.D., Peot, M.A.: Simulation approaches to general probabilistic inference on belief networks. In: Henrion, M., Shachter, R.D., Kanal, L.N., Lemmer, J.F. (eds.) Proceedings of the 5th conference on uncertainty in artificial intelligence (1989), pp. 221–234. North-Holland, Amsterdam (1990)
[45] Sutton, C., McCallum, A.: An introduction to conditional random fields for relational learning. In: [18], ch. 4, pp. 93–127 (2007)
[46] Taskar, B., Abbeel, P., Wong, M.-F., Koller, D.: Relational Markov networks. In: [18], ch. 6, pp. 175–199 (2007)
[47] Thiemann, P.: Combinators for program generation. Journal of Functional Programming 9(5), 483–525 (1999)
[48] Wadler, P.L.: The essence of functional programming. In: POPL 1992: Conference record of the annual ACM symposium on principles of programming languages, pp. 1–14. ACM Press, New York (1992)
[49] Wand, M., Vaillancourt, D.: Relating models of backtracking. In: ICFP 2004: Proceedings of the ACM international conference on functional programming. ACM Press, New York (2004)