What are the Odds? Probabilistic Programming in Scala

Sandro Stucki* Nada Amin* Manohar Jonnalagedda*
Tiark Rompf‡*

*EPFL, LAMP
{first.last}@epfl.ch

‡Oracle Labs
{first.last}@oracle.com

ABSTRACT

Probabilistic programming is a powerful high-level paradigm for probabilistic modeling and inference. We present *Odds*, a small domain-specific language (DSL) for probabilistic programming, embedded in Scala. Odds provides first-class support for random variables and probabilistic choice, while reusing Scala's abstraction and modularity facilities for composing probabilistic computations and for executing deterministic program parts. Odds accurately represents possibly dependent random variables using a probability monad that models committed choice. This monadic representation of probabilistic models can be combined with a range of inference procedures. We present engines for exact inference, rejection sampling and importance sampling with look-ahead, but other types of solvers are conceivable as well. We evaluate Odds on several non-trivial probabilistic programs from the literature and we demonstrate how the basic probabilistic primitives can be used to build higher-level abstractions, such as rule-based logic programming facilities, using advanced Scala features.

Categories and Subject Descriptors

D.3.m [**Software**]: Programming Languages—*Miscellaneous*

General Terms

Languages

Keywords

Probabilistic programming, probability monad, Scala, EDSL, probabilistic inference

1. INTRODUCTION

Probabilistic models and probabilistic inference form the core of the machine learning algorithms that enable sophisticated technology such as self-driving cars, natural language processing systems, or recommender engines. However, building probabilistic reasoning systems is hard and requires expertise across several disciplines. The goal of probabilistic programming is to drastically simplify this task by expressing probabilistic models as high-level programs. Probabilistic programming languages like Church [6], Hansei [7], BLOG [11] or Figaro [15] provide abstractions to represent and manage uncertain information in addition to the usual deterministic control and data abstractions expected from any high-level programming language. For example, in a probabilistic program, an unknown quantity can be represented as a random variable. The language implementation then provides a range of inference procedures to solve the model, i.e. determine probable values for the random variables in the program that represent the model. Thus, probabilistic programming shares some obvious similarities with logic programming.

In this paper, we present *Odds*, an embedded domain-specific language (DSL) for probabilistic programming in Scala. Odds is not a closed system but rather a library that extends Scala with support for first-class random variables, represented as values of type `Rand[T]`. Random variables are created by invoking a probabilistic choice operator that corresponds to a certain probability distribution. For example, the expression `flip(0.5)` creates a random variable of type `Rand[Boolean]` representing a fair coin toss. Random variables can also be combined to form new random variables. For example, given a pair of random variables `a`, `b` of type `Rand[Int]`, the term `a + b` denotes a new random variable with a distribution that depends on both `a` and `b`. Random variables form a monad, and deterministic operations like `+` on `Int` values can be lifted to the monadic level in the usual way. The semantics of random variables correctly models mutually dependent random variables, and demands that, for example, `a + a` will always be equal to `2 * a`. While this fact may seem self-evident, it does not hold for most previous embeddings based on simpler probability monads. A key aspect of our monad implementation is to maintain observable identities and committed choices for random variables, instead of modeling only probability distributions.

Contributions.

In summary, this paper makes the following contributions:

- We present *Odds*, an embedded DSL that extends Scala with first-class random variables `Rand[T]` and probabilistic choice operators.
- We present a monadic interface for probabilistic choice

Scala '13, Montpellier, France

that accurately represents mutually dependent random variables in a referentially transparent way as monad instances, with observable identities and committed choices for random variables. Conceptually, our monad represents lazy weighted non-determinism with call-time choice, similar to those described in [4] and [15].

- Inspired by previous work on the Hansei DSL [7], we implement several probabilistic inference strategies, including rejection sampling, exact inference and importance sampling with look-ahead. These inference algorithms can be used interchangeably or even together in a single probabilistic program.
- We evaluate Odds by implementing several probabilistic programs of realistic complexity from the literature.
- We show how advanced Scala features can be used to build even higher level abstractions on top of the probabilistic primitives. As an example, we present a rule-based logic programming system using virtualized pattern matching.

Odds builds on a large body of previous work on embedded probabilistic programming. Much of the design is inspired by the probabilistic programming DSL Hansei [7], although there are some key differences. The most obvious difference is that Hansei embeds probabilistic choice shallowly, using continuations instead of explicit monad instances, and without distinguishing random variables from deterministic values in the type system.

One of Odds' key design goals is to enable programmers to focus on specifying the actual probabilistic model without worrying about secondary operational aspects such as when and what to memoize; decisions which often need to be made explicitly in other systems.

The rest of this paper is structured as follows: We review the challenges of embedded probabilistic programs as a problem statement in Section 2. We provide introductory programming examples in Section 3. We present key ideas of the implementation in Section 4. We show a rule-based logic programming built on top of Odds in Section 6. We discuss our evaluation in Section 5, related work in Section 7, and future work and conclusion in Section 8.

2. EMBEDDINGS AGAINST ALL ODDS

Probabilistic computation does not exist in a vacuum; large parts of any probabilistic program are deterministic. Moreover, probabilistic programs should be able to benefit from high-level language abstractions like objects, classes or higher order functions, which do not directly contribute to the probabilistic computation but are essential for modularity, composition and reuse of program functionality.

Compared to stand-alone solutions, embedding probabilistic programming in a general-purpose language such as Scala has the decisive advantage of linguistic reuse: all of the host language's abstraction capabilities are immediately available to the deterministic parts of a computation. Following an embedded approach also means that probabilistic programming is easily accessible to programmers already familiar with the host language and with the basic concepts of probability.

Embedding strategies in general come in two flavors, *shallow* or *deep*. A shallow embedding identifies random values with regular values of the host language. For example, a probabilistic computation over integers will just have type `Int`. A deep embedding, by contrast, assigns a separate type to probabilistic values, for example `Rand[Int]`. While a shallow embedding is more seamless and may seem more intuitive at first, it blurs the distinction between the underlying mathematical concepts of *outcomes*, deterministic values that can be part of a probability distribution and *random variables*, entities of uncertainty that can take on outcomes according to their associated probability distribution.

The type-level distinction of a deep embedding models these concepts more accurately and makes it possible to treat random variables as first class objects without necessarily observing their outcomes (probabilistic meta-programming). For example, we can easily define combinator functions on random variables or store random variables in data structures without introducing additional uncertainty.

A key challenge in designing a deep embedding is to prevent the embedding abstraction from leaking, i.e. making sure that the semantics of the embedded language align well with the host language. This is especially true if the model of evaluation of the embedded language and the host language is quite different, as is the case here.

A common approach to build deep embeddings is to use some form of monad. For probabilistic programming, several variants of probability monads have been proposed [16, 5, 3, 16]. What is common to all of them is that the monad makes probabilistic choice points explicit, and sequences probabilistic computations using the monadic bind operator. This works very well as long as the whole computation stays inside the monad, but things get significantly more complicated when we combine host language computations and monadic computations. The root issue is that we are still working with two separate languages, the probabilistic one inside the monad, and the host language outside. Referential transparency and desirable equational properties that hold within the monad do not easily carry over if we start composing instances of the monad, which are first-class host language values. A particular troublesome issue appears when dealing with random variables that may be correlated, i.e. are not independent.

To substantiate this discussion with an example, let us lift the integer + operation to random variables:

```
def infix_+(r1: Rand[Int], r2: Rand[Int]) =
  for (v1 <- r1; v2 <- r2) yield v1 + v2
```

The above definition uses Scala's **for** notation, which is syntactic sugar for the monadic bind operation, and additional support for defining infix methods provided by Scala-Virtualized [17]. The lifted addition works nicely when invoked on *independent* random variables, but given its type signature, we would expect it to be generic and work for all pairs of random variables. It is instructive to see what happens in the following case, where `uniform` models a discrete random choice with uniform probabilities:

```
val r = uniform(0, 1)
r + r
```

Mathematically, if r is a random variable, we would expect $r+r$ to be equal to $2r$, with an outcome of either 0 or 2 with equal probability. However, what we get instead is the sum of two *independent*, identically distributed indicators, with a combined outcome of either 0, 1, 2 with respective proba-

bilities $\frac{1}{4}$, $\frac{1}{2}$, $\frac{1}{4}$. To obtain the desired result and achieve the intuitive equational reasoning, one would have to explicitly bind the r once on the monadic level:

```
for (r <- uniform(0, 1)) yield (r + r)
```

This means that in practice, the whole program needs to be transformed to monadic style, which breaks modularity and encapsulation. Furthermore, even though there is a type-level distinction between probabilistic and regular computations, the types do not help in tracking this intuitive error and they do not offer any guidance on which composition operations are well behaved and which are not.

Shallow approaches, as exemplified by Hansei [7], are less prone to these issues. Since a `Rand[Int]` is just an `Int`, the program is conceptually "forked" at each probabilistic choice point using continuations. Here, the evaluation order of probabilistic choices inherits exactly the evaluation order of the host language. However, the use of continuations comes at the expense of forking the *entire computation* right there, not just the probabilistic parts. To regain efficiency, this requires the programmer to explicitly delay or memoize certain parts of the model (`letlazy` operator in Hansei) to bring choice points and their observations closer together, and thus reduce the search space. In this paper, we are concerned with bringing reuse of evaluation order to monadic deep embeddings and achieve an intuitive behavior similar to shallow approaches, while retaining the deep embedding benefits like distinguishing between choices and observations on the type-level.

3. A TASTE OF ODDS

Let us solve a puzzle related to population estimation from the literature [11]:

> An urn contains an unknown number of balls– say, a number chosen from a uniform distribution. Balls are equally likely to be blue or green. We draw some balls from the urn, observing the color of each and replacing it. We cannot tell two identically colored balls apart; furthermore, observed colors are wrong with probability 0.2. How many balls are in the urn? Were the balls drawn twice?

We first define the models of this problem in Odds, independently of the inference engine. We then illustrate how to run the models. Finally, we discuss the intuition behind Odds' semantics.

3.1 Defining Models

We assume `Color` to be defined as a usual Scala algebraic datatype with two variants, `Blue` and `Green`, and a function `opposite_color` to toggle from one to the other. Now, we can define our first probabilistic function. The observation of color is faulty, with a 20% error rate:

```
def observed_color(c: Color): Rand[Color] =
  flip(0.8).map(if (_) c else opposite_color(c))
```

`flip(p)` returns a random variable of type `Rand[Boolean]` whose distribution is `true` with probability p. Since random variables are monads, we can use a monadic `map` to inspect the result of a flip.

We define some random *processes* which capture the priors of our problem. We assume that the number of balls is a random variable uniformly distributed between 1 and 8 and that the color of each ball is uniformly either `Blue` or `Green`. The true color of a ball never changes. So we create a sequence of independent random variables, one for each ball.

```
val nballs_max = 8
def nballs_prior(): Rand[Int] =
  uniform(1 to nballs_max : _*)
def ball_colors_prior(): IndexedSeq[Rand[Color]] =
  (0 until nballs_max).map(_ => uniform(Blue, Green))
```

Notice that the prior for the ball colors returns an `IndexedSeq[Rand[Color]]` not a `Rand[IndexedSeq[Color]]`. Each ball is represented by an integer from 0 until the number of actual balls. Since we know a bound on the number of actual balls, it is good enough and simpler to *fix* the length of the sequence, rather than to parametrize the random process generating the ball colors on the actual number of balls, which is itself random.

Now, we define the process of drawing a ball from an urn. The outcome of this process depends on the number of balls and the color of the balls. We use the monadic `bind` to sequence inspections on random variables. This process returns a random variable describing the index of the ball, its color, and its observed color.

```
def draw(nballs: Rand[Int], ball_colors: IndexedSeq[Rand[Color]]) = {
  for (n <- nballs;
       b <- uniform(0 until n: _*);
       c <- ball_colors(b);
       o <- observed_color(c)) yield (b, c, o)
}
```

Finally, we can complete our model to answer the problem. First, we want to know the random distribution of the number of balls that match a given sequence of observations:

```
def model_nballs(obs: IndexedSeq[Color]) = {
  val nballs = nballs_prior()
  val ball_colors = ball_colors_prior()
  def matches_draw(obs_color: Color) =
    for ((_, _, o) <- draw(nballs, ball_colors))
    yield o == obs_color
  nballs when forall(obs, matches_draw)
}
```

We can also answer the second question: "Were the balls drawn twice?" by probabilistically calculating a map `drawn: Rand[Map[Int,Int]]` from the ball's index to the (positive) number of times it was drawn. We then calculate the minimum value of this map – if it is 1, some ball was drawn only once.

```
def model_duplicate(obs: IndexedSeq[Color]) = {
  val nballs = nballs_prior()
  val ball_colors = ball_colors_prior()
  val drawn = obs.foldLeft(always(Map.empty[Int, Int])){
    (map,obs_color) =>
    for ((b, c, o) <- draw(nballs, ball_colors);
         if o == obs_color;
         m <- map)
    yield m.updated(b, m.getOrElse(b, 0)+1)
  }
  drawn.map(_.values.min)
}
```

3.2 Running Models

All the code from the previous example is packaged in a Scala trait, say `ColoredBalls`, to be mixed in with the trait `OddsLang` provided by Odds.

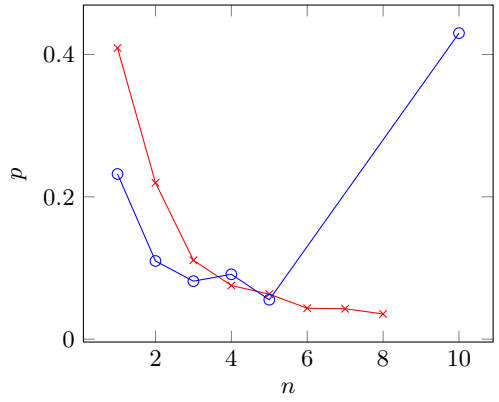

Figure 1: Observing ten blue balls with importance sampling: (1) plots the probability of n balls in the urn; (2) plots the probability of at least n duplicate draws for each ball drawn.

```
trait ColoredBalls extends OddsLang with OddsPrettyPrint {
  // ... code from previous section here ...
  val observations = (1 to 10).map(_ => Blue)
  def ask1 = model_nballs(observations)
  def ask2 = model_duplicate(observations)
}
```

We have also added some default queries for the observations "10 blue balls".

Now, we can run this model by mixing in a trait implementing an inference strategy–for example, importance sampling with look-ahead:

```
new ColoredBalls with LocalImportanceSampling {
  show(normalize(sample(10000, 3)(ask1)), "ask1")
  show(normalize(sample(20000, 3)(ask2)), "ask2")
}
```

The results are plotted in Figure 1.

In addition to importance sampling, Odds provides other strategies for inferring the distribution of a model including rejection sampling and exact inference. Pluggable inference engines, with the same semantics, but different accuracies and capabilities, compelled us towards an intuitive interpretation of embedded probabilistic programs in symbiosis with the host language.

3.3 Intuitive Semantics

Odds' semantics matches that of Scala: a **val** of type `Rand[A]` represents a random *variable*, which is perfectly correlated with itself on each read, while a **def** of *return* type `Rand[A]` represents a random *process*, evaluated anew at each invocation. In the colored balls model, we use both random variables and processes, and seamlessly combine them with deterministic Scala data structures. For example, we represent that balls have fixed colors by associating each ball with a random variable `Rand[Color]`. Since random variables are first-class values, we can keep this association in a regular Scala data structure.

Probabilistic programs return random variables; they are otherwise deterministic programs that make probabilistic choices (such as flipping a coin) every now and then. Since the result of these probabilistic choices may influence the control flow of the program, the results of the different evaluations of the program may differ even for identical inputs. Hence a probabilistic program can also be viewed as a generative processes for a certain probability distribution.

While, in general, some evaluations of a program might not terminate, we will assume that the marginal probability over all such evaluations, i.e. their total probability mass, is zero. Probabilistic programs that terminate with probability one are sometimes called *admissible* [6].

While we expect programs to terminate with probability one, we do not require them to always return a result. An evaluation that does not return a result is said to *fail*. This is useful when designing programs that return *conditioned* random variables, for example, if we want to refine a probabilistic model in light of *empirical observations*. For example, in the colored balls model, we used the `when` operator to rule out scenarios that did not match the observed sequence of colors. Internally, `when` uses a failing computation (embodied by the `never` primitive) whenever its condition does not hold:

```
def infix_when(x: => Rand[Boolean], y: Rand[Boolean]): Rand[A] =
  y flatMap {
    case true  => x
    case false => never
  }
```

The notion of a probabilistic program as a process that returns a random variable suggests a simple algorithm for approximating its distribution: we evaluate the program a certain number of times and build a histogram over the outcomes of the individual runs. Failing runs are not counted in the histogram. If we normalize the histogram by the total number of successful runs, we expect the result to converge to the desired distribution as the number of runs goes to infinity. This method is generally known as *rejection sampling*.

While rejection sampling is an inefficient approximation method, it describes in an intuitive and precise manner the generative process represented by a probabilistic program, and we will therefore adopt it as a model for the operational semantics of the latter. Alternate inference methods presented in this paper should of course converge to the same distribution as that obtained by rejection sampling in the limit but they will be chosen to also to preserve the operational behavior of rejection sampling in as far as it is observable from within a purely functional probabilistic computation.

4. INSIDE ODDS

At the heart of this section, we explain our "committed choice" model, which enables Odds to integrate probabilistic monadic programming with the native evaluation order of the host language. As stepping stones, we summarize Odds' monadic interface and then review rejection sampling, and a classical approach to implementing a probability monad. We conclude this section by touring pluggable inference engines.

4.1 Odds API

The specification of an Odds program is separate from its implementation. The type `Rand[A]` representing a random variable of outcome of type `A` is kept as an abstract zero-plus monad in the API, against which Odds programs are specified. The function `choice` is the primitive discrete random process, yielding fresh random variables according to a given distribution over values of some type `A`. Other discrete random processes, including `always` (the monadic return) `never`

(the monadic zero) are defined in terms `choice`.

```
type Prob = Double
type Dist[+A] = Iterable[(A, Prob)]

type Rand[+A] <: RandIntf[A]

trait RandIntf[+A] {
  def flatMap[B](f: A => Rand[B]): Rand[B]
  def map[B](f: A => B): Rand[B] =
      flatMap(x => always(f(x)))
}

def choice[A](xs: (A, Prob)*): Rand[A]

def always[A](x: A) = choice(x -> 1.0)
def never = choice()
def flip(p: Double): Rand[Boolean] =
    choice(true -> p, false -> (1-p))
def uniform[A](xs: A*): Rand[A] =
    choice(xs.map((_, 1.0 / xs.size)):_*)
```

In rejection sampling, the `Rand[A]` monad is conceptually just the `Option[A]` monad representing a single concrete outcome or a failure. The `choice` function simply samples the given distribution. A probabilistic program is run many times to get an approximate distribution of its outcomes.

4.2 A classical probability monad

We briefly describe a "classical" probability monad [16, 5, 3]. Conceptually, `Rand[A]` represents a random *process* yielding an independent random *variable* at each monadic bind. Thus, the whole program must be in monadic style to correctly track dependencies between random variables.

```
type Rand[+A] = RandVar[A]
final case class RandVar[+A](dist: Dist[A])
  extends RandIntf[A] {
    def flatMap[B](f: A => Rand[B]): Rand[B] = RandVar(
      for ((v, p) <- dist; (w, q) <- f(v).dist)
      yield (w, q * p))
}

def choice[A](xs: (A, Prob)*): Rand[A] = RandVar(xs)
def reify[A](x: Rand[A]): Dist[A] = consolidate(x.dist)
```

The probability monad wraps the distribution associated with a random variable X, and its monadic bind, the `flatMap` method, computes functions on X by marginalizing over X using the chain rule:

$$\Pr\{f(X) = y\} = \sum_x \Pr\{f(X) = y \,|\, X = x\} \Pr\{X = x\}$$

The `reify` function reifies a given random variable `x`, by returning the distribution encapsulated by `x`. The `consolidate` function sums the weights of equal outcomes.

4.3 Committed Choice

Committed choice (also known as *call-time choice*) restores the intuitive semantics that `Rand[A]` represents a random variable that is perfectly correlated with itself. The term "itself" already indicates that we need to be able to observe identities of random variables. The key idea is to assign a *unique ID* to every probabilistic choice and to register the history of choices made by a given program evaluation in an *environment* addressed by choice IDs. Each choice may split the program evaluation into multiple paths. Depending on the inference strategy, several such paths may be followed at the same time. In this case, each of them will maintain a separate "thread local" environment to record choices as they are committed.

In the construction of `Rand[A]` monad instances, we make crucial use of side effects outside of the `Rand` monad to maintain a global counter as a source of unique IDs. Once monad instances are created, all further monadic computation is purely functional and free from side-effects.

Operations in a monad are sequenced through the monadic *bind* operator. In our case, this means that the evaluation order of a probabilistic program is determined by the `flatMap` calls on its `Rand` instances. For our `Rand` monad to work as expected, environments containing committed choices need to be passed between `Rand` instances in evaluation order, that is, from the receiver object of a particular call to `flatMap` to its argument. To decouple the inference algorithm from the monad representation, we use a *delayed evaluation* approach, which allows us to explore possible evaluation paths of a probabilistic program like a search tree during inference. Inspired by the Hansei language [7], this design enables traversals in different orders, and in a lazy fashion, which is a prerequisite for supporting different inference engines. In the more general context of Functional Logic Programming (FLP), our `Rand[T]` monad thus represents a particular variant of *weighted non-determinism* with *call-time choice* and implicit *sharing* [4], and fulfills the corresponding equational laws.

We define our extended probability monad as follows:

```
type Rand[+A] = RandVar[A]

sealed abstract class RandVar[+A] extends RandIntf[A] {
  def flatMap[B](f: A => Rand[B]): Rand[B] =
      RandVarFlatMap(this, f)
}
final case class RandVarChoice[+A](dist: Dist[A])
  extends RandVar[A] with CommittedChoice[A]
final case class RandVarFlatMap[+A, B](
  x: RandVar[B], f: B => Rand[A])
  extends RandVar[A] with CommittedChoice[Rand[A]]

def choice[A](xs: (A, Prob)*): Rand[A] =
    RandVarChoice(xs)
```

Our probability monad is an algebraic data type consisting of application nodes for `choice` and `flatMap`. Calling either one of these functions simply allocates a corresponding node. The `CommittedChoice` trait mixed into the node classes provides a convenient interface for registering choices in an *environment*. While `RandVarChoice` nodes commit to particular values of the underlying distribution, `RandVarFlatMap` nodes commit to the result of applying the closure `f` to a particular choice of bound variable.

Exact Inference

We can then implement exact inference as follows:

```
type Environment = Map[Int, Any];

def explore[A, B](
  x: RandVar[A], p: Prob, env: Environment)(
  cont: (A, Prob, Environment) => Dist[B]): Dist[B] =
    x match {
      case x @ RandVarChoice(dist) =>
        x.choice(env) match {
          case Some(v) => cont(v, p, env)
          case None    => dist flatMap {
            case (v, q) =>
              x.withChoice(env, v) { e =>
                cont(v, p * q, e)
              }
```

```
      }}
    case t @ RandVarFlatMap(x, f) =>
      explore(x, p, env) { (y, q, e) =>
        t.choice(e) match {
          case Some(r) => explore(r, q, e)(cont)
          case None    =>
            val r = f(y)
            t.withChoice(e, r) {
              e1 => explore(r, q, e1)(cont)
            }
  }}}

def reify[A](x: RandVar[A]): Dist[A] =
  consolidate(explore(x, 1, Map()) {
    (y, p, e) => Iterable(y -> p)
  })
```

The exact inference algorithm walks the `Rand` tree to reify the probabilistic computation it represents, committing choices as it traces a given evaluation history.

The core of the algorithm is the `explore` function which reifies the application node at the top of a `RandVar` tree and calls the continuation `cont` of the application with the concrete values of the resulting distribution. The `choice` and `withChoice` methods are part of the `CommittedChoice` trait and are used to look up and register choices in an environment. Whenever `explore` encounters a node that represents a committed choice, it simply passes the committed value on to its continuation `cont`, rather than recomputing it.

Delayed Evaluation

The delayed evaluation of `RandVar` trees has the added benefit that random choices are delayed until the point where the choice and commitment to a concrete value is actually required rather than at the point of definition of a random variable. This can result in a considerable reduction of the number of histories that need to be explored to reify a random computation. The following example adopted from [7] illustrates this point:

```
// Flip `n` coins.
def flips(p: Prob, n: Int): List[Rand[Boolean]] =
  n match {
    case 0 => Nil
    case n => flip(p) :: flips(p, n - 1)
  }

// Check whether a list of coins are all `true`.
def trues(cs: List[Rand[Boolean]]): Rand[Boolean] =
  cs match {
    case Nil     => always(true)
    case c :: cs => c && trues(cs)
  }

val flips20 = reify(trues(flips(0.5, 20)))
```

If `choice` were to commit to choices eagerly, the above model would result in a search tree with 2^{20} leaves because every possible combination of the 20 coin-tosses generated by `flips` would be explored. Using delayed evaluation, the total number of choices to considered by our exact inference algorithm reduces to 40, because the `&&` operation in `trues` short-circuits most evaluations.

4.4 Other Inference Strategies

Apart from rejection sampling and exact inference, Odds provides two additional inference algorithms: *depth-bounded inference* and *importance sampling with look-ahead*. Depth-bounded inference can be used to shallowly explore the search tree programs, while importance sampling with look-ahead yields better approximations of the probability of unlikely outcomes of programs.

Depth-bounded inference

As the name implies, depth-bounded inference is a technique to traverse the search tree of a probabilistic program down to a limited depth. This can be useful in order to explore the subset of the distribution of a probabilistic program over its most likely outcomes. The actual depth to which the search tree is explored depends on the desired number of outcomes in the resulting distribution, or on an upper bound on the probability mass of the unexplored branches in the search tree.

For example, consider the following definition of a probabilistic list of coin tosses:

```
def randomList(): Rand[List[Boolean]] = flip(0.5) flatMap {
  case false => always(Nil)
  case true  => for {
    head <- flip(0.5)
    tail <- randomList()
  } yield head :: tail
}
```

The random lists returned by `randomList` can have any length, but longer lists are less likely.

Given the above definition, we can infer the probability that the concatenation of a random pair `(x, y)` of lists will match a given, finite sequence – say `(true, true, false)`:

```
def infix_++[T](
  x: Rand[List[T]], y: Rand[List[T]]): Rand[List[T]] =
    for (xv <- x; yv <- y) yield xv ++ yv

val x = randomList()
val y = randomList()
val xy = x ++ y
val p = (x, y) when (xy === always(List(true, true, false)))
```

It is easy to see that there are at most four such pairs. Yet, our exact inference algorithm would attempt to compare every value in the support of `xy` to our query sequence, immediately discarding all but four of them. Using depth-bounded inference, we can instead ask for the exploration to stop once the four solutions have been discovered:

```
scala> p.reify(4)
res4: (test.Dist[(List[Boolean], List[Boolean])], test.Prob) =
  (Map((List(true), List(true, false))   -> 0.00390625,
       (List(true, true), List(false))   -> 0.00390625,
       (List(true, true, false), List()) -> 0.00390625,
       (List(), List(true, true, false)) -> 0.00390625),
       0.296875)
```

Importance sampling with look-ahead

An inherent problem of rejection sampling is that it will favor evaluations with high prior probabilities, even though they might eventually fail. Since failing evaluations are not counted as samples towards the final distribution, the resulting approximation might be very poor even for a large number of evaluations. Importance sampling [14, 7] aims to circumvent this problem by excluding early failing evaluations from the sampling procedure.

We implement *importance sampling with look-ahead*, a variant of importance sampling introduced in [7]. Importance sampling with look-ahead combines rejection sampling with depth-bounded inference. The sampler first explores the search tree shallowly, collecting all the leaf nodes it en-

counters, and then picks one of the unexplored sub-trees at random according to the probability of its root node. In this way, early failing branches are excluded quickly.

Consider the following example adopted from [7]:

```
def drunkCoin(): Rand[Boolean] = {
  val toss = flip(0.5)
  val lost = flip(0.9)
  lost flatMap {
    case true  => never
    case false => toss
  }
}

def dcoinAnd(n: Int): Rand[Boolean] = n match {
  case 1 => drunkCoin()
  case n => drunkCoin() && dcoinAnd(n - 1)
}
```

The `drunkCoin` process models a drunk person flipping a coin, with the coin getting lost in the process nine out of ten times. An experiment based on the outcome of ten coin tosses will therefore require many more tosses overall. As a consequence, the rejection sampler generally fails to observe a single streak of ten tosses all coming up heads in as many as 10000 samples:

```
scala> sample(10000)(dcoinAnd(10))
res0: DrunkCoinModel.Dist[Boolean] = Map(false -> 537.0)
```

However, using importance sampling with a look-ahead of four, we can reliably approximate the correct distribution `(false -> 0.99999, true -> 1.85546e-12)` to within 10^{-9} using half the number of samples:

```
scala> normalize(sample(5000,4)(dcoinAnd(10)))
importance sampler: 5000 samples, 22492 solutions.
res7: DrunkCoinModel.Dist[Boolean] =
  ArrayBuffer((false, 0.9999999999979218),
              (true,  2.0781256234326196E-12))
```

5. EVALUATION

We evaluated Odds by modeling several non-trivial examples from the literature, notably Pfeffer's music model [14], and Milch et al.'s radar tracking of aircraft [11]. The source code of these models is available online together with the core Odds sources[1]. Since these examples also exist in Hansei[2], they offer us a nice opportunity to compare both DSLs from a user's perspective.

Music model

Pfeffer's music model studies the evolution of melodic motives in classical music. A melody is a sequence of notes, and can evolve either by notes being removed, inserted, or transformed according to a set of stochastic rules. The model then seeks to understand the likelihood of a certain sequence being an evolution of another sequence.

Of programmatic interest in this example is the use of lazy lists for modeling the evolution of a musical sequence. With Odds, because `Rand[A]` is lazy in the sense that it is evaluated only when `reify` is called (as explained in section 4.3), using normal Scala lists works. We implemented the model both with `Rand[List]` and a custom lazy list implementation. The custom lazy lists have the advantage of short-circuiting computation at the level of the `Rand` monad itself, i.e. before reification. This can potentially reduce the search space to be explored during reification.

Aircraft

The aircraft example is a more complex version of the balls example presented in the section 3. Instead of balls of different colors, we have airplanes on an observable grid, which change position according to stochastic rules. To observe them, the equivalent of drawing balls is an imperfect radar system. The system triggers alarms with a certain probability when it detects a plane; multiple planes in the same region trigger a single blip, and false alarms can also be triggered. This model investigates how many planes a grid contains, given a certain number of blips. Further complexity can be added to the system by modeling entrance and departure of airplanes from the grid.

In this example, we use classes to model plane states and plane co-ordinates, and operate on their stochastic versions.

Experience report

Developing both the above models gives us some insight into how intuitive it is to use Odds. The fact that both models are based on existing implementations in Hansei also allows us to compare and contrast the two languages.

On the one hand, the Odds programs allow programmers to rely on regular Scala facilities like **`val`** and **`def`** to control evaluation order and sharing. Having a type `Rand[T]` to represent random variables is another advantage: it is easier to focus on the "domain-specific" aspects of developing the models once the basic elements of the model are defined as random variables. The actual implementation of the models then consists mainly in using monadic combinators on `Rand[T]` as well as the usual library operations on deterministic data structures in Scala. The Scala type checker helps in detecting improper use of random variables in deterministic contexts quickly.

Hansei, on the other hand, does not distinguish between probabilistic and deterministic computations, and hence random variables do in general not require special typing. In practice, however, random variables are often distinguished through constructs like `letlazy`; as mentioned in section 2, this is an optimization for pruning the search tree. The optimization is essential for operating with realistic examples. While this can be converted into a rule of thumb, our experience suggests that letting Odds' semantics and Scala's type system take care of evaluation is more intuitive for a non-experienced user.

A common difficulty we experienced was the need to manage the lifting of structures to their stochastic representation. For example, it is intuitive to represent the state of a plane in the aircraft example using case classes in Scala:

```
case class PlaneState(idx: Int, p: (Int,Int), dir: Dir)
```

As the state of a plane evolves stochastically, we need to operate over `Rand[PlaneState]`, such that the fields of the state are random variables themselves. We are therefore required to write quite a bit of boilerplate code around `PlaneState` in order to use it with random variables. In Hansei, lazy data structures are necessary to improve performance.

If constructs such as `letlazy` are not needed, Hansei's very shallow embedding without type-level distinction between `Rand[Int]` and `Int` feels very seamless, whereas Odds requires programmers to either use explicit monadic style or to imple-

[1] `https://github.com/sstucki/odds/`
[2] `http://okmij.org/ftp/kakuritu/`

ment lifted versions of the corresponding operations (a task which might be simplified with macros). These observations mirror the expected trade-offs identified in Section 2. In addition, we can take advantage of Scala's implicit conversions for automatically lifting data structures into the `Rand` monad, thereby making their integration with the language more seamless to the user. We have defined such conversions for tuples (as seen in the example in section 3) and the same technique can readily be applied to other data structures. Hence this remains an engineering issue rather than a technical one. For the development of large-scale models, both languages could profit from a standard library of ready-to-use, special-purpose data structures, optimized for probabilistic programming.

Appealing to the "principle of least surprise", we believe the proper placement of operators to control evaluation order and memoization is inherently more difficult than following the guidance of the type system and switching to monadic style were needed.

Internally, Odds uses inference algorithms inspired by Hansei and achieves comparable performance. Since the more advanced inference modules in Odds are still undergoing active developed, we refrain from a more rigorous performance comparison here.

6. EXTENSIONS

We have shown in the preceding section that Odds can express relevant probabilistic programs from the literature. But we can also use the core Odds primitives to build even higher level abstractions. We give one example based on Scala-Virtualized [17]: using virtualized pattern matching, which redefines pattern match expressions as operations on a user-defined zero-plus monad, we can implement a rule-based probabilistic logic programming system. We omit some details here and refer to [17] for a more thorough treatment.

Using the Odds monad as the base for pattern matching expressions, we define a probabilistic extractor `Rule` that wraps a given random process:

```
implicit class Rule(f: String => Rand[String]) {
  def unapply(x: String): Rand[String] = f(x)
}
```

We continue by defining some actual probabilistic rules, assuming an implicit lifting from `String` to `Rand[String]` values:

```
val Likes: Rule = { x: String => x match {
  case "A" => "Coffee"
  case "B" => "Coffee"
  case "D" => "Coffee"
  case "D" => "Coffee" // likes coffee very much!
  case "E" => "Coffee"
}}
val Friend: Rule = { x: String => x match {
  case "A" => "C"
  case "A" => "C" // are really good friends!
  case "C" => "D"
  case "B" => "D"
  case "A" => "E"
}}
```

Since pattern matching is probabilistic, it may explore multiple branches. For example, "A" occurs three times on the left-hand side in rule `Friend`. Repeating a case like "A","C" will double its probability. Rules can also be recursive, which is required e.g. to define reflexive transitive closures:

```
val Knows: Rule = { x: String => x match {
  case Friend(Knows(y)) => y
  case x => x
}}
```

In general, rules compose in an intuitive way:

```
val ShouldGrabCoffee: Rule = { x: String => x match {
  case Likes("Coffee") && Knows(y @ Likes("Coffee")) if x != y =>
    x + " and " + y + " should grab coffee"
}}
```

This definition can be almost read out loud: if x likes coffee, and x knows someone else, y, who also likes coffee, then they should grab coffee together.

Evaluating this model yields a weighted list of coffee matchings:

```
A and D should grab coffee : 0.5714285714285714
B and D should grab coffee : 0.2857142857142857
A and E should grab coffee : 0.14285714285714285
```

In general, the computed weights correspond to relative frequencies and can be interpreted in an application-specific way.

7. RELATED WORK

Our work is very closely related to other functional approaches to probabilistic programming. Hansei [7] is a domain-specific language embedded in OCaml, which allows one to express discrete-distribution models with potentially infinite support, perform exact inference as well as importance sampling with look-ahead, and probabilistic reasoning about inference. Unlike Odds, Hansei does not distinguish between probabilistic and deterministic computations using types. Instead, Hansei uses a very shallow embedding, and implements probabilistic effects in direct style using *delimited continuations*. Church [6] is a universal probabilistic programming language, extending Scheme with probabilistic semantics, and is well suited for describing infinite-dimensional stochastic processes and other recursively-defined generative processes. Unlike Odds, Church is a dynamically-typed, standalone language.

Figaro [15] is another language embedded in Scala but takes an object-oriented approach: it is a library for constructing probabilistic models that also provides a number of built-in inference algorithms that can be applied directly to constructed models. Like Odds, Figaro uses a special type `Element[T]` to distinguish probabilistic computations with support type `T`. The `Element` type forms a probability monad similar to the `Rand` type in Odds, and both languages use Scala's built-in variable binding mechanism and hence track committed choices. However, Figaro programs are represented and built *explicitly* as data structures, that is, instances of appropriate subclasses of `Element`, such as `Flip`, `Apply`, `If`, etc. This allows a programmer to manipulate Figaro programs in Scala and to define custom subclasses of `Element` by overriding appropriate methods. In contrast, Odds programs are constructed *implicitly* through probabilistic operations like `always` or `choice`, **`for`** comprehensions, or operations on deterministic types that have been lifted into the `Rand` domain. The `Rand` type remains abstract in Odds programs until a particular inference algorithm is mixed in, and hence the programmer can not implement custom subclasses of `Rand`. Similarly, Figaro stores choices for its elements directly in the corresponding `Element` instance, while Odds only stores variable IDs in `Rand` instances and uses

separate per-branch environments to keep track of choices during inference. These differences are not accidental, they illustrate the different focus of the two languages: Odds aims to abstract implementation details as much as possible from probabilistic programs, while Figaro gives programmers control over the structures underlying a probabilistic program. Figaro also allows *conditions* and *constraints* to be defined for any element, which, among other things, allows the definition of cyclic models. Currently, no such mechanism exists in Odds.

A logic-programming based approach is taken by ProbLog, which is a probabilistic extension of Prolog based on Sato's distribution semantics [20]. While ProbLog1 focuses on calculating the success probability of a query, ProbLog2 can calculate both conditional probabilities and MPE states. BLOG [11], or Bayesian logic, is a probabilistic programming language with elements of first-order logic, as well as an MCMC-based inference algorithm. BLOG makes it relatively easy to represent uncertainty about the number of underlying objects explaining observed data.

BUGS is a language for specifying finite graphical models and accompanying software for performing B(ayesian) I(nference) U(sing) G(ibbs) S(ampling)[9].

There exist also many software libraries and toolkits for building probabilistic models [10, 13, 1].

8. CONCLUSIONS AND FUTURE WORK

In this paper, we have presented *Odds*, a small embedded (DSL) that provides first-class support for random variables and probabilistic choice. Odds programs can re-uses Scala's abstraction and modularity facilities for composing probabilistic computations, and the probabilistic semantics match that of Scala in an intuitive way. As part of the implementation, we have presented a novel probabilistic monad that accurately represents possibly dependent random variables by modeling committed choice. We have combined this monadic representation with a range of inference procedures, including exact inference, rejection sampling and importance sampling with look-ahead, and we have implemented several non-trivial probabilistic programs.

As part of our future work, we first of all aim to add better off-the-shelf support for lazy lists and lifting of structures to the stochastic world. We also want to explore using Lightweight Modular Staging (LMS) [18] to remove interpretive overhead in the inference algorithms by compiling inference procedures down to low-level code. We also want to use the Delite framework [2, 19, 8] to parallelize inference and run it on heterogenenous hardware like GPUs. In addition, we would like to explore synergies with OptiML [21], a machine learning DSL built on top of Delite. There is also more work to be done on implementing further inference procedures such as Markov-Chain Monte-Carlo (MCMC) solvers [12].

9. ACKNOWLEDGMENTS

The authors would like to thank Chung-chieh Shan, Oleg Kiselyov and the anonymous reviewers for their valuable feedback.

10. REFERENCES

[1] PyMC: Bayesian inference in Python. Online, 2012.

[2] K. J. Brown, A. K. Sujeeth, H. Lee, T. Rompf, H. Chafi, M. Odersky, and K. Olukotun. A heterogeneous parallel framework for domain-specific languages. PACT, 2011.

[3] M. Erwig and S. Kollmansberger. Functional pearls: Probabilistic functional programming in haskell. *J. Funct. Program.*, 16(1):21–34, 2006.

[4] S. Fischer, O. Kiselyov, and C.-c. Shan. Purely functional lazy non-deterministic programming. In *Proceedings of the 14th ACM SIGPLAN international conference on Functional programming*, ICFP '09, pages 11–22, New York, NY, USA, 2009. ACM.

[5] J. Gibbons. Unifying theories of programming with monads. In *Unifying Theories of Programming*, volume 7681, pages 23–67. Springer, 2013.

[6] N. D. Goodman, V. K. Mansinghka, D. M. Roy, K. Bonawitz, and J. B. Tenenbaum. Church: a language for generative models. In *Proc. of Uncertainty in Artificial Intelligence*, 2008.

[7] O. Kiselyov and C.-c. Shan. Embedded probabilistic programming. In W. M. Taha, editor, *Domain-Specific Languages*, volume 5658 of *Lecture Notes in Computer Science*, pages 360–384. Springer, 2009.

[8] H. Lee, K. J. Brown, A. K. Sujeeth, H. Chafi, T. Rompf, M. Odersky, and K. Olukotun. Implementing domain-specific languages for heterogeneous parallel computing. *IEEE Micro*, 31(5):42–53, 2011.

[9] D. J. Lunn, A. Thomas, N. Best, and D. Spiegelhalter. WinBUGS - A Bayesian modelling framework: Concepts, structure, and extensibility. *Statistics and Computing*, 10(4):325–337, Oct. 2000.

[10] A. McCallum, K. Schultz, and S. Singh. Factorie: Probabilistic programming via imperatively defined factor graphs. In *Adv. in Neural Inform. Processing Syst.*, volume 22, 2009.

[11] B. Milch, B. Marthi, S. Russell, D. Sontag, D. L. Ong, and A. Kolobov. BLOG: Probabilistic models with unknown objects. In L. Getoor and B. Taskar, editors, *Statistical Relational Learning*. MIT Press, 2007.

[12] B. Milch and S. Russell. General-purpose MCMC inference over relational structures. In *Proc. 22nd Conference on Uncertainty in Artificial Intelligence*, pages 349–358, 2006.

[13] T. Minka, J. Winn, J. Guiver, and D. Knowles. Infer.NET 2.5, 2012. Microsoft Research Cambridge. http://research.microsoft.com/infernet.

[14] A. Pfeffer. A general importance sampling algorithm for probabilistic programs. Tech. Rep. TR-12-07. Technical report, Harvard University, 2009.

[15] A. Pfeffer. Creating and manipulating probabilistic programs in figaro. UAI Workshop on Statistical Relational AI (StarAI), 2012.

[16] N. Ramsey and A. Pfeffer. Stochastic lambda calculus and monads of probability distributions. *Proc. of the 29th ACM SIGPLAN-SIGACT Symp. on Principles of Program. Lang.*, pages 154–165, 2002.

[17] T. Rompf, N. Amin, A. Moors, P. Haller, and M. Odersky. Scala-virtualized: Linguistic reuse for deep embeddings. In *Higher-Order and Symbolic Computation (Special issue for PEPM'12, to appear)*.

[18] T. Rompf and M. Odersky. Lightweight modular staging: a pragmatic approach to runtime code generation and compiled dsls. *Commun. ACM*, 55(6):121–130, 2012.

[19] T. Rompf, A. K. Sujeeth, H. Lee, K. J. Brown, H. Chafi, M. Odersky, and K. Olukotun. Building-blocks for performance oriented DSLs. DSL, 2011.

[20] T. Sato. Generative modeling by prism. In *ICLP*, pages 24–35, 2009.

[21] A. K. Sujeeth, H. Lee, K. J. Brown, T. Rompf, M. Wu, A. R. Atreya, M. Odersky, and K. Olukotun. OptiML: an implicitly parallel domain-specific language for machine learning. In *Proceedings of the 28th International Conference on Machine Learning*, ICML, 2011.