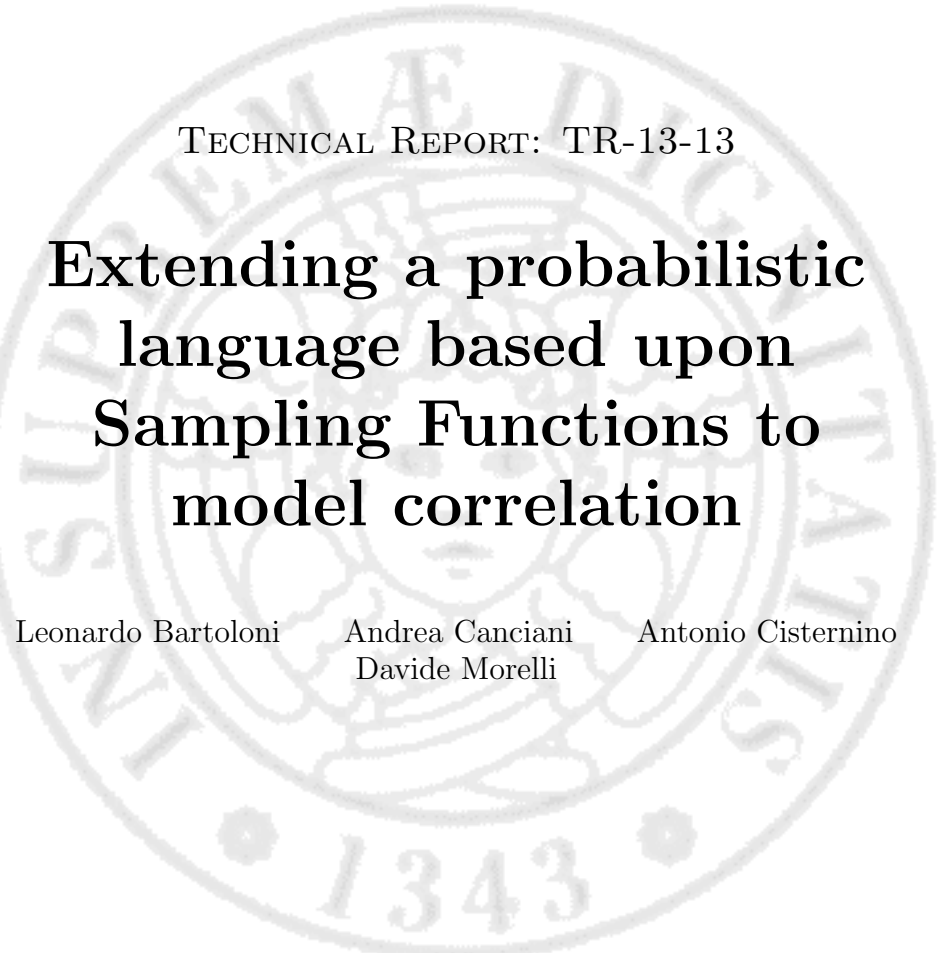
Università di Pisa
Dipartimento di Informatica



Technical Report: TR-13-13

Extending a probabilistic language based upon Sampling Functions to model correlation

Leonardo Bartoloni Andrea Canciani Antonio Cisternino
Davide Morelli

September 3, 2013

address: Largo B. Pontecorvo 3, 56127 Pisa, Italy. tel: +39 050 2212700 fax: +39 050 2212726

Extending a probabilistic language based upon Sampling Functions to model correlation

Leonardo Bartoloni Andrea Canciani Antonio Cisternino
Davide Morelli

September 3, 2013

Abstract

Probability is permeating many applications of computer science, ranging from probabilistic reasoning to stochastic simulations. Therefore, researchers have started working on domain specific languages to target probabilistic computations, in order to support better understanding and development of probabilistic models. Among the proposed approaches sampling functions is one of the most promising: distributions are described as functional mappings from the unit interval (0,1] to probability domains which allows expressing a very broad class of distributions. The key advantage of this approach lies in its ability of lifting operations on values into operations on related distributions. The current state of the art frameworks, however, lack the ability to properly express variable correlation in a clean and composable way, which is a major issue of many real-world problems. In this paper we present Liχely, a probabilistic DSL which extends the sampling functions approach by providing explicit means for expressing variable correlation in a composable way and its implementation in F#.

1 Introduction

Probabilistic approaches are interesting for solving computational problems because often they overcome practical limitations of deterministic modeling when dealing with the uncertainty of reality. They have been applied successfully in many fields of computer science such as artificial intelligence [5] and scientific computing [1]. This led to an effort to define domain specific programming languages which provide types describing distributions to address probabilistic computations. The goal of those languages is to express probabilistic algorithms with a concise and simple code by hiding the mapping of probabilistic computation onto deterministic machines.

Most of those languages only deal with finite support distributions [2, 6], while in many applications the domain is infinite or continuous (for example the distributions representing positions in localization problems). Park et al. have shown with $\lambda_{\bigcirc}$ [4] that it is possible to extend the expressivity of the language by forfeiting an analytical representation of the measure function and by specifying a distribution as the algorithm which generates samples according to it. Limiting the queries allowed on a distribution to sampling is an acceptable

trade-off for expressiveness, because sampling allows to calculate approximate answers to expectation queries by using the Monte Carlo method [3] and to improve those answers simply by averaging over more samples.

The $\lambda_{\bigcirc}$ language however does not offer support for modeling correlation explicitly: while it is possible to define a distribution of correlated sample tuples, is it not possible to preserve correlation between two different distribution objects.

Correlation is a characterizing feature of most probabilistic scenarios, yet it is easy to forget about it even for experienced researchers, introducing subtle errors in the results (for example averaging away systematic error on a series of data). To the best of our knowledge, no other sampling probabilistic language addresses correlation explicitly.

In this paper we describe how it is possible to describe correlation by introducing random variable types. Random variables are distributions enriched with identity information, which encodes dependency and correlation.

A straightforward implementation of the random variable concept requires a global mutable status to ensure that the identity information is different for each random variable. We have chosen to give the first implementation of this extension in F# because of its double nature: the functional syntax allows a one to one mapping of the $\lambda_{\bigcirc}$ language, and its object-oriented interface provides an easy way to implement the equality by instance required for the implementation of random variables.

2 Distributions in Liχely

We want to describe distributions as generalized sampling functions, by implementing the semantics described in $\lambda_{\bigcirc}$. A *generalized sampling function* for a distribution is an algorithm which any number[1] of independent uniform samples in the interval $(0, 1]$ to generate a sample for that distribution. In Liχely we use an imperative syntax to define generalized sampling functions, in a similar way as in $\lambda_{\bigcirc}$. The syntax to specify distributions in Liχely uses the following constituents[2]:

- The `dist{ }` bracketing delimits the environment where the custom syntax is used in addition to the standard F# one.
- The `uniform` keyword represents the uniform distribution in the $(0, 1]$ interval, which is given as a language constant.
- The **`let!`** keyword represents the operation of extracting a sample from an existing distribution to use its value in the enclosed expression.
- The **`return`** keyword is used to yield the value of the computed sample for the distribution which is currently being described.

Here we show an example of how a Bernoulli distribution (i.e. a binary distribution which is true with probability `p`) is expressed in the language:

[1] A single uniform sample can always be split in two (by separating odd and even digits of its decimal representation), therefore this formalism is mathematically equivalent to sampling functions which use exactly one sample, even if this is not true for limited precision numbers as the ones we usually have in calculators

[2] the syntax is extended using F# computational expressions[7]

```
let bernoulli p = dist {
  let! u = uniform in
  return u < p }
```

A sampling algorithm is a standard algorithm over values, hence it inherits all the advantages of classical algorithms. For example, it is possible to define a distribution in terms of another one, and also to give a recursive definition:

```
let rec binomial x n = dist {
  if n = 0 then
    return 0
  else
    let! b = (bernoulli x) in
    let! r = binomial x (n-1) in
    if b then
      return 1 + r
    else
      return r
```

The language can also describe combinations of discrete and continuous distributions over the same domain. The following one is a distribution whose value is 0.5 with 50% probability, and uniformly distributed between 0 and 1 in the other half of the cases:

```
let point_uniform = dist {
  let! x = uniform in
  if x > 0.5 then
    return 0.5
  else
    return 2.0 * x
  }
```

Outside of the sub-language, is it possible to get a sample sequence by using a random source for uniform samples:

```
let sampleSeq = getSamples distribution generator
```

3 Modeling dependence and correlation

Two random variables X and Y defined respectively on domains D_X and D_Y are *independent* when for any pair of events $S_X \subseteq D_X$ and $S_Y \subseteq D_Y$ the probability of them occurring together is the same as the product of the probabilities of them occurring separately, i.e.

$$\forall S_X \subseteq D_X, S_Y \subseteq D_Y : P((X,Y) \in S_X \times S_Y) = P(X \in S_X) \cdot P(Y \in S_Y)$$

If we characterize a random variable only with its distribution, it is impossible to have two random variables which are not independent.

To understand why, let us consider a basket of dice. Half of them are normal dice, the other half are loaded dice, which when thrown always show odd numbers. We pick one die at random from the basket, and roll it twice, considering only whether the result is even or odd. Then, we put it back in the basket, pick another one and roll it once. Let us call X, Y and Z the random variables representing the result of the three rolls in chronological order. Their distribution is

the same: 1/4 probability of getting an even number, 3/4 probability of getting an odd number. If we suppose that the distribution is enough to characterize a random variable, we would not be able to distinguish between any of them, in particular Y and Z. However, this is not correct, because Y is correlated with X while Z is not: if we compute the joint probabilities we obtain

	Y even	Y odd	Z even	Z odd
X even	1/8	1/8	1/16	3/16
X odd	1/8	5/8	3/16	9/16

We want to enrich the definition of random variables with meta-information about their dependencies, which would allow to calculate the distribution for any expression of random variables in a general way, *without assumptions on their correlation*. We will show how this information is tracked in random variable inside Liχely and allow us to correctly model this scenario.

4 Random variables

To overcome the inability to model correlation with distributions alone, we add another sub-language to define random variables declaratively. In our model a random variable is characterized by

- an *instance identity*, i.e. two random variables are equal only if they are aliases of the same object
- a *dependency list*, i.e. a list of the random variables it depends from
- a *conditional distribution function*, i.e. a function which calculates the distribution for the random variable given the value of the dependencies.

We describe a syntax for expressing random variables which is oriented to conditional distributions. The new syntax is introduced by the `rndvar{ }` bracketing. It uses the same **`let!`** and **`return`** keywords used for distributions, but with a different semantic. The code fragment defines the random variable `var`

```
let var = rndvar {
  let! x1 = X1 in
  let! x2 = X2 in
  ...
  let! xn = XN in
  return f(x1,x2, ..., xn)
  }
```

meaning that

$$\forall x_1, x_2, \ldots, x_n : \mathrm{Dist}(\mathrm{var}\,|(X_1 = x_1) \wedge (X_2 = x_2) \wedge \ldots \wedge (X_n = x_n)) = f(x_1, x_2, \ldots, x_n)$$

The **`let!`** keyword introduces a dependency, and the **`return`** keyword expresses the conditional distribution function. We exploit the object-oriented nature of F# to have instance identity, i.e. we want to preseve the ability to create two random variables with the same definition as *different objects*, and thus distinguish them in the dependency tree. This is important because we want to have aliasing (i.e. to be able to address the same event with multiple different names), yet we want to keep the possibility of creating different events with the same definition, for example:

```
let coinflip1 = rndvar { return bernoulli 0.5 }
let coinflip2 = rndvar { return bernoulli 0.5 }
let same x y = rndvar {
  let! a = x in
  let! b = y in
  return dist { return a = b }
  }
```

in this example we want the distribution associated to `same coinflip1 coinflip1` to be always true, while we expect `same coinflip1 coinflip2` to be true only half of the times.

Now, let us consider how the example from section 3 can be modeled using `rndvar`. First we define the distributions of results for normal and loaded die

```
let fairDieDistribution = dist {
  let! isEven = bernoulli 0.5 in
  if isEven
    return Even
  else
    return Odd
  }
let loadedDieDistribution = dist { return Odd }

let basketDiceDistribution = dist {
  let! isLoaded = bernoulli 0.5 in
  if isLoaded then
    return LoadedDie
  else
    return FairDie
  }
```

When we pick a die from the basket we do not know whether it is a normal die or a loaded die, hence we represent the result of this operation with a random variable. Since the result of picking a die from the basket is an independent random variable, we define the `pickDie` function as a random variable constructor, and we define two random variables representing whether the first and the second die we pick are loaded or not:

```
let pickDie () = rndvar {
  return basketDiceDistribution
  }
let D1 = pickDie ()
let D2 = pickDie ()
```

Rolling a die is a similar operation. We define it as a random variable constructor, where the distribution of the roll results depends on which kind of die we picked:

```
let dieRoll pickedDie = rndvar{
  let! d = pickedDie in
  match d with
  | LoadedDie ->  return loadedDieDistribution
  | FairDie ->  return fairDieDistribution
  }
let X = (dieRoll D1)
let Y = (dieRoll D1)
```

```
let Z = (dieRoll D2)
```

Finally, we use the `same` function defined above to express some interesting two variables joint distributions:

```
let XandYsame = same X Y
let XandZsame = same X Z
let XandXsame = same X X
```

Remembering that the sum of two even number or two odd numbers is even, those variables express respectively the events "the sum of the first and second die roll is even", "the sum of the first and third die roll is even", "twice the value of the first roll is even".

4.1 Getting the distribution of a random variable

As promised before, we show a general algorithm to retrieve the distribution of any random variable expression.

```
let getDist r =
  let rec memoizedSampling context var =
    match var with
    | rndvar { return E } ->
        dist {
          let! s = E in
          return context,s
        }
    | rndvar { let! x = R in E } ->
        if context.ContainsKey R.instanceId then
          let x = context.Item(R.instanceId) in
          memoizedSampling context (rndvar{ E })
        else
          dist {
            let! context,v = memoizedSampling context R in
            let context = context.Add(R.instanceId,v)
            let! context,v = memoizedSampling context var in
            return context,v
          }
  dist {
    let! context,v = memoizedSampling Map.empty r
    return v
  }
```

The function `memoizedSampling` calculates a distribution of states for all the variables in the dependency tree of the input variable, conditioned by the values of the variables already bound in the input context. To understand why this algorithm is correct we should notice these properties:

1. When we enter the function `memoizedSampling context var`, the variable `var` is not defined in `context`.

2. In every sample of the returned distribution the context is an extension of the input context, i.e. any variable bound in the input context is bound to the same value in the extension.

3. Only random variables appearing in a `let!` instruction are inserted in the context, and only if they are not there already.

4. In the returned distribution samples, the context contains a binding for every dependency of the input variable but not the variable itself. For this reason, at any step of the computation if a variable is bound in a context every other variable it depends from is bound in the same context.

5. The distributions of contexts returned from `memoizedSampling` are sampled exactly once for each time `getDist r` is sampled.

Hence it is easy to see that for each sample of `getDist r` the conditioned distribution of each random variable in the dependency tree is computed and sampled exactly once. Applying this function results in the following distributions for the random variables expressing joint probability:

```
getDist XandYsame =
  dist {
    let! d1 = basketDiceDistribution in
    match d1 with
      | LoadedDie -> // 50% probability to enter this branch
        let! x = loadedDieDistribution in
        let! y = loadedDieDistribution in
        return x = y // 100% probability to be true in this branch
      | FairDie -> // 50% probability to enter this branch
        let! x = fairDieDistribution in
        let! y = fairDieDistribution in
        return x = y // 50% probability to be true in this branch
    } // true in 50% + 25% = 75% cases

getDist XandZsame =
  dist {
    let! d1 = basketDiceDistribution in
    match d1 with
      | LoadedDie -> // 50% branch
        let! x = loadedDieDistribution in
        let! d2 = basketDiceDistribution in
        match d2 with
          | LoadedDie -> // 25% branch
            let! z = loadedDieDistribution in
            return x = z // 100% truth
          | FairDie -> // 25% branch
            let! z = fairDieDistribution in
            return x = z // 50% truth
      | FairDie -> // 50% branch
        let! x = fairDieDistribution in
        match d2 with
          | LoadedDie -> // 25% branch
            let! z = loadedDieDistribution in
            return x = z // 50% truth
          | FairDie -> // 25% branch
            let! z = fairDieDistribution in
            return x = z // 50% truth
    } // true in 25% + 12.5% + 12.5% + 12.5% = 62.5% cases
```

```
getDist XandXsame =
  dist {
    let! d1 = basketDiceDistribution
    match d1 with
      | LoadedDie -> // 50% branch
        let! x = loadedDieDistribution in
        return x = x // 100% truth
      | FairDie -> // 50% branch
        let! x = fairDieDistribution in
        return x = x // 100% truth
    } // true in 50% + 50% = 100% cases
```

As expected `X` is always identical to itself, and it is more likely to be the same as `Y` which is from the same die than `Z` (the results are consistent with the probabilities calculated in the previous section)

5 Conclusions

In this paper we introduced Liχely, a probabilistic language based on sampling functions, implemented in F#, that extends the state of the art in this field (the handling of arbitrary random distributions) offering an explicit construct to express random variables, making correlation easy to handle.

Our work shows that sampling functions can be extended to model random variables as well as distributions, and that correlation between random variables can be expressed naturally and handled automatically by the language. Therefore, random variables can be used as values in expressions. Moreover, because it has been implemented in F#, it can be used in every .NET dialect, and it can be included in real world systems.

We aim to apply this language to those problems where a probabilistic approach is already being used (e.g. Hidden Markov Models, Montecarlo, etc.), because it allows the developer to describe the problem in terms of distributions and random variables. We believe this could improve the readability of the encoding of the problem. We also aim to apply this language to those problems where a probabilistic approach is not yet being used, because both continuous distributions and an explicit handling of the correlation between random variables are needed.

References

[1] Kurt Binder and Dieter W. Heermann. *Monte Carlo Simulation in Statistical Physics: An Introduction*. Springer, January 2010.

[2] Daphne Koller, David McAllester, and Avi Pfeffer. Effective bayesian inference for stochastic programs. In *Proceedings of the National Conference on Artificial Intelligence*, page 740–747, 1997.

[3] D. J. C. Mackay. Introduction to monte carlo methods. In Michael I. Jordan, editor, *Learning in Graphical Models*, number 89 in NATO ASI Series, pages 175–204. Springer Netherlands, January 1998.

[4] Sungwoo Park, Frank Pfenning, and Sebastian Thrun. A probabilistic language based on sampling functions. *ACM Trans. Program. Lang. Syst.*, 31(1):4:1–4:46, December 2008.

[5] Judea Pearl. *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*. Morgan Kaufmann, 1988.

[6] Avi Pfeffer. IBAL: a probabilistic rational programming language. In *International Joint Conference on Artificial Intelligence*, volume 17, page 733–740, 2001.

[7] Don Syme, Adam Granicz, and Antonio Cisternino. *Expert F# 3.0*. Apress, October 2012.