ProbLog and Applicative Probabilistic Programming

Alexander Vandenbroucke
KU Leuven
alexander.vandenbroucke@kuleuven.be

Tom Schrijvers
KU Leuven
tom.schrijvers@kuleuven.be

1. Introduction

Functional probabilistic programming languages such as Anglican [8] exhibit dynamic structure: a program's structure can arbitrarily depend on the value of a previous probabilistic choice.

On the other hand, an essential feature of the probabilistic logic programming language ProbLog [6] is the separation of probabilistic facts from the program rules. The structure of these rules is static, i.e. independent of the values of the probabilistic facts. Moreover, ProbLog derives much of its efficient exact and approximate inference from this property: the invariance of the rules enables their compilation to forms such as BDDs, sd-DNNFs and SDDs on which efficient inference (weighted model counting) can be performed.

The functional programming community has recently concentrated on studying (functional) probabilistic programs in terms of monads [7]. Monads are a natural choice, as they capture—as algebraic structures—precisely those computations that exhibit the dynamic behaviour of functional probabilistic programming languages.

This begs the question whether a similar algebraic structure exists which disallows dynamic program structure, and thus accurately models the behaviour of ProbLog. Fortunately, such a more restrictive class of algebraic structures indeed exists: Applicative Functors [4]. They restrict monads by not allowing the structure of the program to depend on any previously computed value.

Contribution In this work we explain that any ProbLog program, excluding advanced features such as conditioning and *flexible probabilities*, can be reformulated in terms of a probabilistic applicative program, and vice-versa. Then, ProbLog programs are exactly as powerful as probabilistic applicative programs.

2. Probabilistic Logic Programming

Consider the following ProbLog program, implementing a fair coin flip:

```
0.5 :: heads.
tails :- not(heads).
```

The clauses of the program can be divided in facts $\mathcal{F}$ and rules $\mathcal{R}$. In this particular case, $\mathcal{F} = \{0.5 :: \texttt{heads}\}$ and $\mathcal{R} = \{\texttt{tails :- not(heads)}\}$. Note that the rules $\mathcal{R}$ are entirely non-probabilistic, since they are regular logical clauses.

Semantics of Probabilistic Logic Programs A *total choice* C is any subset of $\mathcal{F}$. A fact f is said to be $true$ ($false$) in C if and only if $f \in C$ ($f \notin C$). A total choice C and a set of clauses $\mathcal{R}$ together form a conventional logic program. We use $P \models a$ to denote that program P logically entails an atom a according to perfect model semantics [5].

Let $\mathcal{F} = \{p_1 :: f_1, \ldots, p_n :: f_n\}$, then the probability of a choice $C \subseteq \mathcal{F}$ is given by the product of the probabilities of the true and false facts:

$$\mathbb{P}(C) = \prod_{p_i :: f_i \in C} p_i \times \prod_{p_j :: f_j \in \mathcal{F} \setminus C} (1 - p_j)$$

The distribution semantics defines the probability of a query q (an atom) for a program $P = \mathcal{F} \cup \mathcal{R}$ as the sum of the probabilities of all total choices that entail q:

$$\mathbb{P}(q|P) = \sum_{\substack{C \subseteq \mathcal{F} \\ C \cup \mathcal{R} \models q}} \mathbb{P}(C) \tag{1}$$

For instance, $\mathbb{P}(heads|P) = 0.5 = \mathbb{P}(tails|P)$ when P is the logic program above.

Clearly, C does not influence the structure of the clauses in $\mathcal{R}$. The computational model does not need the capability to change the structure of the program based on probabilistic choices. Thus, monads—which do posses this capability—are too powerful a model. Instead, applicative functors are more appropriate, since in that setting, the structure of the computation is static.

3. Probabilistic Applicative Functor

An Applicative Functor F is a functor F that is equipped with two additional operations $pure$ and $\circledast$:

$$pure : \forall A.A \to FA$$

$$\circledast : \forall A, B.F(A \to B) \to FA \to FB$$

that satisfy a number of laws, described by McBride and Paterson [4].

To model probabilities, we take the approach described by Gibbons and Hinze [2]. That is, we introduce the following probabilistic binary choice operator:

$$a \blacktriangleleft p \blacktriangleright b : A \to [0, 1] \to A \to FA$$

along with some equational laws. With this operator, the coin toss can be written as $heads \blacktriangleleft 0.5 \blacktriangleright tails$.

We can translate the logic program more systematically, by replacing fact $0.5 :: heads$ with boolean choice $true \blacktriangleleft 0.5 \blacktriangleright false$, and replacing the rule `tails :- not(heads).` with a pure

function performing the logical deduction:

$$heads = true \blacktriangleleft 0.5 \blacktriangleright false$$
$$tails'(h) = \neg h$$
$$tails = pure(tails') \circledast heads$$

Observe that the solution is computed by lifting the pure function $tails'$ in to probabilistic functor F and applying it to $heads$.

Now, consider a more complicated example involving recursion and rules annotated with probabilities.

```
0.3::stress(X) :- person(X).
0.2::influences(X,Y) :- person(X), person(Y).

smokes(X) :- stress(X).
smokes(X) :- friend(X,Y), influences(Y,X), smokes(Y).

person(1). person(2). person(3). person(4).

friend(1,2). friend(2,1). friend(2,4).
friend(3,2). friend(4,2).
```

This program models a small social network, represented by the `friends` relation. The goal is to derive the likelihoods of people smoking given their stress levels and social pressures. The annotations on the rules denote the probability that they are present in the program. This is syntactic sugar, as such rules can always be elaborated into equivalent non-probabilistic rules and probabilistic facts, for example:

```
0.3 :: head :- body.
```

is equivalent to:

```
0.3 :: fact.
head :- fact, body.
```

The corresponding applicative program is as follows:

$$stress_i = true \blacktriangleleft 0.3 \blacktriangleright false$$
$$influences_{ij} = true \blacktriangleleft 0.2 \blacktriangleright false$$
$$friends(i) = \begin{cases} \{2\} & \text{if } i = 1 \\ \{1,4\} & \text{if } i = 2 \\ \{2\} & \text{if } i = 3 \\ \{2\} & \text{if } i = 4 \end{cases}$$
$$smokes'_i(stress, infl) = \mu f.stress_i \bigvee_{j \in friends(i)} (infl_{ji} \wedge f_j)$$
$$smokes_i = pure(smokes'_i) \circledast stress \circledast influences$$

The probabilistic facts are $stress_i$ and $influences_{ij}$ where and i and j range over 1,2,3,4. When these names occur without subscripts, they should be understood as the vector of all such probabilistic facts. Such a vector can be constructed by appending separate facts with $\circledast$. The pure functions are $smokes'_i$ for $i = 1, \ldots, 4$, which are constructed by taking the least fixed point.[1]

Note that the final computation ($smokes_i$) is again of the shape "pure function" ($smokes'_i$) applied to primitive probabilistic facts ($stress$ and $influences$). As a consequence the values of the probabilistic facts do not influence the definition of $smokes'_i$, and so the shape of the program is static with respect to the probabilistic choices.

[1] We use $\mu f.expr$ to denote the least fixed point of repeatedly substituting $\mu f.expr$ for f in $expr$.

4. Transformation

For every ProbLog program there exists a corresponding applicative functional program: Let $P = \mathcal{F} \cup \mathcal{R}$ be a ground ProbLog program with facts $\mathcal{F}$ and rules $\mathcal{R}$. For every fact `p :: f` we introduce a choice $f = true \blacktriangleleft p \blacktriangleright false$. The applicative program then computes the the perfect model semantics of $\mathcal{R}$ over these choices.

Conversely, to transform an applicative program into a ProbLog program, the key idea is that, due to its static structure, *any* probabilistic applicative program can be transformed into a shape

$$pure(r) \circledast true \blacktriangleleft p_1 \blacktriangleright false \circledast \cdots \circledast true \blacktriangleleft p_n \blacktriangleright false$$

In this shape, the rules (function r) and probabilistic facts (choices $p_1, \ldots, p_n$) can be easily separated. The details of this transformation are beyond the scope of this text.

5. Discussion

Since monads admit strictly more programs than applicative functors [3], there must also exist an expression gap between applicative probabilistic functional programs (and equivalently ProbLog) and monadic probabilistic functional programs.

Our analysis breaks down when the number of choices in an applicative program is infinite. For such programs, there is no obvious translation into ProbLog, since ProbLog programs support only a finite number of probabilistic facts.

Moreover, this paper only considers the core features of ProbLog, as implemented in the first version of the language. The newer version, ProbLog2 [1], includes many additional features that are beyond the scope of this text. We believe that these features expand the expressive power of ProbLog beyond those of applicative programs. In particular, ProbLog2 allows more flexible expressions in the probabilities. This feature likely does not make ProbLog2 monadic, but rather something in between Applicative Functors and Monads, such as Arrows [3].

Finally, the equivalence between ProbLog and Applicative Functors has a practical benefit: ProbLog's inference algorithms could be leveraged for more efficient inference on the applicative subset of a functional probabilistic programming language.

References

[1] D. Fierens, G. V. den Broeck, J. Renkens, D. S. Shterionov, B. Gutmann, I. Thon, G. Janssens, and L. D. Raedt. Inference and learning in probabilistic logic programs using weighted boolean formulas. *TPLP*, 15(3):358–401, 2015.

[2] J. Gibbons and R. Hinze. Just do it: simple monadic equational reasoning. In *Proceeding of ICFP 2011, Tokyo, Japan, September 19-21, 2011*, pages 2–14. ACM, 2011.

[3] S. Lindley, P. Wadler, and J. Yallop. Idioms are oblivious, arrows are meticulous, monads are promiscuous. *ENTCS*, 229(5):97–117, 2011.

[4] C. McBride and R. Paterson. Applicative programming with effects. *JFP*, 18(1):1–13, 2008.

[5] T. C. Przymusinski. Every logic program has a natural stratification and an iterated least fixed point model. In *Proceedings of PDS 1989, March 29-31, 1989, Philadelphia, Pennsylvania, USA*, pages 11–21. ACM Press, 1989.

[6] L. D. Raedt and A. Kimmig. Probabilistic (logic) programming concepts. *Machine Learning*, 100(1):5–47, 2015.

[7] A. Scibior, Z. Ghahramani, and A. D. Gordon. Practical probabilistic programming with monads. In B. Lippmeier, editor, *Proceedings of Haskell 2015, Vancouver, BC, Canada, September 3-4, 2015*, pages 165–176. ACM, 2015.

[8] D. Tolpin, J. van de Meent, and F. Wood. Probabilistic programming in anglican. In *Proceedings of ECML PKDD 2015 Part III, Porto, Portugal, September 7-11, 2015*, volume 9286 of *LNCS*, pages 308–311. Springer, 2015.

A. Applicative Programs to ProbLog

A.1 Straight-line applicative programs to ProbLog

Straight-line applicative programs are programs of the following shape:

$$pure(f) \circledast true \blacktriangleleft p_1 \blacktriangleright false \circledast \cdots \circledast true \blacktriangleleft p_n \blacktriangleright false$$

where f is a pure function—modelled, for instance, in the untyped λ-calculus extended with booleans and the boolean connectives negation ($\neg$), conjunction ($\wedge$) and disjunction ($\vee$).

Such programs can be straight-forwardly transformed into a ProbLog program, by introducing a fact $p_i :: f_i$ for every choice $x \blacktriangleleft p_i \blacktriangleright y$, and then evaluating the pure function with a λ-calculus evaluator implemented in (deterministic) ProbLog (i.e. Prolog).[2]

A.2 Applicative programs to ProbLog

We show that any program using probabilistic applicatives can be transformed into a straight-line program.

The idea is to provide instantiations for the applicative functor F and operations $pure$, $\circledast$ and $\blacktriangleleft \cdot \blacktriangleright$ that, when used on an applicative program p, produce a pure function f and a list of probabilities $p_1, \ldots, p_n$ such that

$$pure(f) \circledast true \blacktriangleleft p_1 \blacktriangleright false \circledast \cdots \circledast true \blacktriangleleft p_n \blacktriangleright false$$

is a equivalent to p and straight-line.

Pure function Let $F =$ LC-Bool-List $\times \mathbb{N}$, where LC-Bool-List is the usual extension of the untyped lambda calculus with booleans, if-statements and lists. The corresponding definitions for $pure$, $\circledast$ and $\blacktriangleleft \cdot \blacktriangleright$ are:

$$\begin{aligned} pure(x) &= \langle \lambda \_ . x, 0 \rangle \\ \langle f, n \rangle \circledast \langle g, m \rangle &= \langle \lambda l. f(l_{1\ldots n})(g(l_{n+1\ldots n+m})), n+m \rangle \\ a \blacktriangleleft p \blacktriangleright b &= \langle \lambda x. \texttt{if } x \texttt{ then } a \texttt{ else } b, 1 \rangle \end{aligned}$$

where the notation $\lambda \_ . x$ means we don't care about the argument of the lambda-abstraction and $l_{i\ldots j}$ slicing the list l from the i^{th} element up to and including the j^{th} element.

List of probabilities Let $F =$ List $[0, 1]$, where List a is the type of lists with elements of type a. In this real numbers in the closed interval $[0, 1]$. The corresponding definitions for $pure$, $\circledast$ and $\blacktriangleleft \cdot \blacktriangleright$ are:

$$\begin{aligned} pure(x) &= \varepsilon \\ u \circledast v &= u + v \\ a \blacktriangleleft p \blacktriangleright b &= [p] \end{aligned}$$

where ε means the empty list, $u + v$ is used to denote list concatenation and $[x]$ is a singleton list.

Let $c : fa$ be a computation in an arbitrary probabilistic applicative functor F, and let $\langle f, n \rangle = c :$ LC-Bool-List $\times \mathbb{N}$ and $[p_1, \ldots, p_n] = c :$ List $[0, 1]$, then

$$pure(f) \circledast true \blacktriangleleft p_1 \blacktriangleright false \circledast \cdots \circledast true \blacktriangleleft p_n \blacktriangleright false$$

is the straight-line decomposition of c.

We have shown that any applicative probabilistic program can be transformed into a straight-line applicative program. Since all straight-line applicative programs can be transformed into ProbLog, by transitivity so can all probabilistic applicative programs.

[2] See http://people.cs.kuleuven.be/~alexander.vandenbroucke/AppProb for an example implementation.