ProbLog is Applicative

Probabilistic Programming, Uncertainty in AI, Probabilistic Inference, Knowledge Representation

Abstract

Probabilistic Programming Languages (PPLs) support constructions to natively express probability distributions, making it easier for researchers to develop, share and reuse probabilistic models. They have a long history in both the functional (e.g., Anglican) and logic programming (e.g., ProbLog) paradigms. Unfortunately, these efforts have been conducted mostly in isolation and little is known about the relative merits of the two approaches, creating much confusion for the uninitiated.

In this work we establish a common ground for both approaches in terms of algebraic models of probabilistic computation. It is already well-known that functional PPLs conform to the monadic model. We show that ProbLog's flavour of probabilistic computation is restricted to the applicative functor interface. This means that functional PPLs afford greater expressiveness in terms of dynamic program structure, while ProbLog programs are inherently more amenable to static analysis and thus afford faster inference.

1 Introduction

Probabilistic Programming combines general purpose programming and probabilistic modelling, making it easier for researchers to develop, share their models. However, these languages have been developed mostly in isolation, especially along different paradigms, confusing their relative merits.

Functional PPLs such as Anglican [Tolpin *et al.*, 2015] 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 logic PPL ProbLog [De Raedt and Kimmig, 2015; Fierens *et al.*, 2015] 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 on which efficient inference (weighted model counting) can be performed [Vlasselaer *et al.*, 2016].

The functional programming community has recently concentrated on studying probabilistic programs in terms of monads [Scibior *et al.*, 2015]. Monads are a natural choice, as they capture—as algebraic structures—precisely those computations that exhibit dynamic behaviour.

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 [McBride and Paterson, 2008]. 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, including advanced features such as conditioning, can be transformed into a probabilistic applicative program, and vice-versa. Then, ProbLog is exactly as powerful as applicative PPLs. This suggests a new avenue of optimisation for the probabilistic inference of functional PPLs, by exploiting the same knowledge compilation techniques that were developed for probabilistic logic programming languages, thus, *combining* the expressivity of functional languages with the performance of logical languages.

2 Main Idea

2.1 Probabilistic Logic Programming

Consider the following program written in ProbLog. It implements a fair coin toss:

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

The clauses of the program can be divided into 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 regular non-probabilistic Horn 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 in the perfect model semantics [Przymusinski, 1989].

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:

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

For instance, $\mathbb{P}_P(heads) = 0.5 = \mathbb{P}_P(tails)$ 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.

Grounding The previous discussion is valid only for *ground ProbLog programs*. Ground programs are programs that do not contain any variables. The process of turning a non-ground program into a ground program is called *grounding* and is performed by a *grounder*.

ProbLog's grounder performs some additional tasks in addition to grounding. First, the grounding is restricted to *relevant* programs: only rules that are involved in computing the query are grounded. Second, the grounder transforms rules with more flexible probabilities and syntax into one or more regular ground rules.

For instance, ProbLog allows the following rules:

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

The annotations on the rule denotes the probability that the rule occurs in the program. Such annotations are syntactic sugar, as these rules can be elaborated into equivalent non-probabilistic rules and probabilistic facts. For example, the rule above is equivalent to:

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

An even more flexible format exists, where the annotation is allowed to be a variable:

```
P :: p(I) :- weight(I,W), P is 1/W.
```

Suppose we have two ground atoms `weight(i1,2)` and `weight(i2,5)`, then the grounder returns a program containing two rules:

```
0.5 :: p(i1) :- weight(i1,2), P is 1/2.
0.2 :: p(i2) :- weight(i2,5), P is 1/5.
```

For more details, see the recent paper by De Raedt and Kimmig [2015].

For clarity of presentation, we again assume that all programs are ground (possibly after grounding), with a clear separation between probabilistic facts and (non-probabilistic) logical rules from Section 3 onward.

2.2 Probabilistic Applicative Functor

In this section we give the formal definition of the category-theoretical notion of an applicative functor that models probabilistic computations, and then explain intuitively how programs that exhibit this structure can be transformed into a ProbLog program, and vice-versa.

First is the definition of an applicative functor:

Definition 1 (Applicative Functor)**.** *An* Applicative Functor *F is a functor*[1] *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 the following laws [McBride and Paterson, 2008]:

identity	$pure(id) \circledast u = u$
composition	$pure(\circ) \circledast u \circledast v \circledast w = u \circledast (v \circledast w)$
homomorphism	$pure(f) \circledast pure(x) = pure(f(x))$
interchange	$u \circledast pure(x) = pure(\lambda f.f(x)) \circledast u$

(id and $\circ$ denote the identity function and function composition, respectively).

The idea is that $pure$ lifts a pure (deterministic) computation into a probabilistic one and $\circledast$ applies a probabilistically computed function to a probabilistic value. The laws ensure that lifted pure computations can be moved across $\circledast$ freely, while the order of probabilistic computations is preserved.

To model probabilities, we take the approach described by Gibbons and Hinze [Gibbons and Hinze, 2011].

Definition 2 (Probabilistic Applicative Functor)**.** *A* Probabilistic Applicative Functor *is an applicative functor that also supports a binary choice operation:*

$$\blacktriangleleft \cdot \blacktriangleright \ : A \to [0,1] \to A \to FA$$

Denote $\overline{t} = 1 - t$, then the applicative functor should satisfy the following laws:

0-identity	$x \blacktriangleleft 0 \blacktriangleright y = pure(x)$
1-identity	$x \blacktriangleleft 1 \blacktriangleright y = pure(y)$
skewed commutativity	$x \blacktriangleleft p \blacktriangleright y = y \blacktriangleleft \overline{p} \blacktriangleright x$
idempotence	$pure(x) = x \blacktriangleleft p \blacktriangleright x$

probabilistic interchange

$u \circledast (x \blacktriangleleft p \blacktriangleright y) = (\lambda f.f(x) \blacktriangleleft p \blacktriangleright \lambda f.f(y)) \circledast u$

quasi-associativity

$$(\lambda\_.f) \blacktriangleleft p \blacktriangleright id \circledast (g \blacktriangleleft q \blacktriangleright h)$$
$$= (\lambda\nu.\nu(f)) \blacktriangleleft r \blacktriangleright (\lambda\nu.\nu(g)) \circledast (id \blacktriangleleft s \blacktriangleright \lambda\_.h)$$

where $p = rs$, $\overline{pq} = \overline{s}$. The justification for this law is that the probability of f is p on the left and rs on the right, $\overline{p}q = \overline{r}s$ for g, and $\overline{p}q = \overline{s}$ for h. The condition then follows from simple algebra.

With the $\blacktriangleleft \cdot \blacktriangleright$-operator, the coin toss can be written as $heads \blacktriangleleft 0.5 \blacktriangleright tails$. We arrive at a more systematic translation of the logic program coin toss, by replacing fact $0.5 :: heads$ with Boolean choice $true \blacktriangleleft 0.5 \blacktriangleright false$, and replacing the rule `tails :- not(heads).` with the pure function $tails'$ performing the logical deduction:

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

Throughout the paper we also use a biased coin, defined as:

$$coin(p) = true \blacktriangleleft p \blacktriangleright false$$

Observe that the solution is computed by lifting the pure function $tails'$ into probabilistic functor F (using $pure$) and ap-

```
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).
```

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

Figure 1: The smokers example for ProbLog (left) and Applicative Probabilistic Functors (right).

plying it to *heads* (using $\circledast$).

Now, consider the more complicated example shown on the left-hand side of Figure 1. 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. Modelling this behaviour involves creating a recursive predicate `smokes`.

The corresponding applicative program is shown on the right-hand side of Figure 1. The probabilistic facts are $stress_i$ and $influences_{ij}$ where i and j range over 1,2,3 and 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 computing a least fixed point.[2]

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.

2.3 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 `pi :: fi` we introduce a choice $f_i = coin(p_i)$. Given the results of these choices as input, we then compute the perfect model of the rules $\mathcal{R}$. Finally, we check if the query is a member of this perfect model M_P. Since the computation of the perfect model and membership checking are pure procedures, the final program has the following shape:

$$pure(q \in M_P) \circledast f_1 \circledast \cdots \circledast f_n \text{ where } f_1, \ldots, f_n \in \mathcal{F}$$

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

$$pure(f) \circledast coin(p_1) \circledast \cdots \circledast coin(p_n)$$

In this shape, the pure function f which determines the final value, and probabilistic facts (choices $p_1, \ldots, p_n$) can be easily separated. The pure function f determines the rules of the ProbLog program, and the choices determine the facts (and their probabilities).

3 Syntax and Semantics

The previous section outlined the procedure for transforming a ProbLog program into an Applicative program and vice-versa in an intuitive manner. To actually state the transformation exactly, a more precise model of Applicative Probabilistic languages is required. This section develops a simple typed functional PPL for this purpose.

Types The types consist of real numbers, the unit type, functions, a sum type and a product type.

$$T ::= \mathbf{1} \mid \mathbb{R} \mid T \to T \mid T + T \mid T \times T$$

Syntax The following grammar describes the syntax of the applicative probabilistic programming language:

$$\begin{aligned} e ::= {} & c \in \mathbb{R} \mid \mathbf{1} \\ & \mid \lambda x : T.e \mid e\ e \mid \mu x : T.e \\ & \mid \textbf{case } e \textbf{ of } \{\textbf{inl } x \Rightarrow e; \textbf{inr } x \Rightarrow e\} \mid \textbf{inl } e \mid \textbf{inr } e \\ & \mid (e, e) \mid \textbf{case } e \textbf{ of } \{(x, y) \Rightarrow e\} \end{aligned}$$

The first line introduces real and unit constant terms. The second line introduces (typed) lambda terms, application,and a fix-point operator, in order to allow general recursion. The last two lines introduce sums and products respectively. We also assume derived forms for Booleans as sum types, e.g. $Bool \stackrel{\text{def}}{=} \mathbf{1} + \mathbf{1}$, $true \stackrel{\text{def}}{=} \textbf{inr } \mathbf{1}$ and $false \stackrel{\text{def}}{=} \textbf{inl } \mathbf{1}$. If-statements are also allowed:

$$\textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3 \stackrel{\text{def}}{=} \textbf{case } e_1 \textbf{ of } \{\textbf{ inl } \mathbf{1} \Rightarrow e_3; \textbf{inr } \mathbf{1} \Rightarrow e_2\}$$

Another grammar describes the applicative part:

$$a ::= \textbf{pure } e \mid a \circledast a \mid e \blacktriangleleft e \blacktriangleright e$$

The meaning of these three operations should be clear from the previous section.

[1] A functor is a basic concept of category theory. For our purposes, it suffices to know that the definition of an applicative functor implies functorial structure.

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

Typing relation For this language the typing relation is split into two parts: the deterministic part, which judges pure values, and the probabilistic part, which judges applicative computations. The first is denoted $\Gamma \vdash_d e : T$, meaning that the deterministic term e is judged to be of type T in environment Γ. Its definition is entirely standard [Pierce, 2002] and is therefore elided here.

The second relation is more interesting. It is denoted $\Gamma \vdash_a a : T$, and is defined as follows:

$$\textsc{Pure}\ \frac{\Gamma \vdash_d e : T}{\Gamma \vdash_a \mathbf{pure}\ e : T} \qquad \textsc{App}\ \frac{\Gamma \vdash_a a_1 : T_1 \to T_2 \qquad \Gamma \vdash_a a_2 : T_1}{\Gamma \vdash_a a_1 \circledast a_2 : T_2}$$

$$\textsc{Choose}\ \frac{\Gamma \vdash_d e_1 : T \qquad \Gamma \vdash_d e_2 : \mathbb{R} \qquad \Gamma \vdash_d e_3 : T}{\Gamma \vdash_a e_1 \blacktriangleleft e_2 \blacktriangleright e_3 : T}$$

Big-Step Semantics The big-step trace semantics for our probabilistic programming language is styled after recent work by Borgström et al. [2016]. Let $\cdot \downarrow \cdot$ be the standard big-step semantics for the deterministic part of the language. Then we define the big-step semantics $\cdot \Downarrow_c^w \cdot$ as follows (where $w \in \mathbb{R}$ is the weight and c is a trace of choices L (left) or R (right)):

$$\frac{e \downarrow v}{\mathbf{pure}\ e \Downarrow_\bullet^1 v}\ \text{B-PURE}$$

$$\frac{a_1 \Downarrow_{c_1}^{w_1} v_1 \qquad a_2 \Downarrow_{c_2}^{w_2} v_2 \qquad v_1 v_2 \downarrow v}{a_1 \circledast a_2 \Downarrow_{w_1.w_2}^{c_1 \oplus c_2} v}\ \text{B-APP}$$

$$\frac{e_1 \downarrow v_1 \qquad e_2 \downarrow v_2 \qquad e_3 \downarrow v_3}{e_1 \blacktriangleleft e_2 \blacktriangleright e_3 \Downarrow_{v_2}^{L} v_1}\ \text{B-L-CHOOSE}$$

$$\frac{e_1 \downarrow v_1 \qquad e_2 \downarrow v_2 \qquad e_3 \downarrow v_3}{e_1 \blacktriangleleft e_2 \blacktriangleright e_3 \Downarrow_{\overline{v_2}}^{R} v_3}\ \text{B-R-CHOOSE}$$

The first rule states that a pure term evaluates to a pure value with weight (probability) 1 given an empty trace ($\bullet$). The second rule embodies the applicative homomorphism law: the application ($\circledast$) of two pure values is a new pure value wherein the first operand is applied to the second operand. Note that the weights are multiplied and the choices are concatenated ($\oplus$). The third and fourth rule represent a left or right choice respectively, resulting in a left (L) or right (R) trace and weight v_2 or $\overline{v_2}$. The big-step semantics gives rise to an associated probability:

Definition 3 (Big-Step Probability)**.** *The big-step probability of a value q is given by:*

$$\mathbb{P}_a(q) = \sum_{c \in \{L,R\}^* \wedge a \Downarrow_c^w q} w$$

Note that a and c fully determine w, and that for any a and q there exist only a finite number of cs such that $a \Downarrow_c^w q$, since all c have the same finite length (easily proven by induction).

Applicative Laws The applicative (and probabilistic applicative) laws hold up to Big-Step Semantics. For instance, for the *identity* law we have (the law's left-hand and right-hand side have been highlighted for clarity):

$$\boxed{\mathbf{pure}\ id \circledast u} \Downarrow_w^c q \iff \mathbf{pure}\ id \Downarrow_1^\bullet id \wedge u \Downarrow_w^c q \wedge id\, q \downarrow q$$
$$\iff \boxed{u} \Downarrow_w^c q$$

which implies that their big-step probabilities are the same:

$$\forall q : \mathbb{P}_{\mathbf{pure}\ id \circledast u}(q) = \mathbb{P}_u(q)$$

Similar arguments hold for the other laws as well. So our semantics faithfully models a probabilistic applicative functor.

4 Applicative Programs to ProbLog

In this section we make precise the ideas from Section 2.3.

4.1 Straight-line applicative programs to ProbLog

First, we consider a restricted form of programs, that consist of a single pure probabilistic function applied to a sequence of probabilistic choices. More precisely:

Definition 4 (Straight-line applicative programs)**.** *Straight-line applicative programs are programs of the shape:*

$$pure(f) \circledast coin(p_1) \circledast \cdots \circledast coin(p_n)$$

where f is a pure function.

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 statically evaluating the pure function on the finite range of inputs, encoded as a lookup table in ProbLog. [3]

4.2 Applicative programs to ProbLog

Any program using applicative functors can be transformed into a canonical form [McBride and Paterson, 2008]. In this section, we show that any program using *probabilistic* applicative functors can be canonicalised into a straight-line program. The idea is to define functions $[\![a]\!]_R$ and $[\![a]\!]_W = (w_1, \ldots, w_n)$ such that, for any program a, the program

$$\mathbf{pure}\ [\![a]\!]_R \circledast coin(w_1) \circledast \cdots \circledast coin(w_n)$$

is equivalent to a and straight-line.

Pure function We begin by counting the number of choices in a program:

$$\begin{array}{lcl} [\![\cdot]\!]_C : a \to \mathbb{N} & & \\ [\![\mathbf{pure}\ e]\!]_C & = & 0 \\ [\![a_1 \circledast a_2]\!]_C & = & [\![a_1]\!]_C + [\![a_2]\!]_C \\ [\![e_1 \blacktriangleleft e_2 \blacktriangleright e_3]\!]_C & = & 1 \end{array}$$

A **pure** program has no choices and the number of choices of ($\circledast$) is the sum of the number of choices of its operands.

Next, we extract the pure expression:

$$\begin{array}{ll} [\![\cdot]\!]_R : a \to e & \\ [\![\mathbf{pure}\ t]\!]_R & = \lambda x : \mathbf{1}.t \\ [\![a_1 \circledast a_2]\!]_R & = \lambda(c_1, \ldots, c_m, d_1, \ldots, d_n) : Bool^{m+n}. \\ & \quad [\![a_1]\!]_R(c_1, \ldots, c_m)\ ([\![a_2]\!]_R(d_1, \ldots, c_n)) \\ & \quad \text{where } m = [\![a_1]\!]_C; n = [\![a_2]\!]_C \\ [\![e_1 \blacktriangleleft e_2 \blacktriangleright e_3]\!]_R & = \lambda c : Bool.\mathbf{if}\ c\ \mathbf{then}\ \boldsymbol{e}_1\ \mathbf{else}\ \boldsymbol{e}_3 \end{array}$$

[3] website blinded for review

Note: we use an extended syntax in the definition of $[\![a_1 \circledast a_2]\!]_R$, to allow pattern matching in the λ-abstraction as well as matching against products of more than 2 components. However, this is a conservative extension, that adds no additional expressivity to the language. We also use $Bool^n$ as a short-hand for the product of n $Bool$s. With this convenient extension, it is easier to say what we want: the choices $c_1, \ldots, c_m, d_1, \ldots, d_n$ are divided between a_1 and a_2, and then $[\![a_1]\!]_R$ is applied to $[\![a_2]\!]_R$.

Weights The last piece of information is the weights of the choices in the program. The idea is to flatten the syntax tree into a sequence of choices, where the order corresponds to an in-order traversal of the syntax tree:

$$
\begin{array}{lcl}
[\![\cdot]\!]_W : a \to \mathbb{R}^{[\![\cdot]\!]_C} & & \\
[\![\textbf{pure}\ e]\!]_W & = & \mathbf{1} \\
[\![a_1 \circledast a_2]\!]_W & = & (v_1, \ldots, v_m, w_1, \ldots, w_n) \text{ where} \\
 & & (v_1, \ldots, v_m) = [\![a_1]\!]_W; \\
 & & (w_1, \ldots, w_n) = [\![a_2]\!]_W \\
[\![e_1 \blacktriangleleft e_2 \blacktriangleright e_3]\!]_W & = & w \text{ where } e_2 \downarrow w
\end{array}
$$

The straight-line decomposition Let $(w_1, \ldots, w_n) = [\![a]\!]_W$ and let $r = [\![a]\!]_R$, then[4]

$$a_{sl} \stackrel{\text{def}}{=} \textbf{pure}\ (curry_n(r)) \circledast coin(w_1) \circledast \cdots \circledast coin(w_n)$$

is the straight-line decomposition of c. As an example, consider the following program:

$$\begin{array}{r}\textbf{pure}\ (\vee) \circledast (\textbf{pure}\ (\wedge) \circledast coin(0.9) \circledast coin(0.2)) \\ \circledast (\textbf{pure}\ (\wedge) \circledast coin(0.7) \circledast coin(0.5))\end{array}$$

which, after simplification, becomes:

$$\begin{array}{l}\textbf{pure}\ (\lambda(b_1, b_2, b_3, b_4).(b_1 \wedge b_2) \vee (b_3 \wedge b_4)) \\ \circledast coin(0.9) \circledast coin(0.7) \circledast coin(0.2) \circledast coin(0.5)\end{array}$$

We can prove[5] the following about the canonicalisation:

Theorem 1. *For all terms a, values v, traces c and weights w:*

$$a \Downarrow_w^c v \iff a_{sl} \Downarrow_w^c v$$

As a consequence of this theorem we can also prove that the transformations preserve the semantics, i.e. $\mathbb{P}_a(q) = \mathbb{P}_{a_{sl}}(q)$, $\mathbb{P}_a(q) = \mathbb{P}_P(\texttt{lookup(a,q)})$ and $\mathbb{P}_P(\texttt{q}) = \mathbb{P}_{b\ q}(true)$, where `lookup` implements the lookup table, and b tests if its argument is in the prefect model.

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

5 Observations

ProbLog permits the inclusion of observations in the form of *evidence*. The system then computes the conditional distribution of a query given the evidence. For example:

```
evidence(smokes(p2),true).
```

says that `smokes(p2)` is true. The semantics of the program is given by the regular definition of a conditional probability. Given evidence e and program P, this probability is:

$$\mathbb{P}_P(q|e) = \frac{\mathbb{P}_P(q \wedge e)}{\mathbb{P}_P(e)}$$

5.1 Applicative Observations

In this section we extend our definition of probabilistic applicative functors with an operation $\twoheadrightarrow$ to observe the value of a probabilistic computation. The expression $a \twoheadrightarrow p$ means that the value of a computation a is observed if the predicate p is true for that value.

Definition 5 (Probabilistic Applicative Functor with Observations)**.** *A* Probabilistic Applicative Functor with Observations *is a probabilistic applicative functor that additionally supports the following operation:*

$$\twoheadrightarrow : FA \to (A \to Bool) \to FA$$

which is subject to the following laws:

composition $(a \twoheadrightarrow e_1) \twoheadrightarrow e_2 = a \twoheadrightarrow (\lambda x.e_1\ x \wedge e_2\ x)$

left-interchange

$$(a \twoheadrightarrow e) \circledast b = pure(\pi_3) \circledast \Big((\tau \circledast a \circledast b) \twoheadrightarrow e \circ \pi_1\Big)$$

right-interchange

$$a \circledast (b \twoheadrightarrow e) = pure(\pi_3) \circledast \Big((\tau \circledast a \circledast b) \twoheadrightarrow e \circ \pi_2\Big)$$

where $\tau = pure(\lambda f x.(f, x, f(x)))$ and $\pi_i(x_1, x_2, x_3) = x_i$.

Equipped with these laws, the program can be normalised: observations can float all the way to the outside and to the right, similar to how pure values float to the inside and to the left. Choices are left in the middle. More formally,

Definition 6 (Straight-line applicative programs with Observations)**.** *Straight-line applicative programs with observations are applicative programs of the following-shape:*

$$pure(f) \circledast (coin_n(w_1, \ldots, w_n) \twoheadrightarrow e)$$

where f is a pure function and e is a pure predicate and $coin_n$[6] flips n coins and returns their results in n Booleans.

5.2 Syntax and Semantics

Given the definition of an applicative functor with observations, we can extend the syntax defined in Section 3 by adding an additional production to a:

$$a ::= \cdots \mid a \twoheadrightarrow e$$

This means that the value of a computation a is observed if the predicate e evaluates to $true$ for that value.

The typing rule for $\cdot \twoheadrightarrow \cdot$ is the following:

$$\frac{\Gamma \vdash_d t_1 : T \qquad \Gamma \vdash_a t_2 : T \to Bool}{\Gamma \vdash_a t_1 \twoheadrightarrow t_2 : T}$$

Its big-step semantics is given by:

$$\frac{a \Downarrow_w^c v \qquad e\ v \downarrow true}{a \twoheadrightarrow e \Downarrow_w^c v}\ \text{B-EVIDENCE}$$

$$\frac{a \Downarrow_w^c v \qquad e\ v \downarrow false}{a \twoheadrightarrow e \Downarrow_0^c v}\ \text{B-FAIL}$$

[4] The function $curry_n$ curries the function r, i.e. it turns a function of type $((T_1 \times \cdots \times T_n) \to T)$ into a function of type $(T_1 \to \cdots \to T_n \to T)$.

[5] website blinded for review

[6] $coin_n : \mathbb{R}^n \to F(Bool^n)$

By the first rule, the value, trace and weight are unchanged if v_1 coincides with the evidence, and by the second rule, the weight is fixed to zero if it conflicts with the evidence.

The associated probability $\mathbb{P}_a(q)$ now computes the joint probability of q and the evidence in a. To obtain the proper *conditional* probability, we need to divide the result by the probability of the evidence:

Definition 7 (Conditional Big-Step Probability)**.** *The conditional big-step probability of a value q is given by:*

$$\mathbb{C}_a(q) = \frac{\sum_{c \in \{L,R\}* \wedge\ a \Downarrow_c^w q} w}{\sum_{c \in \{L,R\}*, \exists \mathit{value}\ v \wedge\ a \Downarrow_c^w v} w}$$

Consider the following program, which computes the probability that, when tossing two coins, both will come up heads:

$$\begin{aligned} &\mathit{heads} = \mathit{coin}(0.5) \\ &\mathit{twoheads} = \textbf{pure}\ (\wedge) \circledast \mathit{heads} \circledast \mathit{heads} \end{aligned}$$

Without observations $\mathit{twoheads}$ is true with probability 0.25. However, if we already know the result of the the first coin, the probability changes:

$$\mathit{twoheads} = \textbf{pure}\ (\wedge) \circledast (\mathit{heads} \twoheadrightarrow \lambda x.x) \circledast \mathit{heads}$$

has probability $\mathbb{C}_{\mathit{twoheads}}(\mathit{true}) = 0.25/0.5 = 0.5$.

6 Evidence-preserving Transformation

6.1 ProbLog to Applicative

In addition to checking that the main query is satisfied in the perfect model of the program, every piece of evidence should also be checked. That is, for every piece of evidence `evidence(e,true)`, it should be checked that $M_P \twoheadrightarrow (\models e)$, i.e. that the evidence follows from the prefect model. The resulting program thus has the following shape:

$$\textbf{pure}\ (\models q) \circledast (\textbf{pure}\ \mathcal{R} \circledast \mathcal{F} \twoheadrightarrow (\models e_1) \cdots \twoheadrightarrow (\models e_n))$$

where **pure** $\mathcal{R} \circledast \mathcal{F}$ should be understood as a probabilistic expression that computes the perfect model semantics, and $(\models q) \stackrel{\text{def}}{=} \lambda m.m \models q$, i.e partial application of $\models$.

6.2 Applicative to ProbLog

As in Section 4.2, we define functions $[\![\cdot]\!]_R$, $[\![\cdot]\!]_W$ and $[\![\cdot]\!]_E$, such that for any program a,

$$\textbf{pure}\ [\![a]\!]_R \circledast (\mathit{coin}_n([\![a]\!]_W) \twoheadrightarrow [\![a]\!]_E$$

is an equivalent straight-line program. The definitions of $[\![\cdot]\!]_C, [\![\cdot]\!]_R, [\![\cdot]\!]_W$ for evidence rely on the existing definitions:

$$[\![a \twoheadrightarrow e]\!]_\Delta = [\![a]\!]_\Delta \quad (\Delta \in \{C, R, W\})$$

Observation function $[\![\cdot]\!]_E$ The function $[\![\cdot]\!]_E$ collects all observations into a single pure predicate: Pure computations and choices themselves contain no observations:

$$\begin{aligned} &[\![\cdot]\!]_E : a \to E \\ &[\![\textbf{pure}\ e]\!]_E &&= \lambda x : \mathit{Bool}.\mathit{true} \\ &[\![e_1 \blacktriangleleft e_2 \blacktriangleright e_3]\!]_E &&= \lambda x : \mathit{Bool}.\mathit{true} \end{aligned}$$

Only the predicate of an $\twoheadrightarrow$-expression introduces an observation. This predicate is conjoined with any observations in its sub-computation:

$$[\![a \twoheadrightarrow e]\!]_E = \lambda b : \mathit{Bool}^{[\![a]\!]_C}.[\![a]\!]_E\ b \wedge e\ ([\![a]\!]_R\ b)$$

Similarly, the observations of an application consist of the conjunction of the evidence of its sub-computations:

$$\begin{aligned} [\![a_1 \circledast a_2]\!]_E = &\lambda(c_1, \ldots, c_m, d_1, \ldots, d_n) :\ \mathit{Bool}^{m+n}. \\ &[\![a_1]\!]_E(c_1, \ldots, c_m) \wedge [\![a_2]\!]_R(d_1, \ldots, c_n) \end{aligned}$$

where $n = [\![a_1]\!]_C$ and $m = [\![a_2]\!]_C$.

The straight-line decomposition of a program a is now defined by the triple $\langle [\![a]\!]_R, [\![a]\!]_E, [\![a]\!]_W \rangle$:

$$a_{sl} \stackrel{\text{def}}{=} \textbf{pure}\ [\![a]\!]_R \circledast\ (\mathit{coin}_n [\![a]\!]_W \twoheadrightarrow [\![a]\!]_E)$$

For instance, the straight-line version of $\mathit{twoheads}$ is:

$$\begin{aligned} \mathit{twoheads}_{sl} = &\textbf{pure}\ (\lambda(x, y).x \wedge y) \\ &\circledast\ (\mathit{coin}_2(0.5, 0.5) \twoheadrightarrow \lambda(x, \_).x) \end{aligned}$$

The corresponding ProbLog program only adds a single piece of evidence to the transformation presented in Section 4.1:

```
evidence(
  lookup([|t|]e,f1,...,fn,inr(unit)),true)
```

where `lookup` is the lookup table defined in ProbLog, `[|t|]e` $= [\![t]\!]_E$ and `f1,...,fn` are the probabilistic facts. This evidence merely states that the observations must evaluate to `inr(unit)` (true).

7 Discussion & Related work

As explained in the introduction, monads admit strictly more programs than applicative functors [Lindley *et al.*, 2011], thus there exists an expressivity gap between applicative probabilistic functional programs (and equivalently ProbLog) and monadic probabilistic functional programs.

Guttmann et al [2011] enhance the expressivity in the probabilistic logic paradigm with Distributional Clauses. However, they require inference methods [Nitti *et al.*, 2016] that differ substantially from ProbLog's, losing efficient exact inference in the process. Where this leaves Distributional Clauses on the expressivity scale is future work.

Arrows [Lindley *et al.*, 2011] are another algebraic structure that exists in between the monads and applicatives, the analysis of their expressivity is also future work.

Finally, we believe that 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, ultimately achieving a hybrid language that performs the right inference at the right time.

8 Conclusion

In this paper we have shown that ProbLog, a logic PPL, is equivalent to applicative functional PPLs, by giving an explicit source-to-source semantics-preserving transformation between ProbLog and an idealised applicative PPL. As a consequence we conclude that ProbLog must be strictly less expressive than monadic PPLs.

References

[Borgström *et al.*, 2016] Johannes Borgström, Ugo Dal Lago, Andrew D. Gordon, and Marcin Szymczak. A lambda-calculus foundation for universal probabilistic programming. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming, ICFP 2016, Nara, Japan, September 18-22, 2016*, pages 33–46. ACM, 2016.

[De Raedt and Kimmig, 2015] Luc De Raedt and Angelica Kimmig. Probabilistic (logic) programming concepts. *Machine Learning*, 100(1):5–47, 2015.

[Fierens *et al.*, 2015] Daan Fierens, Guy Van den Broeck, Joris Renkens, Dimitar Sht. Shterionov, Bernd Gutmann, Ingo Thon, Gerda Janssens, and Luc De Raedt. Inference and learning in probabilistic logic programs using weighted boolean formulas. *TPLP*, 15(3):358–401, 2015.

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

[Gutmann *et al.*, 2011] Bernd Gutmann, Ingo Thon, Angelika Kimmig, Maurice Bruynooghe, and Luc De Raedt. The magic of logical inference in probabilistic programming. *TPLP*, 11(4-5):663–680, 2011.

[Lindley *et al.*, 2011] Sam Lindley, Philip Wadler, and Jeremy Yallop. Idioms are oblivious, arrows are meticulous, monads are promiscuous. *ENTCS*, 229(5):97–117, 2011.

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

[Nitti *et al.*, 2016] Davide Nitti, Tinne De Laet, and Luc De Raedt. Probabilistic logic programming for hybrid relational domains. *Machine Learning*, 103(3):407–449, 2016.

[Pierce, 2002] Benjamin C. Pierce. *Types and programming languages*. MIT Press, 2002.

[Przymusinski, 1989] Teodor 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.

[Scibior *et al.*, 2015] Adam Scibior, Zoubin Ghahramani, and Andrew D. Gordon. Practical probabilistic programming with monads. In Ben Lippmeier, editor, *Proceedings of Haskell 2015, Vancouver, BC, Canada, September 3-4, 2015*, pages 165–176. ACM, 2015.

[Tolpin *et al.*, 2015] David Tolpin, Jan-Willem van de Meent, and Frank 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.

[Vlasselaer *et al.*, 2016] Jonas Vlasselaer, Guy Van den Broeck, Angelika Kimmig, Wannes Meert, and Luc De Raedt. T_P-compilation for inference in probabilistic logic programs. *Int. J. Approx. Reasoning*, 78:15–32, 2016.