ProbLog is Applicative

Alexander Vandenbroucke

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:

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

with Tom Schrijvers

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]. [illegible]s are a natural choice, as they capture—as algebraic stru[illegible] precisely those computations that exhibit dynamic beha[illegible]

This begs the question whether a sim[illegible]braic structure exists which disallows dynamic program s[illegible] and thus accurately models the behaviour of ProbLog[illegible]ately, such a more restrictive class of algebraic struc[illegible]d exists: Applicative Functors [McBride and Paterson[illegible] They restrict monads by not allowing the structure of the [illegible] gram 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:

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

with Tom Schrijvers

under review

Introduction

What are PPLs

What are PPLs

General Purpose Programming Language

+

Probabilistic Modelling

=

Probabilistic Programming Language

What are PPLs

General Purpose Programming Language

+

Probabilistic Modelling

=

Probabilistic Programming Language

⇒ easier communication

What are PPLs

General Purpose Programming Language

+

Probabilistic Modelling

=

Probabilistic Programming Language

⇒ easier communication

⇒ more reuse

ProbLog

ProbLog

ProbLog

=

Logic Programming

+

Probabilities

L. De Raedt and A. Kimmig. Probabilistic (logic) programming concepts. Machine Learning, 100:1, pp. 5 - 47, Springer New York LLC, 2015.

Example - Flipping two coins

```
% Probabilistic facts:
0.5 :: heads1.
0.6 :: heads2.

% Rules:
twoHeads :- heads1, heads2.
```

Example - Flipping two coins

```
% Probabilistic facts:
0.5 :: heads1.
0.6 :: heads2.

% Rules:
twoHeads :- heads1, heads2.
```

```
query(heads1).
```

Example - Flipping two coins

```
% Probabilistic facts:
0.5 :: heads1.
0.6 :: heads2.

% Rules:
twoHeads :- heads1, heads2.
```

```
query(heads1).
   → 0.5 (50%)
```

Example - Flipping two coins

```
% Probabilistic facts:
0.5 :: heads1.
0.6 :: heads2.

% Rules:
twoHeads :- heads1, heads2.
```

```
query(heads1).
   → 0.5 (50%)
query(heads2).
```

Example - Flipping two coins

```
% Probabilistic facts:
0.5 :: heads1.
0.6 :: heads2.

% Rules:
twoHeads :- heads1, heads2.
```

```
query(heads1).
   → 0.5 (50%)
query(heads2).
   → 0.6 (60%)
```

Example - Flipping two coins

```
% Probabilistic facts:
0.5 :: heads1.
0.6 :: heads2.

% Rules:
twoHeads :- heads1, heads2.
```

```
query(heads1).
   → 0.5 (50%)
query(heads2).
   → 0.6 (60%)
```

```
query(twoHeads).
```

Example - Flipping two coins

```
% Probabilistic facts:
0.5 :: heads1.
0.6 :: heads2.

% Rules:
twoHeads :- heads1, heads2.
```

```
query(heads1).
   → 0.5 (50%)
query(heads2).
   → 0.6 (60%)
```

```
query(twoHeads).
   → 0.3 (30%)
```

Example - Smokers

```
% Deterministic facts
person(klara). person(george).
friend(klara,george). friend(george,klara).
```

Example - Smokers

```
% Deterministic facts
person(klara). person(george).
friend(klara,george). friend(george,klara).

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

additional facts:

```
0.3 :: stress_k. 0.3 :: stress_g.
```

and rules:

```
stress(klara)  :- stress_k, person(klara).
stress(george) :- stress_g, person(george).
```

Example - Smokers

```
% Deterministic facts
person(klara). person(george).
friend(klara,george). friend(george,klara).

% Probabilistic facts
0.3 :: stress(X) :- person(X).
0.2 :: influences(X,Y) :- person(X),person(Y).

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

Example - Smokers

```
% Deterministic facts
person(klara). person(george).
friend(klara,george). friend(george,klara).

% Probabilistic facts
0.3 :: stress(X) :- person(X).
0.2 :: influences(X,Y) :- person(X),person(Y).

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

query(smokes(george)) → 0.342

Semantics (1/2)

Facts

$\mathcal{F}$ = { `0.5 :: heads1, 0.6 :: heads2` }

Rules

$\mathcal{R}$ = { `twoHeads :- heads1,heads2` }

Semantics (1/2)

Facts

$\mathcal{F}$ = { `0.5 :: heads1, 0.6 :: heads2` }

Rules

$\mathcal{R}$ = { `twoHeads :- heads1,heads2` }

Total Choice

$C \subseteq \mathcal{F}$

for all $f \in \mathcal{F}$:

$f \in C \Rightarrow f$ is true

$f \notin C \Rightarrow f$ is false

Semantics (2/2)

Probability of a total choice

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

probability of true facts

probability of false facts

Semantics (2/2)

Probability of a total choice

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

probability of true facts

probability of false facts

Probability of a query

$$\mathbb{P}_P(q) = \sum_{C \subseteq F \wedge C \cup R \vDash q} P(C)$$

probability of all choices that entail the query

Problem Statement

Problem Statement

Functional PPLs (Anglican)

```
(define (twoHeads)
  (and (flip 0.5)
       (flip 0.6))
```

[Tolpin et al. 2015]

Problem Statement

Functional PPLs (Anglican)

```
(define (trail prob j)
  (if (flip (prob j))
        j
        (trail prob (+ j 1))))
```

recursion depends on `flip`

⇒ structure depends dynamically on `flip`

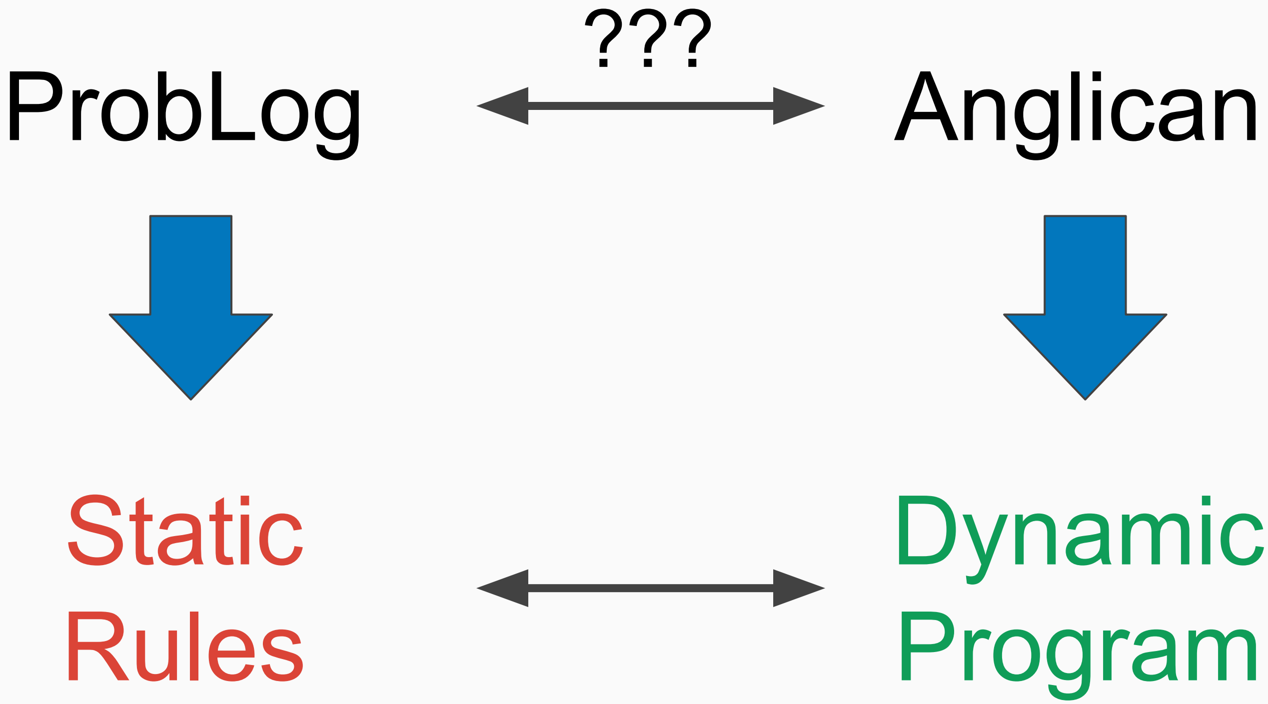
Problem Statement

Functional PPLs (Anglican)

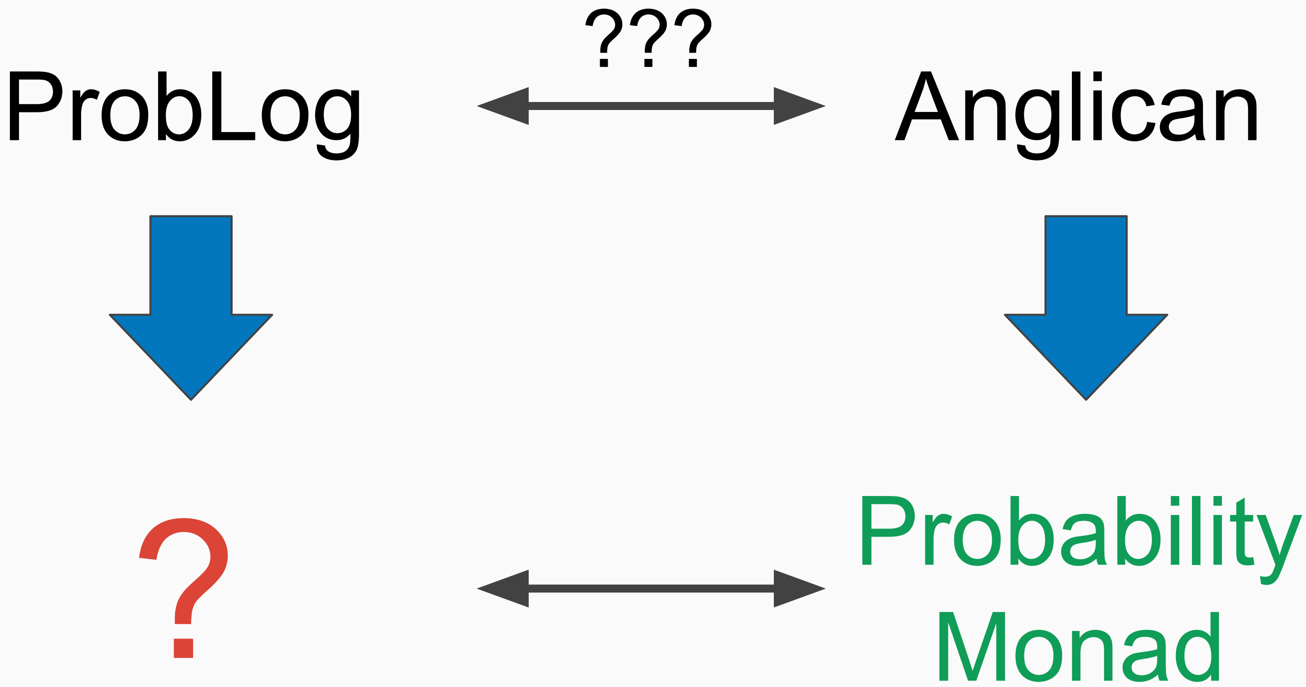
```
(define (trail prob j)
  (if (flip (prob j))
      j
      (trail prob (+ j 1))))
```

recursion depends on `flip`
⇒ structure depends dynamically on `flip`
⇒ semantics requiring static $\mathfrak{R}$ is useless

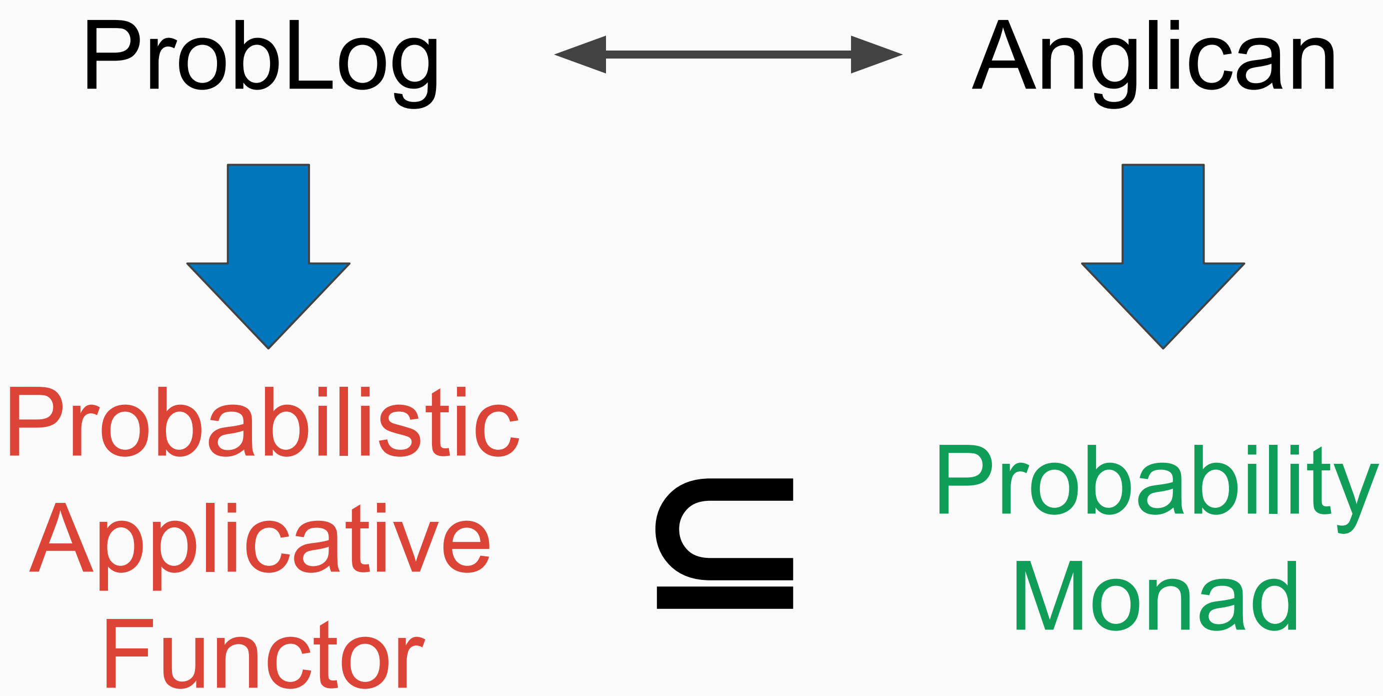
Logic vs Functional



Logic vs Functional



Spoiler Alert



Consequences

ProbLog ⟷ Anglican

Increased Efficiency **vs** Increased Flexibility

Probabilistic Applicative Functors

Probabilistic **Applicative Functors**

Applicative Functor

Functor F

+

$$\textit{pure} : A \rightarrow F\,A$$

$$\circledast : F\,(A \rightarrow B) \rightarrow F\,A \rightarrow F\,B$$

+

Laws

[McBride & Paterson]

Applicative Functor

Functor F

+

$$pure : A \to F\ A$$

"computation" containing an A

$$\circledast : F\ (A \to B) \to F\ A \to F\ B$$

+

Laws

[McBride & Paterson]

Applicative Functor

Functor F

+

wrap value

"computation"
containing an A

$$pure : A \rightarrow F\ A$$

$$\circledast : F\ (A \rightarrow B) \rightarrow F\ A \rightarrow F\ B$$

+

Laws

[McBride & Paterson]

Applicative Functor

Functor F

+

wrap value

"computation"
containing an A

pure : A → F A

⊛ : F (A → B) → F A → F B

"apply" a functor with a function (A → B) to a functor containing a value A

+

Laws

[McBride & Paterson]

Applicative Functor - Laws

identity

$$\mathrm{pure}(id) \circledast u = u$$

composition

$$u \circledast (v \circledast w) = \mathrm{pure}(\circ) \circledast u \circledast v \circledast w$$

homomorphism

$$\mathrm{pure}(f) \circledast \mathrm{pure}(x) = \mathrm{pure}(f\ x)$$

interchange

$$u \circledast \mathrm{pure}(x) = \mathrm{pure}(\backslash f \rightarrow f\ x) \circledast u$$

Applicative Functor - Laws

Pull **pure** left

identity

pure(id) ⊛ u = u

composition

u ⊛ (v ⊛ w) = **pure**(∘) ⊛ u ⊛ v ⊛ w

homomorphism

pure(f) ⊛ pure(x) = **pure**(f x)

interchange

u ⊛ pure(x) = **pure**(\f → f x) ⊛ u

Applicative Functor - Canonical Form

$$pure(f) \circledast (pure(g) \circledast a \circledast b)$$

↓ composition + homomorphism

$$pure(\backslash x\ y \rightarrow f\ (g\ x\ y)) \circledast a \circledast b$$

Applicative Functor - Canonical Form

any applicative expression

laws

$\mathrm{pure}(f) \circledast a_1 \circledast \ldots \circledast a_n$

Canonical Form

Probabilistic Applicative Functor

Applicative Functor *F*

+

$$\langle\ .\ \rangle : A \rightarrow [0,1] \rightarrow A \rightarrow F\ A$$

+

Laws

[Gibbons & Hinze 2011]

Probabilistic Applicative Functor

Applicative Functor *F*

+

$$\langle\ .\ \rangle : A \rightarrow [0,1] \rightarrow A \rightarrow F\ A$$

+

Laws

- Canonical form
- ⟨ . ⟩ behaves like a probability

[Gibbons & Hinze 2011]

Probabilistic App. Fun. - Example

```
% Probabilistic facts:
0.5 :: heads1.
0.6 :: heads2.

% Rules:
twoHeads :- heads1, heads2.
```

```
query(twoHeads).
```

Probabilistic App. Fun. - Example

```
% Probabilistic facts:
0.5 :: heads1.
0.6 :: heads2.

% Rules:
twoHeads :- heads1, heads2.
```

```
query(twoHeads).
```

pure( ) ⊛ ⊛ ***where***

Probabilistic App. Fun. - Example

```
% Probabilistic facts:
0.5 :: heads1.
0.6 :: heads2.

% Rules:
twoHeads :- heads1, heads2.
```

```
query(twoHeads).
```

```
pure(          ) ⊛ heads1 ⊛ heads2 where
    heads1    = True 〈 0.5 〉 False
    heads2    = True 〈 0.6 〉 False
```

Probabilistic App. Fun. - Example

```
% Probabilistic facts:
0.5 :: heads1.
0.6 :: heads2.

% Rules:
twoHeads :- heads1, heads2.
```

```
query(twoHeads).
```

```
pure(twoHeads) ⊛ heads1 ⊛ heads2 where
   heads1   = True ⟨ 0.5 ⟩ False
   heads2   = True ⟨ 0.6 ⟩ False
   twoHeads = \h1 h2 → h1 && h2
```

Probabilistic App. Fun. - Example

```
% Probabilistic facts:
0.5 :: heads1.
0.6 :: heads2.

% Rules:
twoHeads :- heads1, heads2.
```

```
query(twoHeads).
```

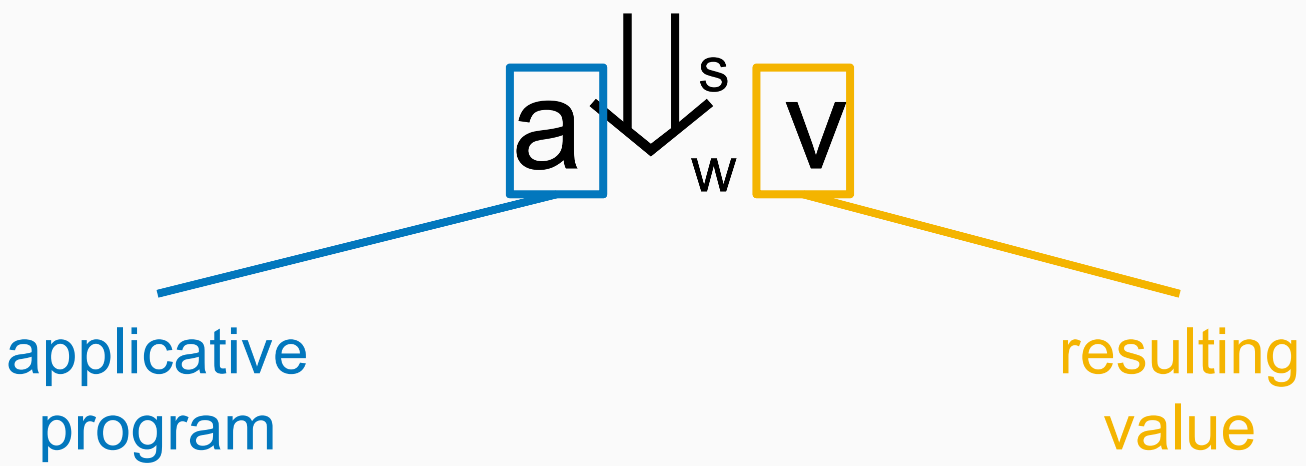
```
pure(twoHeads) ⊛ flip(0.5) ⊛ flip(0.6) where
    twoHeads = \h1 h2 → h1 && h2
    flip p = True ⟨ p ⟩ False
```

a coin flip

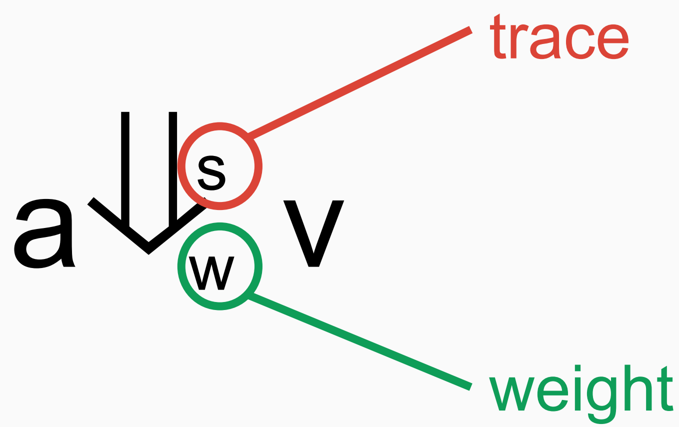
Big-step Relation

$$a \Downarrow^{s}_{w} v$$

Big-step Relation



Big-step Relation



Big-step Relation

$$\text{pure}(v) \Downarrow^{\bullet}_{1} v$$

Prob. App. Fun. - Formal Semantics

Big-step Relation

$$t \langle p \rangle f \Downarrow^{L}_{p} t$$

Prob. App. Fun. - Formal Semantics

Big-step Relation

$$t \langle p \rangle f \Downarrow^{L}_{p} t$$

$$t \langle p \rangle f \Downarrow^{R}_{1-p} f$$

Prob. App. Fun. - Formal Semantics

Big-step Relation

$$u \circledast v \Downarrow^{r++s}_{pq} y$$

$$\Leftrightarrow$$

$$u \Downarrow^{r}_{p} f \;\&\&\; v \Downarrow^{s}_{q} x \;\&\&\; f(x) = y$$

Prob. App. Fun - Formal Semantics

Big-step Relation

$$u \circledast v \Downarrow^{r++s}_{pq} y$$

$$\Leftrightarrow$$

$$u \Downarrow^{r}_{p} f \;\&\&\; v \Downarrow^{s}_{q} x \;\&\&\; f(x) = y$$

Formal Semantics - Associated Probability

$$\mathbb{P}_a(v) = \sum_{\exists s:\, a \Downarrow_w^s v} w$$

all traces evaluating to v

Transformation

ProbLog → Prob. App.

```
% Probabilistic facts:
0.5 :: heads1.
0.6 :: heads2.

% Rules:
twoHeads :- heads1, heads2.
```

```
query(twoHeads).
```

facts → coin tosses

```
pure(twoHeads) ⊛ flip(0.5) ⊛ flip(0.6) where
    twoHeads = \h1 h2 → h1 && h2
    flip p = True 〈 p 〉 False
```

ProbLog → Prob. App.

```
% Probabilistic facts:
0.5 :: heads1.
0.6 :: heads2.

% Rules:
twoHeads :- heads1, heads2.
```

```
query(twoHeads).
```

facts → coin tosses

```
pure(twoHeads) ⊛ flip(0.5) ⊛ flip(0.6) where
    twoHeads = \h1 h2 → h1 && h2
    flip p = True ⟨ p ⟩ False
```

ProbLog → Prob. App.

```
% Probabilistic facts:
0.5 :: heads1.
0.6 :: heads2.

% Rules:
twoHeads :- heads1, heads2.
```

```
query(twoHeads).
```

facts → coin tosses

rules → pure functions

```
pure(twoHeads) ⊛ flip(0.5) ⊛ flip(0.6) where
    twoHeads = \h1 h2 → h1 && h2
    flip p = True ⟨ p ⟩ False
```

ProbLog ← Prob. App. - By Example

$$\mathit{pure}(<) \circledast (1 \langle 0.5 \rangle 10) \circledast (2 \langle 0.3 \rangle 5)$$

ProbLog ← Prob. App. - By Example

```
pure(<) ⊛ (1 ⟨ 0.5 ⟩ 10) ⊛ (2 ⟨ 0.3 ⟩ 5)
   ↓   pick = \x y b -> if b then x else y
pure(<) ⊛ (pure (pick 1 10) ⊛ flip(0.5))
        ⊛ (pure (pick 2 5)  ⊛ flip(0.3))
```

ProbLog ← Prob. App. - By Example

```
pure(<) ⊛ (1 ⟨ 0.5 ⟩ 10) ⊛ (2 ⟨ 0.3 ⟩ 5)
```

↓ `pick = \x y b -> if b then x else y`

```
pure(<) ⊛ (pure (pick 1 10) ⊛ flip(0.5))
        ⊛ (pure (pick 2 5)  ⊛ flip(0.3))
```

↓ **canonicalise**

```
pure(\b1 b2 -> pick 1 10 b1 < pick 2 5 b2)
  ⊛ flip(0.5)
  ⊛ flip(0.3)
```

ProbLog ← Prob. App. - By Example

```
pure(<) ⊛ (1 ⟨ 0.5 ⟩ 10) ⊛ (2 ⟨ 0.3 ⟩ 5)
```

↓ `pick = \x y b -> if b then x else y`

```
pure(<) ⊛ (pure (pick 1 10) ⊛ flip(0.5))
        ⊛ (pure (pick 2 5)  ⊛ flip(0.3))
```

↓ **canonicalise**

```
pure(\b1 b2 -> pick 1 10 b1 < pick 2 5 b2)
  ⊛ flip(0.5)
  ⊛ flip(0.3)
```

```
0.5 :: fact1. 0.3 :: fact2.
```

ProbLog ← Prob. App. - By Example

```
pure(<) ⊛ (1 ⟨ 0.5 ⟩ 10) ⊛ (2 ⟨ 0.3 ⟩ 5)
```

↓

```
pick = \x y b -> if b then x else y
```

```
pure(<) ⊛ (pure (pick 1 10) ⊛ flip(0.5))
        ⊛ (pure (pick 2 5)  ⊛ flip(0.3))
```

↓ **canonicalise**

```
pure(\b1 b2 -> pick 1 10 b1 < pick 2 5 b2)
  ⊛ flip(0.5)
  ⊛ flip(0.3)
```

```
0.5 :: fact1. 0.3 :: fact2.
```

b1	b2	<
F	F	F
F	T	F
T	F	T
T	T	T

ProbLog ← Prob. App. - By Example

```
pure(<) ⊛ (1 ⟨ 0.5 ⟩ 10) ⊛ (2 ⟨ 0.3 ⟩ 5)
```

↓ `pick = \x y b -> if b then x else y`

```
pure(<) ⊛ (pure (pick 1 10) ⊛ flip(0.5))
        ⊛ (pure (pick 2 5)  ⊛ flip(0.3))
```

↓ **canonicalise**

```
pure(\b1 b2 -> pick 1 10 b1 < pick 2 5 b2)
  ⊛ flip(0.5)
  ⊛ flip(0.3)
```

b1	b2	<
F	F	F
F	T	F
T	F	T
T	T	T

```
0.5 :: fact1. 0.3 :: fact2.
lt :- fact1, not(fact2).
lt :- fact1, fact2.
```

Canonicalisation

weights

$$[|.|]_W : F\ A \rightarrow \mathbb{R} ✖ \cdots ✖ \mathbb{R}$$

pure rules

$$[|.|]_R : F\ A \rightarrow (\text{Bool} \rightarrow \cdots \rightarrow \text{Bool} \rightarrow A)$$

canonical form

$$f \cong \mathtt{pure}([|f|]_R) \circledast \mathtt{flip}(w_1) \circledast \cdots \circledast \mathtt{flip}(w_n)$$

$$\textbf{where } (w_1,\ldots,w_n) = [|f|]_W$$

Conclusion

ProbLog $\rightarrow$ Prob. App.

ProbLog $\leftarrow$ Prob. App.

Preserve Probabilities

Evidence

ProbLog evidence

```
% Probabilistic facts:
0.5 :: heads1.
0.6 :: heads2.

% Rules:
twoHeads :- heads1, heads2.
```

query(twoHeads).

→ 0.3

ProbLog evidence

```
% Probabilistic facts:
0.5 :: heads1.
0.6 :: heads2.

% Rules:
twoHeads :- heads1, heads2.
evidence(heads1,true).
```

query(twoHeads).

→ 0.6

Prob. App. with Observations

Applicative Functor *F*

+

$$\twoheadrightarrow : \mathrm{F\ A} \rightarrow (\mathrm{A} \rightarrow \mathrm{Bool}) \rightarrow \mathrm{F\ A}$$

+

Laws

Prob. App. with Observations - Laws

composition

$$(u \twoheadrightarrow p) \twoheadrightarrow q = u \twoheadrightarrow (\backslash x \rightarrow p\ x\ \&\&\ q\ x)$$

left interchange

$$(u \twoheadrightarrow p) \circledast v$$

$$=$$

$$pure(\pi_3) \circledast ((pure(t) \circledast u \circledast v) \twoheadrightarrow p \circ \pi_1)$$

where

$$t = \backslash f\ x \rightarrow (f, x, f(x))$$

. . .

Prob. App. with Observations - Laws

composition

$(u \twoheadrightarrow p) \twoheadrightarrow q = u \twoheadrightarrow (\backslash x \longrightarrow p\ x\ \&\&\ q\ x)$

left interchange

$(u \twoheadrightarrow p) \circledast v$

$=$

$pure(\pi_3) \circledast ((pure(t) \circledast u \circledast v) \twoheadrightarrow p \circ \pi_1)$

where

$t = \backslash f\ x \longrightarrow (f, x, f(x))$

...

Push ↠ to the right

Prob. App. with Observations - Laws

composition

$$(u \twoheadrightarrow p) \twoheadrightarrow q = u \twoheadrightarrow (\backslash x \to p\ x\ \&\&\ q\ x)$$

left interchange

$$(u \twoheadrightarrow p) \circledast v$$

$$=$$

$$pure(\pi_3) \circledast ((pure(t) \circledast u \circledast v) \twoheadrightarrow p \circ \pi_1)$$

where

$$t = \backslash f\ x \to (f, x, f(x))$$

...

$\Rightarrow$ $\exists$ Canonical Form

ProbLog → Prob. App.

```
% Probabilistic facts:
0.5 :: heads1.
0.6 :: heads2.
% Rules:
twoHeads :- heads1, heads2.
evidence(heads1,true).
```

```
query(twoHeads).
```

ProbLog → Prob. App.

```
% Probabilistic facts:
0.5 :: heads1.
0.6 :: heads2.
% Rules:
twoHeads :- heads1, heads2.
evidence(heads1,true).
```

```
query(twoHeads).
```

```
pure(twoHeads) ⊛ heads1 ⊛ heads2 where
   heads1 = flip(0.5)
   heads2 = flip(0.6)
   twoHeads = \h1 h2 -> h1 && h2
```

ProbLog → Prob. App.

```
% Probabilistic facts:
0.5 :: heads1.
0.6 :: heads2.
% Rules:
twoHeads :- heads1, heads2.
evidence(heads1,true).
```

```
query(twoHeads).
```

```
pure(twoHeads) ⊛ (heads1 ↠ id) ⊛ heads2 where
    heads1 = flip(0.5)
    heads2 = flip(0.6)
    twoHeads = \h1 h2 -> h1 && h2
```

ProbLog ← Prob. App.

```
p : F Bool
```

↓ **canonicalise**

```
pure(f) ⊛ ((pure (,...,)⊛flip(w1)⊛...⊛flip(wn))
            ↠ obs)
```

↓

```
w1 :: fact1
...
wn :: factn
f    :- truth table

obs  :- truth table

evidence(obs,true)
```

Summary

Summary

★ ProbLog is a Probabilistic Logic Programming Language

★ whose *computational* model is probabilistic applicative functors

★ with observations.

★ as opposed to monadic computations

Questions

… or just come talk to me.

Smokers Example

```
smokes ⊛ stressA ⊛ stressG ⊛ inflAG ⊛ inflGA where
   stressA = flip(0.3)
   stressG = flip(0.3)
   inflAG  = flip(0.2)
   iflGA   = flip(0.2)
   smokes = \sA sG iAG iGA -> μ smk -> \p ->
     if p == amr then
        sA || (smk george && iGA)
     else
        sG || (smk amr && iAG)
```