CS6462

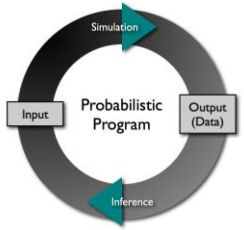
Probabilistic and Explainable AI

Lesson 9
First-Order Probabilistic Programming Language (FOPPL)

by Emil Vassev

February 7, 2022

FOPPL Features



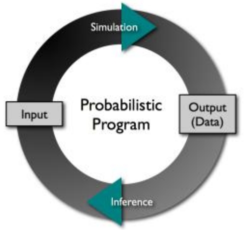
*First-Order Probabilistic Programming Language**:

- includes most common features of programming languages;
- conditional statements, e.g., ***if***
- primitive operations, e.g.,***+,-,*,/***
- user-defined functions:
  - must be first order, i.e., functions cannot accept other functions as arguments
  - cannot be recursive

- FOPPL programs are models describing distributions over a finite number of random variables
- compile any program written in FOPPL to a data structure that represents a graphical model

** "An Introduction to Probabilistic Programming", book by J.W. van de Meent, B. Paige, H. Yang, F. Wood]*

FOPPL Syntax



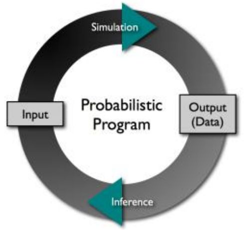
Grammar:

$$
\begin{aligned}
v &::= \text{variable} \\
c &::= \text{constant value or primitive operation} \\
f &::= \text{procedure} \\
e &::= c \mid v \mid (\texttt{let}\ [v\ e_1]\ e_2) \mid (\texttt{if}\ e_1\ e_2\ e_3) \\
&\quad \mid (f\ e_1\ \ldots\ e_n) \mid (c\ e_1\ \ldots\ e_n) \\
&\quad \mid (\texttt{sample}\ e) \mid (\texttt{observe}\ e_1\ e_2) \\
q &::= e \mid (\texttt{defn}\ f\ [v_1\ \ldots\ v_n]\ e)\ q
\end{aligned}
$$

- FOPPL is a Lisp variant that is based on Clojure*
- two sets of production rules: for expressions ***e*** and for programs ***q***

Hickey, R. (2008), ‘The Clojure Programming Language’. In: *Proceedings of the 2008 Symposium on Dynamic Languages. New York, NY, USA, pp. 1:1–1:1,* ACM.

FOPPL Expressions



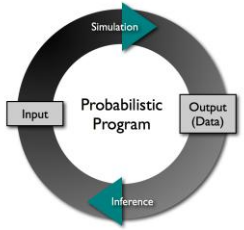
$$
\begin{aligned}
e ::= \; & c \mid v \mid (\texttt{let}\ [v\ e_1]\ e_2) \mid (\texttt{if}\ e_1\ e_2\ e_3) \\
& \mid (f\ e_1\ \ldots\ e_n) \mid (c\ e_1\ \ldots\ e_n) \\
& \mid (\texttt{sample}\ e) \mid (\texttt{observe}\ e_1\ e_2)
\end{aligned}
$$

- constant ***c**:* a value of a primitive data type such as a number, a string, or a Boolean, a built-in primitive function (e.g., ***+***), or a value of any other data type that can be constructed using primitive procedures (lists, vectors, hash-maps, and distributions)

  primitive operation example: **(+ a b)** → *a + b (Python equivalent)*

- variable ***v***: a symbol that references the value of another expression in the program
- ***let*** form (***let [v e_1] e_2***) binds the value of the expression ***e_1*** to the variable ***v**,* which can then be referenced in the expression ***e_2*** (the body of the let expression)
- ***if*** form (***if e_1 e_2 e_3***) takes the value of ***e_2*** when the value of ***e_1*** is logically true and the value of ***e_3*** when ***e_1*** is logically false

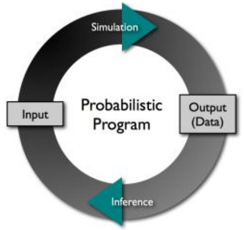
FOPPL Expressions (cont.)



$$
\begin{aligned}
e ::= \; & c \mid v \mid (\texttt{let}\ [v\ e_1]\ e_2) \mid (\texttt{if}\ e_1\ e_2\ e_3) \\
& \mid (f\ e_1\ \ldots\ e_n) \mid (c\ e_1\ \ldots\ e_n) \\
& \mid (\texttt{sample}\ e) \mid (\texttt{observe}\ e_1\ e_2)
\end{aligned}
$$

- function application ***(f*** $\boldsymbol{e_1 \ldots e_n}$***)*** calls the user-defined function ***f*** with arguments $\boldsymbol{e_1}$ through $\boldsymbol{e_n}$
- primitive procedure applications ***(c*** $\boldsymbol{e_1 \ldots e_n}$***)*** calls a built-in function ***c,*** such as ***+***
- sample form ***(sample e)*** represents an unobserved random variable; it accepts a single expression ***e***, which must evaluate to a distribution object, and returns a value that is a sample from this distribution
- observe form ***(observe*** $\boldsymbol{e_1\ e_2}$***)*** represents an observed random variable; it accepts an argument $\boldsymbol{e_1}$, which must evaluate to a distribution that conditions on the next argument $\boldsymbol{e_2}$,i.e., $\boldsymbol{P(e_2 \mid e_1)}$

FOPPL Data Structures



Vector and map data structures:

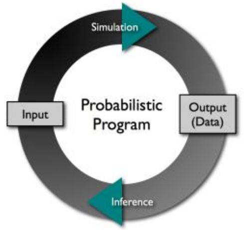
- vectors: constructed by the expression ***(vector e_1 . . . e_n)***

  Example: ***(vector 1 2 3 4 5 6)*** – vector defining the die sample space
- hash maps: constructed by the expression ***(hash-map e_1 e'_1 … e_n e'_n)*** - constructed from a sequence of key-value pairs ***e_1 e'_1***

Functions operating over data structures:

- ***(first e) -*** retrieves the first element of a list or vector ***e***
- ***(rest e)**, **(last e)**, **(append e_1 e_2)***
- ***(get e_1 e_2) -*** retrieves an element at index ***e_2*** from a list or vector ***e_1*** , or the element at key ***e_2*** from a hash map ***e_1***
- ***(put e1 e2 e3)**, **(remove e1 e2)***

FOPPL Loops



For loop syntax:

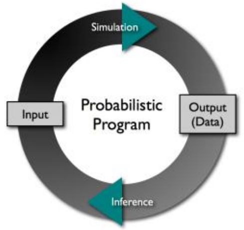
```
(foreach c
  [v_1 e_1 ... v_n e_n]
  e'_1 ... e'_k)
```

```
(vector
  (let [v_1 (get e_1 0) ... v_n (get e_n 0)]
    e'_1 ... e'_k)
  ⋮
  (let [v_1 (get e_1 (- c 1)) ... v_n (get e_n (- c 1))]
    e'_1 ... e'_k))
```

Interpretation:

- ***c*** is a non-negative integer constant
- ***foreach*** form iterates incrementally c-times over the structure ***[v_1 e_1 … v_n e_n]*** where it associates each variable $\boldsymbol{v_i}$ with a vector $\boldsymbol{e_i}$ and returns a vector composed of the c-index elements of vectors $\boldsymbol{e_i}$; consecutively this vector is passed to the $\boldsymbol{e_1'}$ ***…*** $\boldsymbol{e_k'}$ structure and can be referenced form there

FOPPL Loops (cont.)



For loop example:

```
foreach 5
 [x (range 1 6)
  y y-values]
```

Interpretation:

- ***c** = 5 (# of iterations)*
- iterations and vectors produced:

  c=0, (vector 1 2.1)

  c=1, (vector 2 3.9)

  c =3, (vector 3 5.3)

  c=4, (vector 4 7.7)

  c=5, (vector 5 10.2)

```
(let [y-values [2.1 3.9 5.3 7.7 10.2]
      slope (sample (normal 0.0 10.0))
      intercept (sample (normal 0.0 10.0))]
  (foreach 5
   [x (range 1 6)
    y y-values]
   (let [fx (+ (* slope x) intercept)]
     (observe (normal fx 1.0) y))))
```

FOPPL Sampling

Definition:

```
(let [xi sample(d, e1, ... , en)])
```

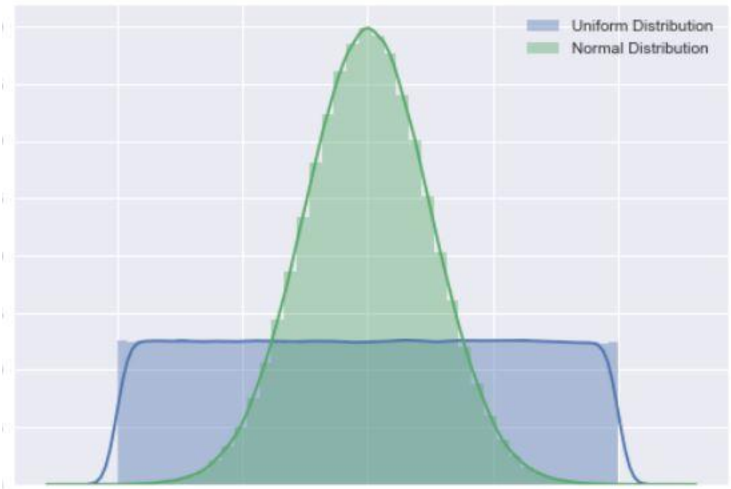
Interpretation:

- sample from the distribution ***d*** is taken and stored under the name ***xi***
- ***sample*** construct accepts a distribution object ***d***, which must evaluate to a distribution object and a set of expressions ***ei*** representing the distribution inputs
- distributions are constructed using primitives provided by the FOPPL

Example:

- ***x Uniform(0, 1)*** is represented as ***(let [x sample(uniform 0 1)])***

  and returns a value that is a sample from this distribution object
- ***(let [x sample (normal 0.0 1.0)])***
- ***(let [x sample (poisson 10.0)])***



FOPPL Observe Conditioning Statement

Definition:

```
observe( (d e1, ... , en)c)
```

Interpretation:

- 1) factors the density according to the distribution ***d***, with ***e1,..., en*** and the observed data ***c*** , and 2) represents an observed random variable
- accepts an argument ***d,*** which must evaluate to a distribution that provides the conditions on the next argument ***c***
- output: ***P(c|d)***

Example:

- (**observe (normal 0 1) 2) → P(2 | Normal(0,1))**
- **(observe (beta 1 5) 2) → P(2 | Beta(0,1))**

FOPPL If Statement

Definition:

If statement***: if (boolean expr.) expr1 expr2***

Examples:

```
(if (> mean 0)
        (observe (normal mean 1) y1)
        (observe (normal mean -1) y2))
```

If (mean > 0)
    (observe (normal mean 1) y1)
else
    (observe (normal mean 1) y1)

(if (< x-0.5 0)

    (observe (normal x 1.0) 2)

    (observe (beta x 5) 2)

)

If ((x-0.5) > 0)
    (observe (normal x 1.0) 2)
else
    (observe (Beta x 5) 2)

FOPPL Function Declaration

Definition:

- ***dffn*** statement***: (dffn*** *name* ***[****args****]*** *(body)* ***)***
- takes a set of arguments and a set of expressions in the body to evaluate during the forward execution

Example:

```
(defn observe-data [slope intercept x y]
  (let [fx (+ (* slope x) intercept)]
    (observe (normal fx 1.0) y)))
```

```
(let [y1 (observe-data slope intercept 1.0 2.1)]
```

FOPPL Example

Example: reasoning about the bias of a coin

```
(let [prior (beta a b)
      x (sample prior)
      likelihood (bernoulli x)
      y 1]
  (observe likelihood y)
  x)
```

Interpretation:

prior (beta a b) → ***beta*** distribution over ***a*** and ***b*** is assigned to ***prior*** variable

x (sample prior) → sample from the distribution ***prior*** is taken and stored in variable ***x***

Likelihood (bernoulli x) → ***bernoulli*** distribution over ***x*** is assigned to ***likelihood*** variable

y 1 → ***1*** (heads) is assigned to ***y*** variable

(observe likelihood y) → computes the posterior conditional probability ***p(y|likelihood)***

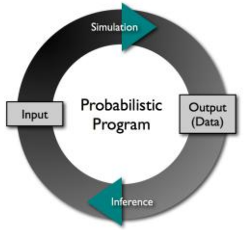
x → stores the value of ***x***

FOPPL Example (cont.)

Example: Bayesian linear regression

```
(let [y-values [2.1 3.9 5.3 7.7 10.2]
      slope (sample (normal 0.0 10.0))
      intercept (sample (normal 0.0 10.0))]
  (foreach 5
    [x (range 1 6)
     y y-values]
    (let [fx (+ (* slope x) intercept)]
      (observe (normal fx 1.0) y)))
[slope intercept])
```

Summary



First-Order Probabilistic Programming Language:

- includes most common features of programming languages;
- conditional statements, e.g., ***if***
- primitive operations, e.g.,***+,-,*,/***
- user-defined functions:
  - must be first order, i.e., functions cannot accept other functions as arguments
  - cannot be recursive

- FOPPL programs are models describing distributions over a finite number of random variables

Expectation:

- you are expected to be able to <u>read</u>/<u>interpret</u> FOPPL examples

Next Lesson – Lesson: Graph-Based Inference

Thank You!

Questions?