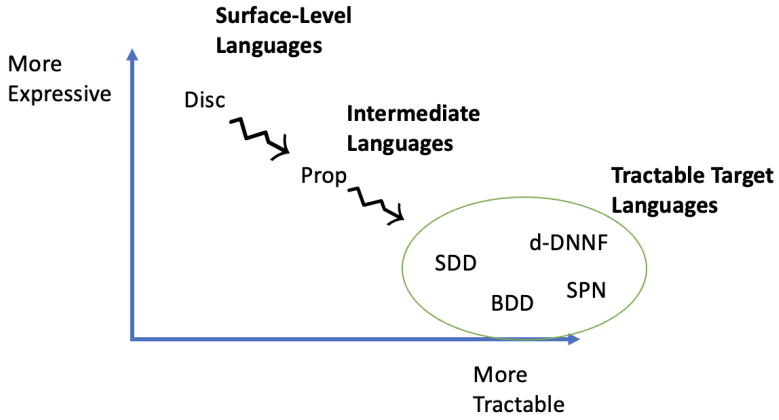
Discrete Probabilistic Programming Languages II[1]

Steven Holtzen
s.holtzen@northeastern.edu

October 3, 2023

[1] CS7470 Fall 2023: Foundations of Probabilistic Programming.

- We have been filling in the PPL compilation picture which organizes different kinds of probabilistic programming languages:



- We now understand every component. Recall:
  - A language consists of syntax and semantics. We customarily give two kinds of semantics: a denotational semantics that describes the mathematical object associated with the program, and an operational semantics that tells us precisely how to compute that object. We typically prove something called "adequacy" that relates the denotational and operational semantics.
  - A compilation $L_1 \rightsquigarrow L_2$ is a procedure for translating programs written in language L_1 into L_2 that preserves semantics.
  - Language L_1 is more expressive than L_2 if we can efficiently compile any program from L_2 into L_1 but not vice versa;
  - We can compare tractability of languages based on their worst-case runtime complexity.
- Some bibliographic notes:
  - The process of compiling a Boolean formula into a tractable target is often called *knowledge compilation* in the literature [Darwiche and Marquis, 2002]. There are many tools for compiling formulae. Some knowledge compilation tools:
    * c2d compiles from CNF to d-DNNF: `http://reasoning.cs.ucla.edu/c2d/`

    * minic2d compiles from CNF to SDD: `http://reasoning.cs.ucla.edu/minic2d/`
    * rsdd compiles from Prop to SDD and BDD: `https://github.com/neuppl/rsdd`
    * sharpSAT compiles from CNF to d-DNNF [Muise et al., 2012]
  - Disc is based on Dice [Holtzen et al., 2020]. There are other PPLs that work by knowledge compilation, such as ProbLog [Fierens et al., 2015, De Raedt et al., 2007], SPPL [Saad et al., 2021]. Historically, inference via compilation has been very effectively used in inference for graphical models [Chavira and Darwiche, 2008, Sang et al., 2005]

- What are some research questions in here?
  - Optimizing compilers for each stage.
  - New tractable target languages.
  - Compilation with performance guarantees.
  - Tools for making building these compilers.

1 Disc: *A simple discrete PPL*

- Syntax:

```
  e ::=
  | x ← e; e
  | flip q                          // q is a rational value
  | if e then e else e
  | return e
  | true | false
  | e ∧ e | e ∨ e | ¬ e |
  | ( e )
p ::= e
```

- Disc looks very similar to a standard functional programming language, but has two some interesting new keywords: `flip`, `observe`, and `return`
- `flip` θ allocates a new random quantity that is `true` with probability θ and `false` with probability $1 - \theta$
- `return e` turns a non-probabilistic quantity into a probabilistic one, i.e. `return true` is a *probabilistic quantity* that is `true` with probability 1 and `false` with probability 0
- Example program and its interpretation:

```
x ← flip 0.5;
y ← flip 0.5;
return x ∧ y
```

This program outputs the probability distribution $[\texttt{true} \mapsto 0.25, \texttt{false} \mapsto 0.75]$.

- **Type system**: terms can either be pure Booleans of type $\mathbb{B}$ or distributions on Booleans of type $\texttt{Dist}(\mathbb{B})$. So, we have the following type definition:

$$\tau ::= \mathbb{B} \mid \texttt{Dist}(\mathbb{B}). \tag{1}$$

- We define a typing judgment $\Gamma \vdash \texttt{e} : \tau$ that associates each term with a type. The typing context Γ is a map from identifiers to types.

$$\Gamma \vdash \texttt{true} : \mathbb{B} \qquad \Gamma \vdash \texttt{false} : \mathbb{B} \qquad \Gamma \vdash \texttt{flip}\ \theta : \texttt{Dist}(\mathbb{B})$$

$$\frac{\Gamma \vdash \texttt{e} : \mathbb{B}}{\Gamma \vdash \texttt{return e} : \texttt{Dist}(\mathbb{B})}$$

$$\frac{\Gamma \vdash \texttt{e}_1 : \texttt{Dist}(\mathbb{B}) \qquad \Gamma \cup [x \mapsto \mathbb{B}] \vdash \texttt{e}_2 : \tau}{\Gamma \vdash x \leftarrow \texttt{e}_1; \texttt{e}_2 : \tau}$$

$$\frac{\Gamma \vdash \texttt{e}_1 : \mathbb{B} \qquad \Gamma \vdash \texttt{e}_2 : \tau \qquad \Gamma \vdash \texttt{e}_3 : \tau}{\Gamma \vdash \texttt{if e}_1\ \texttt{then e}_2\ \texttt{else e}_3 : \tau} \qquad \frac{\Gamma \vdash \texttt{e}_1 : \mathbb{B} \qquad \Gamma \vdash \texttt{e}_2 : \mathbb{B}}{\Gamma \vdash \texttt{e}_1 \wedge \texttt{e}_2 : \mathbb{B}}$$

- This is often called a *monadic* type system [Moggi, 1991]; it waas introduced as a way of handling effects in purely functional languages. We won't get too into monads yet. If you're curious, see Ramsey and Pfeffer [2002].

Denotational semantics of Disc

- We will define our denotation on *closed terms*, i.e. terms with no free variables in them.
- There are two kinds of terms: *probabilistic* and *pure* terms. Probabilistic terms have type $\texttt{Dist}(\mathbb{B})$, and pure terms have type $\mathbb{B}$.
- The denotational semantics of pure terms is standard: they map closed terms to Boolean values. Examples of closed terms:

$$[\![\texttt{true}]\!] = \texttt{true}$$
$$[\![\texttt{false}]\!] = \texttt{false}$$
$$[\![\texttt{e}_1 \wedge \texttt{e}_2]\!] = [\![\texttt{e}_1]\!] \wedge [\![\texttt{e}_2]\!]$$

- The denotational semantics of probabilistic terms maps closed terms to probability distributions, $[\![\texttt{e}]\!] : \texttt{Bool} \to [0, 1]$, and has the following inductive definition:

$$[\![\texttt{flip}\ \theta]\!]\,(v) = \begin{cases} \theta & \text{if } v = T \\ 1-\theta & \text{if } v = F \end{cases}$$

$$[\![\texttt{return e}]\!]\,(v) = \begin{cases} 1 & \text{if } v = [\![\texttt{e}]\!] \\ 0 & \text{otherwise} \end{cases}$$

$$[\![x \leftarrow \texttt{e}_1; \texttt{e}_2]\!]\,(v) = \sum_{v'} [\![\texttt{e}_1]\!]\,(v') \times [\![\texttt{e}_2[v'/x]]\!]\,(v)$$

$$[\![\texttt{if true then e}_1\ \texttt{else e}_2]\!] = [\![\texttt{e}_1]\!]$$

$$[\![\texttt{if false then e}_1\ \texttt{else e}_2]\!] = [\![\texttt{e}_2]\!]$$

- Recall our notation for substitution $\texttt{e}[v/x]$, which substitutes the variable x for value v. Here we define substitution to be *capture-avoiding*. This concept is best illustrated by example. Considering the following program:

```
x ← true;
x ← false;
return true
```

  What should this program return? Intuitively, it should return the distribution $[\texttt{false} \mapsto 1, \texttt{true} \mapsto 0]$.

- A substitution $\texttt{e}[v/x]$ is *capture-avoiding* if it only substitutes *free occurrences* of x with v in e. A variable is **free** if it does not occur inside of a binding. We can accomplish this with the following inductive description of substitution (here we show only a subset of the rules):

$$\texttt{true}[x/v] = \texttt{true} \qquad \texttt{false}[x/v = \texttt{false}] \qquad x[x/v] = v$$

$$\frac{x \neq y}{y[x/v] = y} \qquad (x \leftarrow \texttt{e}_1; \texttt{e}_2)[v/x] = x \leftarrow \texttt{e}_1[v/x]; \texttt{e}_2$$

$$\frac{x \neq y}{(x \leftarrow \texttt{e}_1; \texttt{e}_2)[v/x] = y \leftarrow \texttt{e}_1[v/x]; \texttt{e}_2[v/x]}$$

- In your project, we have implemented capture-avoiding substitution for terms in Disc for you.

- A simple example of the let-rule:

$$[\![x \leftarrow \texttt{flip}\ \theta; \texttt{return}\ x]\!]\,(\texttt{true}) = \sum_{v \in \mathbb{B}} [\![\texttt{flip}\ \theta]\!]\,(v) \times [\![\texttt{return}\ v]\!]\,(\texttt{true}) = \theta$$

2 *Compiling* DISC *to* PROP

- **Goal**: give a semantics-preserving compilation $\rightsquigarrow$ that compiles DISC to PROP.
- We want to define this relation to satisfy the following **adequacy condition**, assuming the program $\mathtt{e} \rightsquigarrow \mathtt{p}$:

$$[\![\mathtt{e}]\!]\,(\mathtt{true}) = [\![\mathtt{p}]\!]\,. \tag{2}$$

- Compilation relation:

$$\mathtt{true} \rightsquigarrow (\mathtt{true}, \emptyset) \qquad \mathtt{false} \rightsquigarrow (\mathtt{false}, \emptyset)$$

$$x \rightsquigarrow (x, [x \mapsto 1/2, \overline{x} \mapsto 1/2]) \qquad \frac{\mathtt{fresh}\ f}{\mathtt{flip}\ \theta \rightsquigarrow (f, [f \mapsto \theta, \overline{f} \mapsto 1-\theta])}$$

$$\frac{\mathtt{e}_1 \rightsquigarrow (\varphi, w) \qquad \mathtt{e}_2 \rightsquigarrow (\varphi', w')}{x \leftarrow \mathtt{e}_1; \mathtt{e}_2 \rightsquigarrow (\varphi'[\varphi/x], w_1 \cup w_2)}$$

$$\frac{\mathtt{e}_1 \rightsquigarrow (\varphi_1, w_1) \qquad \mathtt{e}_2 \rightsquigarrow (\varphi_2, w_2)}{\mathtt{e}_1 \wedge \mathtt{e}_2 \rightsquigarrow (\varphi_1 \wedge \varphi_2, w_1 \cup w_2)}$$

$$\frac{\mathtt{e}_1 \rightsquigarrow (\varphi_1, w_1) \qquad \mathtt{e}_2 \rightsquigarrow (\varphi_2, w_2) \qquad \mathtt{e}_3 \rightsquigarrow (\varphi_3, w_3)}{\mathtt{if}\ \mathtt{e}_1\ \mathtt{then}\ \mathtt{e}_2\ \mathtt{else}\ \mathtt{e}_3 \rightsquigarrow ((\varphi_1 \wedge \varphi_2) \vee (\neg\varphi_1 \wedge \varphi_3), w_1 \cup w_2 \cup w_3)}$$

- One thing to notice above: we have a new substitution rule $\varphi'[\varphi/x]$ that substitutes x with an *arbitrary propositional formula* φ in φ'. For example:

$$(x \vee y)[z \wedge w/x] = (x \wedge y) \vee y.$$

  You can perform this operation directly on BDDs; it is called *BDD composition*.

- Now we are almost ready to state the semantics-preservation property of this compilation. We need one extra piece: *environment substitution*.
- Suppose $\Gamma \vdash \mathtt{e}$. A substitution $\gamma \in [\![\Gamma]\!]$ gives an assignment to every free variable in $\mathtt{e}$. We use the notation $\mathtt{e}[\gamma]$ to denote performing the substituting of all variables in γ in $\mathtt{e}$, i.e.:

$$\mathtt{e}[\{[v_1/x_1, v_2/x_2, \ldots, v_n/x_n]\}] = \mathtt{e}[v_1/x_1][v_2/x_2]\ldots[v_n/x_n].$$

  Additionally, note that we can apply this same substitution notation to propositional formulae, i.e. $\varphi[\gamma]$ is also well-defined according to the above description.

Theorem 1. *Assume* $\Gamma \vdash \mathsf{e}$ *and* $\mathsf{e} \rightsquigarrow \mathsf{p}$*. Then, for any substitution* $\gamma \in [\![\Gamma]\!]$*, is is the case that* $[\![\mathsf{e}[\gamma]]\!]\,(\mathtt{true}) = [\![\mathsf{p}[\gamma]]\!]$.

Proof. The proof is on structural induction on terms in `e`.

The base case for `flip` is simple but a good place to start. Assume $\mathtt{flip}\ \theta \rightsquigarrow (f, [f \mapsto \theta, \overline{f} \mapsto 1-\theta])$. Then,

$$[\![\mathtt{flip}\ \theta]\!]\,(\mathtt{true}) = \theta = \left[\!\!\left[(f, [f \mapsto \theta, \overline{f} \mapsto 1-\theta])\right]\!\!\right].$$

The critical case is the inductive step for binding. Inductive hypothesis: assume that $\mathsf{e}_1 \rightsquigarrow (\varphi_1, w_1)$ and $\mathsf{e}_2 \rightsquigarrow (\varphi_2, w_2)$, and that for any well-typed substitution γ it is the case that:

- $[\![\mathsf{e}_1[\gamma]]\!]\,(\mathtt{true}) = [\![\varphi_1[\gamma]]\!]$
- $[\![\mathsf{e}_2[\gamma]]\!]\,(\mathtt{true}) = [\![\varphi_2[\gamma]]\!]$

We need to show that $[\![x \leftarrow \mathsf{e}_1; \mathsf{e}_2]\!]\,(\mathtt{true}) = [\![\varphi_2[\varphi_1/x], (w_1 \cup w_2)]\!]$.

TODO; if you want, you can take a peek at Holtzen et al. [2020]

□

- Question: Is this an efficient compilation? *Yes*.

3 *Observation*

- `observe` is a powerful keyword that lets us *condition* the program. For instance, suppose I want to model the following scenario: "flip two coins and observe that at least one of them is heads. What is the probability that the first coin was heads?"

  We can encode this scenario as a Disc program:

```
x ← flip 0.5;
y ← flip 0.5;
observe x ∨ y;
return x
```

  This program outputs the probability distribution:

$$[\mathtt{true} \mapsto (0.25 + 0.25)/0.75, \mathtt{false} \mapsto 0.25/0.75]$$

- These semantics give an unnormalized distribution. The main semantic object of interest is the normalized distribution, which is given by the **normalized semantics**:

$$[\![e]\!]_D(T) = \frac{[\![e]\!](T)}{[\![e]\!](T) + [\![e]\!](F)},$$

  defined analogously for the false case.

- In order to handle observations, we will compile DISC programs into *two* PROP programs: one that computes the unnormalized probability of returning `true`, and one that computes the probability of evidence (i.e. normalizing constant)
- Inductive description has the shape $\mathtt{e} \rightsquigarrow (\mathtt{p}_1, \mathtt{p}_2)$.
- Compilation relation:

$$\mathtt{true} \rightsquigarrow (\mathtt{true}, \mathtt{true}, \emptyset) \qquad \mathtt{false} \rightsquigarrow (\mathtt{false}, \mathtt{true}, \emptyset)$$

$$x \rightsquigarrow (x, \mathtt{true}, \emptyset) \qquad \frac{\mathtt{fresh}\ x}{\mathtt{flip}\ \theta \rightsquigarrow (x, \mathtt{true}, [x \mapsto \theta, \overline{x} \mapsto 1 - \theta])}$$

$$\frac{\mathtt{e}_1 \rightsquigarrow (\varphi, \varphi_A, w) \qquad \mathtt{e}_2 \rightsquigarrow (\varphi', \varphi'_A, w')}{x \leftarrow \mathtt{e}_1; \mathtt{e}_2 \rightsquigarrow (\varphi'[\varphi/x], \varphi'_A[\varphi/x] \wedge \varphi_A, w_1 \cup w_2)}$$

$$\frac{\mathtt{e}_1 \rightsquigarrow (\varphi, \varphi_A, w) \qquad \mathtt{e}_2 \rightsquigarrow (\varphi', \varphi'_A, w')}{\mathtt{observe}\ \mathtt{e}_1; \mathtt{e}_2 \rightsquigarrow (\varphi', \varphi'_A \wedge \varphi_A, w_1 \cup w_2)}$$

$$[\![\mathtt{observe}\ e_1; e_2]\!](v) = \sum_{\{v' \mid [\![e_1(v')=T\}]\!]} [\![e_2]\!](v)$$

Theorem 2 (Adequacy)**.** *For well-typed term* $\mathtt{e}$*, assume* $\mathtt{e} \rightsquigarrow (\varphi, \varphi_A, w)$*. Then,* $[\![\mathtt{e}]\!]_D\,(\mathtt{true}) = [\![(\varphi, w)]\!]/[\![(\varphi_A, w)]\!]$.

References

Mark Chavira and Adnan Darwiche. On probabilistic inference by weighted model counting. *Artificial Intelligence*, 172(6-7):772–799, 2008.

Adnan Darwiche and Pierre Marquis. A knowledge compilation map. *Journal of Artificial Intelligence Research*, 17:229–264, 2002.

Luc De Raedt, Angelika Kimmig, Hannu Toivonen, and M Veloso. Problog: A probabilistic prolog and its application in link discovery. In *IJCAI 2007, Proceedings of the 20th international joint conference on artificial intelligence*, pages 2462–2467. IJCAI-INT JOINT CONF ARTIF INTELL, 2007.

Daan Fierens, Guy Van den Broeck, Joris Renkens, Dimitar Shterionov, Bernd Gutmann, Ingo Thon, Gerda Janssens, and Luc De Raedt. Inference and learning in probabilistic logic programs using weighted boolean formulas. *Theory and Practice of Logic Programming*, 15(3):358–401, 2015.

Steven Holtzen, Guy Van den Broeck, and Todd Millstein. Scaling exact inference for discrete probabilistic programs. *Proceedings of the ACM on Programming Languages*, 4(OOPSLA):1–31, 2020.

Eugenio Moggi. Notions of computation and monads. *Information and computation*, 93(1):55–92, 1991.

Christian Muise, Sheila A McIlraith, J Christopher Beck, and Eric I Hsu. D sharp: fast d-dnnf compilation with sharpsat. In *Advances in Artificial Intelligence: 25th Canadian Conference on Artificial Intelligence, Canadian AI 2012, Toronto, ON, Canada, May 28-30, 2012. Proceedings 25*, pages 356–361. Springer, 2012.

Norman Ramsey and Avi Pfeffer. Stochastic lambda calculus and monads of probability distributions. In *Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 154–165, 2002.

Feras A Saad, Martin C Rinard, and Vikash K Mansinghka. Sppl: probabilistic programming with fast exact symbolic inference. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, pages 804–819, 2021.

Tian Sang, Paul Beame, and Henry A Kautz. Performing bayesian inference by weighted model counting. In *AAAI*, volume 5, pages 475–481, 2005.