Discrete Probabilistic Programming from First Principles

Guy Van den Broeck

The Fourth International Workshop on Declarative Learning Based Programming (DeLBP)

Aug 11, 2019
What are probabilistic programs?

What is the formal semantics?

How to do exact inference?

What about approximate inference?
References


...with slides stolen from Steven Holtzen and Tal Friedman.
What are probabilistic programs?
What are probabilistic programs?

```
x ∼ flip(0.5);
y ∼ flip(0.7);
z := x || y;
if(z) {
    ...
}
observe(z);
```

- means “flip a coin, and output true with probability ½”
- Standard programming language constructs
- means “reject this execution if z is not true”
**Semantics** of a Probabilistic Program

A *probability distribution* on its states

\[
\begin{align*}
x & \sim \text{flip}(0.5); \\
y & \sim \text{flip}(0.7);
\end{align*}
\]

Goal: To perform *probabilistic inference*

- Compute the probability of some event
- Can be used for *Bayesian machine learning*: compute posterior (learned) parameters/structure given data

Joint Probability

![Joint Probability Chart](chart.png)
Why Probabilistic Programming?

• PPLs have grown in popularity: there are dozens

Pyro  Edward  Stan  Figaro

Venture, Church

ProbLog, PRISM, LPADs, CPLogic, ICL, PHA, etc.

• They are popular with practitioners
  • Specify a probability model in a familiar language
  • Expressive and concise
  • Cleanly separates model from inference
The Challenge of PPL Inference

Most popular inference algorithms are **black box**

- Treat program as a map from inputs to outputs
- Simplifying assumptions: differentiability, continuity
- Little to no effort to exploit program structure
  (black-box variational, Hamiltonian MC)
- Approximate inference 😞
Why Discrete Models?

1. Real programs have inherent discrete structure (e.g. if-statements)

2. Discrete structure is important in modeling (graphs, topic models, etc.)

3. Many existing systems assume smooth and differentiable densities:

   Discrete probabilistic programming is the important unsolved open problem!
What is the formal semantics?
Simple Discrete PPL Syntax
(stmtements and expressions)

s ::=  
   | s; s
   | x := e
   | x \sim flip(\theta)
   | if e { s } else { s }
   | observe(e)
   | skip

e ::=  
   | x
   | T | F
   | e \lor e
   | e \land e
   | \neg e
Semantics

• The program state is a map from variables to values, denoted $\sigma$

• The goal of our semantics is to associate
  – statements in the syntax with
  – a probability distribution on states

• Notation: semantic brackets $[[s]]$
Sampling Semantics

• The simplest way to give a semantics to our language is to *run the program infinite times*

\[ x \sim \text{flip}(0.5); \]

• The probability distribution of the program is defined as the *long run average* of how often it ends in a particular state
Semantics of \( \omega_1 \)

\[
x = \text{true} \\
y = \text{true}
\]

\( 0.5 \times 0.7 = 0.35 \)

\[
\omega_2
\]

\[
x = \text{false} \\
y = \text{true}
\]

\( 0.5 \times 0.7 = 0.35 \)

\[
\omega_3
\]

\[
x = \text{false} \\
y = \text{false}
\]

\( 0.5 \times 0.3 = 0.15 \)

\[
\omega_4
\]

\[
x = \text{true} \\
y = \text{false}
\]

\( 0.5 \times 0.3 = 0.15 \)

x \sim \text{flip}(0.5); \\
y \sim \text{flip}(0.7);
Semantics of

\[ x = \text{true} \]
\[ y = \text{true} \]
\[ \omega_1 = 0.5 \times 0.7 = 0.35 \]

\[ x = \text{false} \]
\[ y = \text{false} \]
\[ \omega_3 = 0.5 \times 0.3 = 0.15 \]

\[ x = \text{true} \]
\[ y = \text{false} \]
\[ \omega_2 = 0.7 = 0.35 \]

\[ x = \text{false} \]
\[ y = \text{true} \]
\[ \omega_4 = 0.3 \]

Semantics: Throw away all executions that do not satisfy the condition \( x \lor y \).
Rejection Sampling Semantics

• Observes give a *posterior distribution* on the program states

• Semantics of a program: draw (infinite) samples, take the long run average over *accepted samples*

\[
\begin{array}{c|c|c}
\sigma \\
\hline
x=true & y=true \\
\hline
x=false & x=false \\
\hline
x=true & y=false \\
\hline
x=false & y=true \\
\end{array}
\]

\[
x \sim \text{flip}(0.5); \\
y \sim \text{flip}(0.7); \\
\text{observe}(x \ | \ | \ y);
\]
Rejection Sampling Semantics

- Extremely general: you only need to be able to run the program to implement a rejection-sampling semantics.
- This how most AI researchers think about the meaning of their programs (?)

- “Procedural”: the meaning of the program is whatever it executes to ...not entirely satisfying...
- A sample is a full execution: a global property that makes it harder to think modularly about local meaning of code.

Next: the gold standard in programming languages denotational semantics
Denotational Semantics

- Idea: We don’t have to *run* a flip statement to know what its distribution is.
- For some input state $\sigma$ and output state $\sigma'$, we can directly compute the probability of transitioning from $\sigma$ to $\sigma'$ upon executing a flip statement:

$$\text{Run } x \sim \text{flip}(0.4) \text{ on } \sigma$$

- $\Pr = 0.4$
- $\Pr = 0.6$

We can avoid having to think about sampling!
Denotational Semantics of Flip

Idea: Directly define the probability of transitioning upon executing each statement

Call this its *denotation*, written $\llbracket S \rrbracket$

$$\llbracket x \sim \text{flip}(\theta) \rrbracket(\sigma' | \sigma) \triangleq \begin{cases} 
\theta & \text{if } \sigma' = \sigma[x \mapsto T] \\
1 - \theta & \text{if } \sigma' = \sigma[x \mapsto F] \\
0 & \text{otherwise}
\end{cases}$$

Semantic bracket: associate semantics with syntax
Output state
Input State
Assign $x$ to false in the state $\sigma$
Semantics of Expressions

• What about $x := e$?
• Need semantics for expressions: simple
• Just evaluate the expression $e$ on state $\sigma$

\[
\left[ b \lor c \right]_{\sigma}(\{b \mapsto T, c \mapsto F\}) = T
\]
Semantics of Assignments

What about $x := e$?

$$\llbracket x := e \rrbracket(\sigma' \mid \sigma) \triangleq \begin{cases} 1 & \sigma' = \sigma[x \mapsto \llbracket e \rrbracket(\sigma)] \\ 0 & \text{otherwise} \end{cases}$$

(semantics of if-then-else also based on if-test expression)
Semantics of Sequencing

• Assume the program has no observe statements
• We can compute the denotation of sequencing by *marginalizing out the intermediate state*

\[
\begin{split}
\llbracket s_1; s_2 \rrbracket(\sigma' \mid \sigma) &= \sum_{\tau} \llbracket s_1 \rrbracket(\sigma \mid \tau) \times \llbracket s_2 \rrbracket(\sigma' \mid \tau) \\
\end{split}
\]

Example:

\[
\begin{split}
\llbracket x \sim \text{flip}(0.4); y \sim \text{flip}(0.1) \rrbracket(\{x \mapsto T, y \mapsto F\} \mid \emptyset) \\
= \sum_{\tau \in \{\{x \mapsto T\}, \{x \mapsto F\}\}} \llbracket x \sim \text{flip}(0.4) \rrbracket(\tau \mid \emptyset) \times \llbracket y \sim \text{flip}(0.1) \rrbracket(\{x \mapsto T, y \mapsto F\} \mid \tau) \\
= 0.4 \cdot 0.9 + 0.6 \cdot 0
\end{split}
\]
Semantics of Observations

• What if we introduce observations *only at the end* of the program?

\[
[s; \text{observe}(e)](\sigma' \mid \sigma)
\]

\[\Delta \begin{cases} \frac{[s](\sigma' \mid \sigma)}{\sum_{\tau \models [e]} [s](\tau \mid \sigma)} \quad & \sigma' \models [e] \\ 0 \quad & \text{otherwise} \end{cases}\]

• Bayes rule “given that the observe succeeds”
• Look ma! No rejected samples!
What is the meaning of?

\[ bar_1 = \begin{cases} 
\text{if}(x) \{ y \sim \text{flip}(1/4) \} \\
\text{else} \{ y \sim \text{flip}(1/2) \} 
\end{cases} \]

\[ [\bar{bar}_1]_T(\sigma' | \sigma) = \begin{cases} 
1/2 & \text{if } x[\sigma] = x[\sigma'] = F, \\
1/4 & \text{if } x[\sigma] = x[\sigma'] = T \text{ and } y[\sigma'] = T, \\
3/4 & \text{if } x[\sigma] = x[\sigma'] = T \text{ and } y[\sigma'] = F, \\
0 & \text{otherwise.}
\]
What is the meaning of?

\[ \text{bar}_2 = \begin{cases} 
  y \sim \text{flip}(1/2); \\
  \text{observe}(x \lor y); \\
  \text{if}(y) \{ y \sim \text{flip}(1/2) \} \\
  \text{else} \{ y := \text{F} \} 
\end{cases} \]

\[
\begin{align*}
\llbracket \text{bar}_1 \rrbracket_T(\sigma') | \sigma) &= \begin{cases} 
  1/2 & \text{if } x[\sigma] = x[\sigma'] = \text{F}, \\
  1/4 & \text{if } x[\sigma] = x[\sigma'] = \text{T} \text{ and } y[\sigma'] = \text{T}, \\
  3/4 & \text{if } x[\sigma] = x[\sigma'] = \text{T} \text{ and } y[\sigma'] = \text{F}, \\
  0 & \text{otherwise.}
\end{cases}
\end{align*}
\]

\[
\llbracket \text{bar}_2 \rrbracket_T(\sigma') | \sigma) = \begin{cases} 
  1/2 & \text{if } x[\sigma] = x[\sigma'] = \text{F}, \\
  1/4 & \text{if } x[\sigma] = x[\sigma'] = \text{T} \text{ and } y[\sigma'] = \text{T}, \\
  3/4 & \text{if } x[\sigma] = x[\sigma'] = \text{T} \text{ and } y[\sigma'] = \text{F}, \\
  0 & \text{otherwise.}
\end{cases}
\]
Are these programs equivalent?

\[
bar_1 = \begin{cases} 
\text{if}(x) \{ \ y \sim \text{flip}(1/4) \} \\
\text{else} \{ \ y \sim \text{flip}(1/2) \} 
\end{cases}
\]

\[
bar_2 = \begin{cases} 
 y \sim \text{flip}(1/2); \\
\text{observe}(x \lor y); \\
\text{if}(y) \{ \ y \sim \text{flip}(1/2) \} \\
\text{else} \{ \ y := F \} 
\end{cases}
\]
Are these programs equivalent?

\[ \text{foo} = \{ \ x \sim \text{flip}(1/3) \ \} \]

\[ \text{bar}_1 = \begin{cases} \text{if}(x) \{ y \sim \text{flip}(1/4) \} \\ \text{else} \{ y \sim \text{flip}(1/2) \} \end{cases} \]

\[ \text{bar}_2 = \begin{cases} y \sim \text{flip}(1/2); \\ \text{observe}(x \lor y); \\ \text{if}(y) \{ y \sim \text{flip}(1/2) \} \\ \text{else} \{ y := F \} \end{cases} \]

In \( [[\text{foo}; \text{bar}_1]] \) the probability of \( x = F \) in the output state is: \( \frac{2}{3} \)

In \( [[\text{foo}; \text{bar}_2]] \) the probability of \( x = F \) in the output state is:

\[
\frac{2/3 \cdot 1/2}{1/3 + 2/3 \cdot 1/2} = \frac{1}{2}
\]
Accepting and Transition Semantics

\[
\begin{align*}
\llbracket \text{skip}(e) \rrbracket_A(\sigma) & \triangleq 1 \\
\llbracket x \sim \text{flip}(\theta) \rrbracket_A(\sigma) & \triangleq 1 \\
\llbracket x := e \rrbracket_A(\sigma) & \triangleq 1 \\
\llbracket \text{observe}(e) \rrbracket_A(\sigma) & \triangleq \begin{cases} 
 1 & \text{if } \llbracket e \rrbracket(\sigma) = T \\
 0 & \text{otherwise}
\end{cases} \\
\llbracket s_1; s_2 \rrbracket_A(\sigma) & \triangleq \llbracket s_1 \rrbracket_A(\sigma) \times \sum_{\tau \in \Sigma} (\llbracket s_1 \rrbracket_T(\tau | \sigma) \times \llbracket s_2 \rrbracket_A(\tau)) \\
\llbracket \text{if } e \{ s_1 \} \text{ else } \{ s_2 \} \rrbracket_A(\sigma) & \triangleq \begin{cases} 
 \llbracket s_1 \rrbracket_A(\sigma) & \text{if } \llbracket e \rrbracket(\sigma) = T \\
 \llbracket s_2 \rrbracket_A(\sigma) & \text{otherwise}
\end{cases}
\end{align*}
\]

\[
\begin{align*}
\llbracket \text{skip} \rrbracket_T(\sigma' | \sigma) & \triangleq \begin{cases} 
 1 & \text{if } \sigma' = \sigma \\
 0 & \text{otherwise}
\end{cases} \\
\llbracket x \sim \text{flip}(\theta) \rrbracket_T(\sigma' | \sigma) & \triangleq \begin{cases} 
 \theta & \text{if } \sigma' = \sigma[x \mapsto T] \\
 1 - \theta & \text{if } \sigma' = \sigma[x \mapsto F] \\
 0 & \text{otherwise}
\end{cases} \\
\llbracket x := e \rrbracket_T(\sigma' | \sigma) & \triangleq \begin{cases} 
 1 & \text{if } \sigma' = \sigma[x \mapsto \llbracket e \rrbracket(\sigma)] \\
 0 & \text{otherwise}
\end{cases} \\
\llbracket \text{observe}(e) \rrbracket_T(\sigma' | \sigma) & \triangleq \begin{cases} 
 1 & \text{if } \sigma' = \sigma \text{ and } \llbracket e \rrbracket(\sigma) = T \\
 0 & \text{otherwise}
\end{cases} \\
\llbracket s_1; s_2 \rrbracket_T(\sigma' | \sigma) & \triangleq \\
\frac{\sum_{\tau \in \Sigma} \llbracket s_1 \rrbracket_T(\tau | \sigma) \times \llbracket s_2 \rrbracket_T(\sigma' | \tau) \times \llbracket s_2 \rrbracket_A(\tau)}{\sum_{\tau \in \Sigma} \llbracket s_1 \rrbracket_T(\tau | \sigma) \times \llbracket s_2 \rrbracket_A(\tau)} \\
\llbracket \text{if } e \{ s_1 \} \text{ else } \{ s_2 \} \rrbracket_T(\sigma' | \sigma) & \triangleq \\
\begin{cases} 
 \llbracket s_1 \rrbracket_T(\sigma' | \sigma) & \text{if } \llbracket e \rrbracket(\sigma) = T \\
 \llbracket s_2 \rrbracket_T(\sigma' | \sigma) & \text{if } \llbracket e \rrbracket(\sigma) = F
\end{cases}
\end{align*}
\]
Pitfalls of Denotational Semantics

• Intermediate observes:
  • Need accepting semantic
  • Key difference from probabilistic graphical models
  • Sometimes encoded using unnormalized probabilities

• While loops
  • Bounded? “while(i<10)”
  • Almost surely terminating? “while(flip(0.5))”
  • Not almost surely terminating? “while(true)”

• Adding continuous variables:
  • Indian GPA problem [Wu et al. ICML 2018]
  • What is the meaning of “if(Normal(0,1) == 0.34) then ...“
  • Etc.
How to do exact inference for probabilistic programs?
The Challenge of PPL Inference

• Probabilistic inference is \#P-hard
  – Implies there is likely no universal solution

• In practice inference is often feasible
  – Often relies on conditional independence
  – Manifests as *graph properties*

• *Why exact?*
  1. No error propagation
  2. Approximations are intractable in theory as well
  3. Approximates are known to mislead learners
  4. Core of effective approximation techniques
  5. Unaffected by low-probability observations
Techniques for exact inference

<table>
<thead>
<tr>
<th>Exploits independence to decompose inference?</th>
<th>Graphical Model Compilation</th>
<th>Symbolic compilation (This work)</th>
<th>Enumeration</th>
</tr>
</thead>
<tbody>
<tr>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>No</td>
<td>No</td>
<td></td>
<td>Yes</td>
</tr>
</tbody>
</table>

Keeps program structure?
PL Background: Symbolic Execution

• Non-probabilistic programs can be interpreted as *logical formulae* which relate input and output states

\[
x := y; \\
\varphi = (x' \iff y) \land (y' \iff y)
\]

\[
SAT(\varphi \land x' \land y) = T \\
SAT(\varphi \land x' \land \bar{y}) = F
\]

Output state: primed
Input state: unprimed
Our Approach: Inference via Weighted Model Counting

- Probabilistic Program
- Symbolic Compilation
  - Retains Program Structure
- Weighted Boolean Formula
  - Exploits Independence
- WMC
  - Binary Decision Diagram
- WMC
  - Query Result
Inference via Weighted Model Counting

x := flip(0.4);

\[
\begin{array}{|c|c|}
\hline
l & w(l) \\
\hline
f_1 & 0.4 \\
\overline{f_1} & 0.6 \\
\hline
\end{array}
\]

\[
(x' \Leftrightarrow f_1)
\]

WMC(\varphi, w) = \sum_{m \models \varphi} \prod_{l \in m} w(l).

\[
WMC((x' \Leftrightarrow f_1) \land x \land x', w)\]

- A single model: \(m = x' \land x \land f_1\)
- \(w(x') \ast w(x) \ast w(f_1) = 0.4\)
Symbolic compilation: Flip

- Compositional process \( s \sim (\varphi, w) \)

\[
x \sim \text{flip}(\theta) \sim \left( (x' \Leftrightarrow f) \land \text{(rest unchanged)}, w \right)
\]

All variables in the program except for \( x \) are not changed by this statement.
Symbolic compilation: Assignment

• Compositional process $\mathcal{S} \rightsquigarrow (\varphi, w)$

\[
x := e \rightsquigarrow \left( (x' \iff e) \land (\text{rest unchanged}), w \right)
\]
Compiling to BDDs

• BDDs compactly capture complex program structure

\[ x = a \lor b \lor c \lor d \lor e \lor f; \]
Symbolic compilation: Sequencing

• Compositional process \( s \leadsto (\varphi, w) \)

\[
s_1 \leadsto (\varphi_1, w_1) \quad s_2 \leadsto (\varphi_2, w_2)
\]

\[
\varphi'_2 = \varphi_2[x_i \mapsto x'_i, x'_i \mapsto x''_i]
\]

\[
s_1; s_2 \leadsto ((\exists x'_i. \varphi_1 \land \varphi'_2)[x''_i \mapsto x'_i], w_1 \uplus w_2)
\]

• Compile two sub-statements, do some relabeling, then combine them to get the result
Inference via Weighted Model Counting

1. Probabilistic Program
2. Symbolic Compilation
3. Weighted Boolean Formula
4. WMC
5. Query Result
6. Binary Decision Diagram

Process:
- Probabilistic Program is symbolically compiled.
- The symbolic compilation results in a Weighted Boolean Formula.
- The Weighted Boolean Formula is evaluated using WMC to produce a Query Result.
- The Binary Decision Diagram is used to further analyze the results.
Compiling to BDDs

- Consider an example program:

\[
\begin{align*}
x &\sim \text{flip}(0.4); \\
y &\sim \text{flip}(0.6)
\end{align*}
\]

\[
(x \iff f_1) \land (y \iff f_2)
\]

- WMC is efficient for BDDs: *time linear* in size
  - Small BDD = Fast Inference

This sub-function does not depend on x: exploits independence
BDDs Exploit
Conditional Independence

Size of BDD grows linearly with length of Markov chain

1 \( x \sim \text{flip}_x(0.5); \)
2 \( \text{if}(x) \{ y \sim \text{flip}_1(0.6) \} \)
3 \( \text{else} \{ y \sim \text{flip}_2(0.4) \}; \)
4 \( \text{if}(y) \{ z \sim \text{flip}_3(0.6) \} \)
5 \( \text{else} \{ z \sim \text{flip}_4(0.9) \} \)

Given \( y=T \), does not depend on
the value of \( X \): exploits
conditional independence
z ~ flip₁(0.5);
if (z) {
    x ~ flip₂(0.6);
    y ~ flip₃(0.7)
} else {
    x ~ flip₄(0.4);
    y := x
}
Experiments: Markov Chain

- **Symbolic (This Work)**
- **Psi**
- **WebPPL**

Time (s) vs. Length of Markov Chain
Experiment: Bayesian Networks

Large programs (thousands of lines, tens of thousands of flips)

<table>
<thead>
<tr>
<th>Model</th>
<th>Us (s)</th>
<th>BN Time (s) [6]</th>
<th>Size of BDD</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alarm [6]</td>
<td>1.872</td>
<td>0.21</td>
<td>52k</td>
</tr>
<tr>
<td>Halfinder</td>
<td>12.652</td>
<td>1.37</td>
<td>157k</td>
</tr>
<tr>
<td>Hepar2</td>
<td>7.834</td>
<td>0.28 [11]</td>
<td>139k</td>
</tr>
<tr>
<td>pathfinder</td>
<td>62.034</td>
<td>14.94</td>
<td>392k</td>
</tr>
</tbody>
</table>

Specialized BN inference algorithm

Alarm Network

Pathfinder Network
Symbolic Compilation

• Exact inference algorithm for discrete programs
  • Relies on PL ideas to construct state space: symbolic execution, symbolic model checking
  • Relies on AI ideas to perform inference: weighted model counting, knowledge compilation
• Proved **correct** (= denotational semantics)
• Competitive performance
• Will release a language+system soon!
• Also see probabilistic **logic** programming work
What about approximate inference?
Compilation

Exact Independence Properties
Logical Structure

Sampling

Approx Scalable Anytime

Collapsed Compilation
Collapsed Sampling
(Rao-Blackwell)

Sampling on some variables, exact inference conditioned on sample

Sample A,B
Collapsed Sampling (Rao-Blackwell)

Sampling on some variables, exact inference conditioned on sample

Observe sampled values
Collapsed Sampling (Rao-Blackwell)

Sampling on some variables, exact inference conditioned on sample

Compute exactly $P(C|A,B)$
What to Sample?

- Is it even possible to pick a correct set a priori?
- Consider a network of potential smokers, with friendships sampled

Sample 1

Sample 2
Online Collapsed Sampling

Choose \textit{on-the-fly} which variable to sample next, based on result of sampling previous variables

\textbf{Theorem}: Still unbiased
How to do Collapsed Sampling?

1. What/when do we sample?
2. How do we sample?
3. How do we do exact inference?
Collapsed Compilation

Result: A circuit with some sampled variables
How to do Collapsed Compilation?

1. What/when do we sample?
   - *When*: Circuit too big
   - *What*: Heuristic on current circuit
     Intuition: variables with dense weak dependencies

2. How do we sample?

3. How do we do exact inference?
How to do Collapsed Compilation?

1. What/when do we sample?
2. How do we sample?
   – Importance Sampling
   – Need a proposal for any variable conditioned on any other variables
   – Sample according to marginal in current partially compiled circuit
3. How do we do exact inference?
How to do Collapsed Compilation?

1. What/when do we sample?
2. How do we sample?
3. How do we do exact inference?
   - Compiled circuit for each sample
   - Tractable for all required computations (marginals, particle weights, etc.)
Collapsed Compilation Algorithm

To sample a circuit:

1. Compile bottom up until you reach the size limit
2. Pick a variable you want to sample
3. Sample it according to its marginal distribution in the current circuit
4. Condition on the sampled value
5. (Repeat)

Asymptotically unbiased importance sampler 😊
Circuits + importance weights approximate any query
# Experiments

Table 2: Hellinger distances across methods with internal treewidth and size bounds

<table>
<thead>
<tr>
<th>Method</th>
<th>50-20</th>
<th>75-26</th>
<th>DBN</th>
<th>Grids</th>
<th>Segment</th>
<th>linkage</th>
<th>frust</th>
</tr>
</thead>
<tbody>
<tr>
<td>EDBP-100k</td>
<td>2.19e-3</td>
<td>3.17e-5</td>
<td>6.39e-1</td>
<td>1.24e-3</td>
<td>1.63e-6</td>
<td>6.54e-8</td>
<td>4.73e-3</td>
</tr>
<tr>
<td>EDBP-1m</td>
<td>7.40e-7</td>
<td>2.21e-4</td>
<td>6.39e-1</td>
<td>1.98e-7</td>
<td>1.93e-7</td>
<td>5.98e-8</td>
<td>4.73e-3</td>
</tr>
<tr>
<td>SS-10</td>
<td>2.51e-2</td>
<td>2.22e-3</td>
<td>6.37e-1</td>
<td>3.10e-1</td>
<td>3.11e-7</td>
<td>4.93e-2</td>
<td>1.05e-2</td>
</tr>
<tr>
<td>SS-12</td>
<td>6.96e-3</td>
<td>1.02e-3</td>
<td>6.27e-1</td>
<td>2.48e-1</td>
<td>3.11e-7</td>
<td>1.10e-3</td>
<td>5.27e-4</td>
</tr>
<tr>
<td>SS-15</td>
<td>9.09e-6</td>
<td>1.09e-4</td>
<td>(Exact)</td>
<td>8.74e-4</td>
<td>3.11e-7</td>
<td>4.06e-6</td>
<td>6.23e-3</td>
</tr>
<tr>
<td>FD</td>
<td>9.77e-6</td>
<td>1.87e-3</td>
<td>1.24e-1</td>
<td>1.98e-4</td>
<td>6.00e-8</td>
<td>5.99e-6</td>
<td>5.96e-6</td>
</tr>
<tr>
<td>MinEnt</td>
<td>1.50e-5</td>
<td>3.29e-2</td>
<td>1.83e-2</td>
<td>3.61e-3</td>
<td>3.40e-7</td>
<td>6.16e-5</td>
<td>3.10e-2</td>
</tr>
<tr>
<td>RBVar</td>
<td>2.66e-2</td>
<td>4.39e-1</td>
<td>6.27e-3</td>
<td>1.20e-1</td>
<td>3.01e-7</td>
<td>2.02e-2</td>
<td>2.30e-3</td>
</tr>
</tbody>
</table>

Competitive with state-of-the-art approximate inference in graphical models. Outperforms it on several benchmarks!
Conclusions

Programming Languages
- Symbolic Execution
- Abstract Interpretation
- Predicate Abstraction
- Model Checking
- Weakest Precondition

Artificial Intelligence
- Weighted Model Counting
- Bayesian Networks
- Independence
- Lifted Inference
- Knowledge Compilation

Intersecting Concepts:
- Symbolic Compilation
- Probabilistic Predicate Abstraction

Fun with Discrete Structure
Thanks


...with slides stolen from Steven Holtzen and Tal Friedman.