

Scalable Reasoning for Probabilistic Programs

Steven Holtzen and John M. Li

SPLASH/ISSTA 2026 Summer School

October 6, 2026

Northeastern University

`s.holtzen@northeastern.edu`

`li.john@northeastern.edu`

`https://www.khoury.northeastern.edu/home/sholtzen/assets/pdf/splash-2026.pdf`

Talk goal: to give an overview of the challenges and opportunities for probabilistic programming languages.

Talk goal: to give an overview of the challenges and opportunities for probabilistic programming languages.

1. Introduction and overview of probabilistic programming languages (PPLs): programming languages with probabilistic semantics

Talk goal: to give an overview of the challenges and opportunities for probabilistic programming languages.

1. Introduction and overview of probabilistic programming languages (PPLs): programming languages with probabilistic semantics
2. Roulette: deep dive into a particular PPL's implementation

Talk goal: to give an overview of the challenges and opportunities for probabilistic programming languages.

1. Introduction and overview of probabilistic programming languages (PPLs): programming languages with probabilistic semantics
2. Roulette: deep dive into a particular PPL's implementation
3. Symbolic sets: a semantic framework for PPLs (presented by John M. Li)

Where does probability come from?

Where does probability come from?

- **Abstraction:** a tool to abstract a complex computational situation. Physics, the weather, human behavior, etc.



Where does probability come from?

- **Abstraction:** a tool to abstract a complex computational situation. Physics, the weather, human behavior, etc.



- **Algorithmic tool:** a technique to design software with desirable properties. Cryptography, fair scheduling, etc.



Where does probability come from?

- **Abstraction:** a tool to abstract a complex computational situation. Physics, the weather, human behavior, etc.



- **Algorithmic tool:** a technique to design software with desirable properties. Cryptography, fair scheduling, etc.



- **Reasoning framework:** it is an epistemic framework for acquiring and composing together new information.



Two arguments for probabilistic programming languages

Two arguments for probabilistic programming languages

Programming Languages

Two arguments for probabilistic programming languages

Programming Languages

1. It is useful to verify programs.

Two arguments for probabilistic programming languages

Programming Languages

1. It is useful to verify programs.
2. Many kinds of programs have inherent probabilistic uncertainty.

Two arguments for probabilistic programming languages

Programming Languages

1. It is useful to verify programs.
2. Many kinds of programs have inherent probabilistic uncertainty.
3. We need programming languages with probabilistic semantics to give a semantics to these programs.

Two arguments for probabilistic programming languages

Programming Languages

1. It is useful to verify programs.
2. Many kinds of programs have inherent probabilistic uncertainty.
3. We need programming languages with probabilistic semantics to give a semantics to these programs.

Artificial Intelligence

Two arguments for probabilistic programming languages

Programming Languages

1. It is useful to verify programs.
2. Many kinds of programs have inherent probabilistic uncertainty.
3. We need programming languages with probabilistic semantics to give a semantics to these programs.

Artificial Intelligence

1. We want to create agents that reason rationally about the world.

Two arguments for probabilistic programming languages

Programming Languages

1. It is useful to verify programs.
2. Many kinds of programs have inherent probabilistic uncertainty.
3. We need programming languages with probabilistic semantics to give a semantics to these programs.

Artificial Intelligence

1. We want to create agents that reason rationally about the world.
2. To do this, we need a language for describing the world to a computer.

Two arguments for probabilistic programming languages

Programming Languages

1. It is useful to verify programs.
2. Many kinds of programs have inherent probabilistic uncertainty.
3. We need programming languages with probabilistic semantics to give a semantics to these programs.

Artificial Intelligence

1. We want to create agents that reason rationally about the world.
2. To do this, we need a language for describing the world to a computer.
3. The world is too complicated to describe without probabilities.

Two arguments for probabilistic programming languages

Programming Languages

1. It is useful to verify programs.
2. Many kinds of programs have inherent probabilistic uncertainty.
3. We need programming languages with probabilistic semantics to give a semantics to these programs.



Tony Hoare

Artificial Intelligence

1. We want to create agents that reason rationally about the world.
2. To do this, we need a language for describing the world to a computer.
3. The world is too complicated to describe without probabilities.
4. We need programming languages with probabilistic semantics.



Judea Pearl

What are probabilistic programming languages (PPLs)?

Programs that denote probability distributions.

```
> (flip 1/3)
```

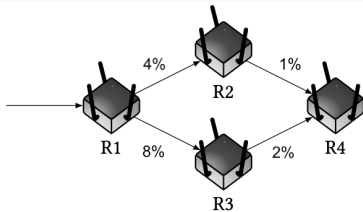
```
Value Probability
```

```
#t    1/3
```

```
#f    2/3
```

-
- Programming in Roulette [29]
 - `(flip 0.4)` *introduces* randomness, `#t` with probability 0.4 and `#f` with probability 0.6.
 - Outputs a probability mass function (PMF), associates values to probabilities

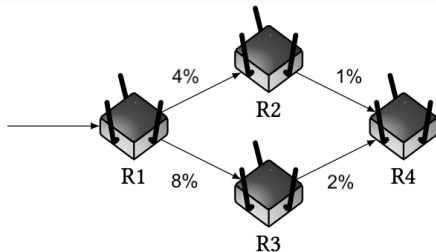
Programming with probabilities: network reachability



Goal: Compute probability an incoming packet reaches R4:

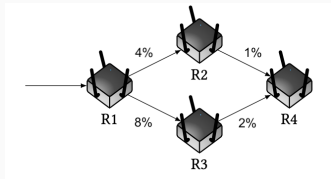
- Packet forwards from R1 uniformly at random
- Each edge fails with the annotated probability

Programming with probabilities: network reachability



```
> (if (flip 0.5) ; if true, forward to R2
    (and (flip 0.96) (flip 0.99))
    (and (flip 0.92) (flip 0.98)))
(pmf | #t  $\mapsto$  0.926 | #f  $\mapsto$  0.074)
```

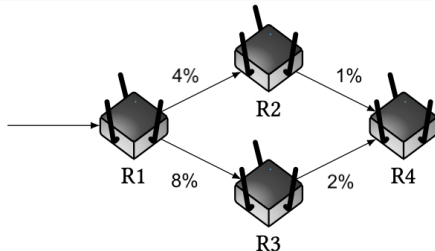
Programming with probabilities: network diagnosis



Now suppose we want to perform *network diagnosis*.

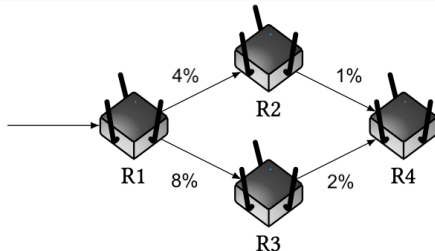
We **observe** a packet does not successfully reach R4. What is the probability that the R1 – R4 link failed?

Programming with probabilities: network diagnosis



```
> (define r1->r2 (flip 0.96))
> (define reaches (if (flip 0.5) ; if true, forward to R2
  (and r1->r2 (flip 0.99))
  (and (flip 0.92) (flip 0.98))))
> (observe! (not reaches))
> r1->r2
(pmf | #t ↦ 0.703135 | #f ↦ 0.296865)
```

Programming with probabilities: network diagnosis



```
> (define r1->r2 (flip 0.96))  
> (define reaches (if (flip 0.5) ; if true, forward to R2  
  (and r1->r2 (flip 0.99))  
  (and (flip 0.92) (flip 0.98))))  
> (observe! (not reaches))  
> r1->r2
```

Called *Bayesian condition-*
ing.

$\#f \mapsto 0.296865$)

Some applications of PPLs, real and imagined

- Teaching probability
- Network verification [12, 35]
- Medical diagnosis
- Scientific discovery
- Differential privacy [31, 3, 9]
- Fraud detection
- Verifying cryptography
- Randomized algorithms
- (Verified) machine learning
- Markov decision processes [19]
- Pharmaceuticals research [30]
- Social sciences [14]
- Content generation for games [28]
- ...

There is *high demand* for effective PPLs!

A monadic probabilistic programming language with
Booleans [13, 32]

TinyPPL: Syntax

A monadic probabilistic programming language with
Booleans [13, 32]

$$e ::= \text{flip } \theta \mid x \mid x \leftarrow e; e \mid \text{if } e \text{ then } e \text{ else } e \mid \text{pure } e \mid \text{T} \mid \text{F}$$
$$\tau ::= \text{Dist Bool} \mid \text{Bool}$$

TinyPPL: Syntax

A monadic probabilistic programming language with
Booleans [13, 32]

$e ::= \text{flip } \theta \mid x \mid x \leftarrow e; e \mid \text{if } e \text{ then } e \text{ else } e \mid \text{pure } e \mid T \mid F$
 $\tau ::= \text{Dist Bool} \mid \text{Bool}$

$$\frac{}{\Gamma \vdash \text{flip } \theta : \text{Dist Bool}} \qquad \frac{\Gamma \vdash e : \text{Bool}}{\Gamma \vdash \text{pure } e : \text{Dist Bool}}$$
$$\frac{\Gamma \vdash e_1 : \text{Dist Bool} \quad \Gamma[x : \tau] \vdash e_2 : \tau_2}{\Gamma \vdash x \leftarrow e_1; e_2 : \tau_2}$$
$$\frac{\Gamma \vdash e_g : \text{Bool} \quad \Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash \text{if } e_g \text{ then } e_1 \text{ else } e_2 : \tau}$$

- Semantics of `flip` is a Bernoulli probability distribution:

- Semantics of `flip` is a Bernoulli probability distribution:

$$\llbracket \text{flip } \theta \rrbracket = v \mapsto \begin{cases} \theta & \text{if } v = \top \\ 1 - \theta & \text{if } v = \perp \end{cases}$$

- Semantics of `flip` is a Bernoulli probability distribution:

$$\llbracket \text{flip } \theta \rrbracket = v \mapsto \begin{cases} \theta & \text{if } v = \top \\ 1 - \theta & \text{if } v = \perp \end{cases}$$

- Semantics of `bind` sums over intermediate probability:

$$\llbracket x \leftarrow e_1; e_2 \rrbracket = v' \mapsto \sum_v \llbracket e_1 \rrbracket(v) \times \llbracket e_2[v/x] \rrbracket(v')$$

- Introduced by Kozen [23]

PPL Semantics More Broadly

- Introduced by Kozen [23]
- Adding continuous random variables requires measure theory [36, 8]
 - Conditioning becomes very challenging [34, 24]!

PPL Semantics More Broadly

- Introduced by Kozen [23]
- Adding continuous random variables requires measure theory [36, 8]
 - Conditioning becomes very challenging [34, 24]!
- Adding loops introduces interesting notions of non-termination:
 - Sure termination: terminates in finitely-many steps on all inputs
 - Almost-sure termination: a measure-zero path might exist, but all non-measure-zero paths terminate in finitely-many steps [21]

PPL Semantics More Broadly

- Introduced by Kozen [23]
- Adding continuous random variables requires measure theory [36, 8]
 - Conditioning becomes very challenging [34, 24]!
- Adding loops introduces interesting notions of non-termination:
 - Sure termination: terminates in finitely-many steps on all inputs
 - Almost-sure termination: a measure-zero path might exist, but all non-measure-zero paths terminate in finitely-many steps [21]
- Adding loops introduces interesting notions of non-termination:

PPL Semantics More Broadly

- Introduced by Kozen [23]
- Adding continuous random variables requires measure theory [36, 8]
 - Conditioning becomes very challenging [34, 24]!
- Adding loops introduces interesting notions of non-termination:
 - Sure termination: terminates in finitely-many steps on all inputs
 - Almost-sure termination: a measure-zero path might exist, but all non-measure-zero paths terminate in finitely-many steps [21]
- Adding loops introduces interesting notions of non-termination:
- Adding loops introduces interesting notions of non-termination:
- Adding continuous random variables and higher-order functions causes interesting challenges! See Heunen et al. [16]

A central challenge: scalability

A central challenge: scalability

- Given a Boolean formula φ , the #SAT problem asks: how many solutions to φ are there?

A central challenge: scalability

- Given a Boolean formula φ , the #SAT problem asks: how many solutions to φ are there?
- Ex: $\text{\#SAT}(A \vee B) = 3$

A central challenge: scalability

- Given a Boolean formula φ , the #SAT problem asks: how many solutions to φ are there?
- Ex: $\text{\#SAT}(A \vee B) = 3$
- Problems that are as hard as #SAT are called **#P-hard**; widely believed to be much harder than SAT [20]

A central challenge: scalability

- **Probabilistic inference** (evaluating semantics) can solve $\#SAT$, is therefore $\#P$ -hard.

A central challenge: scalability

- **Probabilistic inference** (evaluating semantics) can solve $\#SAT$, is therefore $\#P$ -hard.
- Simple reduction:

```
A  $\leftarrow$  flip 1/2;  
B  $\leftarrow$  flip 1/2;  
pure (A  $\vee$  B)
```

A central challenge: scalability

- **Probabilistic inference** (evaluating semantics) can solve $\#SAT$, is therefore $\#P$ -hard.
- Simple reduction:
$$\begin{array}{l} A \leftarrow \text{flip } 1/2; \\ B \leftarrow \text{flip } 1/2; \\ \text{pure } (A \vee B) \end{array}$$
- Multiple output probability by 4 to get the number of satisfying assignments to $A \vee B$

***Hope:** Inference is worst-case intractable, but it is not always intractable: real programs have structure.*

1. Identify classes of programs for which inference is tractable.
 - Type systems, static analysis, programmer discipline, etc.
2. Develop inference algorithms that exploit this tractability.
3. Develop systems that deploy the correct inference algorithm to problems with the right kinds of structure.

Scaling Inference:

Design and Implementation of Roulette

Inference landscape: Direct Sampling

Inference landscape: Direct Sampling

- Also simple idea: when you encounter a flip, sample a value from the Bernoulli probability distribution. Run program many times, take average

Inference landscape: Direct Sampling

- Also simple idea: when you encounter a flip, sample a value from the Bernoulli probability distribution. Run program many times, take average
- Answers are now *approximate*. Scales better (sometimes!), weaker guarantees

Inference landscape: Direct Sampling

- Also simple idea: when you encounter a flip, sample a value from the Bernoulli probability distribution. Run program many times, take average
- Answers are now *approximate*. Scales better (sometimes!), weaker guarantees
- There are many more sophisticated approaches to approximate inference that go beyond sampling to scale better:
 - Markov-Chain Monte Carlo is very popular [6, 33]

Tractability–Expressivity Tradeoff

Expressive Language $\longleftrightarrow$ Tractable Language

Tractability–Expressivity Tradeoff

Expressive Language $\longleftrightarrow$ Tractable Language

- Inference is *undecidable*
- Continuous distributions
- General recursion
- ... other features

Tractability–Expressivity Tradeoff

Expressive Language $\longleftrightarrow$ Tractable Language

- Inference is *undecidable*
- Continuous distributions
- General recursion
- ... other features
- Middle-ground: decidable inference, but high complexity
- Sure termination
- Typically discrete
- ... more?

Tractability–Expressivity Tradeoff

Expressive Language $\longleftrightarrow$ Tractable Language

- Inference is *undecidable*
- Continuous distributions
- General recursion
- ... other features

- Middle-ground: decidable inference, but high complexity
- Sure termination
- Typically discrete
- ... more?

- **Tractable probabilistic models** (TPMs): Inference is *polytime in size*.
- Highly restrictive: an *assembly language* (looks like ANF)
- Examples: binary decision diagrams (BDDs), generating functions, ...

Approach: Inference-via-Compilation



Approach: Inference-via-Compilation



- Separate inference implementation from language design

Approach: Inference-via-Compilation



- Separate inference implementation from language design
 - High-level language has ergonomic features, pleasant to program in

Approach: Inference-via-Compilation



- Separate inference implementation from language design
 - High-level language has ergonomic features, pleasant to program in
 - Target language captures *problem-specific structure*

Approach: Inference-via-Compilation



- Separate inference implementation from language design
 - High-level language has ergonomic features, pleasant to program in
 - Target language captures *problem-specific structure*
 - Compiler may be expensive, inference cost is amortized

Approach: Inference-via-Compilation



- Separate inference implementation from language design
 - High-level language has ergonomic features, pleasant to program in
 - Target language captures *problem-specific structure*
 - Compiler may be expensive, inference cost is amortized
- Develop new tractable target languages that enable new higher-level language features

Approach: Inference-via-Compilation



- Separate inference implementation from language design
 - High-level language has ergonomic features, pleasant to program in
 - Target language captures *problem-specific structure*
 - Compiler may be expensive, inference cost is amortized
- Develop new tractable target languages that enable new higher-level language features
- Increasing number of examples: Dice [18, 11], ProbLog [10], SPPL [10], generating functions [22], Pluck [5], and more [26, 15, 39]

Roulette: an expressive, exact, and efficient PPL

Probabilistic Subset of Racket

- Discrete random variables
- General recursion (sure termination)
- Macros
- Mutable state
- General datatypes (recursive, higher-order, ...)
- Interop with Racket

Tractable PPL

Knowledge compilation



Roulette: an expressive, exact, and efficient PPL

Probabilistic Subset of Racket

- Discrete random variables
- General recursion (sure termination)
- Macros
- Mutable state
- General datatypes (recursive, higher-order, ...)
- Interop with Racket

Tractable PPL

Knowledge compilation

- High-performance exact inference: essentially as fast as state-of-the-art exact inference for Bayesian networks

Roulette: an expressive, exact, and efficient PPL

Probabilistic Subset of Racket

- Discrete random variables
- General recursion (sure termination)
- Macros
- Mutable state
- General datatypes (recursive, higher-order, ...)
- Interop with Racket

Tractable PPL

Knowledge compilation

- High-performance exact inference: essentially as fast as state-of-the-art exact inference for Bayesian networks
- Based on Rosette [38], implemented as a `#lang`

Roulette by Example: Symbolic Unions

Roulette by Example: Symbolic Unions

Step 1: Run a Roulette program under abstract semantics ($\Downarrow$) to a symbolic union and weight map.

Roulette by Example: Symbolic Unions

Step 1: Run a Roulette program under abstract semantics ($\Downarrow$) to a symbolic union and weight map.

```
(define f1 (flip 0.4))  
(define f2 (flip 0.7))  
(or f1 f2)
```

Roulette by Example: Symbolic Unions

Step 1: Run a Roulette program under abstract semantics ($\Downarrow$) to a symbolic union and weight map.

```
(define f1 (flip 0.4))  
(define f2 (flip 0.7))  
(or f1 f2)
```

$\Downarrow$

Roulette by Example: Symbolic Unions

Step 1: Run a Roulette program under abstract semantics ($\Downarrow$) to a symbolic union and weight map.

```
(define f1 (flip 0.4))  
(define f2 (flip 0.7))  
(or f1 f2)
```

$\Downarrow$

$$\left[f_1 \vee f_2 : \#t, \neg(f_1 \vee f_2) : \#f \right]$$

Literal ℓ	$w(\ell)$
f_1	0.4
$\neg f_1$	0.6
f_2	0.7
$\neg f_2$	0.3

Roulette by Example: Symbolic Unions

Step 1: Run a Roulette program under abstract semantics ($\Downarrow$) to a symbolic union and weight map.

```
(define f1 (flip 0.4))  
(define f2 (flip 0.7))  
(or f1 f2)
```

$\Downarrow$

$$\left[f_1 \vee f_2 : \#t, \neg(f_1 \vee f_2) : \#f \right]$$

Literal ℓ	$w(\ell)$
f_1	0.4
$\neg f_1$	0.6
f_2	0.7
$\neg f_2$	0.3

- A *symbolic union* is a list of Racket values guarded by logical formulae [38].

Roulette by Example: Symbolic Unions

Step 1: Run a Roulette program under abstract semantics ($\Downarrow$) to a symbolic union and weight map.

```
(define f1 (flip 0.4))  
(define f2 (flip 0.7))  
(or f1 f2)
```

$\Downarrow$

$$\left[f_1 \vee f_2 : \#t, \neg(f_1 \vee f_2) : \#f \right]$$

Literal ℓ	$w(\ell)$
f_1	0.4
$\neg f_1$	0.6
f_2	0.7
$\neg f_2$	0.3

- A *symbolic union* is a list of Racket values guarded by logical formulae [38].
- Racket operations are *lifted* to operate on symbolic unions.

Another example:

```
(define x0 (flip 0.5))  
(define x1 (flip 0.5))  
(+ (if x0 2 3)  
   (if x1 3 4))
```



Roulette by Example: Symbolic Unions

Another example:

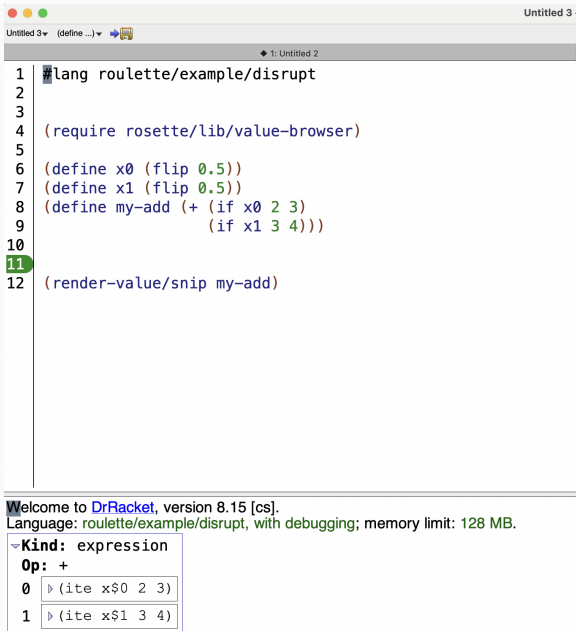
```
(define x0 (flip 0.5))  
(define x1 (flip 0.5))  
(+ (if x0 2 3)  
   (if x1 3 4))
```



$$\left[\begin{array}{l} x_0 \wedge x_1 : 5, \\ (\neg x_0 \wedge x_1) \vee (x_0 \wedge \neg x_1) : 6, \\ (\neg x_0 \wedge \neg x_1) : 7 \end{array} \right]$$

Literal ℓ	$w(\ell)$
x_1	0.5
$\neg x_1$	0.5
x_2	0.5
$\neg x_2$	0.5

Roulette by Example: Symbolic Unions



The screenshot shows the DrRacket IDE interface. The main editor window, titled 'Untitled 3', contains the following Racket code:

```
1 #lang roulette/example/disrupt
2
3
4 (require rosette/lib/value-browser)
5
6 (define x0 (flip 0.5))
7 (define x1 (flip 0.5))
8 (define my-add (+ (if x0 2 3)
9                  (if x1 3 4)))
10
11 (render-value/snip my-add)
12
```

Line 11 is highlighted with a green background. Below the editor, the REPL window shows the following output:

Welcome to [DrRacket](#), version 8.15 [cs].
Language: *roulette/example/disrupt*, with *debugging*; memory limit: 128 MB.

Kind: expression
Op: +
0 ▸ (ite x\$0 2 3)
1 ▸ (ite x\$1 3 4)

$$\hat{\rho} \vdash (e, \hat{\sigma}, w) \Downarrow (\hat{v}, \hat{\sigma}', w', \psi)$$

$$\hat{\rho} \vdash (e, \hat{\sigma}, w) \Downarrow (\hat{v}, \hat{\sigma}', w', \psi)$$

- In (abstract) environment $\hat{\rho}$

$$\hat{\rho} \vdash (e, \hat{\sigma}, w) \Downarrow (\hat{v}, \hat{\sigma}', w', \psi)$$

- In (abstract) environment $\hat{\rho}$
- The program e with store $\hat{\sigma}$ and weight map w

$$\hat{\rho} \vdash (e, \hat{\sigma}, w) \Downarrow (\hat{v}, \hat{\sigma}', w', \psi)$$

- In (abstract) environment $\hat{\rho}$
- The program e with store $\hat{\sigma}$ and weight map w
- Runs to value $\hat{v}$, new store $\hat{\sigma}'$, new weight map w' , and asserting formula ψ (for conditioning)

Computing probabilities

- The *weight of a model* $w(m)$ is the product of the weights of each literal in the model: $w(m) = \prod_{\ell \in m} w(\ell)$

Computing probabilities

- The *weight of a model* $w(m)$ is the product of the weights of each literal in the model: $w(m) = \prod_{\ell \in m} w(\ell)$
- The *weighted model count* $\text{WMC}(\varphi, w)$ is the weighted sum of the models of φ :

$$\text{WMC}(\varphi, w) = \sum_{m \models \varphi} w(m)$$

Computing probabilities

Theorem 5.1, very informal: weighted model counting of the symbolic union computes the correct probabilities.

Computing probabilities

Theorem 5.1, very informal: weighted model counting of the symbolic union computes the correct probabilities.

$$\left[f_1 \vee f_2 : \#t, \neg(f_1 \vee f_2) : \#f \right]$$

```
(define f1 (flip 0.4))  
(define f2 (flip 0.7))  
(or f1 f2)
```



Literal ℓ	$w(\ell)$
f_1	0.4
$\neg f_1$	0.6
f_2	0.7
$\neg f_2$	0.3

Computing probabilities

Theorem 5.1, very informal: weighted model counting of the symbolic union computes the correct probabilities.

$$\left[f_1 \vee f_2 : \#t, \neg(f_1 \vee f_2) : \#f \right]$$

```
(define f1 (flip 0.4))  
(define f2 (flip 0.7))  
(or f1 f2)
```



Literal ℓ	$w(\ell)$
f_1	0.4
$\neg f_1$	0.6
f_2	0.7
$\neg f_2$	0.3

To compute the probability that the program evaluates to $\#t$, compute the WMC of the guard for that value:

$$\text{WMC}(f_1 \vee f_2) = 0.4 \times 0.7 + 0.4 \times 0.3 + 0.6 \times 0.7 = 0.82$$

- We've reduced Roulette inference to performing WMC
- Now we can deploy specialized *weighted model counting tools*!
- Rich catalog of tools for this, similar to SAT!

Historical context: inference via weighted model counting is a state-of-the-art approach for exact inference in discrete Bayesian networks, goes back to Chavira and Darwiche [7]

Knowledge compilation

- **Step 2:** compile formulae to binary decision diagrams (BDDs), which support linear-time weighted model counting

Knowledge compilation

- **Step 2:** compile formulae to binary decision diagrams (BDDs), which support linear-time weighted model counting

$$\left[\begin{array}{l} x_0 \wedge x_1 : 5, \\ (\neg x_0 \wedge x_1) \vee (x_0 \wedge \neg x_1) : 6, \\ (\neg x_0 \wedge \neg x_1) : 7 \end{array} \right]$$

Literal ℓ	$w(\ell)$
x_1	0.5
$\neg x_1$	0.5
x_2	0.5
$\neg x_2$	0.5

Knowledge compilation

- **Step 2:** compile formulae to binary decision diagrams (BDDs), which support linear-time weighted model counting

$$\left[x_0 \wedge x_1 : 5, \right.$$

$$(\neg x_0 \wedge x_1) \vee (x_0 \wedge \neg x_1) : 6,$$

$$\left. (\neg x_0 \wedge \neg x_1) : 7 \right]$$

$\rightsquigarrow$

Compile
guard
for 6

Literal ℓ	$w(\ell)$
x_1	0.5
$\neg x_1$	0.5
x_2	0.5
$\neg x_2$	0.5

Knowledge compilation

- **Step 2:** compile formulae to binary decision diagrams (BDDs), which support linear-time weighted model counting

$$\left[x_0 \wedge x_1 : 5, \right.$$

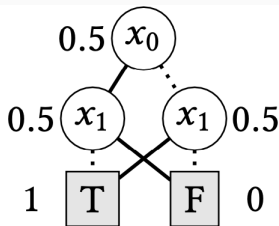
$$(\neg x_0 \wedge x_1) \vee (x_0 \wedge \neg x_1) : 6, \\$$

$$\left. (\neg x_0 \wedge \neg x_1) : 7 \right]$$

Literal ℓ	$w(\ell)$
x_1	0.5
$\neg x_1$	0.5
x_2	0.5
$\neg x_2$	0.5

$\rightsquigarrow$

Compile
guard
for 6



Knowledge compilation

- **Step 2:** compile formulae to binary decision diagrams (BDDs), which support linear-time weighted model counting

$$\left[x_0 \wedge x_1 : 5, \right.$$

$$(\neg x_0 \wedge x_1) \vee (x_0 \wedge \neg x_1) : 6, \\$$

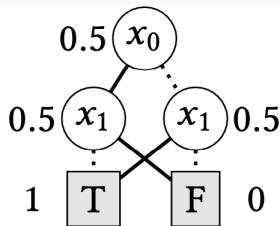
$$\left. (\neg x_0 \wedge \neg x_1) : 7 \right]$$

Literal ℓ	$w(\ell)$
x_1	0.5
$\neg x_1$	0.5

The RSDD knowledge compilation library: <https://github.com/neuppl/rsdd>

$\rightsquigarrow$

Compile
guard
for 6



Scaling PPLs Up: Deductive Reasoning

- Sometimes (often?) automated inference can't scale! What do we do then?

Scaling PPLs Up: Deductive Reasoning

- Sometimes (often?) automated inference can't scale! What do we do then?
- Need *proof rules* to reason about program's behavior deductively (manually).

Scaling PPLs Up: Deductive Reasoning

- Sometimes (often?) automated inference can't scale! What do we do then?
- Need *proof rules* to reason about program's behavior deductively (manually).
- Eventually, this can interact with automated inference

Probabilistic Separation Logics (a taste)

Hope: Probabilistic programs have modular structure for decomposing reasoning into sub-programs.

Probabilistic Separation Logics (a taste)

Hope: Probabilistic programs have modular structure for decomposing reasoning into sub-programs.

- *Idea: generalize separation logic to probabilities! [4]*

Probabilistic Separation Logics (a taste)

Hope: Probabilistic programs have modular structure for decomposing reasoning into sub-programs.

- *Idea: generalize separation logic to probabilities! [4]*
- *Our approach: Lilac, a measure-theoretic separation logic for probability [24]*

Probabilistic Separation Logics (a taste)

Hope: Probabilistic programs have modular structure for decomposing reasoning into sub-programs.

- *Idea: generalize separation logic to probabilities! [4]*
- Our approach: Lilac, a *measure-theoretic separation logic* for probability [24]

```
(define x (flip 0.5))  
  {x ~ bern 0.5}  
(define y (flip 0.5))  
  {x ~ bern 0.5 * y ~ bern 0.5}
```

Probabilistic Separation Logics (a taste)

Hope: Probabilistic programs have modular structure for decomposing reasoning into sub-programs.

- *Idea: generalize separation logic to probabilities! [4]*
- Our approach: Lilac, a *measure-theoretic separation logic* for probability [24]

```
(define x (flip 0.5))  
  {x ~ bern 0.5}  
(define y (flip 0.5))  
  {x ~ bern 0.5 * y ~ bern 0.5}
```

- Separating conjunction (*) denotes *separation of probability spaces*, implies probabilistic independence

An explosion of probabilistic separation logics

An explosion of probabilistic separation logics

- Probability as a resource: PSL [4], Lilac [24], Bluebell [2], BaSL [17]

An explosion of probabilistic separation logics

- Probability as a resource: PSL [4], Lilac [24], Bluebell [2], BaSL [17]
- Iris-family probabilistic separation logics: Eris [25, 1], Amaryllis [27], Polaris [37]

An explosion of probabilistic separation logics

- Probability as a resource: PSL [4], Lilac [24], Bluebell [2], BaSL [17]
- Iris-family probabilistic separation logics: Eris [25, 1], Amaryllis [27], Polaris [37]
- Outcome logics [41, 40]

Symbolic sets

Presenter: John Li

How I got into Roulette



Me in 2024

How I got into Roulette

$$\begin{aligned} \{\top\} \quad & x \leftarrow \text{flip } \frac{1}{2}; \quad y \leftarrow \text{flip } \frac{1}{2} \\ & \{x \sim \text{Ber } \frac{1}{2} * y \sim \text{Ber } \frac{1}{2}\} \end{aligned}$$

$\text{Sch} \simeq \text{Nom}$



Me in 2024

How I got into Roulette

```
(define my-list '(1 2 3))  
(define (add1-randomly l)  
  (map (lambda (x) (if (flip 0.5) x (+ x 1))) l))  
(define (sum-list l)  
  (foldr (lambda (x acc) (+ x acc)) 0 l))  
(equal? (modulo (sum-list (add1-randomly my-list)) 2)  
         0)
```



Me in 2024



How I got into Roulette

```
(define my-list '(1 2 3))  
(define (add1-randomly l)  
  (map (lambda (x) (if (flip 0.5) x (+ x 1))) l))  
(define (sum-list l)  
  (foldr (lambda (x acc) (+ x acc)) 0 l))  
(equal? (modulo (sum-list (add1-randomly my-list)) 2)  
  0)
```

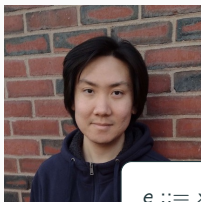


Me in 2024



How I got into Roulette

```
(define my-list '(1 2 3))  
(define (add1-randomly l)  
  (map (lambda (x) (if (flip 0.5) x (+ x 1))) l))  
(define (sum-list l)  
  (foldr (lambda (x acc) (+ x acc)) 0 l))  
(equal? (modulo (sum-list (add1-randomly my-list)) 2)  
  0)
```



Dice (2020)



Me in 2

$e ::= x \mid \text{true} \mid \text{false} \mid (e, e) \mid \text{fst } e \mid \text{snd } e \mid \neg e \mid e \wedge e$
 $\mid \text{flip } \theta \mid \text{observe } e \mid \text{let } x = e \text{ in } e \mid \text{if } e \text{ then } e \text{ else } e \mid f(e)$

Roadmap for this talk

Operational questions:

- How does Roulette reduce inference to WMC?
- How does Roulette keep this reduction efficient?

Denotational questions:

- What is the relationship between Roulette and traditional probability theory?
- What other Roulette-like inference strategies are possible?

- Probabilistic inference, like symbolic execution, simulates all possible paths through a program.
- State merging is essential for controlling the number of paths.
- Type-directed state merging, à la Rosette, helps keep symbolic values small.

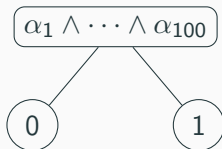
Exact inference as symbolic execution

```
(define x1 (flip 1/2))  
(define x2 (flip 1/2))  
...  
(define x100 (flip 1/2))  
(if (and x1 x2 ... x100) 0 1)
```

Reliability Analysis in Symbolic PathFinder (ICSE 2013)
Symbolic Execution for Randomized Programs (OOPSLA 2022)
Symbolic Semantics for Probabilistic Programs (QEST 2023)

Exact inference as symbolic execution

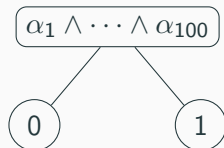
```
(define x1 (flip 1/2))  
(define x2 (flip 1/2))  
...  
(define x100 (flip 1/2))  
(if (and x1 x2 ... x100) 0 1)
```



Reliability Analysis in Symbolic PathFinder (ICSE 2013)
Symbolic Execution for Randomized Programs (OOPSLA 2022)
Symbolic Semantics for Probabilistic Programs (QEST 2023)

Exact inference as symbolic execution

```
(define x1 (flip 1/2))  
(define x2 (flip 1/2))  
...  
(define x100 (flip 1/2))  
(if (and x1 x2 ... x100) 0 1)
```

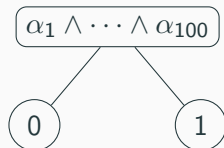


Result: $[\alpha_1 \wedge \dots \wedge \alpha_{100} : 0, \neg(\alpha_1 \wedge \dots \wedge \alpha_{100}) : 1]$

Reliability Analysis in Symbolic PathFinder (ICSE 2013)
Symbolic Execution for Randomized Programs (OOPSLA 2022)
Symbolic Semantics for Probabilistic Programs (QEST 2023)

Exact inference as symbolic execution

```
(define x1 (flip 1/2))  
(define x2 (flip 1/2))  
...  
(define x100 (flip 1/2))  
(if (and x1 x2 ... x100) 0 1)
```



Result: $[\alpha_1 \wedge \dots \wedge \alpha_{100} : 0, \neg(\alpha_1 \wedge \dots \wedge \alpha_{100}) : 1]$

$$\Pr[0] = \text{WMC}(\alpha_1 \wedge \dots \wedge \alpha_{100}) \quad \Pr[1] = \text{WMC}(\neg(\alpha_1 \wedge \dots \wedge \alpha_{100}))$$

Reliability Analysis in Symbolic PathFinder (ICSE 2013)

Symbolic Execution for Randomized Programs (OOPSLA 2022)

Symbolic Semantics for Probabilistic Programs (QEST 2023)

The perils of naive symbolic execution

```
(define (walk n p)
  (if (= n 0)
      p
      (walk (sub1 n) (+ p (if (flip 1/2) 1 -1)))))
```

The perils of naive symbolic execution

```
(define (walk n p)
  (if (= n 0)
      p
      (walk (sub1 n) (+ p (if (flip 1/2) 1 -1)))))
```

(walk 3 0)

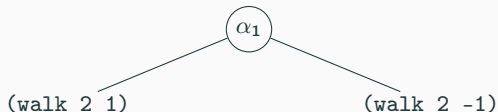
The perils of naive symbolic execution

```
(define (walk n p)
  (if (= n 0)
      p
      (walk (sub1 n) (+ p (if (flip 1/2) 1 -1)))))
```

```
(walk 2 (+ 0 (if (flip 1/2) 1 -1)))
```

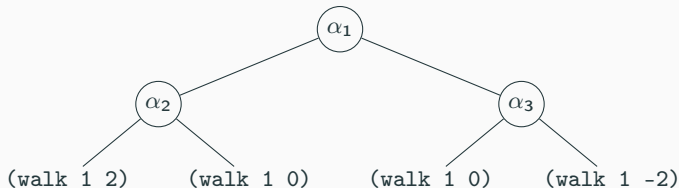
The perils of naive symbolic execution

```
(define (walk n p)
  (if (= n 0)
      p
      (walk (sub1 n) (+ p (if (flip 1/2) 1 -1)))))
```



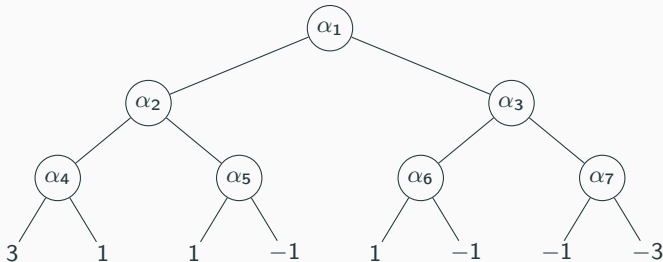
The perils of naive symbolic execution

```
(define (walk n p)
  (if (= n 0)
      p
      (walk (sub1 n) (+ p (if (flip 1/2) 1 -1)))))
```



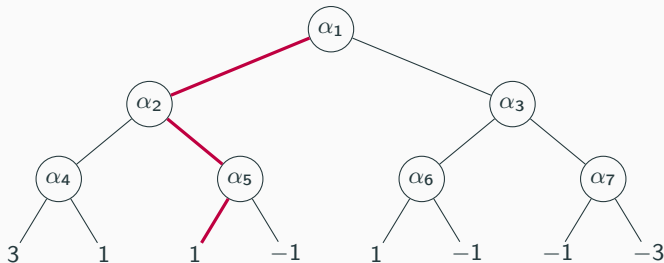
The perils of naive symbolic execution

```
(define (walk n p)
  (if (= n 0)
      p
      (walk (sub1 n) (+ p (if (flip 1/2) 1 -1)))))
```



The perils of naive symbolic execution

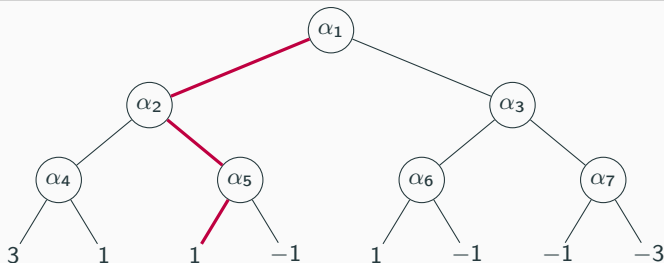
```
(define (walk n p)
  (if (= n 0)
      p
      (walk (sub1 n) (+ p (if (flip 1/2) 1 -1)))))
```



$[\dots, \alpha_1 \wedge \neg\alpha_2 \wedge \alpha_5 : 1, \dots]$

The perils of naive symbolic execution

```
(define (walk n p)
  (if (= n 0)
      p
      (walk (sub1 n) (+ p (if (flip 1/2) 1 -1)))))
```



$[\dots, \alpha_1 \wedge \neg \alpha_2 \wedge \alpha_5 : 1, \dots]$

Number of paths grows exponentially in n

State merging to the rescue

```
(define (walk n p)
  (if (= n 0)
      p
      (walk (sub1 n) (+ p (if (flip 1/2) 1 -1)))))
```

(walk 3 0)

State merging to the rescue

```
(define (walk n p)
  (if (= n 0)
      p
      (walk (sub1 n) (+ p (if (flip 1/2) 1 -1)))))
```

```
(walk 2 (+ 0 (if (flip 1/2) 1 -1)))
```

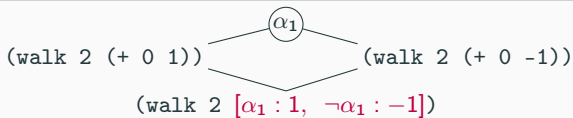
State merging to the rescue

```
(define (walk n p)
  (if (= n 0)
      p
      (walk (sub1 n) (+ p (if (flip 1/2) 1 -1)))))
```



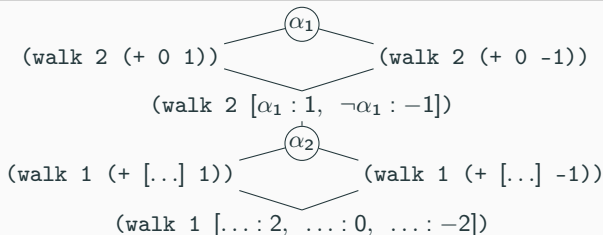
State merging to the rescue

```
(define (walk n p)
  (if (= n 0)
      p
      (walk (sub1 n) (+ p (if (flip 1/2) 1 -1)))))
```



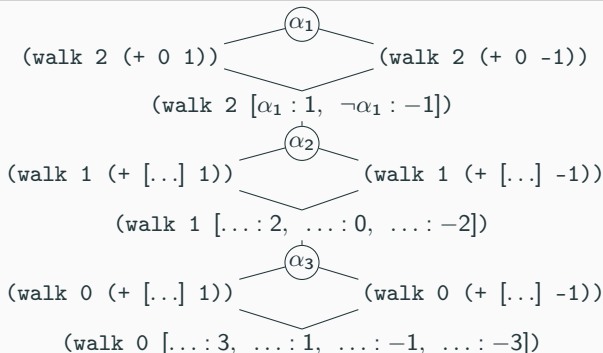
State merging to the rescue

```
(define (walk n p)
  (if (= n 0)
      p
      (walk (sub1 n) (+ p (if (flip 1/2) 1 -1)))))
```



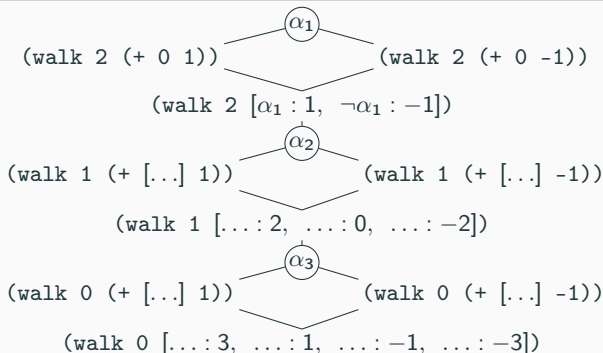
State merging to the rescue

```
(define (walk n p)
  (if (= n 0)
      p
      (walk (sub1 n) (+ p (if (flip 1/2) 1 -1)))))
```



State merging to the rescue

```
(define (walk n p)
  (if (= n 0)
      p
      (walk (sub1 n) (+ p (if (flip 1/2) 1 -1)))))
```



Run time is quadratic in n rather than exponential

Typed state merging keeps values small

Symbolic representation varies by type:

Typed state merging keeps values small

Symbolic representation varies by type:

```
> (define x (flip 1/2))  
> (define y (flip 1/2))  
> (if x y #t)  
(|| (! x$0) x$1)
```

Typed state merging keeps values small

Symbolic representation varies by type:

```
> (define x (flip 1/2))  
> (define y (flip 1/2))  
> (if x y #t)  
(|| (! x$0) x$1)  
  
> (if x (cons #t #f) (cons #t #t))  
(cons #t (! x$0))
```

Typed state merging keeps values small

Symbolic representation varies by type:

```
> (define x (flip 1/2))  
> (define y (flip 1/2))  
> (if x y #t)  
(|| (! x$0) x$1)  
  
> (if x (cons #t #f) (cons #t #t))  
(cons #t (! x$0))  
  
> (if x (lambda (b) b) (lambda (b) (not b)))  
#<procedure>
```

From operational to denotational semantics

Operational semantics is good at explaining how Roulette is implemented and what helps it scale. But it is less good at answering questions like:

- How do symbolic values represent probabilistic objects?
- What optimizations and heuristics from traditional symbolic execution remain sound in the probabilistic setting?

For these we turn to denotational semantics.

- Roulette implements a “symbolic probability theory” that lifts ordinary programs to operate on random variables.
- Making this precise provides a framework for building, verifying, and optimizing Roulette-like inference engines.

Roulette as a symbolic probability theory

Roulette	Probability theory
<code>(define x (flip 1/2))</code>	$X \sim \text{Ber } 1/2$
<code>(define y (flip 1/2))</code>	$Y \sim \text{Ber } 1/2$
<code>(and x y)</code>	$X \wedge Y$
<code>(query (and x y)) = 1/4</code>	$\Pr[X \wedge Y] = 1/4$

Roulette as a symbolic probability theory

Roulette	Probability theory
<code>(define x (flip 1/2))</code>	$X \sim \text{Ber } 1/2$
<code>(define y (flip 1/2))</code>	$Y \sim \text{Ber } 1/2$
<code>(and x y)</code>	$X \wedge Y$
<code>(query (and x y)) = 1/4</code>	$\Pr[X \wedge Y] = 1/4$

Roulette values represent random variables

Roulette as a symbolic probability theory

Roulette	Probability theory
<code>(define x (flip 1/2))</code>	$X \sim \text{Ber } 1/2$
<code>(define y (flip 1/2))</code>	$Y \sim \text{Ber } 1/2$
<code>(and x y)</code>	$X \wedge Y$
<code>(query (and x y)) = 1/4</code>	$\Pr[X \wedge Y] = 1/4$

Roulette programs transform random variables

Roulette as a symbolic probability theory

The probabilistic semantics $\mathcal{R}[\![\!-\!]\!]$ *lifts* the ordinary semantics $\llbracket\!-\!\rrbracket$ to operate on random variables:

Roulette as a symbolic probability theory

The probabilistic semantics $\mathcal{R}[\![\!-\!]\!]$ *lifts* the ordinary semantics $\llbracket\!-\!\rrbracket$ to operate on random variables:

- For types, want $\mathcal{R}[\![A]\!] \cong (\Omega \rightarrow \llbracket A \rrbracket)$, where $\Omega =$ the set of weighted models of Boolean formulae

Contextual Equivalence for Probabilistic Programs with Continuous Random Variables and Scoring (ESOP 2017)

Roulette as a symbolic probability theory

The probabilistic semantics $\mathcal{R}[\![-]\!]$ *lifts* the ordinary semantics $\llbracket - \rrbracket$ to operate on random variables:

- For types, want $\mathcal{R}[\![A]\!] \cong (\Omega \rightarrow \llbracket A \rrbracket)$, where $\Omega =$ the set of weighted models of Boolean formulae

$$\text{Recall: } \text{WMC}(\varphi, w) = \sum_{m \models \varphi} w(m)$$

Contextual Equivalence for Probabilistic Programs with Continuous Random Variables and Scoring (ESOP 2017)

Roulette as a symbolic probability theory

The probabilistic semantics $\mathcal{R}[\![-]\!]$ *lifts* the ordinary semantics $\llbracket - \rrbracket$ to operate on random variables:

- For types, want $\mathcal{R}[\![A]\!] \cong (\Omega \rightarrow \llbracket A \rrbracket)$, where $\Omega =$ the set of weighted models of Boolean formulae
- For terms, want $\mathcal{R}[\![M]\!] : \mathcal{R}[\![\Gamma]\!] \rightarrow \mathcal{R}[\![A]\!]$ to simulate $\llbracket M \rrbracket : \llbracket \Gamma \rrbracket \rightarrow \llbracket A \rrbracket$:

Contextual Equivalence for Probabilistic Programs with Continuous Random Variables and Scoring (ESOP 2017)

Roulette as a symbolic probability theory

The probabilistic semantics $\mathcal{R}[\![-]\!]$ *lifts* the ordinary semantics $\llbracket - \rrbracket$ to operate on random variables:

- For types, want $\mathcal{R}[\![A]\!] \cong (\Omega \rightarrow \llbracket A \rrbracket)$, where Ω = the set of weighted models of Boolean formulae
- For terms, want $\mathcal{R}[\![M]\!] : \mathcal{R}[\![\Gamma]\!] \rightarrow \mathcal{R}[\![A]\!]$ to simulate $\llbracket M \rrbracket : \llbracket \Gamma \rrbracket \rightarrow \llbracket A \rrbracket$:

$$\begin{array}{ccc} \mathcal{R}[\![\Gamma]\!] & \xrightarrow{\mathcal{R}[\![M]\!]} & \mathcal{R}[\![A]\!] \\ \cong \downarrow & & \downarrow \cong \\ (\Omega \rightarrow \llbracket \Gamma \rrbracket) & \xrightarrow{\text{lift } \llbracket M \rrbracket} & (\Omega \rightarrow \llbracket A \rrbracket) \end{array}$$

Contextual Equivalence for Probabilistic Programs with Continuous Random Variables and Scoring (ESOP 2017)

Getting formal: a tiny PPL

$A, B ::= \text{Bool} \mid A \times B \mid A + B$

$M, N ::= x \mid \text{let } x = M \text{ in } N$

$\mid \text{true} \mid \text{false} \mid M \ \&\& \ N \mid ! \ M$

$\mid (M, N) \mid \text{fst } M \mid \text{snd } M$

$\mid \text{inl } M \mid \text{inr } M \mid \text{case}(M, x.N, y.O)$

Getting formal: a tiny PPL

$A, B ::= \text{Bool} \mid A \times B \mid A + B$

$M, N ::= x \mid \text{let } x = M \text{ in } N$

$\mid \text{true} \mid \text{false} \mid M \ \&\& \ N \mid ! \ M$

$\mid (M, N) \mid \text{fst } M \mid \text{snd } M$

$\mid \text{inl } M \mid \text{inr } M \mid \text{case}(M, x.N, y.O)$

Case analysis requires typed state merging

Getting formal: a tiny PPL

$$A, B ::= \text{Bool} \mid A \times B$$
$$M, N ::= x \mid \text{let } x = M \text{ in } N$$
$$\mid \text{true} \mid \text{false} \mid M \ \&\& \ N \mid ! \ M$$
$$\mid (M, N) \mid \text{fst } M \mid \text{snd } M$$

To start, let's just look at pure straight-line programs.

Ordinary semantics of TinyPPL

TinyPPL can be given a standard set-theoretic semantics:

Ordinary semantics of TinyPPL

TinyPPL can be given a standard set-theoretic semantics:

$$\llbracket \text{Bool} \rrbracket = \{0, 1\}$$

$$\llbracket A \times B \rrbracket = \llbracket A \rrbracket \times \llbracket B \rrbracket$$

Types denote sets of values

Ordinary semantics of TinyPPL

TinyPPL can be given a standard set-theoretic semantics:

$$\llbracket \text{Bool} \rrbracket = \{0, 1\}$$

$$\llbracket A \times B \rrbracket = \llbracket A \rrbracket \times \llbracket B \rrbracket$$

$$\llbracket \Gamma \rrbracket = ((x : A) \in \Gamma) \rightarrow \llbracket A \rrbracket$$

Contexts denote sets of
environments

Ordinary semantics of TinyPPL

TinyPPL can be given a standard set-theoretic semantics:

$$\llbracket \text{Bool} \rrbracket = \{0, 1\}$$

$$\llbracket A \times B \rrbracket = \llbracket A \rrbracket \times \llbracket B \rrbracket$$

$$\llbracket \Gamma \rrbracket = ((x : A) \in \Gamma) \rightarrow \llbracket A \rrbracket$$

$$\llbracket \Gamma \vdash M : A \rrbracket : \llbracket \Gamma \rrbracket \rightarrow \llbracket A \rrbracket$$

$$\llbracket x \rrbracket(\gamma) = \gamma(x)$$

$$\llbracket \text{let } x = M \text{ in } N \rrbracket(\gamma) = \llbracket N \rrbracket(\gamma[x \mapsto \llbracket M \rrbracket(\gamma)])$$

$$\llbracket \text{true} \rrbracket(\gamma) = 1$$

$$\llbracket \text{false} \rrbracket(\gamma) = 0$$

$$\llbracket M \ \&\& \ N \rrbracket(\gamma) = \llbracket M \rrbracket(\gamma) \wedge \llbracket N \rrbracket(\gamma)$$

$$\llbracket ! M \rrbracket(\gamma) = 1 - \llbracket M \rrbracket(\gamma)$$

$$\llbracket (M, N) \rrbracket(\gamma) = (\llbracket M \rrbracket(\gamma), \llbracket N \rrbracket(\gamma))$$

$$\llbracket \text{fst } M \rrbracket(\gamma) = \pi_1(\llbracket M \rrbracket(\gamma))$$

$$\llbracket \text{snd } M \rrbracket(\gamma) = \pi_2(\llbracket M \rrbracket(\gamma))$$

$$\mathcal{R}[[A]] = \Omega \rightarrow [[A]]$$

Probabilistic semantics of TinyPPL

$$\mathcal{R}[\llbracket \text{Bool} \rrbracket] = \Omega \rightarrow \llbracket \text{Bool} \rrbracket$$

$$\mathcal{R}[\llbracket A \times B \rrbracket] = \Omega \rightarrow \llbracket A \times B \rrbracket$$

Probabilistic semantics of TinyPPL

$$\mathcal{R}[\![\text{Bool}]\!] = \Omega \rightarrow \{0, 1\}$$

$$\mathcal{R}[A \times B] = \Omega \rightarrow [A \times B]$$

Probabilistic semantics of TinyPPL

$$\mathcal{R}[\text{Bool}] = \{E \mid E \subseteq \Omega\}$$

$$\mathcal{R}[A \times B] = \Omega \rightarrow [A \times B]$$

Probabilistic semantics of TinyPPL

$$\mathcal{R}[\text{Bool}] = \{E \mid E \subseteq \Omega\}$$

$$\mathcal{R}[A \times B] = \Omega \rightarrow [A \times B]$$

Recall:

```
> (if x y #t)
```

```
(|| (! x$0) x$1)
```

Probabilistic semantics of TinyPPL

$$\mathcal{R}[\text{Bool}] = \{E \mid E \subseteq \Omega\}$$

$$\mathcal{R}[A \times B] = \Omega \rightarrow [A] \times [B]$$

Probabilistic semantics of TinyPPL

$$\mathcal{R}[\llbracket \text{Bool} \rrbracket] = \{E \mid E \subseteq \Omega\}$$

$$\mathcal{R}[A \times B] = (\Omega \rightarrow \llbracket A \rrbracket) \times (\Omega \rightarrow \llbracket B \rrbracket)$$

Probabilistic semantics of TinyPPL

$$\mathcal{R}[\text{Bool}] = \{E \mid E \subseteq \Omega\}$$

$$\mathcal{R}[A \times B] = \mathcal{R}[A] \times \mathcal{R}[B]$$

Probabilistic semantics of TinyPPL

$$\mathcal{R}[\text{Bool}] = \{E \mid E \subseteq \Omega\}$$

$$\mathcal{R}[A \times B] = \mathcal{R}[A] \times \mathcal{R}[B]$$

Recall:

```
> (if x (cons #t #f) (cons #t #t))  
(cons #t (! x$0))
```

Probabilistic semantics of TinyPPL

$$\mathcal{R}[\text{Bool}] = \{E \mid E \subseteq \Omega\}$$

$$\mathcal{R}[A \times B] = \mathcal{R}[A] \times \mathcal{R}[B]$$

The interpretation of terms can be calculated in the same way:

$$\mathcal{R}[x](\gamma) = \gamma(x)$$

$$\mathcal{R}[\text{let } x = M \text{ in } N](\gamma) = \mathcal{R}[N](\gamma[x \mapsto \mathcal{R}[M](\gamma)])$$

$$\mathcal{R}[\text{true}](\gamma) = \Omega$$

$$\mathcal{R}[\text{false}](\gamma) = \emptyset$$

$$\mathcal{R}[M \ \&\& \ N](\gamma) = \mathcal{R}[M](\gamma) \cap \mathcal{R}[N](\gamma)$$

$$\mathcal{R}[\text{! } M](\gamma) = \Omega \setminus \mathcal{R}[M](\gamma)$$

$$\mathcal{R}[(M, N)](\gamma) = (\mathcal{R}[M](\gamma), \mathcal{R}[N](\gamma))$$

$$\mathcal{R}[\text{fst } M](\gamma) = \pi_1(\mathcal{R}[M](\gamma))$$

$$\mathcal{R}[\text{snd } M](\gamma) = \pi_2(\mathcal{R}[M](\gamma))$$

Probabilistic semantics of TinyPPL

$$\mathcal{R}[\text{Bool}] = \{E \mid E \subseteq \Omega\}$$

$$\mathcal{R}[A \times B] = \mathcal{R}[A] \times \mathcal{R}[B]$$

The interpretation of terms can be calculated in the same way:

$$\mathcal{R}[x](\gamma) = \gamma(x)$$

$$\mathcal{R}[\text{let } x = M \text{ in } N](\gamma) = \mathcal{R}[N](\gamma[x \mapsto \mathcal{R}[M](\gamma)])$$

$$\mathcal{R}[\text{true}](\gamma) = \Omega$$

$$\mathcal{R}[\text{false}](\gamma) = \emptyset$$

Boolean operations lift to
logical connectives

$$\mathcal{R}[M \ \&\& \ N](\gamma) = \mathcal{R}[M](\gamma) \cap \mathcal{R}[N](\gamma)$$

$$\mathcal{R}[\text{! } M](\gamma) = \Omega \setminus \mathcal{R}[M](\gamma)$$

$$\mathcal{R}[(M, N)](\gamma) = (\mathcal{R}[M](\gamma), \mathcal{R}[N](\gamma))$$

$$\mathcal{R}[\text{fst } M](\gamma) = \pi_1(\mathcal{R}[M](\gamma))$$

$$\mathcal{R}[\text{snd } M](\gamma) = \pi_2(\mathcal{R}[M](\gamma))$$

Probabilistic semantics of TinyPPL

$$\mathcal{R}[\text{Bool}] = \{E \mid E \subseteq \Omega\}$$

$$\mathcal{R}[A \times B] = \mathcal{R}[A] \times \mathcal{R}[B]$$

The interpretation of terms can be calculated in the same way:

$$\mathcal{R}[x](\gamma) = \gamma(x)$$

$$\mathcal{R}[\text{let } x = M \text{ in } N](\gamma) = \mathcal{R}[N](\gamma[x \mapsto \mathcal{R}[M](\gamma)])$$

$$\mathcal{R}[\text{true}](\gamma) = \Omega$$

$$\mathcal{R}[\text{false}](\gamma) = \emptyset$$

$$\mathcal{R}[M \ \&\& \ N](\gamma) = \mathcal{R}[M](\gamma) \cap \mathcal{R}[N](\gamma)$$

$$\mathcal{R}[\text{! } M](\gamma) = \Omega \setminus \mathcal{R}[M](\gamma)$$

$$\mathcal{R}[(M, N)](\gamma) = (\mathcal{R}[M](\gamma), \mathcal{R}[N](\gamma))$$

$$\mathcal{R}[\text{fst } M](\gamma) = \pi_1(\mathcal{R}[M](\gamma))$$

$$\mathcal{R}[\text{snd } M](\gamma) = \pi_2(\mathcal{R}[M](\gamma))$$

Symbolic pairs are
pairs of symbolic values

The following hold by construction:

The following hold by construction:

1. For all types A , have $\mathcal{R}[[A]] \cong (\Omega \rightarrow [[A]])$

Probabilistic semantics of TinyPPL

The following hold by construction:

1. For all types A , have $\mathcal{R}[[A]] \cong (\Omega \rightarrow [[A]])$
2. For all programs M , have

$$\begin{array}{ccc} \mathcal{R}[[\Gamma]] & \xrightarrow{\mathcal{R}[[M]]} & \mathcal{R}[[A]] \\ \cong \downarrow & & \downarrow \cong \\ (\Omega \rightarrow [[\Gamma]]) & \xrightarrow{\text{lift } [[M]]} & (\Omega \rightarrow [[A]]) \end{array}$$

Extending TinyPPL with branching

$A, B ::= \text{Bool} \mid A \times B \mid A + B$

$M, N ::= x \mid \text{let } x = M \text{ in } N$

$\mid \text{true} \mid \text{false} \mid M \ \&\& \ N \mid ! \ M$

$\mid (M, N) \mid \text{fst } M \mid \text{snd } M$

$\mid \text{inl } M \mid \text{inr } M \mid \text{case}(M, x.N, y.O)$

Extending TinyPPL with branching

Let's try to calculate $\mathcal{R}[[A + B]]$ as before:

$$\mathcal{R}[\text{Bool}] = \{E \mid E \subseteq \Omega\}$$

$$\mathcal{R}[A \times B] = \mathcal{R}[A] \times \mathcal{R}[B]$$

$$\mathcal{R}[A + B] = \Omega \rightarrow [A + B]$$

Extending TinyPPL with branching

Let's try to calculate $\mathcal{R}[[A + B]]$ as before:

$$\mathcal{R}[\text{Bool}] = \{E \mid E \subseteq \Omega\}$$

$$\mathcal{R}[A \times B] = \mathcal{R}[A] \times \mathcal{R}[B]$$

$$\mathcal{R}[A + B] = \Omega \rightarrow [[A]] + [[B]]$$

Extending TinyPPL with branching

Let's try to calculate $\mathcal{R}[[A + B]]$ as before:

$$\mathcal{R}[[\text{Bool}]] = \{E \mid E \subseteq \Omega\}$$

$$\mathcal{R}[A \times B] = \mathcal{R}[A] \times \mathcal{R}[B]$$

$$\mathcal{R}[A + B] = \Omega \rightarrow [[A]] + [[B]]$$

$$\neq (\Omega \rightarrow [[A]]) + (\Omega \rightarrow [[B]])$$

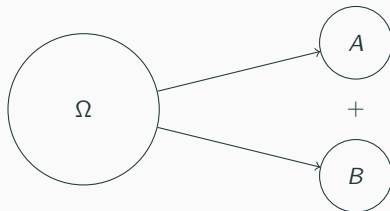
Extending TinyPPL with branching

Let's try to calculate $\mathcal{R}[[A + B]]$ as before:

$$\mathcal{R}[\text{Bool}] = \{E \mid E \subseteq \Omega\}$$

$$\mathcal{R}[A \times B] = \mathcal{R}[A] \times \mathcal{R}[B]$$

$$\begin{aligned}\mathcal{R}[A + B] &= \Omega \rightarrow [[A]] + [[B]] \\ &\neq (\Omega \rightarrow [[A]]) + (\Omega \rightarrow [[B]])\end{aligned}$$



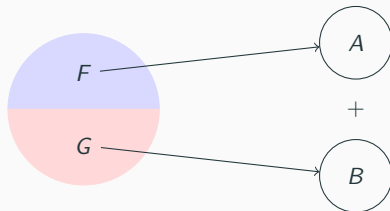
Extending TinyPPL with branching

Let's try to calculate $\mathcal{R}[[A + B]]$ as before:

$$\mathcal{R}[\text{Bool}] = \{E \mid E \subseteq \Omega\}$$

$$\mathcal{R}[A \times B] = \mathcal{R}[A] \times \mathcal{R}[B]$$

$$\begin{aligned}\mathcal{R}[A + B] &= \Omega \rightarrow [[A]] + [[B]] \\ &\neq (\Omega \rightarrow [[A]]) + (\Omega \rightarrow [[B]])\end{aligned}$$



Extending TinyPPL with branching

Let's try to calculate $\mathcal{R}[[A + B]]$ as before:

$$\mathcal{R}[[\text{Bool}]] = \{E \mid E \subseteq \Omega\}$$

$$\mathcal{R}[A \times B] = \mathcal{R}[A] \times \mathcal{R}[B]$$

$$\mathcal{R}[A + B] = \sum_{\Omega=F+G} (F \rightarrow [[A]]) \times (G \rightarrow [[B]])$$

Extending TinyPPL with branching

Let's try to calculate $\mathcal{R}[[A + B]]$ as before:

$$\mathcal{R}[[\text{Bool}]] = \{E \mid E \subseteq \Omega\}$$

$$\mathcal{R}[[A \times B]] = \mathcal{R}[[A]] \times \mathcal{R}[[B]]$$

$$\mathcal{R}[[A + B]] = \sum_{\Omega=F+G} (F \rightarrow [[A]]) \times (G \rightarrow [[B]])$$

Elements of this set are 4-tuples (F, G, X, Y) . Think of each tuple as a symbolic union $[F : \text{inl } X, G : \text{inr } Y]$.

Extending TinyPPL with branching

Let's try to calculate $\mathcal{R}[[A + B]]$ as before:

$$\mathcal{R}[[\text{Bool}]]_{\textcolor{red}{E}} = \{E' \mid E' \subseteq \textcolor{red}{E}\}$$

$$\mathcal{R}[[A \times B]]_{\textcolor{red}{E}} = \mathcal{R}[[A]]_{\textcolor{red}{E}} \times \mathcal{R}[[B]]_{\textcolor{red}{E}}$$

$$\mathcal{R}[[A + B]]_{\textcolor{red}{E}} = \sum_{\textcolor{red}{E} = F + G} (F \rightarrow [[A]]) \times (G \rightarrow [[B]])$$

Extending TinyPPL with branching

Let's try to calculate $\mathcal{R}[A + B]$ as before:

$$\mathcal{R}[\text{Bool}]_E = \{E' \mid E' \subseteq E\}$$

$$\mathcal{R}[A \times B]_E = \mathcal{R}[A]_E \times \mathcal{R}[B]_E$$

$$\mathcal{R}[A + B]_E = \sum_{E=F+G} \mathcal{R}[A]_F \times \mathcal{R}[B]_G$$

Extending TinyPPL with branching

Let's try to calculate $\mathcal{R}[[A + B]]$ as before:

$$\mathcal{R}[[\text{Bool}]]_E = \{E' \mid E' \subseteq E\}$$

$$\mathcal{R}[[A \times B]]_E = \mathcal{R}[[A]]_E \times \mathcal{R}[[B]]_E$$

$$\mathcal{R}[[A + B]]_E = \sum_{E=F+G} \mathcal{R}[[A]]_F \times \mathcal{R}[[B]]_G$$

By construction, have $\mathcal{R}[[A]]_E \cong (E \rightarrow [[A]])$ for all types A and events $E \subseteq \Omega$.

Extending TinyPPL with branching

The interpretation of terms becomes similarly indexed:

$$\mathcal{R}[\![\Gamma \vdash M : A]\!]_E : \mathcal{R}[\![\Gamma]\!]_E \rightarrow \mathcal{R}[\![A]\!]_E$$

Extending TinyPPL with branching

The interpretation of terms becomes similarly indexed:

$$\mathcal{R}[\![\Gamma \vdash M : A]\!]_E : \mathcal{R}[\![\Gamma]\!]_E \rightarrow \mathcal{R}[\![A]\!]_E$$

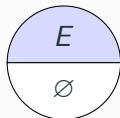
$$\mathcal{R}[\![\text{inl } M : A + B]\!]_E(\gamma) = (E, \emptyset, \mathcal{R}[\![M]\!]_E, \emptyset_B)$$

Extending TinyPPL with branching

The interpretation of terms becomes similarly indexed:

$$\mathcal{R}[\Gamma \vdash M : A]_E : \mathcal{R}[\Gamma]_E \rightarrow \mathcal{R}[A]_E$$

$$\mathcal{R}[\text{inl } M : A + B]_E(\gamma) = (\textcolor{red}{E}, \emptyset, \mathcal{R}[M]_E, \emptyset_B)$$

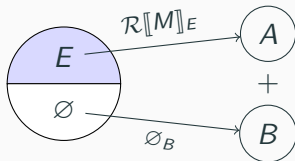


Extending TinyPPL with branching

The interpretation of terms becomes similarly indexed:

$$\mathcal{R}[\Gamma \vdash M : A]_E : \mathcal{R}[\Gamma]_E \rightarrow \mathcal{R}[A]_E$$

$$\mathcal{R}[\text{inl } M : A + B]_E(\gamma) = (E, \emptyset, \mathcal{R}[M]_E, \emptyset_B)$$

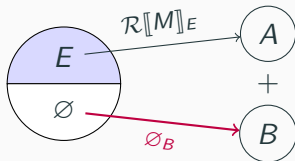


Extending TinyPPL with branching

The interpretation of terms becomes similarly indexed:

$$\mathcal{R}[\Gamma \vdash M : A]_E : \mathcal{R}[\Gamma]_E \rightarrow \mathcal{R}[A]_E$$

$$\mathcal{R}[\text{inl } M : A + B]_E(\gamma) = (E, \emptyset, \mathcal{R}[M]_E, \emptyset_B)$$



Extending TinyPPL with branching

The interpretation of terms becomes similarly indexed:

$$\mathcal{R}[\Gamma \vdash M : A]_E : \mathcal{R}[\Gamma]_E \rightarrow \mathcal{R}[A]_E$$

$$\mathcal{R}[\text{inl } M : A + B]_E(\gamma) = (E, \emptyset, \mathcal{R}[M]_E, \emptyset_B)$$

Every type comes with an empty element:

$$\emptyset_A : \mathcal{R}[A]_{\emptyset}$$

$$\emptyset_{\text{Bool}} = \emptyset$$

$$\emptyset_{A \times B} = (\emptyset_A, \emptyset_B)$$

$$\emptyset_{A+B} = (\emptyset, \emptyset, \emptyset_A, \emptyset_B)$$

Extending TinyPPL with branching

The interpretation of terms becomes similarly indexed:

$$\mathcal{R}[\Gamma \vdash M : A]_E : \mathcal{R}[\Gamma]_E \rightarrow \mathcal{R}[A]_E$$

$$\mathcal{R}[\text{inl } M : A + B]_E(\gamma) = (E, \emptyset, \mathcal{R}[M]_E, \emptyset_B)$$

$$\mathcal{R}[\text{inr } M : A + B]_E(\gamma) = (\emptyset, E, \emptyset_A, \mathcal{R}[M]_E)$$

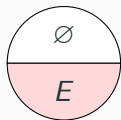
Extending TinyPPL with branching

The interpretation of terms becomes similarly indexed:

$$\mathcal{R}[\Gamma \vdash M : A]_E : \mathcal{R}[\Gamma]_E \rightarrow \mathcal{R}[A]_E$$

$$\mathcal{R}[\text{inl } M : A + B]_E(\gamma) = (E, \emptyset, \mathcal{R}[M]_E, \emptyset_B)$$

$$\mathcal{R}[\text{inr } M : A + B]_E(\gamma) = (\emptyset, \textcolor{red}{E}, \emptyset_A, \mathcal{R}[M]_E)$$



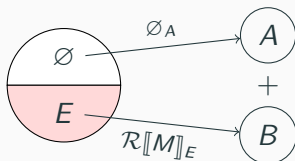
Extending TinyPPL with branching

The interpretation of terms becomes similarly indexed:

$$\mathcal{R}[\Gamma \vdash M : A]_E : \mathcal{R}[\Gamma]_E \rightarrow \mathcal{R}[A]_E$$

$$\mathcal{R}[\text{inl } M : A + B]_E(\gamma) = (E, \emptyset, \mathcal{R}[M]_E, \emptyset_B)$$

$$\mathcal{R}[\text{inr } M : A + B]_E(\gamma) = (\emptyset, E, \emptyset_A, \mathcal{R}[M]_E)$$



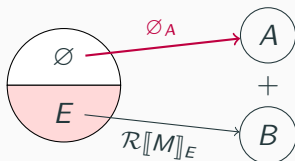
Extending TinyPPL with branching

The interpretation of terms becomes similarly indexed:

$$\mathcal{R}[\Gamma \vdash M : A]_E : \mathcal{R}[\Gamma]_E \rightarrow \mathcal{R}[A]_E$$

$$\mathcal{R}[\text{inl } M : A + B]_E(\gamma) = (E, \emptyset, \mathcal{R}[M]_E, \emptyset_B)$$

$$\mathcal{R}[\text{inr } M : A + B]_E(\gamma) = (\emptyset, E, \emptyset_A, \mathcal{R}[M]_E)$$



Extending TinyPPL with branching

The interpretation of terms becomes similarly indexed:

$$\mathcal{R}[\Gamma \vdash M : A]_E : \mathcal{R}[\Gamma]_E \rightarrow \mathcal{R}[A]_E$$

$$\mathcal{R}[\text{inl } M : A + B]_E(\gamma) = (E, \emptyset, \mathcal{R}[M]_E, \emptyset_B)$$

$$\mathcal{R}[\text{inr } M : A + B]_E(\gamma) = (\emptyset, E, \emptyset_A, \mathcal{R}[M]_E)$$

$$\mathcal{R}[\text{case}(M, x.N, y.O) : C]_E(\gamma) =$$

$$\text{let } (F, G, X, Y) = \mathcal{R}[M]_E(\gamma) \text{ in}$$

$$\mathcal{R}[N]_F(\gamma|_F, X) \sqcup_C \mathcal{R}[O]_G(\gamma|_G, Y)$$

Extending TinyPPL with branching

The interpretation of terms becomes similarly indexed:

$$\mathcal{R}[\Gamma \vdash M : A]_E : \mathcal{R}[\Gamma]_E \rightarrow \mathcal{R}[A]_E$$

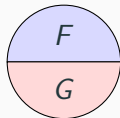
$$\mathcal{R}[\text{inl } M : A + B]_E(\gamma) = (E, \emptyset, \mathcal{R}[M]_E, \emptyset_B)$$

$$\mathcal{R}[\text{inr } M : A + B]_E(\gamma) = (\emptyset, E, \emptyset_A, \mathcal{R}[M]_E)$$

$$\mathcal{R}[\text{case}(M, x.N, y.O) : C]_E(\gamma) =$$

$$\text{let } (F, G, X, Y) = \mathcal{R}[M]_E(\gamma) \text{ in}$$

$$\mathcal{R}[N]_F(\gamma|_F, X) \sqcup_C \mathcal{R}[O]_G(\gamma|_G, Y)$$



Extending TinyPPL with branching

The interpretation of terms becomes similarly indexed:

$$\mathcal{R}[\Gamma \vdash M : A]_E : \mathcal{R}[\Gamma]_E \rightarrow \mathcal{R}[A]_E$$

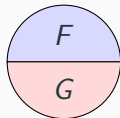
$$\mathcal{R}[\text{inl } M : A + B]_E(\gamma) = (E, \emptyset, \mathcal{R}[M]_E, \emptyset_B)$$

$$\mathcal{R}[\text{inr } M : A + B]_E(\gamma) = (\emptyset, E, \emptyset_A, \mathcal{R}[M]_E)$$

$$\mathcal{R}[\text{case}(M, x.N, y.O) : C]_E(\gamma) =$$

$$\text{let } (F, G, X, Y) = \mathcal{R}[M]_E(\gamma) \text{ in}$$

$$\mathcal{R}[N]_F(\gamma|_F, X) \sqcup_C \mathcal{R}[O]_G(\gamma|_G, Y)$$



Extending TinyPPL with branching

The interpretation of terms becomes similarly indexed:

$$\mathcal{R}[\Gamma \vdash M : A]_E : \mathcal{R}[\Gamma]_E \rightarrow \mathcal{R}[A]_E$$

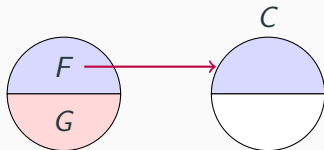
$$\mathcal{R}[\text{inl } M : A + B]_E(\gamma) = (E, \emptyset, \mathcal{R}[M]_E, \emptyset_B)$$

$$\mathcal{R}[\text{inr } M : A + B]_E(\gamma) = (\emptyset, E, \emptyset_A, \mathcal{R}[M]_E)$$

$$\mathcal{R}[\text{case}(M, x.N, y.O) : C]_E(\gamma) =$$

$$\text{let } (F, G, X, Y) = \mathcal{R}[M]_E(\gamma) \text{ in}$$

$$\mathcal{R}[N]_F(\gamma|_F, X) \sqcup_C \mathcal{R}[O]_G(\gamma|_G, Y)$$



Extending TinyPPL with branching

The interpretation of terms becomes similarly indexed:

$$\mathcal{R}[\Gamma \vdash M : A]_E : \mathcal{R}[\Gamma]_E \rightarrow \mathcal{R}[A]_E$$

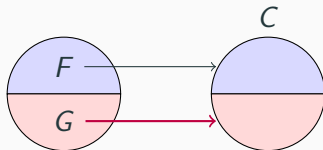
$$\mathcal{R}[\text{inl } M : A + B]_E(\gamma) = (E, \emptyset, \mathcal{R}[M]_E, \emptyset_B)$$

$$\mathcal{R}[\text{inr } M : A + B]_E(\gamma) = (\emptyset, E, \emptyset_A, \mathcal{R}[M]_E)$$

$$\mathcal{R}[\text{case}(M, x.N, y.O) : C]_E(\gamma) =$$

$$\text{let } (F, G, X, Y) = \mathcal{R}[M]_E(\gamma) \text{ in}$$

$$\mathcal{R}[N]_F(\gamma|_F, X) \sqcup_C \mathcal{R}[O]_G(\gamma|_G, Y)$$



Extending TinyPPL with branching

The interpretation of terms becomes similarly indexed:

$$\mathcal{R}[\Gamma \vdash M : A]_E : \mathcal{R}[\Gamma]_E \rightarrow \mathcal{R}[A]_E$$

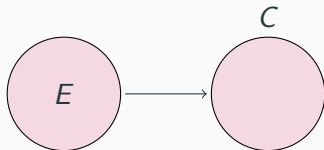
$$\mathcal{R}[\text{inl } M : A + B]_E(\gamma) = (E, \emptyset, \mathcal{R}[M]_E, \emptyset_B)$$

$$\mathcal{R}[\text{inr } M : A + B]_E(\gamma) = (\emptyset, E, \emptyset_A, \mathcal{R}[M]_E)$$

$$\mathcal{R}[\text{case}(M, x.N, y.O) : C]_E(\gamma) =$$

$$\text{let } (F, G, X, Y) = \mathcal{R}[M]_E(\gamma) \text{ in}$$

$$\mathcal{R}[N]_F(\gamma|_F, X) \sqcup_C \mathcal{R}[O]_G(\gamma|_G, Y)$$



Extending TinyPPL with branching

The interpretation of terms becomes similarly indexed:

$$\mathcal{R}[\Gamma \vdash M : A]_E : \mathcal{R}[\Gamma]_E \rightarrow \mathcal{R}[A]_E$$

$$\mathcal{R}[\text{inl } M : A + B]_E(\gamma) = (E, \emptyset, \mathcal{R}[M]_E, \emptyset_B)$$

$$\mathcal{R}[\text{inr } M : A + B]_E(\gamma) = (\emptyset, E, \emptyset_A, \mathcal{R}[M]_E)$$

$$\mathcal{R}[\text{case}(M, x.N, y.O) : C]_E(\gamma) =$$

$$\text{let } (F, G, X, Y) = \mathcal{R}[M]_E(\gamma) \text{ in}$$

$$\mathcal{R}[N]_F(\gamma|_F, X) \sqcup_C \mathcal{R}[O]_G(\gamma|_G, Y)$$

Every type comes with a restriction operation:

$$X|_E^{\text{Bool}} = X \cap E$$

$$(x, y)|_E^{A \times B} = (x|_E^A, y|_E^B)$$

$$(F, G, x, y)|_E^{A+B} = (F \cap E, G \cap E, x|_E^A, y|_E^B)$$

Extending TinyPPL with branching

The interpretation of terms becomes similarly indexed:

$$\mathcal{R}[\Gamma \vdash M : A]_E : \mathcal{R}[\Gamma]_E \rightarrow \mathcal{R}[A]_E$$

$$\mathcal{R}[\text{inl } M : A + B]_E(\gamma) = (E, \emptyset, \mathcal{R}[M]_E, \emptyset_B)$$

$$\mathcal{R}[\text{inr } M : A + B]_E(\gamma) = (\emptyset, E, \emptyset_A, \mathcal{R}[M]_E)$$

$$\mathcal{R}[\text{case}(M, x.N, y.O) : C]_E(\gamma) =$$

$$\text{let } (F, G, X, Y) = \mathcal{R}[M]_E(\gamma) \text{ in}$$

$$\mathcal{R}[N]_F(\gamma|_F, X) \sqcup_{\text{C}} \mathcal{R}[O]_G(\gamma|_G, Y)$$

Every type comes with a merge operation:

$$X \sqcup_{\text{Bool}} Y = X \cup Y$$

$$(a, b) \sqcup_{A \times B} (a', b') = (a \sqcup_A a', b \sqcup_B b')$$

$$(F, G, a, b) \sqcup_{A+B} (F', G', a', b') = (F \cup F', G \cup G', a \sqcup_A a', b \sqcup_B b')$$

Extending TinyPPL with branching

The interpretation of terms becomes similarly indexed:

$$\mathcal{R}[\Gamma \vdash M : A]_E : \mathcal{R}[\Gamma]_E \rightarrow \mathcal{R}[A]_E$$

$$\mathcal{R}[\text{inl } M : A + B]_E(\gamma) = (E, \emptyset, \mathcal{R}[M]_E, \emptyset_B)$$

$$\mathcal{R}[\text{inr } M : A + B]_E(\gamma) = (\emptyset, E, \emptyset_A, \mathcal{R}[M]_E)$$

$$\mathcal{R}[\text{case}(M, x.N, y.O) : C]_E(\gamma) =$$

$$\text{let } (F, G, X, Y) = \mathcal{R}[M]_E(\gamma) \text{ in}$$

$$\mathcal{R}[N]_F(\gamma|_F, X) \sqcup_C \mathcal{R}[O]_G(\gamma|_G, Y)$$

Extending TinyPPL with branching

The following hold by construction:

Extending TinyPPL with branching

The following hold by construction:

1. Across the isomorphism $\mathcal{R}[[A]]_E \cong (E \rightarrow [[A]])$,

Extending TinyPPL with branching

The following hold by construction:

1. Across the isomorphism $\mathcal{R}[[A]]_E \cong (E \rightarrow [[A]])$,
 - $(-)|_{(-)}^A$ corresponds to function restriction;

Extending TinyPPL with branching

The following hold by construction:

1. Across the isomorphism $\mathcal{R}[[A]]_E \cong (E \rightarrow [[A]])$,
 - $(-)|_{(-)}^A$ corresponds to function restriction;
 - $\emptyset_A$ corresponds to the empty function $\emptyset \rightarrow [[A]]$;

Extending TinyPPL with branching

The following hold by construction:

1. Across the isomorphism $\mathcal{R}[[A]]_E \cong (E \rightarrow [[A]])$,
 - $(-)|_{(-)}^A$ corresponds to function restriction;
 - $\emptyset_A$ corresponds to the empty function $\emptyset \rightarrow [[A]]$;
 - $\sqcup_A$ corresponds to unioning of functions with disjoint domain.

Extending TinyPPL with branching

The following hold by construction:

1. Across the isomorphism $\mathcal{R}[\![A]\!]_E \cong (E \rightarrow \mathbb{A})$,
 - $(-)|_{(-)}^A$ corresponds to function restriction;
 - $\emptyset_A$ corresponds to the empty function $\emptyset \rightarrow \mathbb{A}$;
 - $\sqcup_A$ corresponds to unioning of functions with disjoint domain.
2. For all programs M and $E \subseteq \Omega$, have:

$$\begin{array}{ccc} \mathcal{R}[\![\Gamma]\!]_E & \xrightarrow{\mathcal{R}[\![M]\!]_E} & \mathcal{R}[\![A]\!]_E \\ \cong \downarrow & & \downarrow \cong \\ (E \rightarrow \mathbb{A}) & \xrightarrow{\text{lift } \mathbb{A}} & (E \rightarrow \mathbb{A}) \end{array}$$

TinyPPL as a metalanguage for symbolic semantics

TinyPPL as a metalanguage for symbolic semantics

Example: assertions in Roulette.

TinyPPL as a metalanguage for symbolic semantics

Example: assertions in Roulette.

$$\hat{\rho} \vdash (e, \hat{\sigma}) \Downarrow (\hat{v}, \hat{\sigma}', \psi)$$

(weight maps elided)

TinyPPL as a metalanguage for symbolic semantics

Example: assertions in Roulette.

$$\hat{\rho} \vdash (e, \hat{\sigma}) \Downarrow (\hat{v}, \hat{\sigma}', \psi)$$

(weight maps elided)

ASSERT

$$\frac{\hat{\rho}(x) = [\varphi_1 : \text{true}, \varphi_2 : \text{false}]}{\hat{\rho} \vdash (\text{assert } x), \hat{\sigma} \Downarrow (\text{void}, \hat{\sigma}, \varphi_1)}$$

TinyPPL as a metalanguage for symbolic semantics

Example: assertions in Roulette.

$$\hat{\rho} \vdash (e, \hat{\sigma}) \Downarrow (\hat{v}, \hat{\sigma}', \psi)$$

(weight maps elided)

ASSERT

$$\frac{\hat{\rho}(x) = [\varphi_1 : \text{true}, \varphi_2 : \text{false}]}{\hat{\rho} \vdash (\text{assert } x), \hat{\sigma} \Downarrow (\text{void}, \hat{\sigma}, \varphi_1)}$$

LET

$$\frac{\hat{\rho} \vdash (e_1, \hat{\sigma}) \Downarrow (\hat{v}_1, \hat{\sigma}_1, \psi_1) \quad \hat{\rho}[x \mapsto \hat{v}_1] \vdash (e_2, \hat{\sigma}_1) \Downarrow (\hat{v}_2, \hat{\sigma}_2, \psi_2)}{\hat{\rho} \vdash (\text{let } ([x \ e_1]) \ e_2), \hat{\sigma} \Downarrow (\hat{v}_2, \hat{\sigma}_2, \psi_1 \wedge \psi_2)}$$

TinyPPL as a metalanguage for symbolic semantics

Example: assertions in Roulette.

$$\hat{\rho} \vdash (e, \hat{\sigma}) \Downarrow (\hat{v}, \hat{\sigma}', \psi)$$

(weight maps elided)

ASSERT

$$\frac{\hat{\rho}(x) = [\varphi_1 : \text{true}, \varphi_2 : \text{false}]}{\hat{\rho} \vdash ((\text{assert } x), \hat{\sigma}) \Downarrow ((\text{void}), \hat{\sigma}, \varphi_1)}$$

LET

$$\frac{\hat{\rho} \vdash (e_1, \hat{\sigma}) \Downarrow (\hat{v}_1, \hat{\sigma}_1, \psi_1) \quad \hat{\rho}[x \mapsto \hat{v}_1] \vdash (e_2, \hat{\sigma}_1) \Downarrow (\hat{v}_2, \hat{\sigma}_2, \psi_2)}{\hat{\rho} \vdash ((\text{let } ([x \ e_1]) \ e_2), \hat{\sigma}) \Downarrow (\hat{v}_2, \hat{\sigma}_2, \psi_1 \wedge \psi_2)}$$

IF

$$\frac{\hat{\rho}(x) = [\varphi_1 : \text{true}, \varphi_2 : \text{false}] \quad \hat{\rho} \vdash (e_1, \hat{\sigma}) \Downarrow (\hat{v}_1, \hat{\sigma}_1, \psi_1) \quad \hat{\rho} \vdash (e_2, \hat{\sigma}) \Downarrow (\hat{v}_2, \hat{\sigma}_2, \psi_2)}{\hat{\rho} \vdash ((\text{if } x \ e_1 \ e_2), \hat{\sigma}) \Downarrow ([\varphi_1 : \hat{v}_1, \varphi_2 : \hat{v}_2], [\varphi_1 : \hat{\sigma}_1, \varphi_2 : \hat{\sigma}_2], (\varphi_1 \wedge \psi_1) \vee (\varphi_2 \wedge \psi_2))}$$

TinyPPL as a metalanguage for symbolic semantics

Example: assertions in Roulette.

$$\hat{\rho} \vdash (e, \hat{\sigma}) \Downarrow (\hat{v}, \hat{\sigma}', \psi)$$

(weight maps elided)

ASSERT

$$\frac{\hat{\rho}(x) = [\varphi_1 : \text{true}, \varphi_2 : \text{false}]}{\hat{\rho} \vdash ((\text{assert } x), \hat{\sigma}) \Downarrow ((\text{void}), \hat{\sigma}, \varphi_1)}$$

LET

$$\frac{\hat{\rho} \vdash (e_1, \hat{\sigma}) \Downarrow (\hat{v}_1, \hat{\sigma}_1, \psi_1) \quad \hat{\rho}[x \mapsto \hat{v}_1] \vdash (e_2, \hat{\sigma}_1) \Downarrow (\hat{v}_2, \hat{\sigma}_2, \psi_2)}{\hat{\rho} \vdash ((\text{let } ([x \ e_1]) \ e_2), \hat{\sigma}) \Downarrow (\hat{v}_2, \hat{\sigma}_2, \psi_1 \wedge \psi_2)}$$

IF

$$\frac{\hat{\rho}(x) = [\varphi_1 : \text{true}, \varphi_2 : \text{false}] \quad \hat{\rho} \vdash (e_1, \hat{\sigma}) \Downarrow (\hat{v}_1, \hat{\sigma}_1, \psi_1) \quad \hat{\rho} \vdash (e_2, \hat{\sigma}) \Downarrow (\hat{v}_2, \hat{\sigma}_2, \psi_2)}{\hat{\rho} \vdash ((\text{if } x \ e_1 \ e_2), \hat{\sigma}) \Downarrow ([\varphi_1 : \hat{v}_1, \varphi_2 : \hat{v}_2], [\varphi_1 : \hat{\sigma}_1, \varphi_2 : \hat{\sigma}_2], (\varphi_1 \wedge \psi_1) \vee (\varphi_2 \wedge \psi_2))}$$

TinyPPL as a metalanguage for symbolic semantics

Idea: $\text{assert}(M) \equiv \text{if } M \text{ then pure() else fail().}$

TinyPPL as a metalanguage for symbolic semantics

Idea: $\text{assert}(M) \equiv \text{if } M \text{ then pure() else fail()}$.

There is a standard monadic model of failure:

TinyPPL as a metalanguage for symbolic semantics

Idea: $\text{assert}(M) \equiv \text{if } M \text{ then pure() else fail()}$.

There is a standard monadic model of failure:

$$\text{Maybe } A = A + \text{Unit}$$

TinyPPL as a metalanguage for symbolic semantics

Idea: $\text{assert}(M) \equiv \text{if } M \text{ then pure() else fail()}$.

There is a standard monadic model of failure:

$$\text{Maybe } A = A + \text{Unit}$$

$$\text{return}(M) = \text{inl } M$$

$$\text{bind}(M, x.N) = \text{case}(M, x.N, u. \text{inr } u)$$

TinyPPL as a metalanguage for symbolic semantics

Idea: $\text{assert}(M) \equiv \text{if } M \text{ then pure() else fail()}$.

There is a standard monadic model of failure:

$$\text{Maybe } A = A + \text{Unit}$$
$$\text{return}(M) = \text{inl } M$$
$$\text{bind}(M, x.N) = \text{case}(M, x.N, u. \text{inr } u)$$
$$\text{assert} : \text{Bool} \rightarrow \text{Maybe Unit}$$
$$\text{assert}(M) = \text{if } M \text{ then inl() else inr()}$$

TinyPPL as a metalanguage for symbolic semantics

Idea: $\text{assert}(M) \equiv \text{if } M \text{ then pure() else fail()}$.

There is a standard monadic model of failure:

$$\text{Maybe } A = A + \text{Unit}$$
$$\text{return}(M) = \text{inl } M$$
$$\text{bind}(M, x.N) = \text{case}(M, x.N, u. \text{inr } u)$$
$$\text{assert} : \text{Bool} \rightarrow \text{Maybe Unit}$$
$$\text{assert}(M) = \text{if } M \text{ then inl() else inr()}$$

To get a symbolic semantics of assertions, simply interpret the above code under $\mathcal{R}[\![-]\!]$!

TinyPPL as a metalanguage for symbolic semantics

$$\mathcal{R}[\![\text{Maybe } A]\!]_E = \mathcal{R}[\![A + \text{Unit}]\!]_E$$

TinyPPL as a metalanguage for symbolic semantics

$$\begin{aligned}\mathcal{R}[\text{Maybe } A]_E &= \mathcal{R}[A + \text{Unit}]_E \\ &= \sum_{E=F+G} \mathcal{R}[A]_F \times \mathcal{R}[\text{Unit}]_G\end{aligned}$$

TinyPPL as a metalanguage for symbolic semantics

$$\begin{aligned}\mathcal{R}[\text{Maybe } A]_E &= \mathcal{R}[A + \text{Unit}]_E \\ &= \sum_{E=F+G} \mathcal{R}[A]_F \times \mathcal{R}[\text{Unit}]_G \\ &= \sum_{E=F+G} \mathcal{R}[A]_F \times 1\end{aligned}$$

TinyPPL as a metalanguage for symbolic semantics

$$\begin{aligned}\mathcal{R}[\text{Maybe } A]_E &= \mathcal{R}[A + \text{Unit}]_E \\ &= \sum_{E=F+G} \mathcal{R}[A]_F \times \mathcal{R}[\text{Unit}]_G \\ &= \sum_{E=F+G} \mathcal{R}[A]_F \times 1 \\ &= \sum_{F \subseteq E} \mathcal{R}[A]_F\end{aligned}$$

TinyPPL as a metalanguage for symbolic semantics

$$\mathcal{R}[\text{Maybe } A]_E = \sum_{F \subseteq E} \mathcal{R}[A]_F$$

TinyPPL as a metalanguage for symbolic semantics

$$\mathcal{R}[\![\text{Maybe } A]\!]_E = \sum_{F \subseteq E} \mathcal{R}[\![A]\!]_F$$

Recall: $\hat{\rho} \vdash (e, \hat{\sigma}) \Downarrow (\hat{v}, \hat{\sigma}', \psi)$

TinyPPL as a metalanguage for symbolic semantics

$$\mathcal{R}[\text{Maybe } A]_E = \sum_{F \subseteq E} \mathcal{R}[A]_F$$

$$\mathcal{R}[\text{assert}(M)]_E(\gamma) = \mathcal{R}[\text{if } M \text{ then inl() else inr()}]_E(\gamma)$$

TinyPPL as a metalanguage for symbolic semantics

$$\mathcal{R}[\text{Maybe } A]_E = \sum_{F \subseteq E} \mathcal{R}[A]_F$$

$$\mathcal{R}[\text{assert}(M)]_E(\gamma) = \text{let } F = \mathcal{R}[M]_E(\gamma) \text{ in } \mathcal{R}[\text{inl}()]_F \sqcup \mathcal{R}[\text{inr}()]_{E \setminus F}$$

TinyPPL as a metalanguage for symbolic semantics

$$\mathcal{R}[\text{Maybe } A]_E = \sum_{F \subseteq E} \mathcal{R}[A]_F$$

$$\mathcal{R}[\text{assert}(M)]_E(\gamma) = \text{let } F = \mathcal{R}[M]_E(\gamma) \text{ in } (F, E \setminus F, (), ())$$

TinyPPL as a metalanguage for symbolic semantics

$$\mathcal{R}[\text{Maybe } A]_E = \sum_{F \subseteq E} \mathcal{R}[A]_F$$

$$\mathcal{R}[\text{assert}(M)]_E(\gamma) = (\mathcal{R}[M]_E(\gamma), ())$$

TinyPPL as a metalanguage for symbolic semantics

$$\mathcal{R}[\text{Maybe } A]_E = \sum_{F \subseteq E} \mathcal{R}[A]_F$$

$$\mathcal{R}[\text{assert}(M)]_E(\gamma) = (\mathcal{R}[M]_E(\gamma), ())$$

$$\text{ASSERT} \frac{\hat{\rho}(x) = [\varphi_1 : \text{true}, \varphi_2 : \text{false}]}{\hat{\rho} \vdash ((\text{assert } x), \hat{\sigma}) \Downarrow ((\text{void}), \hat{\sigma}, \varphi_1)}$$

TinyPPL as a metalanguage for symbolic semantics

$$\mathcal{R}[\![\text{Maybe } A]\!]_E = \sum_{F \subseteq E} \mathcal{R}[\![A]\!]_F$$

$$\mathcal{R}[\![\text{assert}(M)]\!]_E(\gamma) = (\mathcal{R}[\![M]\!]_E(\gamma), ())$$

$$\begin{aligned} \mathcal{R}[\![\text{bind}(M, x.N)]\!]_E(\gamma) = \\ \mathcal{R}[\![\text{case}(M, x.N, u. \text{inr } u)]\!]_E(\gamma) \end{aligned}$$

TinyPPL as a metalanguage for symbolic semantics

$$\mathcal{R}[\text{Maybe } A]_E = \sum_{F \subseteq E} \mathcal{R}[A]_F$$

$$\mathcal{R}[\text{assert}(M)]_E(\gamma) = (\mathcal{R}[M]_E(\gamma), ())$$

$$\mathcal{R}[\text{bind}(M, x.N)]_E(\gamma) =$$

$$\text{let } (F, G, X, U) = \mathcal{R}[M]_E(\gamma) \text{ in } \mathcal{R}[N]_F(\gamma|_F, X) \sqcup \mathcal{R}[\text{inr } u]_G(\gamma|_G, U)$$

TinyPPL as a metalanguage for symbolic semantics

$$\mathcal{R}[\![\text{Maybe } A]\!]_E = \sum_{F \subseteq E} \mathcal{R}[A]_F$$

$$\mathcal{R}[\![\text{assert}(M)]\!]_E(\gamma) = (\mathcal{R}[M]_E(\gamma), ())$$

$$\mathcal{R}[\![\text{bind}(M, x.N)]\!]_E(\gamma) =$$

$$\text{let } (F, G, X, _) = \mathcal{R}[M]_E(\gamma) \text{ in } \mathcal{R}[N]_F(\gamma|_F, X) \sqcup (\emptyset, \textcolor{red}{G}, \emptyset, ())$$

TinyPPL as a metalanguage for symbolic semantics

$$\mathcal{R}[\text{Maybe } A]_E = \sum_{F \subseteq E} \mathcal{R}[A]_F$$

$$\mathcal{R}[\text{assert}(M)]_E(\gamma) = (\mathcal{R}[M]_E(\gamma), ())$$

$$\begin{aligned} \mathcal{R}[\text{bind}(M, x.N)]_E(\gamma) = \\ \text{let } (F, X) = \mathcal{R}[M]_E(\gamma) \text{ in } \mathcal{R}[N]_{E \cap F}(\gamma|_F, X) \end{aligned}$$

TinyPPL as a metalanguage for symbolic semantics

$$\mathcal{R}[\text{Maybe } A]_E = \sum_{F \subseteq E} \mathcal{R}[A]_F$$

$$\mathcal{R}[\text{assert}(M)]_E(\gamma) = (\mathcal{R}[M]_E(\gamma), ())$$

$$\mathcal{R}[\text{bind}(M, x.N)]_E(\gamma) =$$

$$\text{let } (F, X) = \mathcal{R}[M]_E(\gamma) \text{ in } \mathcal{R}[N]_{E \cap F}(\gamma|_F, X)$$

$$\text{LET} \frac{\hat{\rho} \vdash (e_1, \hat{\sigma}) \Downarrow (\hat{v}_1, \hat{\sigma}_1, \psi_1) \quad \hat{\rho}[x \mapsto \hat{v}_1] \vdash (e_2, \hat{\sigma}_1) \Downarrow (\hat{v}_2, \hat{\sigma}_2, \psi_2)}{\hat{\rho} \vdash (\text{let } ([x \ e_1]) \ e_2), \hat{\sigma}) \Downarrow (\hat{v}_2, \hat{\sigma}_2, \psi_1 \wedge \psi_2)}$$

TinyPPL as a metalanguage for symbolic semantics

$$\mathcal{R}[\![\text{Maybe } A]\!]_E = \sum_{F \subseteq E} \mathcal{R}[\![A]\!]_F$$

$$\mathcal{R}[\![\text{assert}(M)]\!]_E(\gamma) = (\mathcal{R}[\![M]\!]_E(\gamma), ())$$

$$\begin{aligned} \mathcal{R}[\![\text{bind}(M, x.N)]\!]_E(\gamma) = \\ \text{let } (F, X) = \mathcal{R}[\![M]\!]_E(\gamma) \text{ in } \mathcal{R}[\![N]\!]_{E \cap F}(\gamma|_F, X) \end{aligned}$$

$$\begin{aligned} \mathcal{R}[\![\text{if } M \text{ then } N_1 \text{ else } N_2]\!]_E(\gamma) = \\ \text{let } F_1 = \mathcal{R}[\![M]\!]_E(\gamma) \\ F_2 = E \setminus F_1 \\ (G_1, v_1) = \mathcal{R}[\![N_1]\!]_{F_1}(\gamma|_{F_1}) \\ (G_2, v_2) = \mathcal{R}[\![N_2]\!]_{F_2}(\gamma|_{F_2}) \\ \text{in } ((F_1 \cap G_1) \cup (F_2 \cap G_2), v_1 \sqcup v_2) \end{aligned}$$

TinyPPL as a metalanguage for symbolic semantics

$$\mathcal{R}[\text{Maybe } A]_E = \sum_{F \subseteq E} \mathcal{R}[A]_F$$

$$\mathcal{R}[\text{assert}(M)]_E(\gamma) = (\mathcal{R}[M]_E(\gamma), ())$$

$$\mathcal{R}[\text{bind}(M, x.N)]_E(\gamma) =$$

$$\text{let } (F, X) = \mathcal{R}[M]_E(\gamma) \text{ in } \mathcal{R}[N]_{E \cap F}(\gamma|_F, X)$$

$$\mathcal{R}[\text{if } M \text{ then } N_1 \text{ else } N_2]_E(\gamma) =$$

$$\text{let } F_1 = \mathcal{R}[M]_E(\gamma)$$

$$F_2 = E \setminus F_1$$

$$(G_1, v_1) = \mathcal{R}[N_1]_{F_1}(\gamma|_{F_1})$$

$$(G_2, v_2) = \mathcal{R}[N_2]_{F_2}(\gamma|_{F_2})$$

$$\text{in } ((F_1 \cap G_1) \cup (F_2 \cap G_2), v_1 \sqcup v_2)$$

TinyPPL as a metalanguage for symbolic semantics

$$\mathcal{R}[\text{Maybe } A]_E = \sum_{F \subseteq E} \mathcal{R}[A]_F$$

$$\mathcal{R}[\text{assert}(M)]_E(\gamma) = (\mathcal{R}[M]_E(\gamma), ())$$

$$\mathcal{R} \left[\begin{array}{c} \hat{\rho}(x) = [\varphi_1 : \text{true}, \varphi_2 : \text{false}] \\ \text{I}_F \frac{\hat{\rho} \vdash (e_1, \hat{\sigma}) \Downarrow (\hat{v}_1, \hat{\sigma}_1, \psi_1) \quad \hat{\rho} \vdash (e_2, \hat{\sigma}) \Downarrow (\hat{v}_2, \hat{\sigma}_2, \psi_2)}{\hat{\rho} \vdash ((\text{if } x \text{ e}_1 \text{ e}_2), \hat{\sigma}) \Downarrow ([\varphi_1 : \hat{v}_1, \varphi_2 : \hat{v}_2], [\varphi_1 : \hat{\sigma}_1, \varphi_2 : \hat{\sigma}_2], (\varphi_1 \wedge \psi_1) \vee (\varphi_2 \wedge \psi_2))} \end{array} \right]$$

$$\mathcal{R}[\text{if } M \text{ then } N_1 \text{ else } N_2]_E(\gamma) =$$

$$\text{let } F_1 = \mathcal{R}[M]_E(\gamma)$$

$$F_2 = E \setminus F_1$$

$$(G_1, v_1) = \mathcal{R}[N_1]_{F_1}(\gamma|_{F_1})$$

$$(G_2, v_2) = \mathcal{R}[N_2]_{F_2}(\gamma|_{F_2})$$

$$\text{in } ((F_1 \cap G_1) \cup (F_2 \cap G_2), v_1 \sqcup v_2)$$

TinyPPL as a metalanguage for symbolic semantics

$$\mathcal{R}[\![\text{Maybe } A]\!]_E = \sum_{F \subseteq E} \mathcal{R}[\![A]\!]_F$$

$$\mathcal{R}[\![\text{assert}(M)]\!]_E(\gamma) = (\mathcal{R}[\![M]\!]_E(\gamma), ())$$

$$\begin{aligned} \mathcal{R}[\![\text{bind}(M, x.N)]\!]_E(\gamma) = \\ \text{let } (F, X) = \mathcal{R}[\![M]\!]_E(\gamma) \text{ in } \mathcal{R}[\![N]\!]_{E \cap F}(\gamma|_F, X) \end{aligned}$$

$$\begin{aligned} \mathcal{R}[\![\text{if } M \text{ then } N_1 \text{ else } N_2]\!]_E(\gamma) = \\ \text{let } F_1 = \mathcal{R}[\![M]\!]_E(\gamma) \\ F_2 = E \setminus F_1 \\ (G_1, v_1) = \mathcal{R}[\![N_1]\!]_{F_1}(\gamma|_{F_1}) \\ (G_2, v_2) = \mathcal{R}[\![N_2]\!]_{F_2}(\gamma|_{F_2}) \\ \text{in } ((F_1 \cap G_1) \cup (F_2 \cap G_2), v_1 \sqcup v_2) \end{aligned}$$

Making contact with sheaf theory

Each type A is interpreted by:

Making contact with sheaf theory

Each type A is interpreted by:

- A set $\mathcal{R}[[A]]_E$ for each subset $E \subseteq \Omega$

Making contact with sheaf theory

Each type A is interpreted by:

- A set $\mathcal{R}[[A]]_E$ for each subset $E \subseteq \Omega$
- Restriction: $(-)|_F : \mathcal{R}[[A]]_E \rightarrow \mathcal{R}[[A]]_F$ for all $F \subseteq E$

Making contact with sheaf theory

Each type A is interpreted by:

- A set $\mathcal{R}[[A]]_E$ for each subset $E \subseteq \Omega$
- Restriction: $(-)|_F : \mathcal{R}[[A]]_E \rightarrow \mathcal{R}[[A]]_F$ for all $F \subseteq E$
- Merge: $(\sqcup) : \mathcal{R}[[A]]_E \rightarrow \mathcal{R}[[A]]_F \rightarrow \mathcal{R}[[A]]_{E \cup F}$ for disjoint E, F

Making contact with sheaf theory

Each type A is interpreted by:

- A set $\mathcal{R}[[A]]_E$ for each subset $E \subseteq \Omega$
- Restriction: $(-)|_F : \mathcal{R}[[A]]_E \rightarrow \mathcal{R}[[A]]_F$ for all $F \subseteq E$
- Merge: $(\sqcup) : \mathcal{R}[[A]]_E \rightarrow \mathcal{R}[[A]]_F \rightarrow \mathcal{R}[[A]]_{E \cup F}$ for disjoint E, F
- An empty element: $\emptyset_A \in \mathcal{R}[[A]]_\emptyset$

Making contact with sheaf theory

Each type A is interpreted by:

- A set $\mathcal{R}[[A]]_E$ for each subset $E \subseteq \Omega$
- Restriction: $(-)|_F : \mathcal{R}[[A]]_E \rightarrow \mathcal{R}[[A]]_F$ for all $F \subseteq E$
- Merge: $(\sqcup) : \mathcal{R}[[A]]_E \rightarrow \mathcal{R}[[A]]_F \rightarrow \mathcal{R}[[A]]_{E \cup F}$ for disjoint E, F
- An empty element: $\emptyset_A \in \mathcal{R}[[A]]_\emptyset$

satisfying suitable laws.

Making contact with sheaf theory

Each type A is interpreted by:

- A set $\mathcal{R}[[A]]_E$ for each subset $E \subseteq \Omega$
- Restriction: $(-)|_F : \mathcal{R}[[A]]_E \rightarrow \mathcal{R}[[A]]_F$ for all $F \subseteq E$
- Merge: $(\sqcup) : \mathcal{R}[[A]]_E \rightarrow \mathcal{R}[[A]]_F \rightarrow \mathcal{R}[[A]]_{E \cup F}$ for disjoint E, F
- An empty element: $\emptyset_A \in \mathcal{R}[[A]]_\emptyset$

satisfying suitable laws.

Structures of this shape have been extensively studied by mathematicians in the context of sheaf theory.

$$\mathcal{R}[\text{Bool}]_E = \{E' \mid E' \subseteq E\}$$

$$\mathcal{R}[A \times B]_E = \mathcal{R}[A]_E \times \mathcal{R}[B]_E$$

$$\mathcal{R}[A + B]_E = \sum_{E=F+G} \mathcal{R}[A]_F \times \mathcal{R}[B]_G$$

$$\mathcal{R}[\text{Bool}]_E = \{E' \mid E' \subseteq E\}$$

$$\mathcal{R}[A \times B]_E = \mathcal{R}[A]_E \times \mathcal{R}[B]_E$$

$$\mathcal{R}[A + B]_E = \sum_{E=F+G} \mathcal{R}[A]_F \times \mathcal{R}[B]_G$$

The rest of the owl

$$\mathcal{R}[\text{Bool}]_E = \{E' \mid E' \subseteq E\}$$

$$\mathcal{R}[A \times B] = \mathcal{R}[A] \times_{\text{sh}} \mathcal{R}[B]$$

$$\mathcal{R}[A + B]_E = \sum_{E=F+G} \mathcal{R}[A]_F \times \mathcal{R}[B]_G$$

$\times_{\text{sh}}$ is the product of sheaves

$$\mathcal{R}[\text{Bool}]_E = \{E' \mid E' \subseteq E\}$$

$$\mathcal{R}[A \times B] = \mathcal{R}[A] \times_{\text{sh}} \mathcal{R}[B]$$

$$\mathcal{R}[A + B]_E = \sum_{E=F+G} \mathcal{R}[A]_F \times \mathcal{R}[B]_G$$

The rest of the owl

$$\mathcal{R}[\text{Bool}]_E = \{E' \mid E' \subseteq E\}$$

$$\mathcal{R}[A \times B] = \mathcal{R}[A] \times_{\text{sh}} \mathcal{R}[B]$$

$$\mathcal{R}[A + B] = \mathcal{R}[A] +_{\text{sh}} \mathcal{R}[B]$$

$+_{\text{sh}}$ is the coproduct of sheaves

The rest of the owl

$$\mathcal{R}[\text{Bool}] = 1_{\text{Sh}} +_{\text{Sh}} 1_{\text{Sh}}$$

$$\mathcal{R}[A \times B] = \mathcal{R}[A] \times_{\text{Sh}} \mathcal{R}[B]$$

$$\mathcal{R}[A + B] = \mathcal{R}[A] +_{\text{Sh}} \mathcal{R}[B]$$

1_{Sh} is the singleton sheaf

The rest of the owl

$$\mathcal{R}[\text{Bool}] = 1_{\text{Sh}} +_{\text{Sh}} 1_{\text{Sh}}$$

$$\mathcal{R}[A \times B] = \mathcal{R}[A] \times_{\text{Sh}} \mathcal{R}[B]$$

$$\mathcal{R}[A + B] = \mathcal{R}[A] +_{\text{Sh}} \mathcal{R}[B]$$

$$\mathcal{R}[A \rightarrow B] = \mathcal{R}[A] \Rightarrow_{\text{Sh}} \mathcal{R}[B]$$

$\Rightarrow_{\text{Sh}}$ is the exponential of sheaves

The rest of the owl

$$\mathcal{R}[\text{Bool}] = 1_{\text{Sh}} +_{\text{Sh}} 1_{\text{Sh}}$$

$$\mathcal{R}[A \times B] = \mathcal{R}[A] \times_{\text{Sh}} \mathcal{R}[B]$$

$$\mathcal{R}[A + B] = \mathcal{R}[A] +_{\text{Sh}} \mathcal{R}[B]$$

$$\mathcal{R}[A \rightarrow B] = \mathcal{R}[A] \Rightarrow_{\text{Sh}} \mathcal{R}[B]$$

$$\mathcal{R}[\text{List } A] = \mu X. 1_{\text{Sh}} +_{\text{Sh}} \mathcal{R}[A] \times_{\text{Sh}} X$$

$$\mathcal{R}[\text{Tree } A] = \mu X. 1_{\text{Sh}} +_{\text{Sh}} \mathcal{R}[A] \times_{\text{Sh}} X \times_{\text{Sh}} X$$

The rest of the owl

$$\mathcal{R}[\text{Bool}] = 1_{\text{Sh}} +_{\text{Sh}} 1_{\text{Sh}}$$

$$\mathcal{R}[A \times B] = \mathcal{R}[A] \times_{\text{Sh}} \mathcal{R}[B]$$

$$\mathcal{R}[A + B] = \mathcal{R}[A] +_{\text{Sh}} \mathcal{R}[B]$$

$$\mathcal{R}[A \rightarrow B] = \mathcal{R}[A] \Rightarrow_{\text{Sh}} \mathcal{R}[B]$$

$$\mathcal{R}[\text{List } A] = \mu X. 1_{\text{Sh}} +_{\text{Sh}} \mathcal{R}[A] \times_{\text{Sh}} X$$

$$\mathcal{R}[\text{Tree } A] = \mu X. 1_{\text{Sh}} +_{\text{Sh}} \mathcal{R}[A] \times_{\text{Sh}} X \times_{\text{Sh}} X$$

$$\mathcal{R}[\{v : \text{List Bool} \mid \text{len}(v) = n\}] = \{v \in \mathcal{R}[\text{List Bool}] \mid \llbracket \text{len}(v) = n \rrbracket\}$$

The rest of the owl

$$\mathcal{R}[\text{Bool}] = 1_{\text{Sh}} +_{\text{Sh}} 1_{\text{Sh}}$$

$$\mathcal{R}[A \times B] = \mathcal{R}[A] \times_{\text{Sh}} \mathcal{R}[B]$$

$$\mathcal{R}[A + B] = \mathcal{R}[A] +_{\text{Sh}} \mathcal{R}[B]$$

$$\mathcal{R}[A \rightarrow B] = \mathcal{R}[A] \Rightarrow_{\text{Sh}} \mathcal{R}[B]$$

$$\mathcal{R}[\text{List } A] = \mu X. 1_{\text{Sh}} +_{\text{Sh}} \mathcal{R}[A] \times_{\text{Sh}} X$$

$$\mathcal{R}[\text{Tree } A] = \mu X. 1_{\text{Sh}} +_{\text{Sh}} \mathcal{R}[A] \times_{\text{Sh}} X \times_{\text{Sh}} X$$

$$\mathcal{R}[\{v : \text{List Bool} \mid \text{len}(v) = n\}] = \{v \in \mathcal{R}[\text{List Bool}] \mid \llbracket \text{len}(v) = n \rrbracket\}$$

$$\mathcal{R}[\text{Set } A] = \mathcal{R}[\text{List } A] /_{\text{Sh}} \text{reordering}$$

The rest of the owl

$$\mathcal{R}[\text{Bool}] = 1_{\text{Sh}} +_{\text{Sh}} 1_{\text{Sh}}$$

$$\mathcal{R}[A \times B] = \mathcal{R}[A] \times_{\text{Sh}} \mathcal{R}[B]$$

$$\mathcal{R}[A + B] = \mathcal{R}[A] +_{\text{Sh}} \mathcal{R}[B]$$

$$\mathcal{R}[A \rightarrow B] = \mathcal{R}[A] \Rightarrow_{\text{Sh}} \mathcal{R}[B]$$

$$\mathcal{R}[\text{List } A] = \mu X. 1_{\text{Sh}} +_{\text{Sh}} \mathcal{R}[A] \times_{\text{Sh}} X$$

$$\mathcal{R}[\text{Tree } A] = \mu X. 1_{\text{Sh}} +_{\text{Sh}} \mathcal{R}[A] \times_{\text{Sh}} X \times_{\text{Sh}} X$$

$$\mathcal{R}[\{v : \text{List Bool} \mid \text{len}(v) = n\}] = \{v \in \mathcal{R}[\text{List Bool}] \mid \llbracket \text{len}(v) = n \rrbracket\}$$

$$\mathcal{R}[\text{Set } A] = \mathcal{R}[\text{List } A] /_{\text{Sh}} \text{reordering}$$

Everything remains correct by construction!

Takeaways

- Exact probabilistic inference can be implemented as symbolic execution plus weighted model counting.
- State merging keeps the number of paths under control; type-directed merging, à la Rosette, keeps symbolic values small.
- Denotationally, symbolic execution lifts ordinary execution to operate on random variables.
- Sheaves give a systematic, correct-by-construction way to calculate symbolic representations of richer data types.

Conclusion

Theme: PPLs at the intersection of PL and AI

PL

- Deductive reasoning, program logics
- Formal verification
- Language interoperability
- Formal semantics
- Relative expressivity of languages

AI

- Scalable inference techniques
- Probabilistic modeling
- Tractable probabilistic models
- Broad application domains (medicine, science, robotics)

Links and more resources

- Ongoing course: <https://sholtzen.dev/fall26-pwp/>
- The LAFL workshop at POPL every year
- OPLSS course on probabilistic programming: <https://www.khoury.northeastern.edu/home/sholtzen/oplss24-ppl/>
- Roulette source:
<https://github.com/camoy/roulette/>
- Seminar course on PPLs:
<https://neuppl.github.io/CS7470-Fall23/>

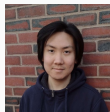
Acknowledgments



Cameron Moy



Jack Czenszak



John M. Li



Brianna Marshall



Sam Stites



John Gouwar



Minsung Cho



Amal Ahmed



Sandia
National
Laboratories



References

- [1] Alejandro Aguirre, Philipp G Haselwarter, Markus De Medeiros, Kwing Hei Li, Simon Oddershede Gregersen, Joseph Tassarotti, and Lars Birkedal. Error credits: Resourceful reasoning about error bounds for higher-order probabilistic programs. *Proceedings of the ACM on Programming Languages*, 8(ICFP):284–316, 2024.
- [2] Jialu Bao, Emanuele D’Osualdo, and Azadeh Farzan. Bluebell: An alliance of relational lifting and independence for probabilistic reasoning. *Proceedings of the ACM on Programming Languages*, 9(POPL):1719–1749, 2025.

- [3] Gilles Barthe, Marco Gaboardi, Benjamin Grégoire, Justin Hsu, and Pierre-Yves Strub. Proving differential privacy via probabilistic couplings. In *Proceedings of the 31st Annual ACM/IEEE Symposium on Logic in Computer Science*, pages 749–758, 2016.
- [4] Gilles Barthe, Justin Hsu, and Kevin Liao. A probabilistic separation logic. *Proceedings of the ACM on Programming Languages*, 4(POPL):1–30, 2019.
- [5] Maddy Bowers, Alexander K Lew, Joshua B Tenenbaum, Armando Solar-Lezama, and Vikash K Mansinghka. Stochastic lazy knowledge compilation for inference in discrete probabilistic programs. *Proceedings of the ACM on Programming Languages*, 9(PLDI):1863–1887, 2025.

- [6] Bob Carpenter, Andrew Gelman, Matthew D Hoffman, Daniel Lee, Ben Goodrich, Michael Betancourt, Marcus Brubaker, Jiqiang Guo, Peter Li, and Allen Riddell. Stan: A probabilistic programming language. *Journal of statistical software*, 76: 1–32, 2017.
- [7] Mark Chavira and Adnan Darwiche. On probabilistic inference by weighted model counting. *Artificial Intelligence*, 172(6-7): 772–799, 2008.
- [8] Ryan Culpepper and Andrew Cobb. Contextual equivalence for probabilistic programs with continuous random variables and scoring. In *European Symposium on Programming*, pages 368–392. Springer, 2017.

- [9] Markus de Medeiros, Muhammad Naveed, Tancrede Lepoint, Temesghen Kahsai, Tristan Ravitch, Stefan Zetsche, Anjali Joshi, Joseph Tassarotti, Aws Albarghouthi, and Jean-Baptiste Tristan. Verified foundations for differential privacy. *arXiv preprint arXiv:2412.01671*, 2024.
- [10] 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.
- [11] Poorva Garg, Steven Holtzen, Guy Van den Broeck, and Todd Millstein. Bit blasting probabilistic programs. *Proceedings of the ACM on Programming Languages*, 8(PLDI):865–888, 2024.

- [12] Timon Gehr, Sasa Misailovic, Petar Tsankov, Laurent Vanbever, Pascal Wiesmann, and Martin Vechev. Bayonet: probabilistic inference for networks. *ACM SIGPLAN Notices*, 53(4):586–602, 2018.
- [13] Michele Giry. A categorical approach to probability theory. In *Categorical Aspects of Topology and Analysis: Proceedings of an International Conference Held at Carleton University, Ottawa, August 11–15, 1981*, pages 68–85. Springer, 2006.
- [14] Benjamin King Goodrich, Gregory Wawro, and Ira Katznelson. Designing quantitative historical social inquiry: an introduction to stan. In *APSA 2012 Annual Meeting Paper*, 2012.

- [15] Tobias Gürtler and Benjamin Lucien Kaminski. notice: Inference for discrete probabilistic programs with nondeterminism and conditioning. *Proceedings of the ACM on Programming Languages*, 10(OOPSLA1):427–455, 2026.
- [16] Chris Heunen, Ohad Kammar, Sam Staton, and Hongseok Yang. A convenient category for higher-order probability theory. In *2017 32nd Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*, pages 1–12. IEEE, 2017.
- [17] Shing Hin Ho, Nicolas Wu, and Azalea Raad. Bayesian separation logic: A logical foundation and axiomatic semantics for probabilistic programming. *Proceedings of the ACM on Programming Languages*, 10(POPL):1557–1585, 2026.

- [18] 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.
- [19] Steven Holtzen, Sebastian Junges, Marcell Vazquez-Chanlatte, Todd Millstein, Sanjit A. Seshia, and Guy Van den Broeck. Model checking finite-horizon markov chains with probabilistic inference. In *Proceedings of the 33rd International Conference on Computer-Aided Verification (CAV)*, July 2021. doi: 10.1007/978-3-030-81688-9_27.
- [20] Mark Jerrum. *Counting, sampling and integrating: algorithms and complexity*. Springer Science & Business Media, 2003.

- [21] Benjamin Lucien Kaminski, Joost-Pieter Katoen, and Christoph Matheja. On the hardness of analyzing probabilistic programs. *Acta Informatica*, 56(3):255–285, 2019.
- [22] Lutz Klinkenberg, Kevin Batz, Benjamin Lucien Kaminski, Joost-Pieter Katoen, Joshua Moerman, and Tobias Winkler. Generating functions for probabilistic programs. In *International Symposium on Logic-Based Program Synthesis and Transformation*, pages 231–248. Springer, 2020.
- [23] Dexter Kozen. Semantics of probabilistic programs. In *20th Annual Symposium on Foundations of Computer Science (sfcs 1979)*, pages 101–114. IEEE, 1979.
- [24] John M Li, Amal Ahmed, and Steven Holtzen. Lilac: a modal separation logic for conditional probability. *Proceedings of the ACM on Programming Languages*, 7(PLDI):148–171, 2023.

- [25] Kwing Hei Li, Alejandro Aguirre, Simon Oddershede Gregersen, Philipp G Haselwarter, Joseph Tassarotti, and Lars Birkedal. Modular reasoning about error bounds for concurrent probabilistic programs. *Proceedings of the ACM on Programming Languages*, 9(ICFP):276–305, 2025.
- [26] Jack Liell-Cock and Sam Staton. Imprecise probabilistic programming, precisely: Credal sets via graded monads, bdds, and semiring-parametric inference (functional pearl). *Proceedings of the ACM on Programming Languages*, 10(ICFP):751–771, 2026.
- [27] Janine Lohse, Tim Rohde, Jimmy Xin, Niklas MĀžck, Iona Kuhn, Derek Dreyer, Deepak Garg, and Emanuele D’Osualdo. First steps towards probabilistic iris: Harmonizing independence, conditioning, and dynamic heap allocation. *arXiv preprint arXiv:2605.13765*, 2026.

- [28] Abdelrahman Madkour, Chris Martens, Steven Holtzen, Casper Hartevelde, and Stacy Marsella. Probabilistic logic programming semantics for procedural content generation. In *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, volume 19, pages 295–305, 2023.
- [29] Cameron Moy, Jack Czenszak, John M. Li, Brianna Marshall, and Steven Holtzen. Roulette: A language for expressive, exact, and efficient discrete probabilistic programming. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2025. doi: 10.1145/3729334.

- [30] Fanni Natanegara, Beat Neuenschwander, John W Seaman Jr, Nelson Kinnersley, Cory R Heilmann, David Ohlssen, and George Rochester. The current state of bayesian methods in medical product development: survey results and recommendations from the dia bayesian scientific working group. *Pharmaceutical statistics*, 13(1):3–12, 2014.
- [31] Lisa Oakley, Steven Holtzen, and Alina Oprea. Synthesizing tight privacy and accuracy bounds via weighted model counting. In *IEEE Computer Security Foundations Symposium*, 2024.
- [32] 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.

- [33] John Salvatier, Thomas V Wiecki, and Christopher Fonnesbeck. Probabilistic programming in python using pymc3. *PeerJ computer science*, 2:e55, 2016.
- [34] Chung-chieh Shan and Norman Ramsey. Exact bayesian inference by symbolic disintegration. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 130–144, 2017.
- [35] Steffen Smolka, Praveen Kumar, David M Kahn, Nate Foster, Justin Hsu, Dexter Kozen, and Alexandra Silva. Scalable verification of probabilistic networks. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 190–203, 2019.

- [36] Sam Staton, Hongseok Yang, Frank Wood, Chris Heunen, and Ohad Kammar. Semantics for probabilistic programming: higher-order functions, continuous distributions, and soft constraints. In *Proceedings of the 31st Annual ACM/IEEE Symposium on Logic in Computer Science*, pages 525–534, 2016.
- [37] Joseph Tassarotti and Robert Harper. A separation logic for concurrent randomized programs. *Proceedings of the ACM on Programming Languages*, 3(POPL):1–30, 2019.
- [38] Emina Torlak and Rastislav Bodik. Growing solver-aided languages with rosette. In *Proceedings of the 2013 ACM international symposium on New ideas, new paradigms, and reflections on programming & software*, pages 135–152, 2013.

- [39] Katherine Wu, Jules Jacobs, Kevin Batz, and Alexandra Silva. Type-directed discretization of probabilistic programs. *Proceedings of the ACM on Programming Languages*, 10(OOPSLA2):2435–2462, 2026.
- [40] Noam Zilberstein, Derek Dreyer, and Alexandra Silva. Outcome logic: A unifying foundation for correctness and incorrectness reasoning. *Proceedings of the ACM on Programming Languages*, 7(OOPSLA1):522–550, 2023.
- [41] Noam Zilberstein, Alexandra Silva, and Joseph Tassarotti. Probabilistic concurrent reasoning in outcome logic: independence, conditioning, and invariants. *Proceedings of the ACM on Programming Languages*, 10(POPL):235–264, 2026.