Reasoning about Probabilistic Programs

Oregon PL Summer School 2021

Justin Hsu
UW–Madison
Cornell University
Day 1: Introducing Probabilistic Programs
- Motivations and key questions
- Mathematical preliminaries

Day 2: First-Order Programs 1
- Probabilistic While language, monadic semantics
- Weakest pre-expectation calculus

Day 3: First-Order Programs 2
- Probabilistic While language, transformer semantics
- Probabilistic separation logic

Day 4: Higher-Order Programs
- Type system: probability monad
- Type system: probabilistic PCF
Equalities

- \( e = e' \) holds in \( s \) iff all variables \( FV(e, e') \subseteq \text{dom}(s) \), and \( e \) is equal to \( e' \) with probability 1 in \( s \)
Atomic Formulas

Equalities

\[ e = e' \text{ holds in } s \text{ iff all variables } FV(e, e') \subseteq \text{dom}(s), \text{ and } e \text{ is equal to } e' \text{ with probability 1 in } s \]

Distribution laws

\[ \text{[e] holds in } s \text{ iff all variables in } FV(e) \subseteq \text{dom}(s) \]
\[ \text{Unif}_S[e] \text{ holds in } s \text{ iff } FV(e) \subseteq \text{dom}(s), \text{ and } e \text{ is uniformly distributed on } S \text{ (e.g., } S = \mathbb{B} \text{ is fair coin flip)} \]
Example: Distribution Assertions

Suppose $\mu$ has two variables $x, y$, indep. fair coin flips

$\mu([x \mapsto tt, y \mapsto tt]) = 1/4$ \hspace{1cm} $\mu([x \mapsto tt, y \mapsto ff]) = 1/4$

$\mu([x \mapsto ff, y \mapsto tt]) = 1/4$ \hspace{1cm} $\mu([x \mapsto ff, y \mapsto ff]) = 1/4$
Example: Distribution Assertions

Suppose $\mu$ has two variables $x, y$, indep. fair coin flips

\[
\begin{align*}
\mu([x \mapsto tt, y \mapsto tt]) &= 1/4 \\
\mu([x \mapsto ff, y \mapsto tt]) &= 1/4 \\
\mu([x \mapsto tt, y \mapsto ff]) &= 1/4 \\
\mu([x \mapsto ff, y \mapsto ff]) &= 1/4
\end{align*}
\]

Then: $\mu$ satisfies $\text{Unif}_B[x] \ast \text{Unif}_B[y]$. Why?
Example: Distribution Assertions

Suppose \( \mu \) has two variables \( x, y \), indep. fair coin flips

\[
\begin{align*}
\mu([x \mapsto tt, y \mapsto tt]) &= 1/4 & \mu([x \mapsto tt, y \mapsto ff]) &= 1/4 \\
\mu([x \mapsto ff, y \mapsto tt]) &= 1/4 & \mu([x \mapsto ff, y \mapsto ff]) &= 1/4
\end{align*}
\]

Then: \( \mu \) satisfies \( \text{Unif}_B[x] \ast \text{Unif}_B[y] \). Why?

- We can decompose \( \mu = \mu_x \otimes \mu_y \), where:

\[
\begin{align*}
\mu_x([x \mapsto tt]) &\triangleq 1/2 & \mu_x([x \mapsto ff]) &\triangleq 1/2 \\
\mu_y([y \mapsto tt]) &\triangleq 1/2 & \mu_y([y \mapsto ff]) &\triangleq 1/2
\end{align*}
\]

So, \( \mu \sqsubseteq \mu_x \circ \mu_y \)
Example: Distribution Assertions

Suppose $\mu$ has two variables $x, y$, indep. fair coin flips

\[
\begin{align*}
\mu([x \mapsto tt, y \mapsto tt]) &= 1/4 & \mu([x \mapsto tt, y \mapsto ff]) &= 1/4 \\
\mu([x \mapsto ff, y \mapsto tt]) &= 1/4 & \mu([x \mapsto ff, y \mapsto ff]) &= 1/4
\end{align*}
\]

Then: $\mu$ satisfies $\text{Unif}_B[x] \ast \text{Unif}_B[y]$. Why?

- We can decompose $\mu = \mu_x \otimes \mu_y$, where:

\[
\begin{align*}
\mu_x([x \mapsto tt]) &\triangleq 1/2 & \mu_x([x \mapsto ff]) &\triangleq 1/2 \\
\mu_y([y \mapsto tt]) &\triangleq 1/2 & \mu_y([y \mapsto ff]) &\triangleq 1/2
\end{align*}
\]

So, $\mu \sqsubseteq \mu_x \circ \mu_y$
Example: Distribution Assertions

Suppose $\mu$ has two variables $x, y$, indep. fair coin flips

\[
\mu([x \mapsto tt, y \mapsto tt]) = 1/4 \quad \mu([x \mapsto tt, y \mapsto ff]) = 1/4 \\
\mu([x \mapsto ff, y \mapsto tt]) = 1/4 \quad \mu([x \mapsto ff, y \mapsto ff]) = 1/4
\]

Then: $\mu$ satisfies $\text{Unif}_B[x] \ast \text{Unif}_B[y]$. Why?

- We can decompose $\mu = \mu_x \otimes \mu_y$, where:

  \[
  \mu_x([x \mapsto tt]) \triangleq 1/2 \quad \mu_x([x \mapsto ff]) \triangleq 1/2 \\
  \mu_y([y \mapsto tt]) \triangleq 1/2 \quad \mu_y([y \mapsto ff]) \triangleq 1/2
  \]

  So, $\mu \sqsubseteq \mu_x \circ \mu_y$

- Next, $\mu_x \models \text{Unif}_B[x]$ and $\mu_y \models \text{Unif}_B[y]$
Example: Distribution Assertions

Suppose $\mu$ has two variables $x, y$, indep. fair coin flips

\[
\mu([x \mapsto tt, y \mapsto tt]) = \frac{1}{4} \quad \mu([x \mapsto tt, y \mapsto ff]) = \frac{1}{4} \\
\mu([x \mapsto ff, y \mapsto tt]) = \frac{1}{4} \quad \mu([x \mapsto ff, y \mapsto ff]) = \frac{1}{4}
\]

Then: $\mu$ satisfies $\operatorname{Unif}_B[x] \ast \operatorname{Unif}_B[y]$. Why?

- We can decompose $\mu = \mu_x \otimes \mu_y$, where:

\[
\mu_x([x \mapsto tt]) \triangleq \frac{1}{2} \quad \mu_x([x \mapsto ff]) \triangleq \frac{1}{2} \\
\mu_y([y \mapsto tt]) \triangleq \frac{1}{2} \quad \mu_y([y \mapsto ff]) \triangleq \frac{1}{2}
\]

So, $\mu \sqsubseteq \mu_x \circ \mu_y$

- Next, $\mu_x \models \operatorname{Unif}_B[x]$ and $\mu_y \models \operatorname{Unif}_B[y]$
Example: Distribution Assertions

Suppose $\mu$ has two variables $x, y$, indep. fair coin flips

\[
\begin{align*}
\mu([x \mapsto tt, y \mapsto tt]) &= 1/4 \\
\mu([x \mapsto tt, y \mapsto ff]) &= 1/4 \\
\mu([x \mapsto ff, y \mapsto tt]) &= 1/4 \\
\mu([x \mapsto ff, y \mapsto ff]) &= 1/4
\end{align*}
\]

Then: $\mu$ satisfies $\text{Unif}_B[x] \ast \text{Unif}_B[y]$. Why?

- We can decompose $\mu = \mu_x \otimes \mu_y$, where:

\[
\begin{align*}
\mu_x([x \mapsto tt]) &\triangleq 1/2 \\
\mu_x([x \mapsto ff]) &\triangleq 1/2 \\
\mu_y([y \mapsto tt]) &\triangleq 1/2 \\
\mu_y([y \mapsto ff]) &\triangleq 1/2
\end{align*}
\]

So, $\mu \sqsubseteq \mu_x \circ \mu_y$

- Next, $\mu_x \models \text{Unif}_B[x]$ and $\mu_y \models \text{Unif}_B[y]$
Example: Distribution Assertions

Suppose $\mu$ has two variables $x, y$, indep. fair coin flips

$\mu([x \mapsto tt, y \mapsto tt]) = 1/4$ \hspace{1cm} $\mu([x \mapsto tt, y \mapsto ff]) = 1/4$

$\mu([x \mapsto ff, y \mapsto tt]) = 1/4$ \hspace{1cm} $\mu([x \mapsto ff, y \mapsto ff]) = 1/4$

Then: $\mu$ satisfies $\text{Unif}_B[x] \ast \text{Unif}_B[y]$. Why?

- We can decompose $\mu = \mu_x \otimes \mu_y$, where:

$\mu_x([x \mapsto tt]) \triangleq 1/2$ \hspace{1cm} $\mu_x([x \mapsto ff]) \triangleq 1/2$

$\mu_y([y \mapsto tt]) \triangleq 1/2$ \hspace{1cm} $\mu_y([y \mapsto ff]) \triangleq 1/2$

So, $\mu \sqsubseteq \mu_x \circ \mu_y$

- Next, $\mu_x \models \text{Unif}_B[x]$ and $\mu_y \models \text{Unif}_B[y]$

- So by definition, $\mu \models \text{Unif}_B[x] \ast \text{Unif}_B[y]$
Example: Distribution Assertions

Suppose $\mu$ has two variables $x, y$, indep. fair coin flips

\[
\begin{align*}
\mu([x \mapsto tt, y \mapsto tt]) &= 1/4 \\
\mu([x \mapsto ff, y \mapsto tt]) &= 1/4 \\
\mu([x \mapsto tt, y \mapsto ff]) &= 1/4 \\
\mu([x \mapsto ff, y \mapsto ff]) &= 1/4
\end{align*}
\]

Then: $\mu$ satisfies $\text{Unif}_B[x] \ast \text{Unif}_B[y]$. Why?

- We can decompose $\mu = \mu_x \otimes \mu_y$, where:

\[
\begin{align*}
\mu_x([x \mapsto tt]) &\triangleq 1/2 \\
\mu_x([x \mapsto ff]) &\triangleq 1/2 \\
\mu_y([y \mapsto tt]) &\triangleq 1/2 \\
\mu_y([y \mapsto ff]) &\triangleq 1/2
\end{align*}
\]

So, $\mu \sqsubseteq \mu_x \circ \mu_y$

- Next, $\mu_x \models \text{Unif}_B[x]$ and $\mu_y \models \text{Unif}_B[y]$

- So by definition, $\mu \models \text{Unif}_B[x] \ast \text{Unif}_B[y]$
Example Axioms

Equality and distributions
\[ x = y \cdot \text{Unif}_B[x] \equiv \text{Unif}_B[y] \]

Uniformity and products
\[ \text{Unif}_B[x] \uplus \text{Unif}_B[y] \equiv \text{Unif}_B[x, y] \]

Uniformity and exclusive-or (\( \uplus \))
\[ \text{Unif}_B[x] \uplus y = x \uplus y \equiv \text{Unif}_B[z] \uplus y \]
Example Axioms

Equality and distributions

$x = y \land \text{Unif}_B[x] \rightarrow \text{Unif}_B[y]$
Example Axioms

Equality and distributions

$\rightarrow x = y \land \text{Unif}_B[x] \rightarrow \text{Unif}_B[y]$

Uniformity and products

$\rightarrow \text{Unif}_B[x] \ast \text{Unif}_B[y] \rightarrow \text{Unif}_{B \times B}[x, y]$
Example Axioms

Equality and distributions

\[ x = y \land \text{Unif}_B[x] \rightarrow \text{Unif}_B[y] \]

Uniformity and products

\[ \text{Unif}_B[x] \times \text{Unif}_B[y] \rightarrow \text{Unif}_{B \times B}[x, y] \]

Uniformity and exclusive-or (\(\oplus\))

\[ \text{Unif}_B[x] \times [y] \land z = x \oplus y \rightarrow \text{Unif}_B[z] \times [y] \]
A Probabilistic Separation Logic
$P$ and $Q$ from probabilistic BI, $c$ a probabilistic program

$\{P\} \; c \; \{Q\}$
Program Logic Judgments in PSL

$P$ and $Q$ from probabilistic BI, $c$ a probabilistic program

\[
\{ P \} \ c \ \{ Q \}
\]

Validity

For all input states $s \in \text{Distr}(\mathcal{M}(\mathcal{X}))$ satisfying the pre-condition $s \models P$, the output state $[c]s$ satisfies the post-condition $[c]s \models Q$.
Program Logic Judgments in PSL

\( P \) and \( Q \) from probabilistic BI, \( c \) a probabilistic program

\[ \{ P \} \ c \ \{ Q \} \]

Validity

For all input states \( s \in \text{Distr}(M(X)) \) satisfying the pre-condition \( s \models P \), the output state \([c]s\) satisfies the post-condition \([c]s \models Q\).
Perfectly fits the transformer semantics for PWHILE

Under transformer semantics:
- $\mathcal{P}$ describes: a distribution over memories (input)
- $\mathcal{Q}$ describes: a distribution over memories (output)

Under monadic semantics: mismatch!
- $\mathcal{P}$ describes: a distribution over memories
- But input to program: a single memory
How do we prove these judgments?

Validity
For all input states $s \in \text{Distr}(\mathcal{M}(\mathcal{X}))$ satisfying the pre-condition $s \models P$, the output state $\llbracket c \rrbracket s$ satisfies the post-condition $\llbracket c \rrbracket s \models Q$. 
How do we prove these judgments?

Validity
For all input states $s \in \text{Distr}(\mathcal{M}(\mathcal{X}))$ satisfying the pre-condition $s \models P$, the output state $\llbracket c \rrbracket s$ satisfies the post-condition $\llbracket c \rrbracket s \models Q$.

Proving validity directly is difficult
- Must unfold definition of $\llbracket c \rrbracket$ as a function
- Then prove property of function by working with definition
How do we prove these judgments?

Validity
For all input states $s \in \text{Distr}(\mathcal{M}(X))$ satisfying the pre-condition $s \models P$, the output state $\llbracket c \rrbracket s$ satisfies the post-condition $\llbracket c \rrbracket s \models Q$.

Proving validity directly is difficult
- Must unfold definition of $\llbracket c \rrbracket$ as a function
- Then prove property of function by working with definition

Things that would make proving judgments easier:
- Compositionality: prove property of bigger program by combining proofs of properties of sub-programs
- Avoid unfolding definition of program semantics
Solution: define a set of proof rules (a proof system)

Each proof rule looks like:

\[
\left\{ P_1 \right\} c_1 \left\{ Q_1 \right\} \quad \cdots \quad \left\{ P_n \right\} c_n \left\{ Q_n \right\} \quad \text{Rule Name}
\]

\[
\left\{ P \right\} c \left\{ Q \right\}
\]
Solution: define a set of proof rules (a proof system)

Each proof rule look like:

\[
\frac{\{P_1\} \ c_1 \ \{Q_1\} \quad \cdots \quad \{P_n\} \ c_n \ \{Q_n\}}{\{P\} \ c \ \{Q\}}
\]

Proof rules mean:

- To prove \(\{P\} \ c \ \{Q\}\)
- We just have to prove \(\{P_1\} \ c_1 \ \{Q_1\}, \ldots, \{P_n\} \ c_n \ \{Q_n\}\)
Solution: define a set of proof rules (a proof system)

Each proof rule look like:

\[
\begin{array}{c}
\{P_1\} c_1 \{Q_1\} & \cdots & \{P_n\} c_n \{Q_n\} \\
\hline
\{P\} & c & \{Q\}
\end{array}
\]

Proof rules mean:

- To prove \(\{P\} c \{Q\}\)
- We just have to prove \(\{P_1\} c_1 \{Q_1\}, \ldots, \{P_n\} c_n \{Q_n\}\)

Why do proof rules help?

- Programs \(c_1, \ldots, c_n\) are smaller/simpler than \(c\)
- If \(c\) can’t be broken down, no premises (\(n = 0\))
The Proof System of PSL

Basic Rules
Basic Proof Rules in PSL: Assignment

Assignment Rule

\[
\begin{align*}
x \notin \text{FV}(e) \\
\{ \top \} \ x & \leftarrow e \ \{ x = e \} \\
\end{align*}
\]

How to read this rule?
From any initial distribution, running \( x \leftarrow e \) will lead to a distribution where \( x \) equals \( e \) with probability 1 (assuming \( x \) doesn't appear in \( e \)).
Basic Proof Rules in PSL: Assignment

Assignment Rule

\[
x \notin FV(e) \quad \text{Assn} \\
\{ \top \} \ x \leftarrow e \ \{ x = e \}
\]

How to read this rule?
From any initial distribution, running \( x \leftarrow e \) will lead to a distribution where \( x \) equals \( e \) with probability 1 (assuming \( x \) doesn’t appear in \( e \)).
Basic Proof Rules in PSL: Sampling

Sampling Rule

$$\{\top\} \ x \ \uparrow \ \text{Flip} \ \{\text{Unif}_B[x]\}$$

How to read this rule?

From any initial distribution, running \(x \ \uparrow \ \text{Flip} \ \{\text{Unif}_B[x]\}\) will lead to a distribution where \(x\) is a uniformly distributed Boolean.
Sampling Rule

\[
\{ \top \} \ x \leftarrow \$ \ \text{Flip} \ \{ \text{Unif}_B[x] \} \quad \text{SAMP}
\]

How to read this rule?
From any initial distribution, running \( x \leftarrow \$ \ \text{Flip} \) will lead to a distribution where \( x \) is a uniformly distributed Boolean.
Sequencing Rule

\[
\begin{align*}
\{P\} & \quad \text{c}_1 \quad \{Q\} \\
\{Q\} & \quad \text{c}_2 \quad \{R\} \\
\{P\} & \quad \text{c}_1 \; ; \; \text{c}_2 \quad \{R\} \\
\hline
\text{SEQ}
\end{align*}
\]
Basic Proof Rules in PSL: Sequencing

Sequencing Rule

\[
\frac{\{P\} c_1 \{Q\} \quad \{Q\} c_2 \{R\}}{\{P\} c_1 ; c_2 \{R\}} \quad \text{SEQ}
\]

How to read this rule?

- If: from any distribution satisfying \( P \), running \( c_1 \) leads to a distribution satisfying \( R \)
- If: from any distribution satisfying \( R \), running \( c_2 \) leads to a distribution satisfying \( Q \)
- Then: from any distribution satisfying \( P \), running \( c_1 ; c_2 \) leads to a distribution satisfying \( Q \)
The Proof System of PSL

Conditional Rule
Conditional Rule: first try

Does this rule work?

\[
\left\{ \begin{array}{l}
e = tt \land P \\
\end{array} \right. \quad c \quad \left\{ Q \right\}
\]

\[
\left\{ \begin{array}{l}
e = ff \land P \\
\end{array} \right. \quad c' \quad \left\{ Q \right\}
\]

\[
\left\{ P \right\} \text{ if } e \text{ then } c \text{ else } c' \quad \left\{ Q \right\}
\]
Rule COND? is not sound!

Take \( P \) to be \( \text{Unif}\left[B\right] \) and \( Q \) to be \( \text{\{e} = t\cdot \text{Unif}\left[B\right] \} \text{c} \{\text{\{e} = \cdot \text{Unif}\left[B\right] \} \} \text{c} \{\text{\{e}} \}

if \( e \) then \( c \) else \( c \) \text{c} {\text{\{e}}

Premises are valid...

There is no distribution satisfying \( e = t\cdot \text{Unif}\left[B\right] \) or \( e = \cdot \text{Unif}\left[B\right] \), so pre-conditions are \( \text{\{e} \) and the premises are trivially valid.

But the conclusion is not!

It is not the case that if \( \text{Unif}\left[B\right] \) in the input distribution, then running if \( e \) then \( c \) else \( c \) will lead to an impossible output distribution!
Rule $\text{COND?}$ is not sound!

Take $P$ to be $\text{Unif}_B[e]$ and $Q$ to be $\bot$:

$$
\frac{
\{e = tt \land \text{Unif}_B[e]\} \quad c \quad \{\bot\}
\quad \{e = ff \land \text{Unif}_B[e]\} \quad c' \quad \{\bot\}
}{
\{\text{Unif}_B[e]\} \quad \text{if } e \text{ then } c \text{ else } c' \quad \{\bot\}
$$
Rule COND? is not sound!

Take $P$ to be $\text{Unif}_B[e]$ and $Q$ to be $\bot$:

\[
\begin{align*}
\{ e = tt \land \text{Unif}_B[e] \} & \quad c \quad \{ \bot \} \quad \{ e = ff \land \text{Unif}_B[e] \} & \quad c' \quad \{ \bot \} \\
\text{COND?}
\end{align*}
\]

Premises are valid...
There is no distribution satisfying $e = tt \land \text{Unif}_B[e]$ or $e = ff \land \text{Unif}_B[e]$, so pre-conditions are $\bot$ and the premises are trivially valid.
Rule \textit{COND?} is not sound!

Take $P$ to be $\text{Unif}_B[e]$ and $Q$ to be $\bot$:

$$\begin{align*}
\{ e = tt \land \text{Unif}_B[e] \} & \quad c \quad \bot \\
\{ e = ff \land \text{Unif}_B[e] \} & \quad c' \quad \bot \\
\{ \text{Unif}_B[e] \} & \quad \text{if } e \text{ then } c \text{ else } c' \quad \bot
\end{align*}$$

Premises are valid...
There is no distribution satisfying $e = tt \land \text{Unif}_B[e]$ or $e = ff \land \text{Unif}_B[e]$, so pre-conditions are $\bot$ and the premises are trivially valid.

But the conclusion is not!
It is not the case that if $\text{Unif}_B[e]$ in the input distribution, then running if $e$ then $c$ else $c'$ will lead to an impossible output distribution!
What went wrong?

The broken rule

\[
\frac{\{e = tt \land P\} \ c \ \{Q\}}{\{P\} \ \text{if } e \ \text{then } c \ \text{else } c'} \ \{Q\} \quad \text{COND?}
\]

The problem: conditioning

We assume:

\(P\) holds in input distribution 

Inputs to branches:

\(P\) conditioned on \(e = tt\) and \(e = ff\)!

But:

\(P\) might not hold on conditional distributions!
What went wrong?

The broken rule

\[
\begin{align*}
\{ e = tt \land P \} & \ c \ \{ Q \} \quad \{ e = ff \land P \} & \ c' \ \{ Q \} \\
\{ P \} & \text{ if } e \text{ then } c \text{ else } c' \ \{ Q \} \\
\end{align*}
\]

The problem: conditioning

- We assume: \( P \) holds in input distribution \( \mu \)
- Inputs to branches: \( \mu \) conditioned on \( e = tt \) and \( e = ff \)
- But: \( P \) might not hold on conditional distributions!
Conditional Rule: second try

Does this rule work?

$$\frac{\{ e = tt * P \} \ c \ \{ Q \}}{\{ e = ff * P \} \ c' \ \{ Q \}} \ \text{if } e \ \text{then } c \ \text{else } c' \ \{ Q \}$$
Conditional Rule: second try

Does this rule work?

\[
\begin{align*}
\{ e = tt \ast P \} & \quad c \quad \{ Q \} & \quad \{ e = ff \ast P \} & \quad c' \quad \{ Q \} \\
\{ [e] \ast P \} & \quad \text{if } e \text{ then } c \text{ else } c' \quad \{ Q \}
\end{align*}
\]

Previous counterexample fails

If we take \( P \) to be \( \text{Unif}_B[e] \), then \( [e] \ast \text{Unif}_B[e] \) is false, and the conclusion is trivially valid.
But rule COND?? is still not sound!

Premises are valid... In the output of each branch, x and e are independent since e is deterministic. But the conclusion is not! In the output of the conditional, x and e are clearly not always independent: they are equal, and they might be randomized!
But rule \textsc{Cond}?? is still not sound!

Consider this proof

\[
\begin{align*}
    &\{e = tt \times \top\} \ x \leftarrow \ e \ \{[x] \times [e]\} & \{e = ff \times P\} \ x \leftarrow \ e \ \{[x] \times [e]\} \\
    &\{[e] \times \top\} \text{ if } e \text{ then } x \leftarrow e \text{ else } x \leftarrow e \ \{[x] \times [e]\}
\end{align*}
\]
But rule `COND??` is still not sound!

Consider this proof

\[
\begin{align*}
\{ e = tt \land T \} & \quad x \leftarrow e \{ [x] \land [e] \} \\
\{ e = ff \land P \} & \quad x \leftarrow e \{ [x] \land [e] \}
\end{align*}
\]

```
{x \land T} \text{ if } e \text{ then } x \leftarrow e \text{ else } x \leftarrow e \{ [x] \land [e] \}
```

Premises are valid...

In the output of each branch, \( x \) and \( e \) are independent since \( e \) is deterministic.
But rule $\text{COND}??$ is still not sound!

Consider this proof

$$
\begin{align*}
\{ e = \text{tt} \land \top \} & \quad x \leftarrow e \{ [x] \land [e] \} \\
\{ e = \text{ff} \land P \} & \quad x \leftarrow e \{ [x] \land [e] \}
\end{align*}
$$

$\{ [e] \land \top \}$ if $e$ then $x \leftarrow e$ else $x \leftarrow e \{ [x] \land [e] \}$

Premises are valid...
In the output of each branch, $x$ and $e$ are independent since $e$ is deterministic.

But the conclusion is not!
In the output of the conditional, $x$ and $e$ are clearly not always independent: they are equal, and they might be randomized!
What went wrong?

The broken rule

\[
\begin{align*}
\{ e = tt \star P \} & \quad c \quad \{ Q \} \\
\{ e = ff \star P \} & \quad c' \quad \{ Q \} \\
\{ [e] \star P \} & \quad \text{if } e \text{ then } c \text{ else } c' \quad \{ Q \}
\end{align*}
\]
What went wrong?

The broken rule

\[
\begin{align*}
&\{ e = tt \ast P \} \ A \ \{ Q \} & \{ e = ff \ast P \} \ A' \ \{ Q \} \\
&\{ [e] \ast P \} \text{ if } e \text{ then } A \text{ else } A' \ \{ Q \} \quad \text{COND??}
\end{align*}
\]

The problem: mixing

- Suppose: \( Q \) holds in the outputs of both branches
- The output of the conditional is a convex combination of the branch outputs
- But: \( Q \) might not hold in the convex combination!
Conditional Rule in PSL

Fixed rule

\[
\begin{align*}
\{ e = tt \ast P \} & \quad c \quad \{ Q \} \\
\{ e = ff \ast P \} & \quad c' \quad \{ Q \}
\end{align*}
\]

Q is closed under mixtures (CM)  

\[
\{ [e] \ast P \} \text{ if } e \text{ then } c \text{ else } c' \quad \{ Q \}
\]  

\text{COND}
Conditional Rule in PSL

**Fixed rule**

\[
\begin{align*}
\{ e = tt \cdot P \} & \quad c \quad \{ Q \} \\
\{ e = ff \cdot P \} & \quad c' \quad \{ Q \}
\end{align*}
\]

\[
Q \text{ is closed under mixtures (CM)} \quad \text{COND}
\]

\[
\{ [e] \cdot P \} \quad \text{if } e \text{ then } c \text{ else } c' \quad \{ Q \}
\]

**Pre-conditions**

- Inputs to branches derived from **conditioning** on \( e \)
- Independence ensures that \( P \) holds after conditioning
Conditional Rule in PSL

Fixed rule

\[
\begin{align*}
\{e = tt \ast P\} & \quad c \quad \{Q\} \\
\{e = ff \ast P\} & \quad c' \quad \{Q\}
\end{align*}
\]

Q is closed under mixtures (CM) \[\text{COND}\]

\[
\begin{align*}
\{[e] \ast P\} & \quad \text{if } e \text{ then } c \text{ else } c' \quad \{Q\}
\end{align*}
\]

Pre-conditions
- Inputs to branches derived from \textit{conditioning} on \(e\)
- Independence ensures that \(P\) holds after conditioning

Post-conditions
- Not all post-conditions \(Q\) can be soundly combined
- “Closed under mixtures” needed for soundness
CM properties: Closed under Mixtures

An assertion $Q$ is CM if it satisfies:

If $\mu_1 \models Q$ and $\mu_2 \models Q$, then $\mu_1 \oplus_p \mu_2 \models Q$ for any $p \in [0, 1]$. 

Examples of CM assertions

$I_x = e^{I_{\text{Unif} B[x]}}$

Examples of non-CM assertions

$I[x] \cup [y]$

$I_x = 1 \rightarrow x = 2$
An assertion $Q$ is CM if it satisfies:
If $\mu_1 \models Q$ and $\mu_2 \models Q$, then $\mu_1 \oplus_p \mu_2 \models Q$ for any $p \in [0, 1]$.

Examples of CM assertions

- $x = e$
- $\text{Unif}_B[x]$
CM properties: Closed under Mixtures

An assertion \( Q \) is CM if it satisfies:

If \( \mu_1 \models Q \) and \( \mu_2 \models Q \), then \( \mu_1 \oplus_p \mu_2 \models Q \) for any \( p \in [0, 1] \).

Examples of CM assertions

- \( x = e \)
- \( \text{Unif}_B[x] \)

Examples of non-CM assertions

- \( [x] \ast [y] \)
- \( x = 1 \lor x = 2 \)
Example: using the conditional rule

Consider the program:

\[
\text{if } x \text{ then } z \leftarrow \neg y \text{ else } z \leftarrow y
\]

If \( x \) is true, negate \( y \) and store in \( z \). Otherwise store \( y \) into \( z \).
Example: using the conditional rule

Consider the program:

\[
\text{if } x \text{ then } z \leftarrow \neg y \text{ else } z \leftarrow y
\]

If \( x \) is true, negate \( y \) and store in \( z \). Otherwise store \( y \) into \( z \).

Using the conditional rule:

\[
\begin{align*}
\{ x = tt \} & \times \text{Unif}_B[y] \} \ z \leftarrow \neg y \ \{ \text{Unif}_B[z] \} \\
\{ x = ff \} & \times \text{Unif}_B[y] \} \ z \leftarrow y \ \{ \text{Unif}_B[z] \} \\
\text{Unif}_B[z] \text{ is closed under mixtures (CM)} & \quad \text{COND}
\end{align*}
\]

\[
\{ [x] \times \text{Unif}_B[y] \} \text{ if } x \text{ then } z \leftarrow \neg y \text{ else } z \leftarrow y \ \{ \text{Unif}_B[z] \}
\]
The Proof System of PSL

Frame Rule
The Frame Rule in SL

Properties about unmodified heaps are preserved

\[
\{P\} \ c \ \{Q\} \quad c \ \text{doesn't modify} \quad FV(R) \\
\{P \ast R\} \ c \ \{Q \ast R\}
\]

So-called "local reasoning" in SL

I Only need to reason about part of heap used by c

Note: doesn't hold if \(\ast\) replaced by \(\cdot\), due to aliasing!
The Frame Rule in SL

Properties about unmodified heaps are preserved

\[
\begin{align*}
\{ P \} \ c \ \{ Q \} & \quad c \ \text{doesn't modify} \ FV(R) \\
\{ P \ast R \} \ c \ \{ Q \ast R \} & \quad \text{FRAME}
\end{align*}
\]

So-called “local reasoning” in SL

- Only need to reason about part of heap used by \( c \)
- Note: doesn’t hold if \( \ast \) replaced by \( \land \), due to aliasing!
Why is the Frame rule important?

In SL: simplify reasoning

- Program may only modify a small part of the heap
- Rest of heap may be complicated (linked lists, trees, etc.)
- Automatically preserve any assertion about rest of heap, as long as rest of heap is separate from what $c$ touches

In PSL: preserve independence

- Assume: in input, variable $x$ is independent of what $c$ uses
- Conclude: in output, $x$ is independent of what $c$ touches
Why is the Frame rule important?

In SL: simplify reasoning

- Program $c$ may only modify a small part of the heap
- Rest of heap may be complicated (linked lists, trees, etc.)
- Automatically preserve any assertion about rest of heap, as long as rest of heap is separate from what $c$ touches
Why is the Frame rule important?

In SL: simplify reasoning
- Program $c$ may only modify a small part of the heap
- Rest of heap may be complicated (linked lists, trees, etc.)
- Automatically preserve any assertion about rest of heap, as long as rest of heap is separate from what $c$ touches

In PSL: preserve independence
- Assume: in input, variable $x$ is independent of what $c$ uses
- Conclude: in output, $x$ is independent of what $c$ touches
The Frame Rule in PSL

The rule

\[
\begin{align*}
\{ P \} c \{ Q \} & \quad FV(R) \cap MV(c) = \emptyset \\
\models P \to [RV(c)] & \quad FV(Q) \subseteq RV(c) \cup WV(c) \\
\{ P \ast R \} c \{ Q \ast R \} & \quad \text{FRAME}
\end{align*}
\]

Side conditions
The Frame Rule in PSL

The rule

\[
\left\{ P \right\} c \left\{ Q \right\} \quad FV(R) \cap MV(c) = \emptyset \\
\models P \rightarrow [RV(c)] \quad FV(Q) \subseteq RV(c) \cup WV(c) \quad \text{FRAME} \\
\left\{ P \ast R \right\} c \left\{ Q \ast R \right\}
\]

Side conditions

1. Variables in \( R \) are not modified
The Frame Rule in PSL

The rule

\[
\begin{align*}
\{ P \} c \{ Q \} & \quad FV(R) \cap MV(c) = \emptyset \\
\models P \rightarrow [RV(c)] & \quad FV(Q) \subseteq RV(c) \cup WV(c) \\
\{ P \ast R \} c \{ Q \ast R \} & \quad \text{FRAME}
\end{align*}
\]

Side conditions

1. Variables in \( R \) are not modified
2. \( P \) describes all variables that might be read
The Frame Rule in PSL

The rule

\[
\begin{align*}
\{P\} & \ c \ \{Q\} \\
\models P \rightarrow [RV(c)] & \quad FV(R) \cap MV(c) = \emptyset \\
FV(Q) \subseteq RV(c) \cup WV(c) & \quad \text{FRAME} \\
\{P \star R\} & \ c \ \{Q \star R\}
\end{align*}
\]

Side conditions

1. Variables in \(R\) are not modified
2. \(P\) describes all variables that might be read
3. Everything in \(Q\) is freshly written, or in \(P\)
The Frame Rule in PSL

The rule

\[
\begin{align*}
\{P\} \ c \ \{Q\} & \quad FV(R) \cap MV(c) = \emptyset \\
\models P \to [RV(c)] & \quad FV(Q) \subseteq RV(c) \cup WV(c) \\
\{P \ast R\} \ c \ \{Q \ast R\} & \quad \text{FRAME}
\end{align*}
\]

Side conditions

1. Variables in \(R\) are not modified
2. \(P\) describes all variables that might be read
3. Everything in \(Q\) is freshly written, or in \(P\)

Variables in the \(Q\) were independent of \(R\), or are newly independent of \(R\)
Example: Deriving a Better Sampling Rule

Original sampling rule:

\[
\{T\} \ x \ \leftarrow^\$ \ \text{Flip} \ \{\text{Unif}_B[x]\}\ \\
\text{SAMP}
\]

Frame rule:

\[
\begin{align*}
\{P\} \ c \ \{Q\} & \quad FV(R) \cap MV(c) = \emptyset \\
\models P \rightarrow [RV(c)] & \quad FV(Q) \subseteq RV(c) \cup WV(c) \\
\{P \ast R\} \ c \ \{Q \ast R\} & \quad \text{FRAME}
\end{align*}
\]
**Example: Deriving a Better Sampling Rule**

**Original sampling rule:**

\[
\{\top\} \ x \triangleleft_{\text{Samp}} \ \text{Flip} \ \{\text{Unif}_B[x]\}
\]

**Frame rule:**

\[
\frac{
\{P\} \ c \ \{Q\} \quad FV(R) \cap MV(c) = \emptyset \quad FV(Q) \subseteq RV(c) \cup WV(c)
}{
\models P \to [RV(c)] \quad \{P \star R\} \ c \ \{Q \star R\}
}
\]

**Can derive:**

\[
\frac{x \notin FV(R) \quad \{R\} \ x \triangleleft_{\text{Samp}} \ \text{Flip} \ \{\text{Unif}_B[x] \star R\}
}{
}
\]
Example: Deriving a Better Sampling Rule

Original sampling rule:

\[
\{ \top \} \ x \xleftarrow{\$} \text{Flip} \ \{ \text{Unif}_B[x] \} \quad \text{SAMP}
\]

Frame rule:

\[
\begin{align*}
\{ P \} \ c \ \{ Q \} & \quad FV(R) \cap MV(c) = \emptyset \\
\models P \rightarrow [RV(c)] & \quad FV(Q) \subseteq RV(c) \cup WV(c) \\
\{ P \ast R \} \ c \ \{ Q \ast R \} & \quad \text{FRAME}
\end{align*}
\]

Can derive:

\[
\begin{align*}
x \notin FV(R) & \\
\{ R \} \ x \xleftarrow{\$} \text{Flip} \ \{ \text{Unif}_B[x] \ast R \} & \quad \text{SAMP}^*
\end{align*}
\]

Intuitively: fresh random sample is independent of everything
A Probabilistic Separation Logic

Soundness Theorem
Proof rules can only show valid judgments

**Theorem**

If \( \{ P \} c \{ Q \} \) is derivable via the proof rules, then \( \{ P \} c \{ Q \} \) is a valid judgment: for all initial distributions \( \mu \), if \( \mu \models P \) then \( \llbracket c \rrbracket \mu \models Q \).

**Key property for soundness: restriction**

Let \( P \) be any formula of probabilistic BI, and suppose that \( s \models P \). Then there exists \( s' \subseteq s \) such that \( s' \models P \) and \( \text{dom}(s') = \text{dom}(s) \cap \text{FV}(P) \).

**Intuition**

- The only variables that “matter” for \( P \) are \( \text{FV}(P) \)
- Tricky for implications; proof “glues” distributions
Verifying an Example
One-Time-Pad (OTP)

Possibly the simplest encryption scheme

- Input: a message $m \in \mathbb{B}$
- Output: a ciphertext $c \in \mathbb{B}$
- Idea: encrypt by taking xor with a uniformly random key $k$
One-Time-Pad (OTP)

Possibly the simplest encryption scheme

- Input: a message $m \in \mathbb{B}$
- Output: a ciphertext $c \in \mathbb{B}$
- Idea: encrypt by taking xor with a uniformly random key $k$

The encoding program:

$$\begin{align*}
    k & \leftarrow \text{Flip}; \\
    c & \leftarrow k \oplus m
\end{align*}$$
How to Formalize Security?

**Method 1:** Uniformity

- Show that $c$ is uniformly distributed
- Always the same, no matter what the message $m$ is

**Method 2:** Input-output independence

- Assume that $m$ is drawn from some (unknown) distribution
- Show that $c$ and $m$ are independent
Method 1: Uniformity

- Show that $c$ is uniformly distributed
- Always the same, no matter what the message $m$ is
How to Formalize Security?

**Method 1: Uniformity**
- Show that $c$ is uniformly distributed
- Always the same, no matter what the message $m$ is

**Method 2: Input-output independence**
- Assume that $m$ is drawn from some (unknown) distribution
- Show that $c$ and $m$ are independent
$k \leftarrow_{\$} \text{Flip}_\phi$

c \leftarrow k \oplus m
Proving Input-Output Independence for OTP in PSL

\[
\{[m]\} \\
 k \leftarrow \text{Flip}_\varphi \\
 c \leftarrow k \oplus m
\]
Proving Input-Output Independence for OTP in PSL

\[
\{[m]\}
\]

\(k \xleftarrow{\$} \text{Flip}_Ω\)

\[
\{[m] * \text{Unif}_B[k]\}
\]

\(c \leftarrow k \oplus m\)

assumption

\([\text{SAMP}^*]\)
Proving Input-Output Independence for OTP in PSL

\{[m]\} \quad \text{assumption}

\begin{align*}
k & \leftarrow \text{Flip}_\odot \\
\{[m] \ast \text{Unif}_B[k]\} & \quad \text{[SAMP*]} \\
c & \leftarrow k \oplus m \\
\{[m] \ast \text{Unif}_B[k] \land c = k \oplus m\} & \quad \text{[ASSN*]} \\
\end{align*}
Proving Input-Output Independence for OTP in PSL

\[
\{[m]\}\quad \text{assumption}
\]

\[k \leftarrow \text{Flip}_9\]

\[
\{[m] \ast \text{Unif}_B[k]\}\quad \text{[SAMP*]}
\]

\[c \leftarrow k \oplus m\]

\[
\{[m] \ast \text{Unif}_B[k] \land c = k \oplus m\}\quad \text{[ASSN*]}
\]

\[
\{[m] \ast \text{Unif}_B[c]\}\quad \text{XOR axiom}
\]
PSL: references and further reading

The original paper on probabilistic semantics

Unifying survey on Bunched Implications

A Probabilistic Separation Logic (POPL20)
- Extensions to PSL: deterministic variables, loops, etc.
- Many examples from cryptography, security of ORAM

A Bunched Logic for Conditional Independence (LICS21)
- A BI-style logic called DIBI for conditional independence
- A separation logic (CPSL) based on DIBI
Reasoning about Probabilistic Programs

Higher-Order Languages
So far: reasoning about PWHILE programs

First part
- Monadic semantics: $\{c\} : \mathcal{M} \rightarrow \text{Distr}(\mathcal{M})$
- Verification method: weakest pre-expectations ($wpe$)

Second part
- Transformer semantics: $[c] : \text{Distr}(\mathcal{M}) \rightarrow \text{Distr}(\mathcal{M})$
- Verification method: probabilistic separation logic (PSL)
Today: probabilistic higher-order programs

What’s missing from PWHILE?
- First-order programs only
- That is: can’t pass functions to other functions

This is OPLSS: where are the functions?
- How about probabilistic functional languages?
- What do the type systems look like?
With a Probability Monad:

A Simple Functional Language
Operations on distributions: unit

The simplest possible distribution

Dirac distribution: Probability 1 of producing a particular element, and probability 0 of producing anything else.

Distribution unit
Let \( a \in A \). Then \( \text{unit}(a) \in \text{Distr}(A) \) is defined to be:

\[
\text{unit}(a)(x) = \begin{cases} 
1 & : x = a \\
0 & : \text{otherwise}
\end{cases}
\]

Why “unit”? The unit ("return") of the distribution monad.
Operations on distributions: bind

Sequence two sampling instructions together
Draw a sample $x$, then draw a sample from a distribution $f(x)$ depending on $x$. Transformation map $f$ is randomized: function $A \to \text{Distr}(B)$.

Distribution bind
Let $\mu \in \text{Distr}(A)$ and $f : A \to \text{Distr}(B)$. Then $\text{bind}(\mu, f) \in \text{Distr}(B)$ is defined to be:

$$\text{bind}(\mu, f)(b) \triangleq \sum_{a \in A} \mu(a) \cdot f(a)(b)$$
Language: probabilistic monadic lambda calculus

Language grammar: core

\[ E \ni e := x \in X \mid \lambda X. \ E \mid E \ E \mid \text{fix } X. \lambda X. \ E \quad \text{(lambda calc.)} \]
Language: probabilistic monadic lambda calculus

Language grammar: core

\[ E \ni e := x \in X \mid \lambda X. E \mid E \ E \mid \text{fix } X. \lambda X. E \quad \text{ (lambda calc.)} \]

Language grammar: base types

\[ E \ni e ::= \cdots \mid b \in B \mid \text{if } E \text{ then } E \text{ else } E \quad \text{ (booleans)} \]

\[ \mid n \in \mathbb{N} \mid \text{add}(E, E) \quad \text{ (numbers)} \]
Language: probabilistic monadic lambda calculus

Language grammar: core

\[ \mathcal{E} \ni e \ ::= \ x \in \mathcal{X} \mid \lambda \mathcal{X}. \mathcal{E} \mid \mathcal{E} \mathcal{E} \mid \text{fix } \mathcal{X}. \lambda \mathcal{X}. \mathcal{E} \quad \text{(lambda calc.)} \]

Language grammar: base types

\[ \mathcal{E} \ni e \ ::= \cdots \mid b \in \mathbb{B} \mid \text{if } \mathcal{E} \text{ then } \mathcal{E} \text{ else } \mathcal{E} \quad \text{(booleans)} \]
\[ \mid n \in \mathbb{N} \mid \text{add}(\mathcal{E}, \mathcal{E}) \quad \text{(numbers)} \]

Language grammar: probabilistic part

\[ \mathcal{E} \ni e \ ::= \cdots \mid \text{Flip} \mid \text{Roll} \quad \text{(distributions)} \]
\[ \mid \text{return}(\mathcal{E}) \quad \text{(unit)} \]
\[ \mid \text{sample } \mathcal{X} = \mathcal{E} \text{ in } \mathcal{E} \quad \text{(bind)} \]
Example programs

**Sum of two dice rolls**

```plaintext
sample x = Roll in
sample y = Roll in
return(add(x, y))
```
Example programs

Sum of two dice rolls

sample $x = \text{Roll}$ in
sample $y = \text{Roll}$ in
return($\text{add}(x, y)$)

Geometric distribution

$$(\text{fix } geo. \lambda n. \quad \text{sample } stop = \text{Flip} \text{ in} \quad \text{if } stop \text{ then return}(n) \text{ else } geo \text{ add}(n, 1)) \; 0$$
Operational semantics: setup

One-step reduction
The one-step relation $\rightarrow : \mathcal{CE} \rightarrow \text{SDistr}(\mathcal{CE})$ maps closed expressions to sub-distributions on closed expressions. Read:

$$e \rightarrow \mu$$

as “$e$ steps to sub-distribution $\mu$ on expressions in one step”.
Operational semantics: setup

One-step reduction
The one-step relation \( \rightarrow : \mathcal{CE} \rightarrow \text{SDistr}(\mathcal{CE}) \) maps closed expressions to sub-distributions on closed expressions. Read:

\[
e \rightarrow \mu
\]
as “\( e \) steps to sub-distribution \( \mu \) on expressions in one step”.

Multi-step reduction
For every \( n \in \mathbb{N} \), the multi-step relation \( \Rightarrow_n : \mathcal{CE} \rightarrow \text{SDistr}(\mathcal{CE}) \) maps closed expressions to sub-distributions on closed expressions. Read:

\[
e \Rightarrow_n \mu
\]
as “\( e \) steps to sub-distribution \( \mu \) on values in exactly \( n \) steps”.
Operational semantics: non-probabilistic part

Standard call-by-value semantics

\[(\lambda x. \, e) \, v \rightarrow \text{unit}(e[v/x])\]
\[\text{if } \text{tt then } e \text{ else } e' \rightarrow \text{unit}(e)\]
\[\text{if } \text{ff then } e \text{ else } e' \rightarrow \text{unit}(e')\]
\[(\text{fix } f. \, \lambda x. \, e) \, v \rightarrow \text{unit}(e[(\text{fix } f. \, \lambda x. \, e)/f][v/x])\]
\[\text{add}(n, n') \rightarrow \text{unit}(n + n')\]
\[\ldots\]
Notation
We write \( \{v_1 : p_1, \ldots, v_n : p_n\} \) or \( \{v_i : p_i\}_{i \in I} \) for the distribution that produces \( v_i \) with probability \( p_i \).

Step to distributions on values

- **Flip** → \( \{tt : 1/2, ff : 1/2\} \)
- **Roll** → \( \{1 : 1/6, \ldots, 6 : 1/6\} \)
Operational semantics: unit and bind

**Unit**

\[ e \rightarrow e' \]

\[ \text{return}(e) \rightarrow \text{return}(e') \]

**Bind**

\[ e \rightarrow \{v_i : p_i\}_{i \in I} \]

\[ \text{sample} \ x = e \ \text{in} \ e' \rightarrow \sum_{i \in I} p_i \cdot e'[v_i/x] \]
A Simple Probabilistic Type System
Types in our language

\[ \mathcal{T} \ni \tau := \mathbb{B} \mid \mathbb{N} \]
\[ \mid \mathcal{T} \rightarrow \mathcal{T} \]
\[ \mid \bigcirc \mathcal{T} \]

(base types)
(functions)
(distributions)
Typing judgment basics

The main judgment
Let \( e \in \mathcal{E} \), \( \tau \in \mathcal{T} \), and \( \Gamma \) be a finite list of bindings \( x_1 : \tau_1, \ldots, x_n : \tau_n \). Then the typing judgment is:

\[
\Gamma \vdash e : \tau
\]

Reading
If we substitute closed values \( v_1, \ldots, v_n \) for variables \( x_1, \ldots, x_n \) in \( e \), then the result either reduces to \( \text{unit}(v) \) if \( \tau \) is non-probabilistic, or reduces to a sub-distribution over closed values if \( \tau \) is probabilistic (of the form \( \bigcirc \tau \)).
Typing rules: variables and functions

Exactly the same as in lambda calculus

\[
\begin{align*}
\text{VAR} & \quad \frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau} \\
\text{LAM} & \quad \frac{\Gamma, x : \tau \vdash e : \tau'}{\Gamma \vdash \lambda x. e : \tau \rightarrow \tau'} \\
\text{APP} & \quad \frac{\Gamma \vdash e : \tau \rightarrow \tau' \quad \Gamma \vdash e' : \tau}{\Gamma \vdash e \ e' : \tau'} \\
\text{FIX} & \quad \frac{\Gamma, f : \tau \rightarrow \tau' \vdash \lambda x. e : \tau \rightarrow \tau'}{\Gamma \vdash \text{fix } f \ . \ \lambda x. e : \tau \rightarrow \tau'}
\end{align*}
\]
Typing rules: booleans and integers

Hopefully not too surprising

\[ b = tt, ff \quad \frac{}{\Gamma \vdash b : \mathbb{B}} \text{ BOOL} \]

\[ n \in \mathbb{N} \quad \frac{}{\Gamma \vdash n : \mathbb{N}} \text{ NAT} \]

\[ \Gamma \vdash e : \mathbb{N} \quad \Gamma \vdash e' : \mathbb{N} \quad \frac{}{\Gamma \vdash \text{add}(e, e') : \mathbb{N}} \text{ ADD} \]
Typing rules: primitive distributions

Assign distribution types

\[ \Gamma \vdash \text{Flip} : \mathcal{B} \]  \hspace{2cm} \[ \Gamma \vdash \text{Roll} : \mathcal{N} \]
Typing rules: unit and bind

**Unit**

\[
\frac{\Gamma \vdash e : \tau}{\Gamma \vdash \text{return}(e) : \bigcirc \tau}
\]

**Bind**

\[
\frac{\Gamma \vdash e : \bigcirc \tau \quad \Gamma, x : \tau \vdash e' : \bigcirc \tau'}{\Gamma \vdash \text{sample } x = e \text{ in } e' : \bigcirc \tau'}
\]
What property do we want the types to ensure?

**Non-probabilistic types**
If \( e \in \mathcal{C}\mathcal{E} \) has non-probabilistic type \( \tau \), then \( e \) should reduce to \( \text{unit}(v) \) with \( v \in \mathcal{C}\mathcal{V} \) of type \( \tau \), or loop forever.

**Probabilistic types**
If \( e \in \mathcal{C}\mathcal{E} \) has probabilistic type \( \bigcirc \tau \), then \( e \) should reduce to \( \mu \in \text{SDistr}(\mathcal{C}\mathcal{V}) \) where every element in the support of \( \mu \) has type \( \tau \).
Monadic Type Systems: A Closer Look
What else can we do with a monadic type system?

So far: describe type of a distribution

If a program $e$ has type $\bigcirc \mathbb{N}$, then:

- It evaluates to a sub-distribution over $\mathbb{N}$: samples drawn from the distribution will always be natural numbers.
- It never gets stuck (runtime error) during evaluation.

But what other properties can we handle?

- Produces a uniform distribution
- Produces a distribution that has probability $1/4$ of returning an even number
- ...
The key typing rule: SAMPLE

\[
\Gamma \vdash e : \bigodot \tau \quad \Gamma, x : \tau \vdash e' : \bigodot \tau' \\
\Gamma \vdash \text{sample } x = e \text{ in } e' : \bigodot \tau' \quad \text{SAMPLE}
\]
The key typing rule: SAMPLE

\[
\Gamma \vdash e : \bigcirc \tau \\
\Gamma, x : \tau \vdash e' : \bigcirc \tau' \\
\Gamma \vdash \text{sample } x = e \text{ in } e' : \bigcirc \tau'
\]

Let’s unpack this rule

1. \( e \) is a \textbf{distribution over } \tau
The key typing rule: SAMPLE

\[
\begin{align*}
\Gamma \vdash e : \bigcirc \tau \\
\Gamma, x : \tau \vdash e' : \bigcirc \tau'
\end{align*}
\]

\[
\Gamma \vdash \text{sample } x = e \text{ in } e' : \bigcirc \tau'
\]

Let’s unpack this rule

1. \(e\) is a distribution over \(\tau\)
2. Given a sample \(x : \tau\), \(e'\) produces a distribution over \(\tau'\)
The key typing rule: SAMPLE

\[
\Gamma \vdash e : \bigcirc \tau \quad \Gamma, x : \tau \vdash e' : \bigcirc \tau'
\]

\[
\Gamma \vdash \text{sample } x = e \text{ in } e' : \bigcirc \tau'
\]

Let’s unpack this rule

1. \(e\) is a distribution over \(\tau\)
2. Given a sample \(x : \tau\), \(e'\) produces a distribution over \(\tau'\)
3. Sampling from \(e\) and plugging into \(e'\): distribution over \(\tau'\)
Generalizing the rule

\[
\Gamma \vdash e : P\tau \quad \Gamma, x : \tau \vdash e' : Q\tau' \\
\Gamma \vdash \text{sample } x = e \text{ in } e' : Q\tau' \quad \text{SAMPLEGEN}
\]
Generalizing the rule

\[
\Gamma \vdash e : P_\tau \\
\Gamma, x : \tau \vdash e' : Q_{\tau'} \\
\Gamma \vdash \text{sample } x = e \text{ in } e' : Q_{\tau'}
\]

Let's change the meaning of the distribution type

1. \( e \) is a distribution over \( \tau \) satisfying \( P \)
Generalizing the rule

\[
\Gamma \vdash e : P\tau \quad \Gamma, x : \tau \vdash e' : Q\tau' \quad \text{SAMPLEGEN}
\]
\[
\Gamma \vdash \text{sample } x = e \text{ in } e' : Q\tau'
\]

Let's change the meaning of the distribution type

1. \( e \) is a distribution over \( \tau \) satisfying \( P \)
2. Given a sample \( x : \tau \), \( e' \) produces a distribution over \( \tau' \) satisfying \( Q \)
Generalizing the rule

\[ \Gamma \vdash e : P \tau \quad \Gamma, x : \tau \vdash e' : Q \tau' \]

\[ \Gamma \vdash \text{sample } x = e \text{ in } e' : Q \tau' \]

Let’s change the meaning of the distribution type

1. \( e \) is a distribution over \( \tau \) satisfying \( P \)
2. Given a sample \( x : \tau, e' \) produces a distribution over \( \tau' \) satisfying \( Q \)
3. Sampling from \( e \) and plugging into \( e' \) produces a distribution over \( \tau' \) satisfying \( Q \)
Generalizing the rule

\[
\Gamma \vdash e : P\tau \\
\Gamma, x : \tau \vdash e' : Q\tau' \\
\Gamma \vdash \text{sample } x = e \text{ in } e' : Q\tau' \\
\]

**SAMPLEGEN**

Let’s change the meaning of the distribution type

1. \( e \) is a distribution over \( \tau \) satisfying \( P \)
2. Given a sample \( x : \tau, e' \) produces a distribution over \( \tau' \) satisfying \( Q \)
3. Sampling from \( e \) and plugging into \( e' \) produces a distribution over \( \tau' \) satisfying \( Q \)
Generalizing the rule

\[ \Gamma \vdash e : P\tau \quad \Gamma, x : \tau \vdash e' : Q\tau' \]

\[ \Gamma \vdash \text{sample } x = e \text{ in } e' : Q\tau' \]

Let's change the meaning of the distribution type

1. \( e \) is a distribution over \( \tau \) satisfying \( P \)
2. Given a sample \( x : \tau, e' \) produces a distribution over \( \tau' \) satisfying \( Q \)
3. Sampling from \( e \) and plugging into \( e' \) produces a distribution over \( \tau' \) satisfying \( Q \)

For what distribution properties \( Q \) is this rule OK? Does this remind you of something we have seen already?
An assertion $Q$ is CM if it satisfies:
If $\mu_1 \models Q$ and $\mu_2 \models Q$, then $\mu_1 \oplus_p \mu_2 \models Q$ for any $p \in [0, 1]$.

Examples of CM assertions

- $x = e$
- $\text{Unif}_B[x]$

Examples of non-CM assertions

- $[x] * [y]$
- $x = 1 \lor x = 2$
The main requirement: closed under mixtures (CM)

\[
\Gamma \vdash e : P\tau \quad \Gamma, x : \tau \vdash e' : Q\tau' \\
\Gamma \vdash \text{Sample } x = e \text{ in } e' : Q\tau'
\]
The main requirement: closed under mixtures (CM)

\[
\Gamma \vdash e : P\tau \quad \Gamma, x : \tau \vdash e' : Q\tau' \\
\hline
\Gamma \vdash \text{sample } x = e \text{ in } e' : Q\tau' \\
\]

The property \( Q \) must be closed under mixtures (CM)

1. We have a bunch of distributions over \( \tau' \) satisfying \( Q \)
The main requirement: closed under mixtures (CM)

\[ \Gamma \vdash e : P \tau \quad \Gamma, x : \tau \vdash e' : Q \tau' \]

\[ \Gamma \vdash \text{SampleGen} \]

\[ \Gamma \vdash \text{sample } x = e \in e' : Q \tau' \]

The property \( Q \) must be closed under mixtures (CM)

1. We have a bunch of distributions over \( \tau' \) satisfying \( Q \)
2. We are blending these distributions together
The main requirement: closed under mixtures (CM)

\[ \Gamma \vdash e : P\tau \quad \Gamma, x : \tau \vdash e' : Q\tau' \]

\[ \Gamma \vdash \text{sample } x = e \text{ in } e' : Q\tau' \]

The property \( Q \) must be closed under mixtures (CM)

1. We have a bunch of distributions over \( \tau' \) satisfying \( Q \)
2. We are blending these distributions together
3. We want the resulting distribution to also satisfy \( Q \)
Example: monadic types for uniformity

Type of uniform distributions $U\tau$

Meaning: when $\tau$ is a finite type (e.g., $\mathbb{B}$), a program $e$ has type $U\tau$ if it evaluates to the uniform distribution over $\tau$ without encountering any runtime errors.

Then the sampling rule is sound:

$$ \Gamma \vdash e : \bigcirc\tau \quad \Gamma, x : \tau \vdash e' : U\tau' \quad \text{SampleUnif} \quad \Gamma \vdash \text{sample } x = e \text{ in } e' : U\tau' $$
Monadic Type Systems: Generalizing to Graded Monads
From monads to graded monads

Instead of one monad, have a family of monads

- $M$ is a monoid with a pre-order (e.g., $(\mathbb{R}, 0, +, \leq)$)
- Each monadic type has an **index** $\alpha \in M$
From monads to graded monads

Instead of one monad, have a family of monads

- \( M \) is a monoid with a pre-order (e.g., \((\mathbb{R}, 0, +, \leq)\))
- Each monadic type has an index \( \alpha \in M \)

Intuition

- Graded monads: different kinds of the same monad
- Smaller index: less information/weaker guarantee
- Index carries additional information “on the side”
- Indexes combine through the bind rule
Changes to the type system

New types

\[ \mathcal{T} \ni \tau ::= \cdots \mid \bigcirc_{\alpha} \tau \quad (\alpha \in M) \]

New typing rules

\[
\begin{align*}
\Gamma \vdash e : \tau &\quad \text{GRETURN} \\
\Gamma \vdash \text{return}(e) : \bigcirc_{0} \tau &\quad \text{GRETURN}
\end{align*}
\]

\[
\begin{align*}
\Gamma \vdash e : \bigcirc_{\alpha} \tau &\quad \text{GSAMPLE} \\
\Gamma, x : \tau \vdash e' : \bigcirc_{\beta} \tau' &\quad \text{GSAMPLE}
\end{align*}
\]

\[
\begin{align*}
\Gamma \vdash e : \bigcirc_{\alpha} &\quad \text{GSUBTY} \\
\alpha \leq \beta &\quad \text{GSUBTY}
\end{align*}
\]

\[
\begin{align*}
\Gamma \vdash e : \bigcirc_{\alpha} &\quad \text{GSUBTY} \\
\alpha \leq \beta &\quad \text{GSUBTY}
\end{align*}
\]

\[
\begin{align*}
\Gamma \vdash e : \bigcirc_{\beta} &\quad \text{GSUBTY} \\
\alpha \leq \beta &\quad \text{GSUBTY}
\end{align*}
\]
**Monadic types: references and further readings**

**Original papers on probabilistic monadic types**

**Differential privacy typing**
- Key ingredients: (bounded) linear types and a monad
- Reed and Pierce. Distance makes the types grow stronger: a calculus for differential privacy. ICFP 2010.

**HOARE$^2$: probabilistic relational properties by typing**
- Key ingredients: Refinement types and a graded monad.
Beyond Monadic Types:

Two Representative Systems
Monadic type systems: the good and the bad

The good
- Clean separation between deterministic and randomized
- Always treat variables as values, not distributions

The bad
- Class of properties is limited
- All properties everywhere must be CM (cf. PSL)
Main features

- Makes $\tau$ and $\bigcirc\tau$ the same: no more monad!
- Call-by-value: sample when passing arguments to fn.

What kinds of properties can be expressed in types?

- No monad type, but let-binding rule is similar to SAMPLE
- Seems to need the CM condition
Judgments look like

\[ x_1 : \tau_1, \ldots, x_n : \tau_n \vdash e : \tau \]

Reading

For any well-typed closing substitution of values \( v_1, \ldots, v_n \) for \( x_1, \ldots x_n \), the expression \( e \) evaluates to distribution over \( \tau \).
Main features

- Makes $\tau$ and $\bigcirc \tau$ the same: no more monad!
- **Call-by-name**: functions can take distributions
- Let-binding construct used to force sampling

What kinds of properties can be expressed in types?

- Function calls don’t force sampling
- Let-binding, if-then-else, all force sampling
Judgments look like

\[ x_1 : \tau_1, \ldots, x_n : \tau_n \vdash e : \tau \]

Reading

For any well-typed closing substitution of distributions \( \mu_1, \ldots, \mu_n \) for \( \mu_1, \ldots, \mu_n \), the expression \( e \) evaluates to some distribution over \( \tau \).

But note that \( \mu_1, \ldots, \mu_n \) are entirely separate distributions: draws from \( \mu_1, \ldots, \mu_n \) are always independent.
Many technical extensions

Richer distributions
  ➤ Continuous distributions
  ➤ Distributions over function spaces

Richer types
  ➤ Recursive types, linear types, ...

Richer language features
  ➤ Most notably: conditioning constructs (“observe”/“score”)
Higher-order programs: references and readings

**Semantics**

**Type systems**
- PCF\(\oplus\): Dal Lago ([https://doi.org/10.1017/9781108770750.005](https://doi.org/10.1017/9781108770750.005))
Reasoning about Probabilistic Programs

Wrapping up
Day 1: Introducing Probabilistic Programs
- Motivations and key questions
- Mathematical preliminaries

Day 2: First-Order Programs 1
- Probabilistic While language, monadic semantics
- Weakest pre-expectation calculus

Day 3: First-Order Programs 2
- Probabilistic While language, transformer semantics
- Probabilistic separation logic

Day 4: Higher-Order Programs
- Type system: probability monad
- Type system: probabilistic PCF
Main takeaways

There are multiple semantics for probabilistic programs

- We saw: monadic semantics, and transformer semantics
- Choice of semantics influences what verification is possible

Standard verification methods, to probabilistic programs

- Weakest pre-conditions to weakest pre-expectations
- Separation logic to Probabilistic separation logic
- Type systems, monads, ...

Verification currently better for imperative programs

- Wide variety of Hoare logics proving interesting properties
- Type systems for probabilistic programs: active research
Where to go next

More semantics
- Lots of recent research on categorical semantics (e.g., QBS)

Learn about conditioning
- Mostly implementation (hard), but recently verification too

Verifying specific properties
- Expected running time, probabilistic termination, ...

Interesting applications
- Cryptography, differential privacy, machine learning, ...

Read: Foundations of Probabilistic Programming
- Open-access book, 15 chapters by leading researchers

https://doi.org/10.1017/9781108770750
Reasoning about Probabilistic Programs
Oregon PL Summer School 2021

Justin Hsu
UW–Madison
Cornell University