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

Last time: monadic semantics for pWHILE

The pWHILE language

- Core imperative language extended with random sampling

Last time: monadic semantics for pWHILE

The pWHILE language

- Core imperative language extended with random sampling

Monadic semantics

$$(\!|c|\!) : \mathcal{M} \to \mathsf{Distr}(\mathcal{M})$$

- Input: memory
- Output: distribution over memories

Last time: weakest pre-expectations

Weakest pre-expectation calculus

- Given: PWHILE program c
- Given: post-expectation $E : \mathcal{M} \to \mathbb{R}^+$
- Compute $wpe(c, E)$: maps an input m to c to the expected value of E in the output of c executed on m.

Last time: weakest pre-expectations

Weakest pre-expectation calculus

- Given: pWhile program c
- Given: post-expectation $E : \mathcal{M} \to \mathbb{R}^+$
- Compute $wpe(c, E)$: maps an input m to c to the expected value of E in the output of c executed on m.

What is this useful for?

- "The probability of $x = y$ is $1/2$" in the output
- "The expected value of t in the output is $n + 42$"

How to compute Weakest Pre-expectations easier?

Same idea as for wp: define wpe compositionally

- Compute wpe of a program from wpe of sub-programs
- Break down a complicated computation into simpler parts

How to compute Weakest Pre-expectations easier?

Same idea as for wp: define wpe compositionally

- Compute wpe of a program from wpe of sub-programs
- Break down a complicated computation into simpler parts

Overall framework developed by Morgan and McIver

- Work over multiple decades, building on work by Kozen
- Also covered non-deterministic choice (we won't do this)

WPE Calculus: Skip

Intuition

- Program: skip
- Post-expectation: E
- Average value of E after is just E before

WPE Calculus: Skip

Intuition

- Program: skip
- Post-expectation: E
- Average value of E after is just E before

WPE for Skip

$$wpe(\mathsf{skip}, E) = E$$

WPE Calculus: Assignment

Intuition

- Program: $x \leftarrow e$
- Post-expectation: E
- Average value of E after is E with $x \mapsto e$ before

WPE Calculus: Assignment

Intuition

- Program: $x \leftarrow e$
- Post-expectation: E
- Average value of E after is E with $x \mapsto e$ before

WPE for Assignment

$$wpe(x \leftarrow e, E) = E[x \mapsto e]$$

WPE Calculus: Random sampling

Intuition

- Program: $x \xleftarrow{\$} d$
- Post-expectation: E
- Average value of E computed from averaging over x

WPE Calculus: Random sampling

Intuition

- Program: $x \xleftarrow{\$} d$
- Post-expectation: E
- Average value of E computed from averaging over x

WPE for sampling $\mathbf{Flip}(p)$

$$wpe(x \xleftarrow{\$} \mathbf{Flip}(p), E) = p \cdot E[x \mapsto tt] + (1 - p) \cdot E[x \mapsto ff]$$

Try this at home!

What is $wpe(x \xleftarrow{\$} \mathbf{Roll}, E)$?

WPE Calculus: Sequencing

Intuition

- Program: $c_1 \ ; c_2$
- Post-expectation: E
- Average value of E after c_2 is $wpe(c_2, E)$ before c_2
- Average value of $wpe(c_2, E)$ before c_1: another wpe

WPE Calculus: Sequencing

Intuition

- Program: $c_1 \ ; c_2$
- Post-expectation: E
- Average value of E after c_2 is $wpe(c_2, E)$ before c_2
- Average value of $wpe(c_2, E)$ before c_1: another wpe

WPE for Sequencing

$$wpe(c_1 \ ; c_2, E) = wpe(c_1, wpe(c_2, E))$$

WPE Calculus: Conditionals

Intuition

- Program: if e then c_1 else c_2
- Post-expectation: E
- Average value of E after is $wpe(c_1, E)$ before if $e = tt$, else $wpe(c_2, E)$ before if $e = ff$

WPE Calculus: Conditionals

Intuition

- Program: if e then c_1 else c_2
- Post-expectation: E
- Average value of E after is $wpe(c_1, E)$ before if $e = tt$, else $wpe(c_2, E)$ before if $e = ff$

WPE for Conditionals

$$wpe(\text{if } e \text{ then } c_1 \text{ else } c_2, E) = [e] \cdot wpe(c_1, E) + [\neg e] \cdot wpe(c_2, E)$$

Indicator functions play the role of if-then-else.

WPE Calculus: Main soundness theorem

Theorem

Let c be a pWHILE program, E be an expectation, and $m \in \mathcal{M}$ be any input state. If $\mu = (\!|c|\!)m$ is the output memory, then:

$$\mathbb{E}_{m' \sim \mu}[E(m')] = wpe(c, E)(m).$$

WPE Calculus: Main soundness theorem

Theorem

Let c be a PWHILE *program, E be an expectation, and $m \in \mathcal{M}$ be any input state. If $\mu = (\!|c|\!)m$ is the output memory, then:*

$$\mathbb{E}_{m' \sim \mu}[E(m')] = wpe(c, E)(m).$$

Try this at home!

Prove this for loop-free programs, by induction on the program structure.

Weakest Pre-expectations

for Probabilistic Loops

Can you guess this WPE?

Program:

$$\begin{aligned}&n \leftarrow 100;\\&\text{while } n > 42 \text{ do}\\&\quad n \leftarrow n - 1\end{aligned}$$

Post-expectation: n

Can you guess this WPE?

Program:

$$\begin{array}{l} n \leftarrow 100; \\ \text{while } n > 42 \text{ do} \\ \quad n \leftarrow n - 1 \end{array}$$

Post-expectation: n

Answer

Deterministic program, always terminates with $n = 42$. So $wpe(c, n) = 42$.

What about this one?

Program:

$$
\begin{array}{l}
n \leftarrow 100; \\
\text{while } n > 42 \text{ do} \\
\quad dec \xleftarrow{\$} \mathbf{Flip}; \\
\quad \text{if } dec \text{ then } n \leftarrow n - 1
\end{array}
$$

Post-expectation: n

What about this one?

Program:

$$
\begin{array}{l}
n \leftarrow 100; \\
\text{while } n > 42 \text{ do} \\
\quad dec \xleftarrow{\$} \mathbf{Flip}; \\
\quad \text{if } dec \text{ then } n \leftarrow n - 1
\end{array}
$$

Post-expectation: n

Answer

Randomized program, but always terminates with $n = 42$. So $wpe(c, n) = 42$.

What about this one?

Program:

$$
\begin{array}{l}
t \leftarrow 0; \mathit{stop} \leftarrow \mathit{ff}; \\
\text{while } \neg \mathit{stop} \text{ do} \\
\quad t \leftarrow t + 1; \\
\quad \mathit{stop} \xleftarrow{\$} \mathbf{Flip}(1/4)
\end{array}
$$

Post-expectation: t

What about this one?

Program:

$$
\begin{array}{l}
t \leftarrow 0; stop \leftarrow \mathit{ff}; \\
\text{while } \neg stop \text{ do} \\
\quad t \leftarrow t + 1; \\
\quad stop \xleftarrow{\$} \mathbf{Flip}(1/4)
\end{array}
$$

Post-expectation: t

Starting to get more complicated...

Can we give a general method to compute wpe for loops?

What is the WPE of a loop?

Can define wpe for loops mathematically, but...

- Defined in terms of a least fixed point
- Hard to compute $wpe(\text{while } b \text{ do } c, E)$ in terms of $wpe(c, -)$

What is the WPE of a loop?

Can define wpe for loops mathematically, but...

- Defined in terms of a least fixed point
- Hard to compute $wpe(\text{while } b \text{ do } c, E)$ in terms of $wpe(c, -)$

Idea: prove upper and lower bounds on wpe

- Analog of wp: implication becomes inequality
- Don't aim to compute wpe exactly

Making it easier to bound WPE: super-invariant rule

Setup: check upper-bounds on wpe

- Program: while e do c
- Pre-expectation E', Post-expectation E
- Goal: Check if $wpe(\text{while } e \text{ do } c, E) \leq E'$

Making it easier to bound WPE: super-invariant rule

Setup: check upper-bounds on wpe

- Program: while e do c
- Pre-expectation E', Post-expectation E
- Goal: Check if $wpe(\text{while } e \text{ do } c, E) \leq E'$

Super-invariant rule

Suppose we have an expectation I (the invariant) satisfying the super-invariant conditions:

- $I \leq E'$
- $[e] \cdot wpe(c, I) + [\neg e] \cdot E \leq I$

Making it easier to bound WPE: super-invariant rule

Setup: check upper-bounds on wpe

- Program: while e do c
- Pre-expectation E', Post-expectation E
- Goal: Check if $wpe(\text{while } e \text{ do } c, E) \leq E'$

Super-invariant rule

Suppose we have an expectation I (the invariant) satisfying the super-invariant conditions:

- $I \leq E'$
- $[e] \cdot wpe(c, I) + [\neg e] \cdot E \leq I$

Then we can conclude the upper-bound:

$$wpe(\text{while } e \text{ do } c, E) \leq E'$$

Making it easier to bound WPE: sub-invariant rule

Setup: check lower-bounds on wpe

- Program: while e do c
- Pre-expectation E', Post-expectation E
- Goal: Check if $E' wpe(\text{while } e \text{ do } c, E)$

Making it easier to bound WPE: sub-invariant rule

Setup: check lower-bounds on wpe

- Program: while e do c
- Pre-expectation E', Post-expectation E
- Goal: Check if $E' wpe($while e do $c, E)$

Sub-invariant rule

Suppose we have an expectation I (the invariant) satisfying the sub-invariant conditions and I is bounded in $[0, 1]$:

- $E' \leq I$
- $I \leq [e] \cdot wpe(c, I) + [\neg e] \cdot E$

Making it easier to bound WPE: sub-invariant rule

Setup: check lower-bounds on wpe

- Program: while e do c
- Pre-expectation E', Post-expectation E
- Goal: Check if $E' wpe($while e do $c, E)$

Sub-invariant rule

Suppose we have an expectation I (the invariant) satisfying the sub-invariant conditions and I is bounded in $[0, 1]$:

- $E' \leq I$
- $I \leq [e] \cdot wpe(c, I) + [\neg e] \cdot E$

Then we can conclude the lower-bound:

$$E' \leq wpe(\text{while } e \text{ do } c, E)$$

An example: Fair

Simulate a fair coin flip from biased coin flips

while $x = y$ do
$x \xleftarrow{\$} \mathbf{Flip}(p)$;
$y \xleftarrow{\$} \mathbf{Flip}(p)$;

An example: Fair

Simulate a fair coin flip from biased coin flips

$$
\begin{array}{l}
\text{while } x = y \text{ do} \\
\quad x \xleftarrow{\$} \mathbf{Flip}(p); \\
\quad y \xleftarrow{\$} \mathbf{Flip}(p);
\end{array}
$$

Goal: show that if $x = y$ initially, then final x is fair coin

In terms of wpe, this follows from proving:

$$wpe(\textsc{Fair}, [x]) = [x = y] \cdot 0.5 + [x \neq y] \cdot [x]$$

An example: FAIR

Simulate a fair coin flip from biased coin flips

$$\begin{array}{l} \text{while } x = y \text{ do} \\ \quad x \xleftarrow{\$} \mathbf{Flip}(p); \\ \quad y \xleftarrow{\$} \mathbf{Flip}(p); \end{array}$$

Goal: show that if $x = y$ initially, then final x is fair coin

In terms of wpe, this follows from proving:

$$wpe(\text{FAIR}, [x]) = [x = y] \cdot 0.5 + [x \neq y] \cdot [x]$$

Prove this in two steps:

1. Upper-bound: $wpe(\text{FAIR}, [x]) \leq [x = y] \cdot 0.5 + [x \neq y] \cdot [x]$
2. Lower-bound: $wpe(\text{FAIR}, [x]) \geq [x = y] \cdot 0.5 + [x \neq y] \cdot [x]$

FAIR: proving the upper-bound

Want I satisfying super-invariant conditions:

$$I \leq [x = y] \cdot wpe(x \xleftarrow{\$} \mathbf{Flip}(p); y \xleftarrow{\$} \mathbf{Flip}(p), I) + [x \neq y] \cdot [x]$$

$\textsc{Fair}$: proving the upper-bound

Want I satisfying super-invariant conditions:

$$I \leq [x = y] \cdot wpe(x \xleftarrow{\$} \mathbf{Flip}(p); y \xleftarrow{\$} \mathbf{Flip}(p), I) + [x \neq y] \cdot [x]$$

Take the following invariant:

$$I \triangleq [x = y] \cdot 0.5 + [x \neq y] \cdot [x]$$

FAIR: checking the super-invariant condition

Apply the wpe calculus rules

$$[x = y] \cdot wpe(x \xleftarrow{\$} \mathbf{Flip}(p); y \xleftarrow{\$} \mathbf{Flip}(p), I) + [x \neq y] \cdot [x]$$

FAIR: checking the super-invariant condition

Apply the *wpe* calculus rules

$$[x = y] \cdot wpe(x \xleftarrow{\$} \mathbf{Flip}(p); y \xleftarrow{\$} \mathbf{Flip}(p), I) + [x \neq y] \cdot [x]$$
$$= [x = y] \cdot wpe(x \xleftarrow{\$} \mathbf{Flip}(p),$$
$$p \cdot I[y \mapsto tt] + (1 - p) \cdot I[y \mapsto \mathit{ff}]) + [x \neq y] \cdot [x]$$

Fair: checking the super-invariant condition

Apply the wpe calculus rules

$$[x = y] \cdot wpe(x \xleftarrow{\$} \mathbf{Flip}(p); y \xleftarrow{\$} \mathbf{Flip}(p), I) + [x \neq y] \cdot [x]$$

$$= [x = y] \cdot wpe(x \xleftarrow{\$} \mathbf{Flip}(p),$$

$$p \cdot I[y \mapsto tt] + (1 - p) \cdot I[y \mapsto ff]) + [x \neq y] \cdot [x]$$

$$= [x = y] \cdot (p \cdot p \cdot I[x, y \mapsto tt] + p \cdot (1 - p) \cdot I[x, y \mapsto tt, ff]$$

$$+ p \cdot (1 - p) \cdot I[x, y \mapsto ff, tt] + (1 - p)^2 \cdot I[x, y \mapsto ff]) + [x \neq y] \cdot [x]$$

FAIR: checking the super-invariant condition

Apply the *wpe* calculus rules

$$
\begin{aligned}
&[x = y] \cdot wpe(x \xleftarrow{\$} \mathbf{Flip}(p); y \xleftarrow{\$} \mathbf{Flip}(p), I) + [x \neq y] \cdot [x] \\
&= [x = y] \cdot wpe(x \xleftarrow{\$} \mathbf{Flip}(p), \\
&\quad p \cdot I[y \mapsto tt] + (1 - p) \cdot I[y \mapsto ff]) + [x \neq y] \cdot [x] \\
&= [x = y] \cdot (p \cdot p \cdot I[x, y \mapsto tt] + p \cdot (1 - p) \cdot I[x, y \mapsto tt, ff] \\
&\quad + p \cdot (1 - p) \cdot I[x, y \mapsto ff, tt] + (1 - p)^2 \cdot I[x, y \mapsto ff]) + [x \neq y] \cdot [x] \\
&= [x = y] \cdot (p \cdot p \cdot 0.5 + p \cdot (1 - p) \cdot 1 \\
&\quad + p \cdot (1 - p) \cdot 0 + (1 - p)^2 \cdot 0.5) + [x \neq y] \cdot [x]
\end{aligned}
$$

FAIR: checking the super-invariant condition

Apply the *wpe* calculus rules

$$
\begin{aligned}
&[x = y] \cdot wpe(x \xleftarrow{\$} \mathbf{Flip}(p); y \xleftarrow{\$} \mathbf{Flip}(p), I) + [x \neq y] \cdot [x] \\
&= [x = y] \cdot wpe(x \xleftarrow{\$} \mathbf{Flip}(p), \\
&\quad p \cdot I[y \mapsto tt] + (1 - p) \cdot I[y \mapsto \mathit{ff}]) + [x \neq y] \cdot [x] \\
&= [x = y] \cdot (p \cdot p \cdot I[x, y \mapsto tt] + p \cdot (1 - p) \cdot I[x, y \mapsto tt, \mathit{ff}] \\
&\quad + p \cdot (1 - p) \cdot I[x, y \mapsto \mathit{ff}, tt] + (1 - p)^2 \cdot I[x, y \mapsto \mathit{ff}]) + [x \neq y] \cdot [x] \\
&= [x = y] \cdot (p \cdot p \cdot 0.5 + p \cdot (1 - p) \cdot 1 \\
&\quad + p \cdot (1 - p) \cdot 0 + (1 - p)^2 \cdot 0.5) + [x \neq y] \cdot [x] \\
&= [x = y] + [x \neq y] \cdot [x] \leq I
\end{aligned}
$$

Fair: checking the super-invariant condition

Apply the *wpe* calculus rules

$$[x = y] \cdot wpe(x \overset{\$}{\leftarrow} \mathbf{Flip}(p); y \overset{\$}{\leftarrow} \mathbf{Flip}(p), I) + [x \neq y] \cdot [x]$$

$$\begin{aligned}
&= [x = y] \cdot wpe(x \overset{\$}{\leftarrow} \mathbf{Flip}(p), \\
&\quad p \cdot I[y \mapsto tt] + (1 - p) \cdot I[y \mapsto \mathit{ff}]) + [x \neq y] \cdot [x] \\
&= [x = y] \cdot (p \cdot p \cdot I[x, y \mapsto tt] + p \cdot (1 - p) \cdot I[x, y \mapsto tt, \mathit{ff}] \\
&\quad + p \cdot (1 - p) \cdot I[x, y \mapsto \mathit{ff}, tt] + (1 - p)^2 \cdot I[x, y \mapsto \mathit{ff}]) + [x \neq y] \cdot [x] \\
&= [x = y] \cdot (p \cdot p \cdot 0.5 + p \cdot (1 - p) \cdot 1 \\
&\quad + p \cdot (1 - p) \cdot 0 + (1 - p)^2 \cdot 0.5) + [x \neq y] \cdot [x] \\
&= [x = y] + [x \neq y] \cdot [x] \leq I
\end{aligned}$$

FAIR: checking the super-invariant condition

Apply the wpe calculus rules

$$
\begin{aligned}
&[x = y] \cdot wpe(x \xleftarrow{\$} \mathbf{Flip}(p); y \xleftarrow{\$} \mathbf{Flip}(p), I) + [x \neq y] \cdot [x] \\
&= [x = y] \cdot wpe(x \xleftarrow{\$} \mathbf{Flip}(p), \\
&\quad p \cdot I[y \mapsto tt] + (1-p) \cdot I[y \mapsto ff]) + [x \neq y] \cdot [x] \\
&= [x = y] \cdot (p \cdot p \cdot I[x, y \mapsto tt] + p \cdot (1-p) \cdot I[x, y \mapsto tt, ff] \\
&\quad + p \cdot (1-p) \cdot I[x, y \mapsto ff, tt] + (1-p)^2 \cdot I[x, y \mapsto ff]) + [x \neq y] \cdot [x] \\
&= [x = y] \cdot (p \cdot p \cdot 0.5 + p \cdot (1-p) \cdot 1 \\
&\quad + p \cdot (1-p) \cdot 0 + (1-p)^2 \cdot 0.5) + [x \neq y] \cdot [x] \\
&= [x = y] + [x \neq y] \cdot [x] \leq I
\end{aligned}
$$

Thus the super-invariant rule proves the upper-bound:

$$wpe(\textsf{FAIR}, [x]) \leq [x = y] \cdot 0.5 + [x \neq y] \cdot [x]$$

FAIR: proving the lower-bound

Want I satisfying sub-invariant conditions:

$$I \geq [x = y] \cdot wpe(x \xleftarrow{\$} \mathbf{Flip}(p); y \xleftarrow{\$} \mathbf{Flip}(p), I) + [x \neq y] \cdot [x]$$

The same invariant works:

$$I \triangleq [x = y] \cdot 0.5 + [x \neq y] \cdot [x]$$

And I is bounded in $[0, 1]$.

Thus the sub-invariant rule proves the lower-bound:

$$wpe(\text{FAIR}, [x]) \geq [x = y] \cdot 0.5 + [x \neq y] \cdot [x]$$

WPE: references and further reading

Recent survey of the area

Kaminski. Advanced Weakest Precondition Calculi for Probabilistic Programs. PhD Thesis (RWTH Aachen), 2019.
`https://moves.rwth-aachen.de/people/kaminski/thesis/`

Comprehensive book

McIver and Morgan. Abstraction, Refinement and Proof for Probabilistic Systems. Springer, 2004.

Related methods: Hoare logics for monadic pWHILE

Prove judgments of the following form:

$$\{P\}\ c\ \{Q\}$$

- Pre-condition P describes input memory
- Post-condition Q describes output memory distribution

Example systems

- A program logic for union bounds (ICALP16)
- Formal certification of code-based cryptographic proofs (POPL09)
- Probabilistic relational reasoning for differential privacy (POPL12)
- A pre-expectation calculus for probabilistic sensitivity (POPL21)

A Second Semantics for pWhile

Transformer Semantics

Why a second semantics?

Why a second semantics?

Alternative view of what the program does

- Gives us a new way of understanding the program behavior

Why a second semantics?

Alternative view of what the program does

- Gives us a new way of understanding the program behavior

Enable new extensions of the language

- Allows extending the language with different features

Why a second semantics?

Alternative view of what the program does

- Gives us a new way of understanding the program behavior

Enable new extensions of the language

- Allows extending the language with different features

Support different verification methods

- Can make some properties easier (or harder) to verify

Semantics of expressions/distributions: unchanged

Recall: program states are memories

Memory m maps each variable to a value:

$$m \in \mathcal{M} = \mathcal{X} \to \mathcal{V}$$

Semantics of expressions/distributions: unchanged

Recall: program states are memories

Memory m maps each variable to a value:

$$m \in \mathcal{M} = \mathcal{X} \to \mathcal{V}$$

Expression semantics: map memory to value

$$[\![-]\!] : \mathcal{E} \to \mathcal{M} \to \mathcal{V}$$

Semantics of expressions/distributions: unchanged

Recall: program states are memories

Memory m maps each variable to a value:

$$m \in \mathcal{M} = \mathcal{X} \to \mathcal{V}$$

Expression semantics: map memory to value

$$[\![-]\!] : \mathcal{E} \to \mathcal{M} \to \mathcal{V}$$

D-expression semantics: distribution over values

$$[\![-]\!] : \mathcal{DE} \to \mathsf{Distr}(\mathcal{V})$$

Transformer semantics of commands: overview

Last time: monadic semantics

$$(\!|-|\!) : \mathcal{C} \to \mathcal{M} \to \mathsf{Distr}(\mathcal{M})$$

Command: input memory to output distribution over memories.

Transformer semantics of commands: overview

Last time: monadic semantics

$$(\!|-|\!) : \mathcal{C} \to \mathcal{M} \to \mathsf{Distr}(\mathcal{M})$$

Command: input memory to output distribution over memories.

This time: transformer semantics (Kozen)

$$[\![-]\!] : \mathcal{C} \to \mathsf{Distr}(\mathcal{M}) \to \mathsf{Distr}(\mathcal{M})$$

Command: input distribution over memories to output distribution over memories.

Semantics of commands: skip

Intuition

- Input: memory distribution μ
- Output: the same memory distribution μ

Semantics of commands: skip

Intuition

- Input: memory distribution μ
- Output: the same memory distribution μ

Semantics of skip

$$[\![\mathsf{skip}]\!] \mu \triangleq \mu$$

Semantics of commands: assignment

Intuition

- Input: memory distribution μ
- Output: distribution from sampling m from μ, and mapping to m with $x \mapsto v$, where v is the original value of e in m.

Semantics of commands: assignment

Intuition

- Input: memory distribution μ
- Output: distribution from sampling m from μ, and mapping to m with $x \mapsto v$, where v is the original value of e in m.

Semantics of assignment

Let $f(m) = m[x \mapsto [\![e]\!]m]$. Then:

$$[\![x \leftarrow e]\!]\mu \triangleq map(f)(\mu)$$

Semantics of commands: sampling

Intuition

- Input: memory distribution μ
- Sample m from μ, and sample v from d-expression
- Output: return updated memory, m with $x \mapsto v$

Semantics of commands: sampling

Intuition

- Input: memory distribution μ
- Sample m from μ, and sample v from d-expression
- Output: return updated memory, m with $x \mapsto v$

Semantics of sampling

Let $g(m)(v) = m[x \mapsto v]$. Then:

$$[\![x \xleftarrow{\$} d]\!]\mu \triangleq bind(\mu, \lambda m.\ map(g(m))([\![d]\!]))$$

Semantics of commands: sequencing

Intuition

- Input: memory distribution μ
- Transform μ to μ' using first command
- Output: transform μ' to μ'' using second command

Semantics of commands: sequencing

Intuition

- Input: memory distribution μ
- Transform μ to μ' using first command
- Output: transform μ' to μ'' using second command

Semantics of sequencing

$$[\![c_1 \ ; \ c_2]\!]\mu \triangleq [\![c_2]\!]([\![c_1]\!]\mu)$$

Semantics of commands: conditionals (first try)

Intuition

- Input: memory distribution μ
- ???

Semantics of commands: conditionals (first try)

Intuition

- Input: memory distribution μ
- ???

Problem: what should input to branches be?

- First branch: distribution where guard holds
- Second branch: distribution where guard doesn't hold
- But μ may have some probability of both cases
- Can't case analysis on guard in μ (cf. monadic semantics)

Operations on distributions: conditioning

Restrict a distribution to a smaller subset

Given a distribution over A, assume that the result is in $E \subseteq A$. Then what probabilities should we assign elements in A?

Distribution conditioning

Let $\mu \in \text{Distr}(A)$, and $E \subseteq A$. Then μ conditioned on E is the distribution in $\text{Distr}(A)$ defined by:

$$(\mu \mid E)(a) \triangleq \begin{cases} \mu(a)/\mu(E) & : a \in E \\ 0 & : a \notin E \end{cases}$$

Idea: probability of a “assuming that” the result must be in E. Only makes sense if $\mu(E)$ is not zero!

Semantics of commands: conditionals (second try)

Intuition

- Input: memory distribution μ
- Condition μ on guard true; transform with first branch
- Condition μ on guard false; transform with second branch
- Output: ???

Semantics of commands: conditionals (second try)

Intuition

- Input: memory distribution μ
- Condition μ on guard true; transform with first branch
- Condition μ on guard false; transform with second branch
- Output: ???

Problem: how to combine outputs of branches?

- First branch: some output distribution
- Second branch: some other output distribution
- But we want a single output for the if-then-else

Operations on distributions: convex combination

Blending/mixing two distributions

Say we have distributions μ_1, μ_2 over the same set. Blending the distributions: with probability p, draw something from μ_1. Else, draw something from μ_2.

Convex combination

Let $\mu_1, \mu_2 \in \mathsf{Distr}(A)$, and let $p \in [0, 1]$. Then the convex combination of μ_1 and μ_2 is defined by:

$$\mu_1 \oplus_p \mu_2(a) \triangleq p \cdot \mu_1(a) + (1 - p) \cdot \mu_2(a).$$

Semantics of commands: conditionals

Intuition

- Input: memory distribution μ
- Record probability p of guard true
- Condition μ on guard true; transform with first branch
- Condition μ on guard false; transform with second branch
- Output: take p-convex combination of two results

Semantics of commands: conditionals

Intuition

- Input: memory distribution μ
- Record probability p of guard true
- Condition μ on guard true; transform with first branch
- Condition μ on guard false; transform with second branch
- Output: take p-convex combination of two results

Semantics of conditionals

Let $p = \mu(\llbracket e \rrbracket)$ be the probability the guard is true. Then:

$$\llbracket \textsf{if } e \textsf{ then } c_1 \textsf{ else } c_2 \rrbracket \mu \triangleq \llbracket c_1 \rrbracket(\mu \mid \llbracket e = tt \rrbracket) \oplus_p \llbracket c_2 \rrbracket(\mu \mid \llbracket e = ff \rrbracket)$$

Semantics of commands: loops

Same strategy works as before

- Define sequence of loop approximants $\mu_1, \mu_2, \ldots$
- Each μ_n: outputs terminating after n iterations
- Take limit μ_n as $n \to \infty$ to define output of loop

Semantics of commands: loops

Same strategy works as before

- Define sequence of loop approximants $\mu_1, \mu_2, \ldots$
- Each μ_n: outputs terminating after n iterations
- Take limit μ_n as $n \to \infty$ to define output of loop

Maybe don't try this at home:

Work out the gory details and define a transformer semantics for loops.

Comparing the two semantics:

Monadic versus Transformer

Monadic semantics to transformer semantics

Useful construction

- Given: $f : \mathcal{M} \to \mathsf{Distr}(\mathcal{M})$
- Define $f^{\#} : \mathsf{Distr}(\mathcal{M}) \to \mathsf{Distr}(\mathcal{M})$ by "averaging f" over input distribution:

$$f^{\#}(\mu)(m') \triangleq \sum_{m \in \mathcal{M}} \mu(m) \cdot f(m)(m')$$

Monadic semantics to transformer semantics

Useful construction

- Given: $f : \mathcal{M} \to \mathsf{Distr}(\mathcal{M})$
- Define $f^{\#} : \mathsf{Distr}(\mathcal{M}) \to \mathsf{Distr}(\mathcal{M})$ by "averaging f" over input distribution:

$$f^{\#}(\mu)(m') \triangleq \sum_{m \in \mathcal{M}} \mu(m) \cdot f(m)(m')$$

Relation between semantics

For any pWHILE program c and input distribution μ, we have:

$$(\!| c |\!)^{\#}(\mu) = [\![c]\!] \mu$$

Good sanity check: would be strange if monadic semantics disagrees with transformer semantics when we feed in the same input distribution.

Transformer semantics to monadic semantics?

Not so useful fact

- Given: $\overline{f} : \mathsf{Distr}(\mathcal{M}) \to \mathsf{Distr}(\mathcal{M})$
- There does not always exist $f : \mathcal{M} \to \mathsf{Distr}(\mathcal{M})$ such that $\overline{f} = f^{\#}$.
- Transformer semantics supports fancier PPL features

Transformer semantics to monadic semantics?

Not so useful fact

- Given: $\overline{f} : \mathsf{Distr}(\mathcal{M}) \to \mathsf{Distr}(\mathcal{M})$
- There does not always exist $f : \mathcal{M} \to \mathsf{Distr}(\mathcal{M})$ such that $\overline{f} = f^{\#}$.
- Transformer semantics supports fancier PPL features

Notable example: conditioning

New command to condition the input distribution on a guard being true:

$$[\![\mathsf{observe}(e)]\!]\mu \triangleq \mu \mid [\![e = tt]\!]$$

Not possible to give a monadic semantics to this command.

For verification: what is the tradeoff?

Why prefer monadic semantics?

- Memory assertions are simpler than distribution assertions
- Can do case analysis on memory if input is a memory

For verification: what is the tradeoff?

Why prefer monadic semantics?

- Memory assertions are simpler than distribution assertions
- Can do case analysis on memory if input is a memory

Why prefer transformer semantics?

- Sometimes, want to assume property of input distribution
- Can enable verifying richer probabilistic properties

Reasoning about pWhile Programs

Probabilistic Separation Logic

What Is Independence, Intuitively?

Two random variables x and y are **independent** if they are uncorrelated: the value of x gives no information about the value or distribution of y.

Things that are independent

Fresh random samples

- x is the result of a fair coin flip
- y is the result of another, “fresh” coin flip
- More generally: “separate” sources of randomness

Things that are independent

Fresh random samples

- x is the result of a fair coin flip
- y is the result of another, “fresh” coin flip
- More generally: “separate” sources of randomness

Uncorrelated things

- x is today’s winning lottery number
- y is the closing price of the stock market

Things that are not independent

Re-used samples

- x is the result of a fair coin flip
- y is the result of the same coin flip

Things that are not independent

Re-used samples

- x is the result of a fair coin flip
- y is the result of the same coin flip

Common cause

- x is today's ice cream sales
- y is today's sunglasses sales

What Is Independence, Formally?

Definition

Two random variables x and y are independent (in some implicit distribution over x and y) if for all values a and b:

$$\Pr(x = a \wedge y = b) = \Pr(x = a) \cdot \Pr(y = b)$$

That is, the distribution over (x, y) is the product of a distribution over x and a distribution over y.

Why Is Independence Useful for Program Reasoning?

Ubiquitous in probabilistic programs

- A “fresh” random sample is independent of the state.

Why Is Independence Useful for Program Reasoning?

Ubiquitous in probabilistic programs

- A “fresh” random sample is independent of the state.

Simplifies reasoning about groups of variables

- Complicated: general distribution over many variables
- Simple: product of distributions over each variable

Why Is Independence Useful for Program Reasoning?

Ubiquitous in probabilistic programs

- A “fresh” random sample is independent of the state.

Simplifies reasoning about groups of variables

- Complicated: general distribution over many variables
- Simple: product of distributions over each variable

Preserved under common program operations

- Local operations independent of “separate” randomness
- Behaves well under conditioning (prob. control flow)

Reasoning about Independence: Challenges

Formal definition isn't very promising

- Quantification over all values: lots of probabilities!
- Computing exact probabilities: often difficult

> How can we leverage the intuition behind probabilistic independence?

Main Observation: Independence is Separation

Two variables x and y in a distribution μ are independent if μ is the product of two distributions μ_x and μ_y with disjoint domains, containing x and y.

Leverage separation logic to reason about independence

- Pioneered by O'Hearn, Reynolds, and Yang
- Highly developed area of program verification research
- Rich logical theory, automated tools, etc.

Our Approach: Two Ingredients

- Develop a probabilistic model of the logic BI

- Design a probabilistic separation logic PSL

Bunched Implications and Separation Logics

What Goes into a Separation Logic?

What Goes into a Separation Logic?

1. Programs
   - Transform input states to output states
   - Done: pWhile with transformer semantics

What Goes into a Separation Logic?

1. Programs
   - Transform input states to output states
   - Done: PWHILE with transformer semantics
2. Assertions
   - Formulas describe pieces of program states
   - Semantics defined by a model of BI (Pym and O'Hearn)

What Goes into a Separation Logic?

1. Programs
   - Transform input states to output states
   - Done: pWHILE with transformer semantics
2. Assertions
   - Formulas describe pieces of program states
   - Semantics defined by a model of BI (Pym and O'Hearn)
3. Program logic
   - Formulas describe programs
   - Assertions specify pre- and post-conditions

Classical Setting: Heaps

Program states (s, h)

- A store $s : \mathcal{X} \to \mathcal{V}$, map from variables to values
- A heap $h : \mathbb{N} \rightharpoonup \mathcal{V}$, partial map from addresses to values

Classical Setting: Heaps

Program states (s, h)

- A **store** $s : \mathcal{X} \to \mathcal{V}$, map from variables to values
- A **heap** $h : \mathbb{N} \rightharpoonup \mathcal{V}$, partial map from addresses to values

Pointer-manipulating programs

- Control flow: sequence, if-then-else, loops
- Read/write addresses in heap
- Allocate/free heap cells

Assertion Logic: Bunched Implications (BI)

Substructural logic (O'Hearn and Pym)

- Start with regular propositional logic $(\top, \bot, \wedge, \vee, \rightarrow)$
- Add a new conjunction ("star"): $P * Q$
- Add a new implication ("magic wand"): $P \mathrel{-\!\!*} Q$

Assertion Logic: Bunched Implications (BI)

Substructural logic (O'Hearn and Pym)

- Start with regular propositional logic $(\top, \bot, \wedge, \vee, \to)$
- Add a new conjunction ("star"): $P * Q$
- Add a new implication ("magic wand"): $P \mathrel{-\!\!*} Q$

Star is a multiplicative conjunction

- $P \wedge Q$: P and Q hold on the entire state
- $P * Q$: P and Q hold on disjoint parts of the entire state

Resource Semantics of BI (O'Hearn and Pym)

Suppose states form a pre-ordered, partial monoid

- Set S of states, pre-order $\sqsubseteq$ on S
- Partial operation $\circ : S \times S \rightharpoonup S$ (assoc., comm., $\ldots$)

Resource Semantics of BI (O'Hearn and Pym)

Suppose states form a pre-ordered, partial monoid

- Set S of states, pre-order $\sqsubseteq$ on S
- Partial operation $\circ : S \times S \rightharpoonup S$ (assoc., comm., …)

Inductively define states that satisfy formulas

Resource Semantics of BI (O'Hearn and Pym)

Suppose states form a pre-ordered, partial monoid

- Set S of states, pre-order $\sqsubseteq$ on S
- Partial operation $\circ : S \times S \rightharpoonup S$ (assoc., comm., …)

Inductively define states that satisfy formulas

$s \models \top$	always
$s \models \bot$	never

Resource Semantics of BI (O'Hearn and Pym)

Suppose states form a pre-ordered, partial monoid

- Set S of states, pre-order $\sqsubseteq$ on S
- Partial operation $\circ : S \times S \rightharpoonup S$ (assoc., comm., ...)

Inductively define states that satisfy formulas

$s \models \top$	always
$s \models \bot$	never
$s \models P \land Q$	iff $s \models P$ and $s \models Q$

Resource Semantics of BI (O'Hearn and Pym)

Suppose states form a pre-ordered, partial monoid

- Set S of states, pre-order $\sqsubseteq$ on S
- Partial operation $\circ : S \times S \rightharpoonup S$ (assoc., comm., ...)

Inductively define states that satisfy formulas

$s \models \top$	always
$s \models \bot$	never
$s \models P \land Q$	iff $s \models P$ and $s \models Q$
$s \models P * Q$	iff $s_1 \circ s_2 \sqsubseteq s$ with $s_1 \models P$ and $s_2 \models Q$

State s can be split into two "disjoint" states, one satisfying P and one satisfying Q

Example: Heap Model of BI

Set of states: heaps

- $S = \mathbb{N} \rightharpoonup \mathcal{V}$, partial maps from addresses to values

Example: Heap Model of BI

Set of states: heaps

- $S = \mathbb{N} \rightharpoonup \mathcal{V}$, partial maps from addresses to values

Monoid operation: combine disjoint heaps

- $s_1 \circ s_2$ is defined to be union iff $\mathsf{dom}(s_1) \cap \mathsf{dom}(s_2) = \emptyset$

Example: Heap Model of BI

Set of states: heaps

- $S = \mathbb{N} \rightharpoonup \mathcal{V}$, partial maps from addresses to values

Monoid operation: combine disjoint heaps

- $s_1 \circ s_2$ is defined to be union iff $\mathrm{dom}(s_1) \cap \mathrm{dom}(s_2) = \emptyset$

Pre-order: extend/project heaps

- $s_1 \sqsubseteq s_2$ iff $\mathrm{dom}(s_1) \subseteq \mathrm{dom}(s_2)$, and s_1, s_2 agree on $\mathrm{dom}(s_1)$

Propositions for Heaps

Atomic propositions: “points-to”

- $x \mapsto v$ holds in heap s iff $x \in \text{dom}(s)$ and $s(x) = v$

Example axioms (not complete)

- Deterministic: $x \mapsto v \land y \mapsto w \land x = y \to v = w$
- Disjoint: $x \mapsto v * y \mapsto w \to x \neq y$

The Separation Logic Proper

Programs c from a basic imperative language

- Read from location: $x := *e$
- Write to location: $*e := e'$

The Separation Logic Proper

Programs c from a basic imperative language

- Read from location: $x := *e$
- Write to location: $*e := e'$

Program logic judgments

$$\{P\}\ c\ \{Q\}$$

Reading

Executing c on any input state satisfying P leads to an output state satisfying Q, without invalid reads or writes.

A Probabilistic Model of BI

States: Distributions over Memories

States: Distributions over Memories

Memories (not heaps)

- Fix sets $\mathcal{X}$ of variables and $\mathcal{V}$ of values
- Memories indexed by domains $A \subseteq \mathcal{X}$: $\mathcal{M}(A) = A \to \mathcal{V}$

States: Distributions over Memories

Memories (not heaps)

- Fix sets $\mathcal{X}$ of variables and $\mathcal{V}$ of values
- Memories indexed by domains $A \subseteq \mathcal{X}$: $\mathcal{M}(A) = A \to \mathcal{V}$

Program states: randomized memories

- States are distributions over memories with same domain
- Formally: $S = \{s \mid s \in \mathsf{Distr}(\mathcal{M}(A)), A \subseteq \mathcal{X}\}$
- When $s \in \mathsf{Distr}(\mathcal{M}(A))$, write $\mathsf{dom}(s)$ for A

Monoid: “Disjoint” Product Distribution

Intuition

- Two distributions can be combined iff domains are disjoint
- Combine by taking product distribution, union of domains

Monoid: “Disjoint” Product Distribution

Intuition

- Two distributions can be combined iff domains are disjoint
- Combine by taking product distribution, union of domains

More formally...

Suppose that $s \in \text{Distr}(\mathcal{M}(A))$ and $s' \in \text{Distr}(\mathcal{M}(B))$. If A, B are disjoint, then:

$$(s \circ s')(m \cup m') = s(m) \cdot s'(m')$$

for $m \in \mathcal{M}(A)$ and $m' \in \mathcal{M}(B)$. Otherwise, $s \circ s'$ is undefined.

Pre-Order: Extension/Projection

Intuition

- Define $s \sqsubseteq s'$ if s "has less information than" s'
- In probabilistic setting: s is a projection of s'

Pre-Order: Extension/Projection

Intuition

- Define $s \sqsubseteq s'$ if s "has less information than" s'
- In probabilistic setting: s is a projection of s'

More formally...

Suppose that $s \in \text{Distr}(\mathcal{M}(A))$ and $s' \in \text{Distr}(\mathcal{M}(B))$. Then $s \sqsubseteq s'$ iff $A \subseteq B$, and for all $m \in \mathcal{M}(A)$, we have:

$$s(m) = \sum_{m' \in \mathcal{M}(B)} s'(m \cup m').$$

That is, s is obtained from s' by marginalizing variables in $B \setminus A$.