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

Please ask questions!

OPLSS Slack: `#probabilistic`

- I will check in periodically for offline questions

Zoom chat/raise hand

- Thanks to Breandan Considine for moderating!

We don't have to get through everything

- We will have to skip over many topics, anyways

Requests are welcome!

- Tell me if you're curious about something not on the menu

Probabilistic Programs

Are Everywhere!

Executable code: Randomized Algorithms

Executable code: Randomized Algorithms

Better performance in exchange for chance of failure

- Check if $n \times n$ matrices $A \cdot B = C$: $O(n^{2.37\cdots})$ operations
- Freivalds' randomized algorithm: $O(n^2)$ operations

Executable code: Randomized Algorithms

Better performance in exchange for chance of failure

- Check if $n \times n$ matrices $A \cdot B = C$: $O(n^{2.37\cdots})$ operations
- Freivalds' randomized algorithm: $O(n^2)$ operations

Improve performance against "worst-case" inputs

- Quicksort: if input is worst-case, $O(n^2)$ comparisons
- Randomized quicksort: $O(n \log n)$ comparisons on average

Executable code: Randomized Algorithms

Better performance in exchange for chance of failure

- Check if $n \times n$ matrices $A \cdot B = C$: $O(n^{2.37\cdots})$ operations
- Freivalds' randomized algorithm: $O(n^2)$ operations

Improve performance against "worst-case" inputs

- Quicksort: if input is worst-case, $O(n^2)$ comparisons
- Randomized quicksort: $O(n \log n)$ comparisons on average

Other benefits

- Randomized algorithms can be simpler to describe
- Sometimes: more efficient than deterministic algorithms

Executable code: Security and Privacy

Executable code: Security and Privacy

Cryptography

- Generate secrets the adversary doesn't know
- Example: draw encryption/decryption keys randomly

Executable code: Security and Privacy

Cryptography

- Generate secrets the adversary doesn't know
- Example: draw encryption/decryption keys randomly

Privacy

- Add random noise to blur private data
- Example: differential privacy

Executable code: Randomized Testing

Executable code: Randomized Testing

Randomly generate inputs to a program

- Search a huge space of potential inputs
- Avoid human bias in selecting testcases

Executable code: Randomized Testing

Randomly generate inputs to a program

- Search a huge space of potential inputs
- Avoid human bias in selecting testcases

Very common strategy for testing programs

- Property-based testing (e.g., QuickCheck)
- Fuzz testing (e.g., AFL, OSS-Fuzz)

Modeling tool: Representing Uncertainty

Modeling tool: Representing Uncertainty

Think of uncertain things as drawn from a distribution

- Example: whether a network link fails or not
- Example: tomorrow's temperature

Modeling tool: Representing Uncertainty

Think of uncertain things as drawn from a distribution

- Example: whether a network link fails or not
- Example: tomorrow's temperature

Different motivation from executable code

- Aim: model some real-world data generation process
- Less important: generating data from this distribution

Modeling tool: Fitting Empirical Data

Modeling tool: Fitting Empirical Data

Foundation of machine learning

- Human designs a model of how data is generated, with unknown parameters
- Based on data collected from the world, infer parameters of the model

Modeling tool: Fitting Empirical Data

Foundation of machine learning

- Human designs a model of how data is generated, with unknown parameters
- Based on data collected from the world, infer parameters of the model

Example: learning the bias of a coin

- Boolean data generated by coin flips
- Unknown parameter: bias of the coin
- Flip coin many times, try to infer the bias

Modeling tool: Approximate Computing

Modeling tool: Approximate Computing

Computing on unreliable hardware

- Hardware operations may occasionally give wrong answer
- Motivation: lower power usage if we allow more errors

Modeling tool: Approximate Computing

Computing on unreliable hardware

- Hardware operations may occasionally give wrong answer
- Motivation: lower power usage if we allow more errors

Model failures as drawn from a distribution

- Run hardware many times, estimate failures rate
- Randomized program describes approximate computing

Main Questions
and Research Directions

What to know about probabilistic programs?

Four general categories

- Semantics
- Verification
- Automation
- Implementation

Semantics: what do programs mean mathematically?

Specify what programs are supposed to do

- Programs may generate complicated distributions
- Desired behavior of programs may not be obvious

Common tools

- Denotational semantics: define program behavior using mathematical concepts from probability theory (distributions, measures, ...)
- Operational semantics: define how programs step

Verification: how to prove programs correct?

Design ways to prove probabilistic program properties

- Target properties can be highly mathematical, subtle
- Goal: reusable techniques to prove these properties

Common tools

- Low-level: interactive theorem provers (e.g., Coq, Agda)
- Higher-level: type systems, Hoare logic, and custom logics

Automation: how to analyze programs automatically?

Prove correctness without human help

- Benefit: don't need any human expertise to run
- Drawback: less expressive than manual techniques

Common tools

- Probabilistic model checking (e.g., PRISM, Storm)
- Abstract interpretation

Implementation: how to run programs efficiently?

Executing a probabilistic program is not always easy

- Especially: in languages supporting conditioning
- Algorithmic insights to execute probabilistic programs

Common tools: sampling algorithms

- Markov Chain Monte Carlo (MCMC)
- Sequential Monte Carlo (SMC)

Important division: conditioning or not?

Important division: conditioning or not?

No conditioning in language

- Semantics is more straightforward
- Easier to implement; closer to executable code
- Verification and automation are more tractable

Important division: conditioning or not?

No conditioning in language

- Semantics is more straightforward
- Easier to implement; closer to executable code
- Verification and automation are more tractable

Yes conditioning in language

- Semantics is more complicated
- Difficult to implement efficiently, but useful for modeling
- Verification and automation are very difficult

Our focus, and the plan (can't cover everything!)

Our focus, and the plan (can't cover everything!)

Primary focus: verification

- Main course goal: reasoning about probabilistic programs

Our focus, and the plan (can't cover everything!)

Primary focus: verification

- Main course goal: reasoning about probabilistic programs

Secondary focus: semantics

- Introduce a few semantics for probabilistic languages

Our focus, and the plan (can't cover everything!)

Primary focus: verification

- Main course goal: reasoning about probabilistic programs

Secondary focus: semantics

- Introduce a few semantics for probabilistic languages

Programs without conditioning

- Simpler, and covers many practical applications

What does semantics have to do with verification?

What does semantics have to do with verification?

Semantics is the foundation of verification

- Semantics: definition of program behavior
- Verification: prove program behavior satisfies property

What does semantics have to do with verification?

Semantics is the foundation of verification

- Semantics: definition of program behavior
- Verification: prove program behavior satisfies property

Semantics can make properties easier or harder to verify

- Probabilistic programs: several natural semantics
- Choice of semantics strongly affects verification

Verifying Probabilistic Programs

What Are the Challenges?

Traditional verification: big code, general proofs

Traditional verification: big code, general proofs

It computes a super-set of the possible run-time errors. ASTRÉE is designed for efficiency on large software: hundreds of thousands of lines of code are analyzed in a matter of hours, while producing very few false alarms. For example, some fly-by-wire avionics reactive control codes (70 000 and 380 000 lines respectively, the latter of a much more complex design) are analyzed in 1 h and 10 h 30' respectively on current single-CPU PCs, with *no false alarm* [1,2,9].

Traditional verification: big code, general proofs

It computes a super-set of the possible run-time errors. ASTRÉE is designed for efficiency on large software: hundreds of thousands of lines of code are analyzed in a matter of hours, while producing very few false alarms. For example, some fly-by-wire avionics reactive control codes (70 000 and 380 000 lines respectively, the latter of a much more complex design) are analyzed in 1 h and 10 h 30' respectively on current single-CPU PCs, with *no false alarm* [1,2,9].

Key lessons for designing static analyses tools deployed to find bugs in hundreds of millions of lines of code.

BY DINO DISTEFANO, MANUEL FÄHNDRICH, FRANCESCO LOGOZZO, AND PETER W. O'HEARN

Scaling Static Analyses at Facebook

Traditional verification: big code, general proofs

It computes a super-set of the possible run-time errors. ASTRÉE is designed for efficiency on large software: hundreds of thousands of lines of code are analyzed in a matter of hours, while producing very few false alarms. For example, some fly-by-wire avionics reactive control codes (70 000 and 380 000 lines respectively, the latter of a much more complex design) are analyzed in 1 h and 10 h 30' respectively on current single-CPU PCs, with *no false alarm* [1,2,9].

Key lessons for designing static analyses tools deployed to find bugs in hundreds of millions of lines of code.

BY DINO DISTEFANO, MANUEL FÄHNDRICH, FRANCESCO LOGOZZO, AND PETER W. O'HEARN

Scaling Static Analyses at Facebook

How Coverity built a bug-finding tool, and a business, around the unlimited supply of bugs in software systems.

BY AL BESSEY, KEN BLOCK, BEN CHELF, ANDY CHOU, BRYAN FULTON, SETH HALLEM, CHARLES HENRI-GROS, ASYA KAMSKY, SCOTT MCPEAK, AND DAWSON ENGLER

A Few Billion Lines of Code Later

Randomized programs: small code, specialized proofs

Small code

- Usually: on the order of 10s of lines of code
- 100-line algorithm: unthinkable (and un-analyzable)

Specialized proofs

- Often: apply combination of known and novel techniques
- Proofs (and techniques) can be research contributions

Simple programs, but complex program states

Programs manipulate distributions over program states

- Each state has a numeric probability
- Probabilities of different states may be totally unrelated

Simple programs, but complex program states

Programs manipulate distributions over program states

- Each state has a numeric probability
- Probabilities of different states may be totally unrelated

Example: program with 10 Boolean variables

- Non-probabilistic programs: $2^{10} = 1024$ possible states
- Probabilistic programs: each state also has a probability
- 1024 possible states versus uncountably many states

Properties are fundamentally quantitative

Properties are fundamentally quantitative

Key probabilistic properties often involve...

- Probabilities of events (e.g., returning wrong result)
- Average value of randomized quantities (e.g., running time)

Properties are fundamentally quantitative

Key probabilistic properties often involve...

- Probabilities of events (e.g., returning wrong result)
- Average value of randomized quantities (e.g., running time)

Can't just "ignore" probabilities

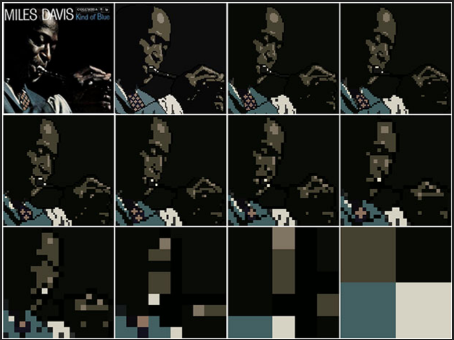
- Treat probabilities as zero or non-zero (non-determinism)
- Simplifies verification, but can't prove most properties

Needed: good abstractions for probabilistic programs

Discard unneeded aspects of a program's state/behavior

Needed: good abstractions for probabilistic programs

Discard unneeded aspects of a program's state/behavior



— Andy Baio, Jay Maisel

What do we want from these abstractions?

Desired features

1. Retain enough info to show target probabilistic properties

What do we want from these abstractions?

Desired features

1. Retain enough info to show target probabilistic properties
2. Be easy to establish (or at least not too difficult)

What do we want from these abstractions?

Desired features

1. Retain enough info to show target probabilistic properties
2. Be easy to establish (or at least not too difficult)
3. Behave well under program composition

Mathematical Preliminaries

Distributions and sub-distributions

Distribution over A assigns a probability to each $a \in A$

Let A be a countable set. A (discrete) distribution over A, $\mu \in \mathsf{Distr}(A)$, is a function $\mu : A \to [0, 1]$ such that:

$$\sum_{a \in A} \mu(a) = 1.$$

For modeling non-termination: sub-distributions

A (discrete) subdistribution over A, $\mu \in \mathsf{SDistr}(A)$, is a function $\mu : A \to [0, 1]$ such that:

$$\sum_{a \in A} \mu(a) \leq 1.$$

"Missing" mass is probability of non-termination.

Examples of distributions

Fair coin: $\mathbf{Flip}$

- Distribution over $\mathbb{B} = \{tt, ff\}$
- $\mu(tt) = \mu(ff) \triangleq 1/2$

Examples of distributions

Fair coin: $\mathbf{Flip}$

- Distribution over $\mathbb{B} = \{tt, ff\}$
- $\mu(tt) = \mu(ff) \triangleq 1/2$

Biased coin: $\mathbf{Flip}(1/4)$

- Distribution over $\mathbb{B} = \{tt, ff\}$
- $\mu(tt) \triangleq 1/4, \mu(ff) \triangleq 3/4$

Examples of distributions

Fair coin: $\mathbf{Flip}$

- Distribution over $\mathbb{B} = \{tt, ff\}$
- $\mu(tt) = \mu(ff) \triangleq 1/2$

Biased coin: $\mathbf{Flip}(1/4)$

- Distribution over $\mathbb{B} = \{tt, ff\}$
- $\mu(tt) \triangleq 1/4, \mu(ff) \triangleq 3/4$

Dice roll: $\mathbf{Roll}$

- Distribution over $\mathbb{N} = \{0, 1, 2, \dots\}$
- $\mu(1) = \cdots = \mu(6) \triangleq 1/6$
- Otherwise: $\mu(n) \triangleq 0$

Notation for distributions

Probability of a set

Let $E \subseteq A$ be an event, and let $\mu \in \mathsf{Distr}(A)$ be a distribution. Then the probability of E in μ is:

$$\mu(E) \triangleq \sum_{x \in E} \mu(x).$$

Notation for distributions

Probability of a set

Let $E \subseteq A$ be an event, and let $\mu \in \mathrm{Distr}(A)$ be a distribution. Then the probability of E in μ is:

$$\mu(E) \triangleq \sum_{x \in E} \mu(x).$$

Expected value

Let $\mu \in \mathrm{Distr}(A)$ be a distribution, and $f : A \to \mathbb{R}^+$ be a non-negative function. Then the expected value of f in μ is:

$$\mathbb{E}_{x \sim \mu}[f(x)] \triangleq \sum_{x \in A} f(a) \cdot \mu(a).$$

Operations on distributions: unit

The simplest possible distribution

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

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 $unit(a) \in \mathsf{Distr}(A)$ is defined to be:

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

Why “unit”? The unit (“return”) of the distribution monad.

Operations on distributions: map

Translate each distribution output to something else

Whenever sample x, sample $f(x)$ instead. Transformation map f is deterministic: function $A \to B$.

Operations on distributions: map

Translate each distribution output to something else

Whenever sample x, sample $f(x)$ instead. Transformation map f is deterministic: function $A \to B$.

Distribution map

Let $f : A \to B$. Then $map(f) : \mathsf{Distr}(A) \to \mathsf{Distr}(B)$ takes $\mu \in \mathsf{Distr}(A)$ to:

$$map(f)(\mu)(b) \triangleq \sum_{a \in A : f(a) = b} \mu(a)$$

Probability of $b \in B$ is sum probability of $a \in A$ mapping to b.

Example: distribution map

Swap results of a biased coin flip

- Let $neg : \mathbb{B} \to \mathbb{B}$ map $tt \mapsto ff$, and $ff \mapsto tt$.
- Then $\mu = map(neg)(\mathbf{Flip}(1/4))$ swaps the results of a biased coin flip.
- By definition of map: $\mu(tt) = 3/4, \mu(ff) = 1/4$.

Example: distribution map

Swap results of a biased coin flip

- Let $neg : \mathbb{B} \to \mathbb{B}$ map $tt \mapsto ff$, and $ff \mapsto tt$.
- Then $\mu = map(neg)(\mathbf{Flip}(1/4))$ swaps the results of a biased coin flip.
- By definition of map: $\mu(tt) = 3/4, \mu(ff) = 1/4$.

Try this at home!

What is the distribution obtained by adding 1 to the result of a dice roll $\mathbf{Roll}$? Compute the probabilities using map.

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 $A \to \text{Distr}(B)$.

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 $bind(\mu, f) \in \text{Distr}(B)$ is defined to be:

$$bind(\mu, f)(b) \triangleq \sum_{a \in A} \mu(a) \cdot f(a)(b)$$

Unpacking the formula for bind

$$bind(\mu, f)(b) \triangleq \sum_{a \in A} \mu(a) \cdot f(a)(b)$$

Unpacking the formula for bind

$$bind(\mu, f)(b) \triangleq \sum_{a \in A} \mu(a) \cdot f(a)(b)$$

Probability of sampling b is ...

1. Sample $a \in A$ from μ: probability $\mu(a)$

Unpacking the formula for bind

$$bind(\mu, f)(b) \triangleq \sum_{a \in A} \mu(a) \cdot f(a)(b)$$

Probability of sampling b is …

1. Sample $a \in A$ from μ: probability $\mu(a)$
2. Sample b from $f(a)$: probability $f(a)(b)$

Unpacking the formula for bind

$$bind(\mu, f)(b) \triangleq \sum_{a \in A} \mu(a) \cdot f(a)(b)$$

Probability of sampling b is ...

1. Sample $a \in A$ from μ: probability $\mu(a)$
2. Sample b from $f(a)$: probability $f(a)(b)$
3. Sum over all possible "intermediate samples" $a \in A$

Example: distribution bind

Summing two dice rolls

- For $n \in \mathbb{N}$, let $f(n) \in \text{Distr}(\mathbb{N})$ be the distribution of adding n to the result of a fair dice roll $\mathbf{Roll}$.
- Then: $\mu = bind(\mathbf{Roll}, f)$ is the distribution of the sum of two fair dice rolls.
- Can check from definition of bind:
  $\mu(2) = (1/6) \cdot (1/6) = 1/36$

Example: distribution bind

Summing two dice rolls

- For $n \in \mathbb{N}$, let $f(n) \in \text{Distr}(\mathbb{N})$ be the distribution of adding n to the result of a fair dice roll $\mathbf{Roll}$.
- Then: $\mu = bind(\mathbf{Roll}, f)$ is the distribution of the sum of two fair dice rolls.
- Can check from definition of bind: $\mu(2) = (1/6) \cdot (1/6) = 1/36$

Try this at home!

- Define f in terms of distribution map.
- What if you try to define μ with map instead of bind?

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?

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!

Example: conditioning

Rolling a dice until even number

Suppose we repeatedly roll a dice until it produces an even number. What distribution over even numbers will we get?

Example: conditioning

Rolling a dice until even number

Suppose we repeatedly roll a dice until it produces an even number. What distribution over even numbers will we get?

Model as a conditional distribution

- Let $E = \{2, 4, 6\}$
- Resulting distribution is $\mu = (\mathbf{Roll} \mid E)$
- From definition of conditioning: $\mu(2) = \mu(4) = \mu(6) = 1/3$

Try this at home!

Suppose we keep rolling two dice until the sum of the dice is 6 or larger. What is the distribution of the final sum?

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.

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

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

Example: convex combination

Blend two biased coin flips

- Let $\mu_1 = \mathbf{Flip}(1/4), \mu_2 = \mathbf{Flip}(3/4)$
- From definition of mixing, $\mu_1 \oplus_{1/2} \mu_2$ is a fair coin $\mathbf{Flip}$

Example: convex combination

Blend two biased coin flips

- Let $\mu_1 = \mathbf{Flip}(1/4), \mu_2 = \mathbf{Flip}(3/4)$
- From definition of mixing, $\mu_1 \oplus_{1/2} \mu_2$ is a fair coin $\mathbf{Flip}$

Try this at home!

- Show that $\mathbf{Flip}(r) \oplus_p \mathbf{Flip}(s) = \mathbf{Flip}(p \cdot r + (1 - p) \cdot s)$.
- Show this relation between mixing and conditioning:

$$\mu = (\mu \mid E) \oplus_{\mu(E)} (\mu \mid \overline{E})$$

Operations on distributions: independent product

Distribution of two “fresh” samples

Common operation in probabilistic programming languages: draw a sample, and then draw another, “fresh” sample.

Operations on distributions: independent product

Distribution of two “fresh” samples

Common operation in probabilistic programming languages: draw a sample, and then draw another, “fresh” sample.

Independent product

Let $\mu_1 \in \text{Distr}(A_1)$ and $\mu_2 \in \text{Distr}(A_2)$. Then the independent product is the distribution in $\text{Distr}(A_1 \times A_2)$ defined by:

$$(\mu_1 \otimes \mu_2)(a_1, a_2) \triangleq \mu_1(a_1) \cdot \mu_2(a_2).$$

Example: independent product

Distribution of two fair coin flips

- Let $\mu_1 = \mu_2 = \mathbf{Flip}$
- Then distribution of pair of fair coin flips is $\mu = \mu_1 \otimes \mu_2$
- By definition, can show $\mu(b_1, b_2) = (1/2) \cdot (1/2) = 1/4$.

Example: independent product

Distribution of two fair coin flips

- Let $\mu_1 = \mu_2 = \mathbf{Flip}$
- Then distribution of pair of fair coin flips is $\mu = \mu_1 \otimes \mu_2$
- By definition, can show $\mu(b_1, b_2) = (1/2) \cdot (1/2) = 1/4$.

Try this at home!

- Show that $unit(a_1) \otimes unit(a_2) = unit((a_1, a_2))$.
- Can you formulate and prove an interesting property relating independent product and distribution bind?

Our First Probabilistic Language

Probabilistic WHILE (pWhile)

pWHILE by Example

The language, in a nutshell

- Core imperative WHILE-language
- Assignment, sequencing, if-then-else, while-loops
- Main extension: a command for random sampling $x \xleftarrow{\$} d$, where d is a built-in distribution

pWHILE by Example

The language, in a nutshell

- Core imperative WHILE-language
- Assignment, sequencing, if-then-else, while-loops
- Main extension: a command for random sampling $x \xleftarrow{\$} d$, where d is a built-in distribution

Can you guess what this program does?

$$
\begin{aligned}
&x \xleftarrow{\$} \mathbf{Roll};\\
&y \xleftarrow{\$} \mathbf{Roll};\\
&z \leftarrow x + y
\end{aligned}
$$

pWhile by Example

Control flow can be probabilistic

- Branches can depend on random samples
- Challenge for verification: can't do a simple case analysis
- In some sense, an execution takes both branches

pWhile by Example

Control flow can be probabilistic

- Branches can depend on random samples
- Challenge for verification: can't do a simple case analysis
- In some sense, an execution takes both branches

Can you guess what this program does?

$$\begin{array}{l} choice \xleftarrow{\$} \mathbf{Flip}; \\ \text{if } choice \text{ then} \\ \quad res \xleftarrow{\$} \mathbf{Flip}(1/4) \\ \text{else} \\ \quad res \xleftarrow{\$} \mathbf{Flip}(3/4) \end{array}$$

pWhile by Example

Loops can also be probabilistic

- Number of iterations can be randomized
- Termination can be probabilistic

pWhile by Example

Loops can also be probabilistic

- Number of iterations can be randomized
- Termination can be probabilistic

Can you guess what this program does?

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

More formally: pWhile expressions

Grammar of boolean and numeric expressions

$$
\begin{aligned}
\mathcal{E} \ni e \coloneqq\ & x \in \mathcal{X} && \text{(variables)} \\
& \mid b \in \mathbb{B} \mid \mathcal{E} > \mathcal{E} \mid \mathcal{E} = \mathcal{E} && \text{(booleans)} \\
& \mid n \in \mathbb{N} \mid \mathcal{E} + \mathcal{E} \mid \mathcal{E} \cdot \mathcal{E} && \text{(numbers)}
\end{aligned}
$$

Basic expression language

- Expression language can be extended if needed
- Assume: programs only use well-typed expressions

More formally: pWHILE d-expressions

Grammar of d-expressions

$$\begin{aligned} \mathcal{DE} \ni d &:= \mathbf{Flip} && \text{(fair coin flip)} \\ &\mid \mathbf{Flip}(p) && \text{(p-biased coin flip, $p \in [0,1]$)} \\ &\mid \mathbf{Roll} && \text{(fair dice roll)} \end{aligned}$$

“Built-in” or “primitive” distributions

- Distributions can be extended if needed
- “Mathematically standard” distributions
- Distributions that can be sampled from in hardware

More formally: pWHILE commands

Grammar of commands

$$
\begin{aligned}
\mathcal{C} \ni c &:= \mathsf{skip} && \text{(do nothing)}\\
&\mid \mathcal{X} \leftarrow \mathcal{E} && \text{(assignment)}\\
&\mid \mathcal{X} \xleftarrow{\$} \mathcal{DE} && \text{(sampling)}\\
&\mid \mathcal{C}\,;\,\mathcal{C} && \text{(sequencing)}\\
&\mid \mathsf{if}\ \mathcal{E}\ \mathsf{then}\ \mathcal{C}\ \mathsf{else}\ \mathcal{C} && \text{(if-then-else)}\\
&\mid \mathsf{while}\ \mathcal{E}\ \mathsf{do}\ \mathcal{C} && \text{(while-loop)}
\end{aligned}
$$

Imperative language with sampling

- Bare-bones imperative language
- Many possible extensions: procedures, pointers, etc.

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: PWHILE programs

Can you guess what this program does?

$$
\begin{array}{l}
r \leftarrow 0; \\
\text{while } r < 4 \text{ do} \\
\quad r \xleftarrow{\$} \mathbf{Roll}
\end{array}
$$

Last time: PWHILE programs

Can you guess what this program does?

$$\begin{array}{l} r \leftarrow 0; \\ \text{while } r < 4 \text{ do} \\ \quad r \xleftarrow{\$} \mathbf{Roll} \end{array}$$

Uniform sample from $\{4, 5, 6\}$

- Start with dice roll, condition on $r \geq 4$

More formally: pWHILE expressions

Grammar of boolean and numeric expressions

$$
\begin{aligned}
\mathcal{E} \ni e := x \in \mathcal{X} & & \text{(variables)} \\
\mid b \in \mathbb{B} \mid \mathcal{E} > \mathcal{E} \mid \mathcal{E} = \mathcal{E} & & \text{(booleans)} \\
\mid n \in \mathbb{N} \mid \mathcal{E} + \mathcal{E} \mid \mathcal{E} \cdot \mathcal{E} & & \text{(numbers)}
\end{aligned}
$$

Basic expression language

- Expression language can be extended if needed
- Assume: programs only use well-typed expressions

More formally: pWHILE d-expressions

Grammar of d-expressions

$$\mathcal{DE} \ni d := \mathbf{Flip} \qquad \text{(fair coin flip)}$$
$$\mid \mathbf{Flip}(p) \qquad \text{(p-biased coin flip, $p \in [0,1]$)}$$
$$\mid \mathbf{Roll} \qquad \text{(fair dice roll)}$$

“Built-in” or “primitive” distributions

- Distributions can be extended if needed
- “Mathematically standard” distributions
- Distributions that can be sampled from in hardware

More formally: pWHILE commands

Grammar of commands

$$
\begin{aligned}
\mathcal{C} \ni c ::= {} & \mathsf{skip} && \text{(do nothing)} \\
\mid {} & \mathcal{X} \leftarrow \mathcal{E} && \text{(assignment)} \\
\mid {} & \mathcal{X} \overset{\$}{\leftarrow} \mathcal{DE} && \text{(sampling)} \\
\mid {} & \mathcal{C} \,;\, \mathcal{C} && \text{(sequencing)} \\
\mid {} & \mathsf{if}\ \mathcal{E}\ \mathsf{then}\ \mathcal{C}\ \mathsf{else}\ \mathcal{C} && \text{(if-then-else)} \\
\mid {} & \mathsf{while}\ \mathcal{E}\ \mathsf{do}\ \mathcal{C} && \text{(while-loop)}
\end{aligned}
$$

Imperative language with sampling

- Bare-bones imperative language
- Many possible extensions: procedures, pointers, etc.

A First Semantics for pWhile

Monadic Semantics

Program states

Programs modify memories

- Memories m assign a value $v \in \mathcal{V}$ to each variable $x \in \mathcal{X}$
- Just like memories in imperative languages

Program states

Programs modify memories

- Memories m assign a value $v \in \mathcal{V}$ to each variable $x \in \mathcal{X}$
- Just like memories in imperative languages

More formally:

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

Semantics of expressions

The value of an expression depends on the memory

- Example: value of $x + 1$ depends on the memory m
- Semantics of expressions takes memory as parameter

Semantics of expressions

The value of an expression depends on the memory

- Example: value of $x + 1$ depends on the memory m
- Semantics of expressions takes memory as parameter

More formally:

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

For example:

- Expression $x + 1$
- Memory m with $m(x) = 3$
- $[\![x + 1]\!]m \triangleq [\![x]\!]m + [\![1]\!]m \triangleq m(x) + 1 = 3 + 1 = 4$

Semantics of distributions

Semantics of d-expression is distribution over values

- From d-expression to a (mathematical) distribution
- (Easy) extension: d-expression with parameters

More formally:

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

For example:

- D-expression $\mathbf{Flip}$
- $[\![\mathbf{Flip}]\!] \triangleq \mu \in \mathsf{Distr}(\mathbb{B})$, where $\mu(tt) = \mu(f\!f) = 1/2$

Monadic semantics of commands: overview

First choice:

1. Command takes a memory as input, or:
2. Command takes a distribution over memories as input?

Monadic semantics of commands: overview

First choice:

1. Command takes a memory as input, or:
2. Command takes a distribution over memories as input?

Monadic semantics of commands: overview

First choice:

1. Command takes a memory as input, or:
2. Command takes a distribution over memories as input?

This lecture: monadic semantics

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

Command: input memory to output distribution over memories.

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 $unit(a) \in \mathsf{Distr}(A)$ is defined to be:

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

Why “unit”? The unit (“return”) of the distribution monad.

Semantics of commands: skip

Intuition

- Input: memory m
- Output: distribution that always returns m

Semantics of commands: skip

Intuition

- Input: memory m
- Output: distribution that always returns m

Semantics of skip

$$(\!|\mathsf{skip}|\!)m \triangleq unit(m)$$

Semantics of commands: assignment

Intuition

- Input: memory m
- Output: distribution that always returns m with $x \mapsto v$, where v is the original value of e in m.

Semantics of commands: assignment

Intuition

- Input: memory m
- Output: distribution that always returns m with $x \mapsto v$, where v is the original value of e in m.

Semantics of assignment

Let $v \triangleq [\![e]\!]m$. Then:

$$(\!|x \leftarrow e|\!)m \triangleq unit(m[x \mapsto v])$$

Operations on distributions: map

Translate each distribution output to something else

Whenever sample x, sample $f(x)$ instead. Transformation map f is deterministic: function $A \to B$.

Distribution map

Let $f : A \to B$. Then $map(f) : \text{Distr}(A) \to \text{Distr}(B)$ takes $\mu \in \text{Distr}(A)$ to:

$$map(f)(\mu)(b) \triangleq \sum_{a \in A : f(a) = b} \mu(a)$$

Probability of $b \in B$ is sum probability of $a \in A$ mapping to b.

Semantics of commands: sampling

Intuition

- Input: memory m
- Draw sample from $[\![d]\!]$, call it v
- Given v, map to updated output memory $m[x \mapsto v]$

Semantics of commands: sampling

Intuition

- Input: memory m
- Draw sample from $[\![d]\!]$, call it v
- Given v, map to updated output memory $m[x \mapsto v]$

Semantics of sampling

Let $f(v) \triangleq m[x \mapsto v]$. Then:

$$(\!| x \xleftarrow{\$} d |\!) m \triangleq map(f)([\![d]\!])$$

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 $bind(\mu, f) \in \text{Distr}(B)$ is defined to be:

$$bind(\mu, f)(b) \triangleq \sum_{a \in A} \mu(a) \cdot f(a)(b)$$

Semantics of commands: sequencing

Intuition

- Input: memory m
- Run first command, get distribution μ_1
- Sample m' from μ_1, bind into second command

Semantics of commands: sequencing

Intuition

- Input: memory m
- Run first command, get distribution μ_1
- Sample m' from μ_1, bind into second command

Semantics of sequencing

$$(\!| c_1 \ ; \ c_2 |\!) m \triangleq bind((\!| c_1 |\!) m, (\!| c_2 |\!))$$

Semantics of commands: conditionals

Intuition

- Input: memory m
- If guard is true in m run c_1, else run c_2
- Note: m is a memory, not a distribution!

Semantics of commands: conditionals

Intuition

- Input: memory m
- If guard is true in m run c_1, else run c_2
- Note: m is a memory, not a distribution!

Semantics of conditionals

$$(\!|\text{if } e \text{ then } c_1 \text{ else } c_2|\!)m \triangleq \begin{cases} (\!|c_1|\!)m & : [\![e]\!]m = tt \\ (\!|c_2|\!)m & : [\![e]\!]m = \mathit{ff} \end{cases}$$

Semantics of loops: first try

Intuition

- Input: memory m
- Idea: while e do c should be sequence of if-then-else:

$$(\text{if } e \text{ then } c); \cdots ; (\text{if } e \text{ then } c)$$

Semantics of loops: first try

Intuition

- Input: memory m
- Idea: while e do c should be sequence of if-then-else:

$$(\text{if } e \text{ then } c); \cdots ; (\text{if } e \text{ then } c)$$

- Define loop semantics as limit?

$$(\!|\text{while } e \text{ do } c|\!)m \stackrel{?}{=} \lim_{n\to\infty} (\!|(\text{if } e \text{ then } c)^n|\!)m$$

Semantics of loops: first try

Intuition

- Input: memory m
- Idea: while e do c should be sequence of if-then-else:

$$(\text{if } e \text{ then } c); \cdots ; (\text{if } e \text{ then } c)$$

- Define loop semantics as limit?

$$(\!|\text{while } e \text{ do } c|\!)m \stackrel{?}{=} \lim_{n\to\infty} (\!|(\text{if } e \text{ then } c)^n|\!)m$$

What does this limit mean?

- Say $\mu_n \triangleq (\!|(\text{if } e \text{ then } c)^n|\!)m$
- Each μ_n is a distribution in $\text{Distr}(\mathcal{M})$. Does limit exist?

Intuitive loop semantics: limit may not exist!

Simple example: flipper

$$\text{while } tt \text{ do if } x \text{ then } x \leftarrow \mathit{ff} \text{ else } x \leftarrow tt$$

What does this program do?

Intuitive loop semantics: limit may not exist!

Simple example: flipper

$$\text{while } tt \text{ do if } x \text{ then } x \leftarrow \mathit{ff} \text{ else } x \leftarrow tt$$

What does this program do?

Repeatedly changes x to tt and ff

- Suppose input m has $m(x) = tt$
- Can verify: $\mu_n = (\!|(\text{if } e \text{ then } c)^n|\!)m$ has all mass on m for even n, and all mass on $m[x \mapsto \mathit{ff}]$ for odd n
- Oscillates: no sensible limit!

Semantics of loops: approximants

Problem with the flipper example: loop not terminating

- Idea: only "count" probability mass that has terminated
- Why? Once loop terminates, it is always terminated
- Terminated states can't oscillate: values remain constant

Semantics of loops: approximants

Problem with the flipper example: loop not terminating

- Idea: only "count" probability mass that has terminated
- Why? Once loop terminates, it is always terminated
- Terminated states can't oscillate: values remain constant

More formally...

- For $\mu \in \text{Distr}(\mathcal{M})$, define:

$$\mu[e](m) \triangleq \begin{cases} \mu(m) & : [\![e]\!]m = tt \\ 0 & : \text{otherwise} \end{cases}$$

- Erase weight of memories where $e = f\!f$ (not conditioning)

Semantics of loops: limit of approximants

Loop approximants

Idea: mass that has terminated after n iterations

$$\mu_n \triangleq ((\!|(\text{if } e \text{ then } c)^n|\!) m)[\neg e]$$

Sub-distributions μ_n are increasing in n: for any m',

$$\mu_n(m') \leq \mu_{n+1}(m').$$

Thus limit exists!

Semantics of loops: limit of approximants

Loop approximants

Idea: mass that has terminated after n iterations

$$\mu_n \triangleq (\llparenthesis(\text{if } e \text{ then } c)^n\rrparenthesis m)[\neg e]$$

Sub-distributions μ_n are increasing in n: for any m',

$$\mu_n(m') \leq \mu_{n+1}(m').$$

Thus limit exists!

Finally: define loop semantics

$$\llparenthesis\text{while } c \text{ do } e\rrparenthesis m \triangleq \lim_{n\to\infty} \mu_n$$

Semantics of loops: example

Consider this loop:

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

Suppose input memory m has $m(t) = 0, m(stop) = \mathit{ff}$

Semantics of loops: example

Consider this loop:

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

Suppose input memory m has $m(t) = 0, m(stop) = ff$

- After 1 iters: terminates with prob. $1/4$ with $t = 1$
- After 2 iters: terminates with prob. $3/4 \cdot 1/4$ with $t = 2$
- After n iters: terminates with prob. $(3/4)^{n-1} \cdot 1/4$ with $t = n$

Thus approximants are:

$$\mu_n([\![t = k]\!]) = (3/4)^{k-1} \cdot 1/4$$

for $k = 1, \ldots, n$. Taking limit as $n \to \infty$ gives loop semantics.

Reasoning about pWhile Programs

Weakest Pre-Expectation Calculus

Standard programs: Weakest Pre-conditions (Dijkstra)

Given a program and a post-condition, find pre-condition

- Given: program c and post-condition Q
- Find $wp(c, Q)$: general pre-condition that ensures Q holds

To check Q on output, check $wp(c, Q)$ on input

- If input state m satisfies $wp(c, Q)$, then $[\![c]\!]m$ satisfies Q

Example: Weakest Pre-conditions

Example

- Program: $x \leftarrow y$
- Post-condition: $x > 0$

What is the wp?

Example: Weakest Pre-conditions

Example

- Program: $x \leftarrow y$
- Post-condition: $x > 0$

What is the wp?

Answer: $wp(x \leftarrow y, x > 0) = (y > 0)$

Why?

Condition $y > 0$ is the least we need to ensure that $x > 0$ holds after running $x \leftarrow y$.

Example: Weakest Pre-conditions

Example

- Program: $x \leftarrow y; x \leftarrow x + 1$
- Post-condition: $x > 0$

What is the wp?

Example: Weakest Pre-conditions

Example

- Program: $x \leftarrow y; x \leftarrow x + 1$
- Post-condition: $x > 0$

What is the wp?

Answer: $wp(x \leftarrow y; x \leftarrow x + 1, x > 0) = (y > -1)$

Why?

Condition $y > -1$ is the least we need to ensure that $x > 0$ holds after running $x \leftarrow y; x \leftarrow x + 1$.

Example: Weakest Pre-conditions

Example

- Program: if $z > 0$ then $x \leftarrow y; x \leftarrow x + 1$ else $x \leftarrow 5$
- Post-condition: $x > 0$

What is the wp?

Example: Weakest Pre-conditions

Example

- Program: if $z > 0$ then $x \leftarrow y; x \leftarrow x + 1$ else $x \leftarrow 5$
- Post-condition: $x > 0$

What is the wp?

Possible to work out by hand, but getting a bit cumbersome...

How to make computing WP easier?

Idea: compute WP compositionally

- WP of complex command defined in terms of WP for sub-commands

Benefits

- Simplify computation of WP for complicated programs
- WP can be computed “mechanically” (and automatically)

WP Calculus: Skip

Intuition

- Program: skip
- Post-condition: Q
- To ensure Q holds after, Q must hold before

WP Calculus: Skip

Intuition

- Program: skip
- Post-condition: Q
- To ensure Q holds after, Q must hold before

WP for Skip

$$wp(\mathsf{skip}, Q) = Q$$

WP Calculus: Assignment

Intuition

- Program: $x \leftarrow e$
- Post-condition: Q
- To ensure Q holds after, Q with $x \mapsto e$ must hold before

WP Calculus: Assignment

Intuition

- Program: $x \leftarrow e$
- Post-condition: Q
- To ensure Q holds after, Q with $x \mapsto e$ must hold before

WP for Assignment

$$wp(x \leftarrow e, Q) = Q[x \mapsto e]$$

WP Calculus: Assignment

Intuition

- Program: $x \leftarrow e$
- Post-condition: Q
- To ensure Q holds after, Q with $x \mapsto e$ must hold before

WP for Assignment

$$wp(x \leftarrow e, Q) = Q[x \mapsto e]$$

Brief check

$$wp(x \leftarrow x + 1, x > 0) = (x + 1 > 0) = (x > -1)$$

WP Calculus: Sequencing

Intuition

- Program: $c_1 \ ; c_2$
- Post-condition: Q
- To ensure Q holds after c_2, $wp(c_2, Q)$ must hold after c_1
- To ensure $wp(c_2, Q)$ holds after c_1, compute another wp

WP Calculus: Sequencing

Intuition

- Program: $c_1 \; ; c_2$
- Post-condition: Q
- To ensure Q holds after c_2, $wp(c_2, Q)$ must hold after c_1
- To ensure $wp(c_2, Q)$ holds after c_1, compute another wp

WP for Sequencing

$$wp(c_1 \; ; c_2, Q) = wp(c_1, wp(c_2, Q))$$

WP Calculus: Conditionals

Intuition

- Program: if e then c_1 else c_2
- Post-condition: Q
- To ensure Q holds after, $wp(c_1, Q)$ must hold before if $e = tt$, and $wp(c_2, Q)$ must hold before if $e = ff$

WP Calculus: Conditionals

Intuition

- Program: if e then c_1 else c_2
- Post-condition: Q
- To ensure Q holds after, $wp(c_1, Q)$ must hold before if $e = tt$, and $wp(c_2, Q)$ must hold before if $e = ff$

WP for Conditionals

$$wp(\text{if } e \text{ then } c_1 \text{ else } c_2, Q) = (e \rightarrow wp(c_1, Q)) \land (\neg e \rightarrow wp(c_2, Q))$$

Example: using the WP calculus

Example

- Program: if $z > 0$ then $x \leftarrow y; x \leftarrow x + 1$ else $x \leftarrow 5$
- Post-condition: $x > 0$

Example: using the WP calculus

Example

- Program: if $z > 0$ then $x \leftarrow y; x \leftarrow x + 1$ else $x \leftarrow 5$
- Post-condition: $x > 0$

What is the wp? A bit ugly, but entirely mechanical:

$$
\begin{aligned}
& wp(\text{if } z > 0 \text{ then } x \leftarrow y; x \leftarrow x + 1 \text{ else } x \leftarrow 5, x > 0) \\
&= (z > 0 \rightarrow wp(x \leftarrow y; x \leftarrow x + 1, x > 0)) \\
&\quad \wedge (z \leq 0 \rightarrow wp(x \leftarrow 5, x > 0)) \\
&= (z > 0 \rightarrow wp(x \leftarrow y; x > -1)) \wedge (z \leq 0 \rightarrow 5 > 0)) \\
&= (z > 0 \rightarrow y > -1)
\end{aligned}
$$

What is WP for loops?

Problem: WP for loops is not easy to compute

- Defined in terms of a least fixed-point
- Might have to unroll loop arbitrarily far to compute wp

What is WP for loops?

Problem: WP for loops is not easy to compute

- Defined in terms of a least fixed-point
- Might have to unroll loop arbitrarily far to compute wp

Idea: we often don't need to compute WP for loops

- Just want to know: does P imply $wp(\text{while } e \text{ do } c, Q)$?
- Use simpler, sufficient conditions to prove this implication

WP for loops: invariant rule

Setup

- Program while e do c
- Pre-condition P, post-condition Q

WP for loops: invariant rule

Setup

- Program while e do c
- Pre-condition P, post-condition Q

If we know I satisfying the invariant conditions...

- $P \to I$
- $I \land \neg e \to Q$
- $I \land e \to wp(c, I)$

then we are done:

$$P \to wp(\text{while } e \text{ do } c, Q)$$

WP for loops: invariant rule

Setup

- Program while e do c
- Pre-condition P, post-condition Q

If we know I satisfying the invariant conditions...

- $P \to I$
- $I \land \neg e \to Q$
- $I \land e \to wp(c, I)$

then we are done:

$$P \to wp(\text{while } e \text{ do } c, Q)$$

What's the catch? Need to magically find an invariant I

- Invariant conditions are easy to check

Example: using the invariant rule

Example

- Program: while $n > 0$ do $n \leftarrow n - 2$
- Pre-condition: P is $n\%2 = 0 \wedge n \geq 0$ (n is even)
- Post-condition: Q is $n = 0$

Example: using the invariant rule

Example

- Program: while $n > 0$ do $n \leftarrow n - 2$
- Pre-condition: P is $n\%2 = 0 \wedge n \geq 0$ (n is even)
- Post-condition: Q is $n = 0$

Invariant:

$$I = (n > 0 \rightarrow n\%2 = 0) \wedge (n \leq 0 \rightarrow n = 0)$$

Example: using the invariant rule

Example

- Program: while $n > 0$ do $n \leftarrow n - 2$
- Pre-condition: P is $n\%2 = 0 \wedge n \geq 0$ (n is even)
- Post-condition: Q is $n = 0$

Invariant:

$$I = (n > 0 \rightarrow n\%2 = 0) \wedge (n \leq 0 \rightarrow n = 0)$$

Check these invariant conditions:

- $P \rightarrow I$
- $I \wedge \neg e \rightarrow Q$
- $I \wedge e \rightarrow wp(c, I)$

Generalizing Weakest Preconditions to Probabilistic Programs

Idea: generalize predicates to expectations

“Real-valued” version of predicates

- Predicate: $P : \mathcal{M} \to \mathbb{B}$
- Expectation: $E : \mathcal{M} \to \mathbb{R}^+$

Idea: generalize predicates to expectations

“Real-valued” version of predicates

- Predicate: $P : \mathcal{M} \to \mathbb{B}$
- Expectation: $E : \mathcal{M} \to \mathbb{R}^+$

Example: numeric expression

- If x, y, z are numeric, then they are all expectations
- Also expressions like $x + y, x \cdot y, \ldots$

Idea: generalize predicates to expectations

"Real-valued" version of predicates

- Predicate: $P : \mathcal{M} \to \mathbb{B}$
- Expectation: $E : \mathcal{M} \to \mathbb{R}^+$

Example: numeric expression

- If x, y, z are numeric, then they are all expectations
- Also expressions like $x + y, x \cdot y, \ldots$

Example: indicator function

- If P is a (binary) predicate, then the indicator function is:

$$[P](m) = \begin{cases} 1 & : P(m) = tt \\ 0 & : P(m) = f\!f \end{cases}$$

- Turns a predicate into an expectation

What do expectations “mean” in a probabilistic state?

Intuition

- The “value” of a predicate P in a memory m is $[P](m)$: 0 if false, and 1 if true.
- The “value” of an expectation E in a distribution over memories μ is the average of E over μ.

Example: encoding a probability as an expectation

Suppose that:

- μ is a distribution over memories
- E is the expectation $[x = y]$

Then we have:

The probability of $x = y$ in μ is the average of E over μ.

Example: encoding an average as an expectation

Suppose that:

- μ is a distribution over memories
- E is the expectation t, where t is the running time

Then we have:

The average running time in μ is the average of E over μ.

Weakest pre-expectation (Morgan and McIver)

Looks similar to weakest pre-conditions

- Given: probabilistic program c and expectation E
- Find $wpe(c, E)$: an expectation that computes the average value of E in the output distribution after running c

To find average value of E after, evaluate $wpe(c, E)$

- For any input state m, the average value of E in the output distribution $(\!|c|\!)m$ is exactly $wpe(c, E)(m)$.

Tailored to the monadic semantics for pWHILE

Key property satisfied by wpe

For any program c, expectation E, and input memory m:

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

Tailored to the monadic semantics for pWhile

Key property satisfied by wpe

For any program c, expectation E, and input memory m:

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

Expectation evaluated on input

- Input is a single memory m
- Evaluate expectation on the memory

Tailored to the monadic semantics for pWHILE

Key property satisfied by wpe

For any program c, expectation E, and input memory m:

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

Expectation evaluated on input

- Input is a single memory m
- Evaluate expectation on the memory

Expectation evaluated on output

- Output is a distribution over memories $(\!|c|\!) m$
- Average the expectation over the output distribution

Example: Reasoning with Weakest Pre-expectation

Example

- Program: $z \xleftarrow{\$} \mathbf{Flip}(p)$
- Expectation: $[z]$

What is the wpe?

Example: Reasoning with Weakest Pre-expectation

Example

- Program: $z \xleftarrow{\$} \mathbf{Flip}(p)$
- Expectation: $[z]$

What is the wpe?

Answer: $wpe(z \xleftarrow{\$} \mathbf{Flip}(p), [z]) = p$

Why?

Average value of $[z]$ after running $z \xleftarrow{\$} \mathbf{Flip}(p)$ is the probability that $z = tt$, which is p.

Example: Reasoning with Weakest Pre-expectation

Example

- Program: $x \xleftarrow{\$} \mathbf{Roll}; y \xleftarrow{\$} \mathbf{Roll}$
- Expectation: $x + y$

What is the wpe?

Example: Reasoning with Weakest Pre-expectation

Example

- Program: $x \xleftarrow{\$} \mathbf{Roll}; y \xleftarrow{\$} \mathbf{Roll}$
- Expectation: $x + y$

What is the wpe?

Answer: $wpe(x \xleftarrow{\$} \mathbf{Roll}; y \xleftarrow{\$} \mathbf{Roll}, x + y) = 7$

Why? Already not so easy to see...

Average value of $x + y$ after running $x \xleftarrow{\$} \mathbf{Roll}; y \xleftarrow{\$} \mathbf{Roll}$ is the average value of x plus the average value of y, which is $3.5 + 3.5 = 7$.

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

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

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 $e = f\!f$

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 $e = \mathit{ff}$

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:

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

Post-expectation: n

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

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:

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

Post-expectation: t

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

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

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

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

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:

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

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

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

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

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

$\textsc{Fair}$: checking the super-invariant condition

Apply the wpe calculus rules

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

FAIR: checking the super-invariant condition

Apply the *wpe* calculus rules

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

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

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

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

Fair: checking the super-invariant condition

Apply the wpe calculus rules

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

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

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

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

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

$$= [x = y] \cdot (p \cdot p \cdot 0.5 + p \cdot (1 - p) \cdot 1$$

$$+ p \cdot (1 - p) \cdot 0 + (1 - p)^2 \cdot 0.5) + [x \neq y] \cdot [x]$$

$$= [x = y] + [x \neq y] \cdot [x] \leq I$$

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:

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

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

$$\llbracket c_1 \ ; \ c_2 \rrbracket \mu \triangleq \llbracket c_2 \rrbracket (\llbracket c_1 \rrbracket \mu)$$

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 $\mu \in \mathsf{Distr}(A)$, and $E \subseteq A$. Then μ conditioned on E is the distribution in $\mathsf{Distr}(A)$ 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 $\mu_1, \mu_2 \in \text{Distr}(A)$, 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:

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

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 $\mu_1, \mu_2, \dots$
- 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., $\ldots$)

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
$s \models P \wedge Q$	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 $\mathrm{dom}(s_1) \cap \mathrm{dom}(s_2) = \emptyset$

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 $x \in \mathrm{dom}(s)$ and $s(x) = v$

Example axioms (not complete)

- Deterministic: $x \mapsto v \wedge y \mapsto w \wedge x = y \rightarrow v = w$
- Disjoint: $x \mapsto v * y \mapsto w \rightarrow x \neq y$

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 $B \setminus A$.

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

Atomic Formulas

Equalities

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

Atomic Formulas

Equalities

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

Distribution laws

- $[e]$ holds in s iff all variables in $FV(e) \subseteq \mathrm{dom}(s)$
- $\mathbf{Unif}_S[e]$ holds in s iff $FV(e) \subseteq \mathrm{dom}(s)$, and e is uniformly distributed on S (e.g., $S = \mathbb{B}$ is fair coin flip)

Example: Distribution Assertions

Suppose μ has two variables x, y, indep. fair coin flips

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

Example: Distribution Assertions

Suppose μ has two variables x, y, indep. fair coin flips

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

Then: μ satisfies $\mathbf{Unif}_{\mathbb{B}}[x] * \mathbf{Unif}_{\mathbb{B}}[y]$. Why?

Example: Distribution Assertions

Suppose μ has two variables x, y, indep. fair coin flips

$$\mu([x \mapsto tt, y \mapsto tt]) = 1/4 \qquad \mu([x \mapsto tt, y \mapsto f\!f]) = 1/4$$
$$\mu([x \mapsto f\!f, y \mapsto tt]) = 1/4 \qquad \mu([x \mapsto f\!f, y \mapsto f\!f]) = 1/4$$

Then: μ satisfies $\mathbf{Unif}_{\mathbb{B}}[x] * \mathbf{Unif}_{\mathbb{B}}[y]$. Why?

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

$$\mu_x([x \mapsto tt]) \triangleq 1/2 \qquad \mu_x([x \mapsto f\!f]) \triangleq 1/2$$
$$\mu_y([y \mapsto tt]) \triangleq 1/2 \qquad \mu_y([y \mapsto f\!f]) \triangleq 1/2$$

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

Example: Distribution Assertions

Suppose μ has two variables x, y, indep. fair coin flips

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

Then: μ satisfies $\mathbf{Unif}_{\mathbb{B}}[x] * \mathbf{Unif}_{\mathbb{B}}[y]$. Why?

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

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

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

Example: Distribution Assertions

Suppose μ has two variables x, y, indep. fair coin flips

$$\mu([x \mapsto tt, y \mapsto tt]) = 1/4 \qquad \mu([x \mapsto tt, y \mapsto f\!f]) = 1/4$$
$$\mu([x \mapsto f\!f, y \mapsto tt]) = 1/4 \qquad \mu([x \mapsto f\!f, y \mapsto f\!f]) = 1/4$$

Then: μ satisfies $\mathbf{Unif}_{\mathbb{B}}[x] * \mathbf{Unif}_{\mathbb{B}}[y]$. Why?

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

$$\mu_x([x \mapsto tt]) \triangleq 1/2 \qquad \mu_x([x \mapsto f\!f]) \triangleq 1/2$$
$$\mu_y([y \mapsto tt]) \triangleq 1/2 \qquad \mu_y([y \mapsto f\!f]) \triangleq 1/2$$

  So, $\mu \sqsubseteq \mu_x \circ \mu_y$
- Next, $\mu_x \models \mathbf{Unif}_{\mathbb{B}}[x]$ and $\mu_y \models \mathbf{Unif}_{\mathbb{B}}[y]$

Example: Distribution Assertions

Suppose μ has two variables x, y, indep. fair coin flips

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

Then: μ satisfies $\mathbf{Unif}_{\mathbb{B}}[x] * \mathbf{Unif}_{\mathbb{B}}[y]$. Why?

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

$$\mu_x([x \mapsto tt]) \triangleq 1/2 \qquad \mu_x([x \mapsto \mathit{ff}]) \triangleq 1/2$$
$$\mu_y([y \mapsto tt]) \triangleq 1/2 \qquad \mu_y([y \mapsto \mathit{ff}]) \triangleq 1/2$$

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

- Next, $\mu_x \models \mathbf{Unif}_{\mathbb{B}}[x]$ and $\mu_y \models \mathbf{Unif}_{\mathbb{B}}[y]$

Example: Distribution Assertions

Suppose μ has two variables x, y, indep. fair coin flips

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

Then: μ satisfies $\mathbf{Unif}_{\mathbb{B}}[x] * \mathbf{Unif}_{\mathbb{B}}[y]$. Why?

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

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

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

- Next, $\mu_x \models \mathbf{Unif}_{\mathbb{B}}[x]$ and $\mu_y \models \mathbf{Unif}_{\mathbb{B}}[y]$

Example: Distribution Assertions

Suppose μ has two variables x, y, indep. fair coin flips

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

Then: μ satisfies $\mathbf{Unif}_{\mathbb{B}}[x] * \mathbf{Unif}_{\mathbb{B}}[y]$. Why?

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

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

  So, $\mu \sqsubseteq \mu_x \circ \mu_y$
- Next, $\mu_x \models \mathbf{Unif}_{\mathbb{B}}[x]$ and $\mu_y \models \mathbf{Unif}_{\mathbb{B}}[y]$
- So by definition, $\mu \models \mathbf{Unif}_{\mathbb{B}}[x] * \mathbf{Unif}_{\mathbb{B}}[y]$

Example: Distribution Assertions

Suppose μ has two variables x, y, indep. fair coin flips

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

Then: μ satisfies $\mathbf{Unif}_{\mathbb{B}}[x] * \mathbf{Unif}_{\mathbb{B}}[y]$. Why?

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

$$\mu_x([x \mapsto tt]) \triangleq 1/2 \qquad \mu_x([x \mapsto \mathit{ff}]) \triangleq 1/2$$
$$\mu_y([y \mapsto tt]) \triangleq 1/2 \qquad \mu_y([y \mapsto \mathit{ff}]) \triangleq 1/2$$

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

- Next, $\mu_x \models \mathbf{Unif}_{\mathbb{B}}[x]$ and $\mu_y \models \mathbf{Unif}_{\mathbb{B}}[y]$
- So by definition, $\mu \models \mathbf{Unif}_{\mathbb{B}}[x] * \mathbf{Unif}_{\mathbb{B}}[y]$

Example Axioms

Example Axioms

Equality and distributions

- $x = y \wedge \mathbf{Unif}_{\mathbb{B}}[x] \to \mathbf{Unif}_{\mathbb{B}}[y]$

Example Axioms

Equality and distributions

- $x = y \wedge \mathbf{Unif}_{\mathbb{B}}[x] \to \mathbf{Unif}_{\mathbb{B}}[y]$

Uniformity and products

- $\mathbf{Unif}_{\mathbb{B}}[x] * \mathbf{Unif}_{\mathbb{B}}[y] \to \mathbf{Unif}_{\mathbb{B}\times\mathbb{B}}[x, y]$

Example Axioms

Equality and distributions

- $x = y \land \mathbf{Unif}_{\mathbb{B}}[x] \to \mathbf{Unif}_{\mathbb{B}}[y]$

Uniformity and products

- $\mathbf{Unif}_{\mathbb{B}}[x] * \mathbf{Unif}_{\mathbb{B}}[y] \to \mathbf{Unif}_{\mathbb{B}\times\mathbb{B}}[x, y]$

Uniformity and exclusive-or ($\oplus$)

- $\mathbf{Unif}_{\mathbb{B}}[x] * [y] \land z = x \oplus y \to \mathbf{Unif}_{\mathbb{B}}[z] * [y]$

A Probabilistic Separation Logic

Program Logic Judgments in PSL

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 $s \in \mathsf{Distr}(\mathcal{M}(\mathcal{X}))$ 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 $s \in \mathsf{Distr}(\mathcal{M}(\mathcal{X}))$ satisfying the pre-condition $s \models P$, the output state $[\![c]\!]s$ satisfies the post-condition $[\![c]\!]s \models Q$.

Perfectly fits the transformer semantics for pWHILE

Under transformer semantics:

- P describes: a distribution over memories (input)
- Q describes: a distribution over memories (output)

Under monadic semantics: mismatch!

- 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 $[\![c]\!]s$ satisfies the post-condition $[\![c]\!]s \models Q$.

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 $[\![c]\!]s$ satisfies the post-condition $[\![c]\!]s \models Q$.

Proving validity directly is difficult

- Must unfold definition of $[\![c]\!]$ as a function
- Then prove property of function by working with definition

How do we prove these judgments?

Validity

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

Proving validity directly is difficult

- Must unfold definition of $[\![c]\!]$ 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 look like:

$$\frac{\{P_1\}\ c_1\ \{Q_1\} \qquad \cdots \qquad \{P_n\}\ c_n\ \{Q_n\}}{\{P\}\ c\ \{Q\}}\ \textsc{RuleName}$$

Solution: define a set of proof rules (a proof system)

Each proof rule look like:

$$\frac{\{P_1\}\ c_1\ \{Q_1\} \qquad \cdots \qquad \{P_n\}\ c_n\ \{Q_n\}}{\{P\}\ c\ \{Q\}}\ \textsc{RuleName}$$

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:

$$\frac{\{P_1\}\ c_1\ \{Q_1\} \qquad \cdots \qquad \{P_n\}\ c_n\ \{Q_n\}}{\{P\}\ c\ \{Q\}}\ \textsc{RuleName}$$

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

$$\frac{x \notin FV(e)}{\{\top\}\ x \leftarrow e\ \{x = e\}}\ \textsc{Assn}$$

Basic Proof Rules in PSL: Assignment

Assignment Rule

$$\frac{x \notin FV(e)}{\{\top\}\ x \leftarrow e\ \{x = e\}}\ \textsc{Assn}$$

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

$$\frac{}{\{\top\}\ x \xleftarrow{\$} \mathbf{Flip}\ \{\mathbf{Unif}_{\mathbb{B}}[x]\}}\ \textsc{Samp}$$

Basic Proof Rules in PSL: Sampling

Sampling Rule

$$\frac{}{\{\top\}\ x \xleftarrow{\$} \mathbf{Flip}\ \{\mathbf{Unif}_{\mathbb{B}}[x]\}}\ \textsc{Samp}$$

How to read this rule?

From any initial distribution, running $x \xleftarrow{\$} \mathbf{Flip}$ will lead to a distribution where x is a uniformly distributed Boolean.

Basic Proof Rules in PSL: Sequencing

Sequencing Rule

$$\frac{\{P\}\ c_1\ \{Q\} \qquad \{Q\}\ c_2\ \{R\}}{\{P\}\ c_1\ ;\ c_2\ \{R\}}\ \textsc{Seq}$$

Basic Proof Rules in PSL: Sequencing

Sequencing Rule

$$\frac{\{P\}\ c_1\ \{Q\} \qquad \{Q\}\ c_2\ \{R\}}{\{P\}\ c_1\ ;\ c_2\ \{R\}}\ \textsc{Seq}$$

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?

$$\frac{\{e = tt \land P\}\; c\; \{Q\} \qquad \{e = \mathit{ff} \land P\}\; c'\; \{Q\}}{\{P\}\ \text{if } e \text{ then } c \text{ else } c'\ \{Q\}}\ \textsc{Cond?}$$

Rule COND? is not sound!

Rule COND? is not sound!

Take P to be $\mathbf{Unif}_{\mathbb{B}}[e]$ and Q to be $\bot$:

$$\frac{\{e = tt \wedge \mathbf{Unif}_{\mathbb{B}}[e]\}\ c\ \{\bot\} \qquad \{e = ff \wedge \mathbf{Unif}_{\mathbb{B}}[e]\}\ c'\ \{\bot\}}{\{\mathbf{Unif}_{\mathbb{B}}[e]\}\ \text{if } e \text{ then } c \text{ else } c'\ \{\bot\}}\ \textsc{Cond?}$$

Rule $\textsc{Cond?}$ is not sound!

Take P to be $\mathbf{Unif}_{\mathbb{B}}[e]$ and Q to be $\bot$:

$$\frac{\{e = tt \wedge \mathbf{Unif}_{\mathbb{B}}[e]\}\ c\ \{\bot\} \qquad \{e = \mathit{ff} \wedge \mathbf{Unif}_{\mathbb{B}}[e]\}\ c'\ \{\bot\}}{\{\mathbf{Unif}_{\mathbb{B}}[e]\}\ \text{if } e \text{ then } c \text{ else } c'\ \{\bot\}}\ \textsc{Cond?}$$

Premises are valid...

There is no distribution satisfying $e = tt \wedge \mathbf{Unif}_{\mathbb{B}}[e]$ or $e = \mathit{ff} \wedge \mathbf{Unif}_{\mathbb{B}}[e]$, so pre-conditions are $\bot$ and the premises are trivially valid.

Rule COND? is not sound!

Take P to be $\mathbf{Unif}_{\mathbb{B}}[e]$ and Q to be $\perp$:

$$\frac{\{e = tt \wedge \mathbf{Unif}_{\mathbb{B}}[e]\}\ c\ \{\perp\} \qquad \{e = ff \wedge \mathbf{Unif}_{\mathbb{B}}[e]\}\ c'\ \{\perp\}}{\{\mathbf{Unif}_{\mathbb{B}}[e]\}\ \text{if } e \text{ then } c \text{ else } c'\ \{\perp\}}\ \textsc{Cond?}$$

Premises are valid...

There is no distribution satisfying $e = tt \wedge \mathbf{Unif}_{\mathbb{B}}[e]$ or $e = ff \wedge \mathbf{Unif}_{\mathbb{B}}[e]$, so pre-conditions are $\perp$ and the premises are trivially valid.

But the conclusion is not!

It is not the case that if $\mathbf{Unif}_{\mathbb{B}}[e]$ 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

$$\frac{\{e = tt \land P\}\ c\ \{Q\} \qquad \{e = f\!f \land P\}\ c'\ \{Q\}}{\{P\}\ \textsf{if}\ e\ \textsf{then}\ c\ \textsf{else}\ c'\ \{Q\}}\ \textsc{Cond?}$$

What went wrong?

The broken rule

$$\frac{\{e = tt \wedge P\}\ c\ \{Q\} \qquad \{e = ff \wedge P\}\ c'\ \{Q\}}{\{P\}\ \text{if } e \text{ then } c \text{ else } c'\ \{Q\}}\ \textsc{Cond?}$$

The problem: conditioning

- We assume: P holds in input distribution μ
- Inputs to branches: μ conditioned on $e = tt$ and $e = ff$
- But: P might not hold on conditional distributions!

Conditional Rule: second try

Does this rule work?

$$\frac{\{e = tt * P\}\ c\ \{Q\} \qquad \{e = \mathit{ff} * P\}\ c'\ \{Q\}}{\{[e] * P\}\ \mathsf{if}\ e\ \mathsf{then}\ c\ \mathsf{else}\ c'\ \{Q\}}\ \textsc{Cond??}$$

Conditional Rule: second try

Does this rule work?

$$\frac{\{e = tt * P\}\ c\ \{Q\} \qquad \{e = \mathit{ff} * P\}\ c'\ \{Q\}}{\{[e] * P\}\ \text{if } e \text{ then } c \text{ else } c'\ \{Q\}}\ \textsc{Cond}??$$

Previous counterexample fails

If we take P to be $\mathbf{Unif}_{\mathbb{B}}[e]$, then $[e] * \mathbf{Unif}_{\mathbb{B}}[e]$ is false, and the conclusion is trivially valid.

But rule Cond?? is still not sound!

But rule Cond?? is still not sound!

Consider this proof

$$\dfrac{\{e = tt * \top\}\ x \leftarrow e\ \{[x] * [e]\} \qquad \{e = \mathit{ff} * P\}\ x \leftarrow e\ \{[x] * [e]\}}{\{[e] * \top\}\ \text{if } e \text{ then } x \leftarrow e \text{ else } x \leftarrow e\ \{[x] * [e]\}}\ \textsc{Cond??}$$

But rule COND?? is still not sound!

Consider this proof

$$\dfrac{\{e = tt * \top\}\ x \leftarrow e\ \{[x] * [e]\} \qquad \{e = ff * P\}\ x \leftarrow e\ \{[x] * [e]\}}{\{[e] * \top\}\ \text{if } e \text{ then } x \leftarrow e \text{ else } x \leftarrow e\ \{[x] * [e]\}}\ \text{COND??}$$

Premises are valid...

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

But rule COND?? is still not sound!

Consider this proof

$$\frac{\{e = tt * \top\}\ x \leftarrow e\ \{[x] * [e]\} \qquad \{e = ff * P\}\ x \leftarrow e\ \{[x] * [e]\}}{\{[e] * \top\}\ \text{if } e \text{ then } x \leftarrow e \text{ else } x \leftarrow e\ \{[x] * [e]\}}\ \textsc{Cond??}$$

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

$$\frac{\{e = tt * P\}\ c\ \{Q\} \qquad \{e = ff * P\}\ c'\ \{Q\}}{\{[e] * P\}\ \text{if } e \text{ then } c \text{ else } c'\ \{Q\}}\ \textsc{Cond??}$$

What went wrong?

The broken rule

$$\frac{\{e = tt * P\}\ c\ \{Q\} \qquad \{e = f\!f * P\}\ c'\ \{Q\}}{\{[e] * P\}\ \text{if}\ e\ \text{then}\ c\ \text{else}\ c'\ \{Q\}}\ \textsc{Cond??}$$

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

$$\frac{\begin{array}{c}\{e = tt * P\}\ c\ \{Q\} \\ \{e = ff * P\}\ c'\ \{Q\} \\ Q \text{ is closed under mixtures (CM)}\end{array}}{\{[e] * P\}\ \text{if } e \text{ then } c \text{ else } c'\ \{Q\}}\ \textsc{Cond}$$

Conditional Rule in PSL

Fixed rule

$$\frac{\{e = tt * P\}\ c\ \{Q\} \qquad \{e = \mathit{ff} * P\}\ c'\ \{Q\} \qquad Q \text{ is closed under mixtures (CM)}}{\{[e] * P\}\ \text{if } e \text{ then } c \text{ else } c'\ \{Q\}}\ \textsc{Cond}$$

Pre-conditions

- Inputs to branches derived from conditioning on e
- Independence ensures that P holds after conditioning

Conditional Rule in PSL

Fixed rule

$$\frac{\{e = tt * P\}\ c\ \{Q\} \quad \{e = ff * P\}\ c'\ \{Q\} \quad Q \text{ is closed under mixtures (CM)}}{\{[e] * P\}\ \text{if } e \text{ then } c \text{ else } c'\ \{Q\}}\ \textsc{Cond}$$

Pre-conditions

- Inputs to branches derived from 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 $p \in [0, 1]$.

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

- $x = e$
- $\mathbf{Unif}_{\mathbb{B}}[x]$

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

- $x = e$
- $\mathbf{Unif}_{\mathbb{B}}[x]$

Examples of non-CM assertions

- $[x] * [y]$
- $x = 1 \vee x = 2$

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:

$$\frac{\begin{array}{c}\{x = tt * \mathbf{Unif}_{\mathbb{B}}[y]\}\ z \leftarrow \neg y\ \{\mathbf{Unif}_{\mathbb{B}}[z]\} \\ \{x = ff * \mathbf{Unif}_{\mathbb{B}}[y]\}\ z \leftarrow y\ \{\mathbf{Unif}_{\mathbb{B}}[z]\} \\ \mathbf{Unif}_{\mathbb{B}}[z] \text{ is closed under mixtures (CM)}\end{array}}{\{[x] * \mathbf{Unif}_{\mathbb{B}}[y]\}\ \text{if } x \text{ then } z \leftarrow \neg y \text{ else } z \leftarrow y\ \{\mathbf{Unif}_{\mathbb{B}}[z]\}}\ \textsc{Cond}$$

The Proof System of PSL

Frame Rule

The Frame Rule in SL

Properties about unmodified heaps are preserved

$$\frac{\{P\}\,c\,\{Q\} \qquad c \text{ doesn't modify } FV(R)}{\{P * R\}\,c\,\{Q * R\}}\ \textsc{Frame}$$

The Frame Rule in SL

Properties about unmodified heaps are preserved

$$\frac{\{P\}\, c\, \{Q\} \qquad c \text{ doesn't modify } FV(R)}{\{P * R\}\, c\, \{Q * R\}}\ \textsc{Frame}$$

So-called “local reasoning” in SL

- Only need to reason about part of heap used by c
- Note: doesn’t hold if $*$ replaced by $\wedge$, due to aliasing!

Why is the Frame rule important?

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

$$\frac{\{P\}\, c\, \{Q\} \qquad FV(R) \cap MV(c) = \emptyset \qquad \models P \to [RV(c)] \qquad FV(Q) \subseteq RV(c) \cup WV(c)}{\{P * R\}\, c\, \{Q * R\}}\ \textsc{Frame}$$

Side conditions

The Frame Rule in PSL

The rule

$$\frac{\{P\}\ c\ \{Q\} \qquad FV(R) \cap MV(c) = \emptyset \qquad \models P \to [RV(c)] \qquad FV(Q) \subseteq RV(c) \cup WV(c)}{\{P * R\}\ c\ \{Q * R\}}\ \textsc{Frame}$$

Side conditions

1. Variables in R are not modified

The Frame Rule in PSL

The rule

$$\frac{\{P\}\, c\, \{Q\} \qquad FV(R) \cap MV(c) = \emptyset \\ \models P \to [RV(c)] \qquad FV(Q) \subseteq RV(c) \cup WV(c)}{\{P * R\}\, c\, \{Q * R\}}\ \textsc{Frame}$$

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

$$\frac{\{P\}\ c\ \{Q\} \qquad FV(R) \cap MV(c) = \emptyset \qquad \models P \to [RV(c)] \qquad FV(Q) \subseteq RV(c) \cup WV(c)}{\{P * R\}\ c\ \{Q * R\}}\ \textsc{Frame}$$

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

$$\frac{\{P\}\ c\ \{Q\} \qquad FV(R) \cap MV(c) = \emptyset \qquad \models P \to [RV(c)] \qquad FV(Q) \subseteq RV(c) \cup WV(c)}{\{P * R\}\ c\ \{Q * R\}}\ \textsc{Frame}$$

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:

$$\frac{}{\{\top\}\; x \xleftarrow{\$} \mathbf{Flip}\; \{\mathbf{Unif}_{\mathbb{B}}[x]\}}\ \textsc{Samp}$$

Frame rule:

$$\frac{\begin{array}{cc} \{P\}\; c\; \{Q\} & FV(R) \cap MV(c) = \emptyset \\ \models P \to [RV(c)] & FV(Q) \subseteq RV(c) \cup WV(c) \end{array}}{\{P * R\}\; c\; \{Q * R\}}\ \textsc{Frame}$$

Example: Deriving a Better Sampling Rule

Original sampling rule:

$$\frac{}{\{\top\}\ x \xleftarrow{\$} \mathbf{Flip}\ \{\mathbf{Unif}_{\mathbb{B}}[x]\}}\ \textsc{Samp}$$

Frame rule:

$$\frac{\begin{array}{cc} \{P\}\ c\ \{Q\} & FV(R) \cap MV(c) = \emptyset \\ \models P \to [RV(c)] & FV(Q) \subseteq RV(c) \cup WV(c) \end{array}}{\{P * R\}\ c\ \{Q * R\}}\ \textsc{Frame}$$

Can derive:

$$\frac{x \notin FV(R)}{\{R\}\ x \xleftarrow{\$} \mathbf{Flip}\ \{\mathbf{Unif}_{\mathbb{B}}[x] * R\}}\ \textsc{Samp*}$$

Example: Deriving a Better Sampling Rule

Original sampling rule:

$$\frac{}{\{\top\}\ x \xleftarrow{\$} \mathbf{Flip}\ \{\mathbf{Unif}_{\mathbb{B}}[x]\}}\ \textsc{Samp}$$

Frame rule:

$$\frac{\{P\}\ c\ \{Q\} \qquad FV(R) \cap MV(c) = \emptyset \qquad \models P \to [RV(c)] \qquad FV(Q) \subseteq RV(c) \cup WV(c)}{\{P * R\}\ c\ \{Q * R\}}\ \textsc{Frame}$$

Can derive:

$$\frac{x \notin FV(R)}{\{R\}\ x \xleftarrow{\$} \mathbf{Flip}\ \{\mathbf{Unif}_{\mathbb{B}}[x] * R\}}\ \textsc{Samp}^*$$

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 μ, if $\mu \models P$ then $[\![c]\!]\mu \models Q$.

Key property for soundness: restriction

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

Intuition

- The only variables that "matter" for P are $FV(P)$
- 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:

$$k \xleftarrow{\$} \mathbf{Flip}\,⨾$$
$$c \leftarrow k \oplus m$$

How to Formalize Security?

How to Formalize Security?

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

Proving Input-Output Independence for OTP in PSL

$k \xleftarrow{\$} \mathbf{Flip} ⨾$

$c \leftarrow k \oplus m$

Proving Input-Output Independence for OTP in PSL

$\{[m]\}$ assumption

$k \xleftarrow{\$} \mathbf{Flip};$

$c \leftarrow k \oplus m$

Proving Input-Output Independence for OTP in PSL

$\{[m]\}$ assumption

$k \xleftarrow{\$} \textbf{Flip};$

$\{[m] * \textbf{Unif}_{\mathbb{B}}[k]\}$ [SAMP*]

$c \leftarrow k \oplus m$

Proving Input-Output Independence for OTP in PSL

$\{[m]\}$	assumption
$k \xleftarrow{\$} \mathbf{Flip};$	
$\{[m] * \mathbf{Unif}_{\mathbb{B}}[k]\}$	[Samp*]
$c \leftarrow k \oplus m$	
$\{[m] * \mathbf{Unif}_{\mathbb{B}}[k] \wedge c = k \oplus m\}$	[Assn*]

Proving Input-Output Independence for OTP in PSL

$\{[m]\}$	assumption
$k \xleftarrow{\$} \mathbf{Flip};$	
$\{[m] * \mathbf{Unif}_{\mathbb{B}}[k]\}$	[SAMP*]
$c \leftarrow k \oplus m$	
$\{[m] * \mathbf{Unif}_{\mathbb{B}}[k] \wedge c = k \oplus m\}$	[ASSN*]
$\{[m] * \mathbf{Unif}_{\mathbb{B}}[c]\}$	XOR axiom

PSL: references and further reading

The original paper on probabilistic semantics

Kozen. Semantics of Probabilistic Programs. FOCS 1980.

Unifying survey on Bunched Implications

Docherty. Bunched Logics: A Uniform Approach. PhD Thesis (UCL), 2019.

A Probabilistic Separation Logic (POPL20)

- Extensions to PSL: deterministic variables, loops, etc.
- Many examples from cryptography, security of ORAM
- https://arxiv.org/abs/1907.10708

A Bunched Logic for Conditional Independence (LICS21)

- A BI-style logic called DIBI for conditional independence
- A separation logic (CPSL) based on DIBI
- https://arxiv.org/abs/2008.09231

Reasoning about Probabilistic Programs

Higher-Order Languages

So far: reasoning about pWHILE programs

First part

- Monadic semantics: $\llparenthesis c \rrparenthesis : \mathcal{M} \to \mathsf{Distr}(\mathcal{M})$
- Verification method: weakest pre-expectations (wpe)

Second part

- Transformer semantics: $[\![c]\!] : \mathsf{Distr}(\mathcal{M}) \to \mathsf{Distr}(\mathcal{M})$
- 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 $unit(a) \in \mathsf{Distr}(A)$ is defined to be:

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

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 $bind(\mu, f) \in \text{Distr}(B)$ is defined to be:

$$bind(\mu, f)(b) \triangleq \sum_{a \in A} \mu(a) \cdot f(a)(b)$$

Language: probabilistic monadic lambda calculus

Language grammar: core

$$\mathcal{E} \ni e \coloneqq x \in \mathcal{X} \mid \lambda \mathcal{X}.\, \mathcal{E} \mid \mathcal{E}\ \mathcal{E} \mid \mathsf{fix}\ \mathcal{X}.\, \lambda \mathcal{X}.\, \mathcal{E} \qquad \text{(lambda calc.)}$$

Language: probabilistic monadic lambda calculus

Language grammar: core

$$\mathcal{E} \ni e \coloneqq x \in \mathcal{X} \mid \lambda \mathcal{X}.\, \mathcal{E} \mid \mathcal{E}\ \mathcal{E} \mid \mathsf{fix}\ \mathcal{X}.\, \lambda \mathcal{X}.\, \mathcal{E} \qquad \text{(lambda calc.)}$$

Language grammar: base types

$$\begin{aligned} \mathcal{E} \ni e \coloneqq \cdots &\mid b \in \mathbb{B} \mid \mathsf{if}\ \mathcal{E}\ \mathsf{then}\ \mathcal{E}\ \mathsf{else}\ \mathcal{E} && \text{(booleans)} \\ &\mid n \in \mathbb{N} \mid \mathsf{add}(\mathcal{E}, \mathcal{E}) && \text{(numbers)} \end{aligned}$$

Language: probabilistic monadic lambda calculus

Language grammar: core

$$\mathcal{E} \ni e \coloneqq x \in \mathcal{X} \mid \lambda \mathcal{X}.\, \mathcal{E} \mid \mathcal{E}\ \mathcal{E} \mid \mathsf{fix}\ \mathcal{X}.\, \lambda \mathcal{X}.\, \mathcal{E} \qquad \text{(lambda calc.)}$$

Language grammar: base types

$$\begin{aligned} \mathcal{E} \ni e \coloneqq \cdots &\mid b \in \mathbb{B} \mid \mathsf{if}\ \mathcal{E}\ \mathsf{then}\ \mathcal{E}\ \mathsf{else}\ \mathcal{E} && \text{(booleans)} \\ &\mid n \in \mathbb{N} \mid \mathsf{add}(\mathcal{E}, \mathcal{E}) && \text{(numbers)} \end{aligned}$$

Language grammar: probabilistic part

$$\begin{aligned} \mathcal{E} \ni e \coloneqq \cdots &\mid \mathbf{Flip} \mid \mathbf{Roll} && \text{(distributions)} \\ &\mid \mathsf{return}(\mathcal{E}) && \text{(unit)} \\ &\mid \mathsf{sample}\ \mathcal{X} = \mathcal{E}\ \mathsf{in}\ \mathcal{E} && \text{(bind)} \end{aligned}$$

Example programs

Sum of two dice rolls

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

Example programs

Sum of two dice rolls

$$\begin{array}{l}\text{sample } x = \mathbf{Roll} \text{ in} \\ \text{sample } y = \mathbf{Roll} \text{ in} \\ \text{return}(\text{add}(x, y))\end{array}$$

Geometric distribution

$$\begin{array}{l}(\text{fix } geo.\ \lambda n. \\ \quad \text{sample } stop = \mathbf{Flip} \text{ in} \\ \quad \text{if } stop \text{ then return}(n) \text{ else } geo \text{ add}(n, 1))\ 0\end{array}$$

Operational semantics: One-step reduction

Definition

The one-step relation $\rightarrow : \mathcal{CE} \rightarrow \mathsf{Distr}(\mathcal{CE})$ maps closed expressions to distributions on closed expressions:

$$e \rightarrow \mu$$

Reading

"Expression e steps to distribution μ on expressions in one step".

Operational semantics: Multi-step reduction

Definition

For any $n \in \mathbb{N}$, the multi-step relation $\Rightarrow_n : \mathcal{CE} \to \mathsf{SDistr}(\mathcal{CV})$ maps closed expressions to sub-distributions on closed values.

$$e \Rightarrow_n \mu$$

Reading

"Expression e steps to sub-distribution μ on values in exactly n steps".

Operational semantics: Big-step reduction

Definition

The multi-step relation $\Rightarrow : \mathcal{CE} \to \mathsf{SDistr}(\mathcal{CV})$ maps closed expressions to sub-distributions on closed values.

$$e \Rightarrow \mu$$

Reading

"Expression e steps to sub-distribution μ on values".
Define as limit of approximants: if $e \Rightarrow_n \mu_n$, then $e \Rightarrow \lim_{k\to\infty} \sum_{n=1}^{k} \mu_n$

Operational semantics: non-probabilistic part

Standard call-by-value semantics

$$
\begin{aligned}
(\lambda x.\ e)\ v &\to unit(e[v/x]) \\
\text{if } tt \text{ then } e \text{ else } e' &\to unit(e) \\
\text{if } \mathit{ff} \text{ then } e \text{ else } e' &\to unit(e') \\
(\text{fix } f.\ \lambda x.\ e)\ v &\to unit(e[(\text{fix } f.\ \lambda x.\ e)/f][v/x]) \\
\text{add}(n, n') &\to unit(n + n') \\
&\cdots
\end{aligned}
$$

Operational semantics: primitive distributions

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

$$\mathbf{Flip} \to \{tt : 1/2, ff : 1/2\}$$
$$\mathbf{Roll} \to \{1 : 1/6, \ldots, 6 : 1/6\}$$

Operational semantics: unit and bind

Unit

$$\frac{e \to e'}{\mathsf{return}(e) \to \mathsf{return}(e')}$$

Bind

$$\frac{e \to \{v_i : p_i\}_{i \in I}}{\mathsf{sample}\ x = e\ \mathsf{in}\ e' \to \{e'[v_i/x] : p_i\}_{i \in I}}$$

A Simple Probabilistic Type System

Types in our language

$$
\begin{aligned}
\mathcal{T} \ni \tau \coloneqq {} & \mathbb{B} \mid \mathbb{N} && \text{(base types)} \\
\mid {} & \mathcal{T} \to \mathcal{T} && \text{(functions)} \\
\mid {} & \bigcirc \mathcal{T} && \text{(distributions)}
\end{aligned}
$$

Typing judgment basics

The main judgment

Let $e \in \mathcal{E}$, $\tau \in \mathcal{T}$, and Γ be a finite list of 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 $unit(v)$ if τ is non-probabilistic, or reduces to a sub-distribution over closed values if τ is probabilistic (of the form $\bigcirc\tau$).

Typing rules: variables and functions

Exactly the same as in lambda calculus

$$\frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau}\ \textsc{Var} \qquad \frac{\Gamma, x : \tau \vdash e : \tau'}{\Gamma \vdash \lambda x.\ e : \tau \to \tau'}\ \textsc{Lam}$$

$$\frac{\begin{array}{c}\Gamma \vdash e : \tau \to \tau' \\ \Gamma \vdash e' : \tau\end{array}}{\Gamma \vdash e\ e' : \tau'}\ \textsc{App} \qquad \frac{\Gamma, f : \tau \to \tau' \vdash \lambda x.\ e : \tau \to \tau'}{\Gamma \vdash \mathsf{fix}\ f.\ \lambda x.\ e : \tau \to \tau'}\ \textsc{Fix}$$

Typing rules: booleans and integers

Hopefully not too surprising

$$\frac{b = tt, \mathit{ff}}{\Gamma \vdash b : \mathbb{B}}\ \textsc{Bool} \qquad \frac{n \in \mathbb{N}}{\Gamma \vdash n : \mathbb{N}}\ \textsc{Nat}$$

$$\frac{\Gamma \vdash e : \mathbb{N} \quad \Gamma \vdash e' : \mathbb{N}}{\Gamma \vdash \mathsf{add}(e, e') : \mathbb{N}}\ \textsc{Add}$$

Typing rules: primitive distributions

Assign distribution types

$$\frac{}{\Gamma \vdash \mathbf{Flip} : \bigcirc\mathbb{B}}\ \textsf{FLIP} \qquad \frac{}{\Gamma \vdash \mathbf{Roll} : \bigcirc\mathbb{N}}\ \textsf{ROLL}$$

Typing rules: unit and bind

Unit

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash \mathsf{return}(e) : \bigcirc\tau}\ \textsc{Return}$$

Bind

$$\frac{\Gamma \vdash e : \bigcirc\tau \qquad \Gamma, x : \tau \vdash e' : \bigcirc\tau'}{\Gamma \vdash \mathsf{sample}\ x = e\ \mathsf{in}\ e' : \bigcirc\tau'}\ \textsc{Sample}$$

What property do we want the types to ensure?

Non-probabilistic types

If $e \in \mathcal{CE}$ has non-probabilistic type τ, then e should reduce to $unit(v)$ with $v \in \mathcal{CV}$ of type τ, or loop forever.

Probabilistic types

If $e \in \mathcal{CE}$ has probabilistic type $\bigcirc\tau$, then e should reduce to $\mu \in$ SDistr$(\mathcal{CV})$ where every element in the support of μ has type τ.

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

$$\frac{\Gamma \vdash e : \bigcirc\tau \qquad \Gamma, x : \tau \vdash e' : \bigcirc\tau'}{\Gamma \vdash \mathsf{sample}\ x = e\ \mathsf{in}\ e' : \bigcirc\tau'}\ \textsc{Sample}$$

The key typing rule: Sample

$$\frac{\Gamma \vdash e : \bigcirc\tau \qquad \Gamma, x : \tau \vdash e' : \bigcirc\tau'}{\Gamma \vdash \text{sample } x = e \text{ in } e' : \bigcirc\tau'}\ \textsc{Sample}$$

Let's unpack this rule

1. e is a distribution over τ

The key typing rule: Sample

$$\frac{\Gamma \vdash e : \bigcirc\tau \qquad \Gamma, x : \tau \vdash e' : \bigcirc\tau'}{\Gamma \vdash \mathsf{sample}\ x = e\ \mathsf{in}\ e' : \bigcirc\tau'}\ \textsc{Sample}$$

Let's unpack this rule

1. e is a distribution over τ
2. Given a sample $x : \tau$, e' produces a distribution over τ'

The key typing rule: Sample

$$\frac{\Gamma \vdash e : \bigcirc\tau \qquad \Gamma, x : \tau \vdash e' : \bigcirc\tau'}{\Gamma \vdash \textsf{sample}\ x = e\ \textsf{in}\ e' : \bigcirc\tau'}\ \textsc{Sample}$$

Let's unpack this rule

1. e is a distribution over τ
2. Given a sample $x : \tau$, e' produces a distribution over τ'
3. Sampling from e and plugging into e': distribution over τ'

Generalizing the rule

$$\frac{\Gamma \vdash e : P\tau \qquad \Gamma, x : \tau \vdash e' : Q\tau'}{\Gamma \vdash \mathsf{sample}\ x = e\ \mathsf{in}\ e' : Q\tau'}\ \textsc{SampleGen}$$

Generalizing the rule

$$\frac{\Gamma \vdash e : P\tau \qquad \Gamma, x : \tau \vdash e' : Q\tau'}{\Gamma \vdash \textsf{sample } x = e \textsf{ in } e' : Q\tau'} \textsc{SampleGen}$$

Let's change the meaning of the distribution type

1. e is a distribution over τ satisfying P

Generalizing the rule

$$\frac{\Gamma \vdash e : P\tau \qquad \Gamma, x : \tau \vdash e' : Q\tau'}{\Gamma \vdash \textsf{sample } x = e \textsf{ in } e' : Q\tau'}\ \textsc{SampleGen}$$

Let's change the meaning of the distribution type

1. e is a distribution over τ satisfying P
2. Given a sample $x : \tau$, e' produces a distribution over τ' satisfying Q

Generalizing the rule

$$\frac{\Gamma \vdash e : P\tau \qquad \Gamma, x : \tau \vdash e' : Q\tau'}{\Gamma \vdash \textsf{sample } x = e \textsf{ in } e' : Q\tau'}\ \textsc{SampleGen}$$

Let's change the meaning of the distribution type

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

Generalizing the rule

$$\frac{\Gamma \vdash e : P\tau \qquad \Gamma, x : \tau \vdash e' : Q\tau'}{\Gamma \vdash \textsf{sample } x = e \textsf{ in } e' : Q\tau'} \textsc{ SampleGen}$$

Let's change the meaning of the distribution type

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

Generalizing the rule

$$\frac{\Gamma \vdash e : P\tau \qquad \Gamma, x : \tau \vdash e' : Q\tau'}{\Gamma \vdash \text{sample } x = e \text{ in } e' : Q\tau'} \textsc{SampleGen}$$

Let's change the meaning of the distribution type

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

For what distribution properties Q is this rule OK?

Does this remind you of something we have seen already?

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

- $x = e$
- $\mathbf{Unif}_{\mathbb{B}}[x]$

Examples of non-CM assertions

- $[x] * [y]$
- $x = 1 \vee x = 2$

The main requirement: closed under mixtures (CM)

$$\frac{\Gamma \vdash e : P\tau \qquad \Gamma, x : \tau \vdash e' : Q\tau'}{\Gamma \vdash \mathsf{sample}\ x = e\ \mathsf{in}\ e' : Q\tau'}\ \textsc{SampleGen}$$

The main requirement: closed under mixtures (CM)

$$\frac{\Gamma \vdash e : P\tau \qquad \Gamma, x : \tau \vdash e' : Q\tau'}{\Gamma \vdash \textsf{sample } x = e \textsf{ in } e' : Q\tau'} \textsc{SampleGen}$$

The property Q must be closed under mixtures (CM)

1. We have a bunch of distributions over τ' satisfying Q

The main requirement: closed under mixtures (CM)

$$\frac{\Gamma \vdash e : P\tau \qquad \Gamma, x : \tau \vdash e' : Q\tau'}{\Gamma \vdash \textsf{sample } x = e \textsf{ in } e' : Q\tau'} \textsc{ SampleGen}$$

The property Q must be closed under mixtures (CM)

1. We have a bunch of distributions over τ' satisfying Q
2. We are blending these distributions together

The main requirement: closed under mixtures (CM)

$$\frac{\Gamma \vdash e : P\tau \qquad \Gamma, x : \tau \vdash e' : Q\tau'}{\Gamma \vdash \textsf{sample}\ x = e\ \textsf{in}\ e' : Q\tau'}\ \textsc{SampleGen}$$

The property Q must be closed under mixtures (CM)

1. We have a bunch of distributions over τ' 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 τ is a finite type (e.g., $\mathbb{B}$), a program e has type $U\tau$ if it evaluates to the uniform distribution over τ without encountering any runtime errors.

Then the sampling rule is sound:

$$\frac{\Gamma \vdash e : \bigcirc\tau \qquad \Gamma, x : \tau \vdash e' : U\tau'}{\Gamma \vdash \text{sample } x = e \text{ in } e' : U\tau'}\ \textsc{SampleUnif}$$

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

$$\mathcal{T} \ni \tau \coloneqq \cdots \mid \bigcirc_\alpha \tau \qquad (\alpha \in M)$$

New typing rules

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash \mathsf{return}(e) : \bigcirc_0 \tau}\ \textsc{GReturn}$$

$$\frac{\Gamma \vdash e : \bigcirc_\alpha \tau \qquad \Gamma, x : \tau \vdash e' : \bigcirc_\beta \tau'}{\Gamma \vdash \mathsf{sample}\ x = e\ \mathsf{in}\ e' : \bigcirc_{\alpha+\beta} \tau'}\ \textsc{GSample}$$

$$\frac{\Gamma \vdash e : \bigcirc_\alpha \qquad \alpha \leq \beta}{\Gamma \vdash e : \bigcirc_\beta}\ \textsc{GSubty}$$

Monadic types: references and further readings

Original papers on probabilistic monadic types

- Ramsey and Pfeffer. Stochastic lambda calculus and monads of probability distributions. POPL 2002.
- Park, Pfenning, and Thrun. A Probabilistic Language based upon Sampling Functions. POPL 2005.

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.
- Higher-Order Approximate Relational Refinement Types for Mechanism Design and Differential Privacy. POPL 2015.

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)

$\mathrm{PCF}_{\oplus}$

Main features

- Makes τ 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

$\mathrm{PCF}_{\oplus}$: Reading the typing judgment

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 τ.

PPCF

Main features

- Makes τ 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

PPCF: Reading the typing judgment

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 τ.
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

- Saheb-Djahromi. CPO's of Measures for Non-determinism. 1979.
- Jones and Plotkin. A Probabilistic Powerdomain of Evaluations. 1989.
- Heunen, Kammar, Staton, Yang. A Convenient Category for Higher-Order Probability Theory. 2017.

Type systems

- $\mathrm{PCF}_{\oplus}$: Dal Lago (https://doi.org/10.1017/9781108770750.005)
- PPCF: Erhard, Pagani, Tasson. Measurable Cones and Stable, Measurable Functions. 2018.
- Darais, Sweet, Liu, Hicks. A language for probabilistically oblivious computation. POPL 2020.

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