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

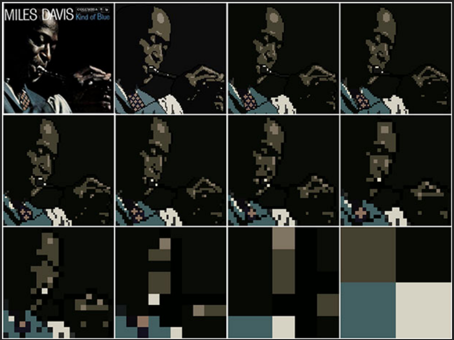
- Treat probabilities as zero or non-zero (non-determinism)
- Simplifies verification, but can’t prove most properties

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 $\mathbb{B} = \{tt, f\!f\}$
- $\mu(tt) = \mu(f\!f) \triangleq 1/2$

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 $\mu \in \text{Distr}(A)$ 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 $\mu \in \text{Distr}(A)$ be a distribution. Then the probability of E in μ is:

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

Expected value

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

Example: distribution map

Swap results of a biased coin flip

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

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

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

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: $\mu = \mathit{bind}(\mathbf{Roll}, f)$ 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 $\mu \in \mathsf{Distr}(A)$, 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!

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 $\mu_1 \in \mathsf{Distr}(A_1)$ and $\mu_2 \in \mathsf{Distr}(A_2)$. Then the independent product is the distribution in $\mathsf{Distr}(A_1 \times A_2)$ 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

$\textsc{pWhile}$ by Example

The language, in a nutshell

- Core imperative $\textsc{While}$-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?

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

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?

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

More formally: pWHILE expressions

Grammar of boolean and numeric expressions

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

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

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

Imperative language with sampling

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