

# **P**robabilistic & **A**pproximate **C**omputing

**Sasa Misailovic**

**UIUC**

# Operational Semantics (*deterministic*)

Simulates how the program executes on an abstract state-machine

31: ...  
32:  $y = x + 2$   
33: ...

X	Y	PC
3	0	32

X	Y	PC
3	5	33

## Two flavors:

- Small-step operational semantics  
*read from location pointed to by x, do addition, store to location pointed to by y*
- Big-step operational semantics  
 *$x + 2$  evaluates to the value 5; store this value to the location pointed to by y*

# Operational Semantics (*deterministic*)

## Execution Stages:

- Map the program and inputs to the *initial configuration*

init: (x, 3)

01: def program(x) {...}

X	Y
3	0

PC
01

- Execute the steps* that represent the instructions of the abstract machine

32:  $y = x + 2$

X	Y
3	5

PC
33

- Map the *final configuration* (if it exists) to the output

33: return y

X	Y
3	5

PC
33

# Simple Deterministic Language

$x$	$\in$	Vars	$S$	$::=$	
$\mathcal{T}$	$::=$	bool			$x := \mathcal{E}$
<b>uop</b>	$::=$	not			$\mathcal{S}_1; \mathcal{S}_2$
<b>bop</b>	$::=$	and   or			if $\mathcal{E}$ then $\mathcal{S}_1$ else $\mathcal{S}_2$
$\mathcal{D}$	$::=$	$\mathcal{T} x_1, x_2, \dots, x_n$			while $\mathcal{E}$ do $\mathcal{S}_1$
$\mathcal{E}$	$::=$		$\mathcal{P}$	$::=$	$\mathcal{D} \mathcal{S}$
		$x$			
		$c$			
		$\mathcal{E}_1$ <b>bop</b> $\mathcal{E}_2$			
		<b>uop</b> $\mathcal{E}_1$			

# Operational Semantics

Tuple:  $S = (\mathcal{C}, \rightarrow, \mathcal{C}_{final}, \mathcal{J}, \mathcal{O})$

- $\mathcal{C}$ : Set of configurations (e.g.,  $\mathcal{C} = Stmt \times Stack \times Mem$ )
- $\rightarrow$ : Transition relation, which defines possible transitions between the configurations
  - Deterministic if for each start configuration  $c \in \mathcal{C}$  there exists a single result configuration  $c' \in \mathcal{C}$
  - Nondeterministic if there can be multiple  $c \in \mathcal{C}$
- $\mathcal{C}_{final}$ : Set of final configurations ( $\mathcal{C}_{final} \subseteq \mathcal{C}$ ) in which the program successfully ends
- $\mathcal{J}$  maps program source and input to initial configuration  $\mathcal{C}_0$
- $\mathcal{O}$  maps final configuration  $\mathcal{C}_{final}$  to the output

# Operational Semantics (*small step*)

**Configuration:**  $c \in \mathcal{C} ::= Stmt \times \Sigma$

$\sigma \in \Sigma ::= (x_1 \rightarrow v_1, \dots, x_n \rightarrow v_n)$

*Stmt* – next statement or no statement to run ( $\cdot$ )

$x_i$  – variables,  $v_i$  – values

## Final Configuration:

- Has the form  $(\text{skip}, \sigma) = (\text{skip}, (x_1 \rightarrow v_1, \dots, x_n \rightarrow v_n))$
- Specifies configurations in which program terminated normally
- But, execution can get stuck (there is a statement for which there is no transition)
- Or, execution may never terminate (the execution loops infinitely)

# Operational Semantics (*small step*)

**For expressions:**  $\rightarrow_b \in Expr \times \Sigma \rightarrow Expr$        $e1, e2$  – expressions  
 $v1, v2$  – values

$$(x, \sigma) \rightarrow_b \sigma(x)$$

*Reading a variable*

$$(uop\ v1, \sigma) \rightarrow_b\ v2$$

*Unary operation on constants  
(v2 is the result of unary operation)*

$$\frac{(e1, \sigma) \rightarrow_b\ e1'}{(uop\ e1, \sigma) \rightarrow_b\ uop\ e1'}$$

*Unary operation on subexpressions*

$$(v1\ bop\ v2, \sigma) \rightarrow_b\ v3$$

*Binary operation on constants  
(v3 is the result of binary operation)*

$$\frac{(e1, \sigma) \rightarrow_b\ e1'}{(e1\ bop\ e2, \sigma) \rightarrow_b\ e1'\ bop\ e2}$$

*Binary operation on subexpressions*

$$\frac{(e2, \sigma) \rightarrow_b\ e2'}{(v1\ bop\ e2, \sigma) \rightarrow_b\ v1\ bop\ e2'}$$

*Binary operation on subexpressions*

# Operational Semantics (*small step*)

**For statements:**

$\rightarrow \in \mathcal{C} \rightarrow \mathcal{C}$

$e_1, e_2$  – expressions

$v_1, v_2$  – values

$(x = v, \sigma) \rightarrow (\text{skip}, \sigma[x \leftarrow v])$

*Assigning a constant*

$(x = e, \sigma) \rightarrow (x = e', \sigma)$   
where  $(e, \sigma) \rightarrow_b e'$

*Assigning an expression*

$(\text{skip}; s_1, \sigma) \rightarrow (s_1, \sigma)$

*Sequence rule (1)*

$(s_1; s_2, \sigma) \rightarrow (s_1'; s_2, \sigma')$   
where  $(s_1, \sigma) \rightarrow (s_1', \sigma')$

*Sequence rule (2)*



# Operational Semantics (*small step*)

**For statements:**

$\rightarrow \in \mathcal{C} \rightarrow \mathcal{C}$

$e_1, e_2$  – expressions

$v_1, v_2$  – values

$(\text{if true then } s_1 \text{ else } s_2, \sigma) \rightarrow (s_1, \sigma)$

**Conditional (then)**

$(\text{if false then } s_1 \text{ else } s_2, \sigma) \rightarrow (s_2, \sigma)$

**Conditional (else)**

$(e_1, \sigma) \rightarrow_b e_1'$

---

$(\text{if } e \text{ then } s_1 \text{ else } s_2, \sigma) \rightarrow$   
 $(\text{if } e' \text{ then } s_1 \text{ else } s_2, \sigma)$

**Conditional (expr)**

$(\text{while } e \text{ do } s, \sigma) \rightarrow$

**While loop**

$(\text{if } e \text{ then } \{s_1; \text{while } e \text{ do } s\} \text{ else skip, } \sigma)$

# Simple Probabilistic Language

$r \in \mathbb{R}$

$x \in \text{Vars}$

$\mathcal{T} ::= \text{bool}$

**uop** ::= not

**bop** ::= and | or

$\mathcal{D} ::= | \mathcal{T} x_1, x_2, \dots, x_n$

$\mathcal{E} ::=$

$x$

|  $c$

|  $\mathcal{E}_1$  **bop**  $\mathcal{E}_2$

| **uop**  $\mathcal{E}_1$

$\mathcal{S} ::=$

$x := \mathcal{E}$

|  $x := \text{Bernoulli}(r)$

| observe ( $\mathcal{E}$ )

| skip

|  $\mathcal{S}_1; \mathcal{S}_2$

| if  $\mathcal{E}$  then  $\mathcal{S}_1$  else  $\mathcal{S}_2$

| while  $\mathcal{E}$  do  $\mathcal{S}_1$

$\mathcal{P} ::= \mathcal{D} \mathcal{S}$

# Probabilistic State

## Deterministic

**State:**  $\sigma_0 \in \Sigma_0 ::= (x_1 \rightarrow v_1, \dots, x_n \rightarrow v_n)$   
 $x_i$  – variables     $v_i$  – values

## Probabilistic

**State:**  $\sigma \in \Sigma = \Sigma_0$

**Expressions:**  $A \in \mathcal{E} \times \Sigma \rightarrow \text{Bool}$

**Statements:**  $T \subseteq (S \times \Sigma) \times (\Sigma \times [0,1])$

(we use notation  $(\cdot, \cdot) \dot{\rightarrow} \cdot$ )

# Probabilistic Assignment

Statements:  $T \subseteq (S \times \Sigma) \times (\Sigma \times [0,1])$

$(x = v, \sigma) \rightarrow (\text{skip}, \sigma[x \leftarrow v])$       *Assigning a constant*

$(x = e, \sigma) \rightarrow (x = e', \sigma)$       *Assigning an expression*  
where  $(e, \sigma) \rightarrow_b e'$

$(x = \text{Bern}(p_B), \sigma) \xrightarrow{p_B} (\text{skip}, \sigma[x \leftarrow \text{True}])$

$(x = \text{Bern}(p_B), \sigma) \xrightarrow{1-p_B} (\text{skip}, \sigma[x \leftarrow \text{False}])$

# Example: Bernoulli Program

(

X	Y
--	--

, 1.0)

**X := Bernoulli(0.7);**

(

X	Y
True	--

, 0.7), (

X	Y
False	--

, 0.3)

**Y := not X;**

(

X	Y
True	False

, 0.7), (

X	Y
False	True

, 0.3)

# Probabilistic Control Flow

$$(\text{skip}; s1, \sigma) \xrightarrow{1} (s1, \sigma)$$

**Sequence rule (1)**

$$(s1, \sigma) \xrightarrow{p_1} (s1', \sigma')$$

---

$$(s1; s2, \sigma) \xrightarrow{p_1} (s1'; s2, \sigma')$$

**Sequence rule (2)**

*Recall:*

$$(x = \text{Bern}(p_B), \sigma) \xrightarrow{p_B} (\text{skip}, \sigma[x \leftarrow \text{True}])$$

$$(x = \text{Bern}(p_B), \sigma) \xrightarrow{1-p_B} (\text{skip}, \sigma[x \leftarrow \text{False}])$$

# Example: Bernoulli Program

(

X	Y
--	--

, 1.0)

**X := Bernoulli(0.7);**

(

X	Y
True	--

, 0.7), (

X	Y
False	--

, 0.3)

**Y := Bernoulli(0.7);**

(

X	Y
True	True

, 0.49), (

X	Y
True	False

, 0.21), (

X	Y
False	True

, 0.21), (

X	Y
False	False

, 0.09)

# Probability of a Trace

**(Finite) trace** of executing a statement/program ( $c \in \mathcal{C}$ ):

$$\theta = c_1 \xrightarrow{p_1} c_2 \xrightarrow{p_2} c_3 \xrightarrow{p_3} \dots \xrightarrow{p_n} c_{n+1}$$

- $c_1 = (S, \sigma_{init})$  is an initial configuration
- $c_{n+1} = (\text{skip}, \sigma_{final})$  is the final configuration, assuming that  $S$  terminates
- The execution took specific transitions from  $c_1$  to  $c_{n+1}$

**Probability of the trace:**  $\Pr(\theta) = p_1 \cdot p_2 \cdot \dots \cdot p_n$



# Example: Bernoulli Program

(

X	Y
--	--

, 1.0)

**X := Bernoulli(0.7);**

(

X	Y
True	--

, 0.7), (

X	Y
False	--

, 0.3)

**Y := Bernoulli(0.7);**

(

X	Y
True	True

, 0.49), (

X	Y
True	False

, 0.21), (

X	Y
False	True

, 0.21), (

X	Y
False	False

, 0.09)

# Observations

$$(\text{observe True}, \sigma) \xrightarrow{1} (\text{skip}, \sigma)$$

$$(\text{observe False}, \sigma) \xrightarrow{1} \emptyset$$

***(doesn't exist: no such transition)***

**Normalization:** If the probabilities  $p_1 \dots p_k$  do not sum up to 1, rescale them so that they do:

1. Compute sum:  $Z = p_1 + p_2 + \dots + p_k$
2. Rescale:  $p'_1 = \frac{p_1}{Z}, p'_2 = \frac{p_2}{Z}, \dots, p'_k = \frac{p_k}{Z}$

Do only at the end of the program – although still expensive

# Example: Bernoulli Program

(

X	Y
--	--

, 1.0)

**X := Bernoulli(0.7);**

**Y := Bernoulli(0.7);**

(

X	Y
True	True

, 0.49), (

X	Y
True	False

, 0.21), (

X	Y
False	True

, 0.21), (

X	Y
False	False

, 0.09)

**condition ( X == True );**

**return Y;**

(

X	Y
True	True

, 0.49/0.7), (

X	Y
True	False

, 0.21/0.7),

# Example: Bernoulli Program

(

X	Y
--	--

, 1.0)

**X := Bernoulli(0.7);**

**Y := Bernoulli(0.7);**

(

X	Y
True	True

, 0.49), (

X	Y
True	False

, 0.21), (

X	Y
False	True

, 0.21), (

X	Y
False	False

, 0.09)

**condition ( X == Y );**

**return Y;**

(

X	Y
True	True

, 0.49/0.58), (

X	Y
False	False

, 0.09/0.58)

# Example: Bernoulli Program

(

X	Y
--	--

, 1.0)

**X := Bernoulli(0.7);**

**Y := Bernoulli(0.7);**

(

X	Y
True	True

, 0.49), (

X	Y
True	False

, 0.21), (

X	Y
False	True

, 0.21), (

X	Y
False	False

, 0.09)

**factor ( X ? 0 : -1 );**

$e^0 = 1$   
 $e^{-1} = 0.37$

(

X	Y
True	True

, 0.49), (

X	Y
True	False

, 0.21), (

X	Y
False	True

, 0.077), (

X	Y
False	False

, 0.033)

# Example: Bernoulli Program

*factor*  $\Rightarrow$  *condition*

(

X	Y
--	--

, 1.0)

**X := Bernoulli(0.7);**

**Y := Bernoulli(0.7);**

(

X	Y
True	True

, 0.49), (

X	Y
True	False

, 0.21), (

X	Y
False	True

, 0.21), (

X	Y
False	False

, 0.09)

**factor ( X ? 0 :  $-\infty$  );**

$e^0 = 1$   
 $e^{-\infty} = 0$

(

X	Y
True	True

, 0.49), (

X	Y
True	False

, 0.21), (

X	Y
False	True

, 0), (

X	Y
False	False

, 0)

# Denotational Semantics

**Syntactic Domain:** describes the syntax of the language (grammar): elements are e.g., nodes in the abstract syntax tree (AST)

**Semantic Domain:** mathematical entities and operations on them

- For instance, sets of numbers, sets of tuples
- For probabilistic programs expectations are easy to envision

**Meaning Function:** Translates elements of the syntactic domain to the elements and operations in the semantic domain

- **Compositionality:** the meaning of the AST is composite of the meaning of its nodes

**How Do We Get Randomness?**



# Linear Congruental Generator

```
int rseed = 0;  
  
inline void srand(int x)  
    rseed = x;  
}
```

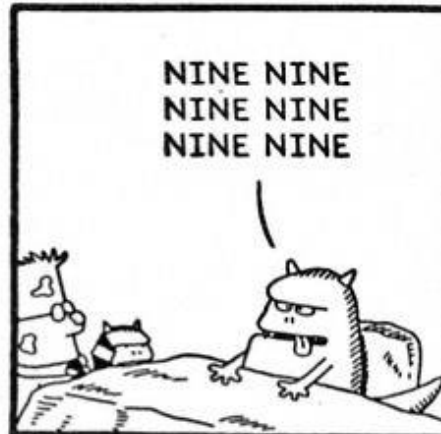


```
#define RAND_MAX ((1U << 31) - 1)  
  
inline int rand() {  
    return rseed =  
        (rseed * 1103515245 + 12345) & RAND_MAX;  
}
```

# Still Better Than...

```
int getRandomNumber()  
{  
    return 4; // chosen by fair dice roll.  
             // guaranteed to be random.  
}
```

© xkcd



10/25/01 © 2001 United Feature Syndicate, Inc.

# Better Pseudorandom Generators

## **Mersenne Twister**

- Large cycle (up to  $2^{19937} - 1$ )
- Fast, but requires lots of (cache) memory
- Default choice for the languages from this century

## **Xorshift**

- Moderate cycle (from  $2^{64} - 1$  to  $2^{1024} - 1$ )
- Very fast, uses only bitshifts and xor operators
- May not pass all tests for uniformity, but good for simulation

**Simulation vs. cryptography** (e.g., Yarrow/Fortuna)

# True Randomness?

## Hardware generators

- Based on thermal noise (or other natural phenomena)
- Main use is cryptographic, speed is less of a concern
- E.g., Intel IvyBridge-EP microarchitecture uses hardware RNG (see RDRAND instruction)

## True random sequences from the Internet

- E.g., <https://www.random.org/> gets numbers from atmospheric noise

## Tests for pseudorandom number generators

- DieHard (Marsaglia)
- TestU01 (L'Ecuyer and Simard)