

Language-Parametric Formal Methods in the Field

Daejun Park
May 23, 2018 @ Fasoo



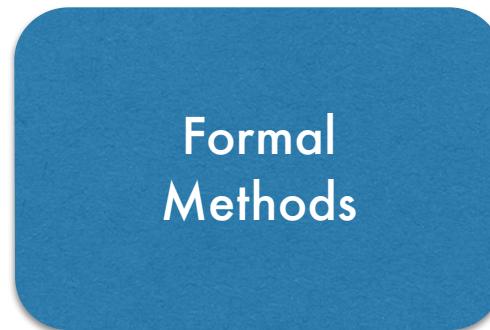
Reusability problem of formal tools

- Fragmentation: crafted for a fixed language
 - Implemented similar heuristics/optimizations: duplicating efforts
- Hard to re-target
 - Replace hardcoded semantics, or
 - Implement a translation to the original language

Language-parametric formal methods

- Mitigate reusability issue
- Develop universal formal methods
 - Parameterized by language semantics
- Instantiate formal tools
 - By plugging-in language semantics
 - Admit operational semantics

Language-parametric formal methods



- C (c11, gcc, clang, ...)
 - Java (6, 7, 8, ...)
 - JavaScript (ES5, ES6, ...)
 - ...
-
- Four lines connect the four items above to the corresponding boxes below:
- C (c11, gcc, clang, ...) connects to Deductive verification
 - Java (6, 7, 8, ...) connects to Model checking
 - JavaScript (ES5, ES6, ...) connects to Program equivalence checking
 - ... connects to ...

Defined/implemented once, and reused for all others

Language-independent verification framework

Semantics

\models

Program & Properties

Language-independent uniform notation (logic)

Language-independent proof systems

Proof automation

\vdash

- Provides a nice interface (logic) in which both language semantics and program properties can be described.
- Proof search in this logic becomes completely language-independent.

Operational semantics

- Easy to define and understand than axiomatic semantics
 - Require little mathematical knowledge
 - Similar to implement language interpreter
- Executable, thus testable
 - Important when defining real large languages
- Shown to scale to defining full language semantics
 - C, Java, JavaScript, Python, PHP, ...

Missing details of language behaviors

```
1 unsigned x = UINT_MAX;  
2 unsigned y = x + 1;  
→ 3 _assert(y == 0)
```

VCC incorrectly reported an overflow error

Missing details of language behaviors

```
1 int assign(int *p, int x)
2     _(ensures *p == x)
3     _(writes p)
4 {
5     return (*p = x);
6 }
7
8 void main() {
9     int r;
10    assign(&r, 0) == assign(&r, 1);
11    _(assert r == 1)
12 }
```



VCC incorrectly proved it, missing non-determinism

Missing details of language behaviors

→ **14% of SV-COMP's "Correct Programs" are Undefined!**

Posted on September 18, 2016 by Grigore Rosu



Last April (2016), I gave a [tutorial on K](#) at ETAPS'16 in Eindhoven, Netherlands, where I also demonstrated [RV-Match](#). During the week that I spent there, I heard several friends and colleagues who were involved with the Competition on Software Verification, [SV-COMP](#), that some of the benchmark's *correct* programs appear to be undefined. What? So some of the assumed-correct C programs that are used to evaluate the best program verifiers in the world are actually *wrong* programs? [Continue reading →](#)

* Grigore Rosu, <https://runtimeverification.com/blog/?p=200>

Overview

- Specifying real-world language semantics and measuring the specification effort
- Developing language-parametric formal methods
- Instantiating them by plugging-in various language semantics
- Applying the derived formal tools to real-world applications, demonstrating their practical feasibility

Application 1

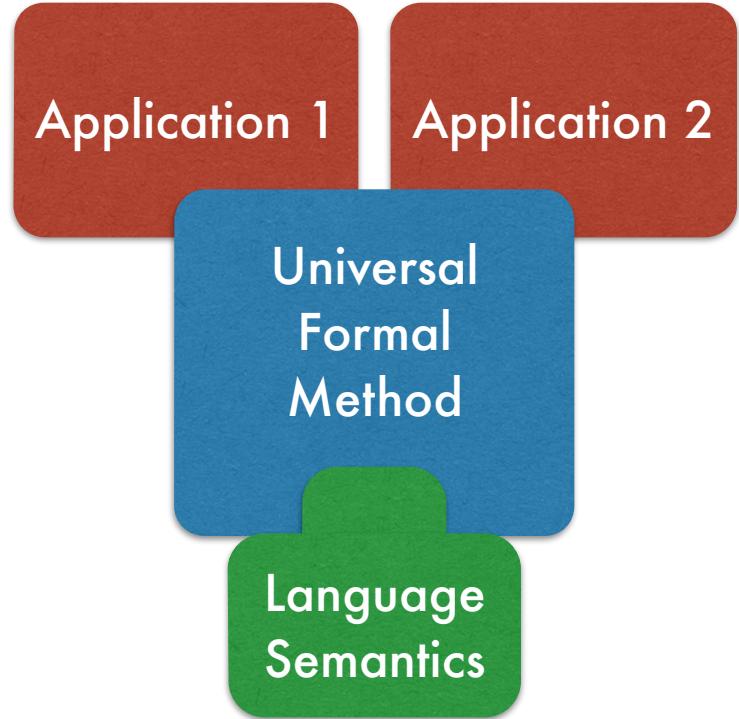
Application 2

Universal
Formal
Method

Language
Semantics

Overview

- Specifying real-world language semantics and measuring the specification effort
- Developing language-parametric formal methods
- Instantiating them by plugging-in various language semantics
- Applying the derived formal tools to real-world applications, demonstrating their practical feasibility



Overview

Heap
manipulating
programs

Smart
contracts

Compiler
verification

Specification
refinement

Deductive
Verifier

Program
Equivalence
Checker

C

Java

JavaScript

WASM

JAUS

Vyper

EVM

Specifying language semantics

Heap manipulating programs

Smart contracts

Compiler verification

Specification refinement

Deductive Verifier

Program Equivalence Checker

C

Java

JavaScript

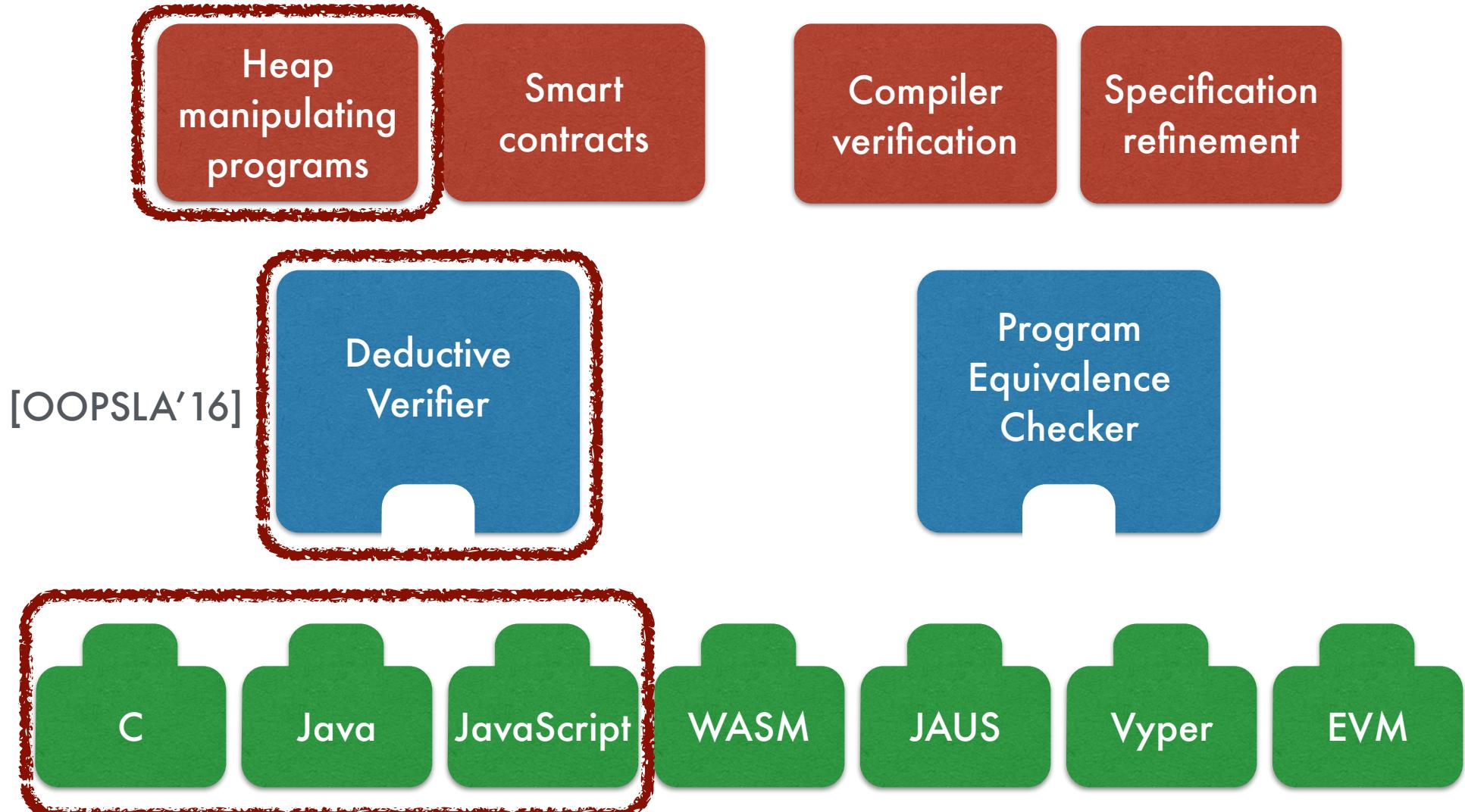
WASM

JAUS

Vyper

EVM

Instantiating with various languages



Applying to real-world systems

[TR'17] [TR'18]

Heap
manipulating
programs

Smart
contracts

Compiler
verification

Specification
refinement

Deductive
Verifier

Program
Equivalence
Checker

C

Java

JavaScript

WASM

JAUS

Vyper

EVM

[CSF'18]

Developing new formal method

Heap manipulating programs

Smart contracts

Compiler verification

Specification refinement

Deductive Verifier

Program Equivalence Checker

[TR'17]

C

Java

JavaScript

WASM

JAUS

Vyper

EVM

Specifying language semantics

Heap manipulating programs

Smart contracts

Compiler verification

Specification refinement

Deductive Verifier

Program Equivalence Checker

C

Java

JavaScript

WASM

JAUS

Vyper

EVM

KJS: complete formal semantics of JavaScript

KJS *faithfully* formalizes ECMAScript 5.1 standard.

informal

The expression “`++ Expression`” is evaluated as follows:

1. Let `expr` be the result of evaluating `Expression`.
2. Let `oldValue` be `ToNumber(GetValue(expr))`.
3. Let `newValue` be the result of adding the value 1 to `oldValue`.
4. Call `PutValue(expr, newValue)`.
5. Return `newValue`.

ECMAScript 5.1 standard

KJS: complete formal semantics of JavaScript

KJS *faithfully* formalizes ECMAScript 5.1 standard.

informal

The expression “`++ Expression`” is evaluated as follows:

1. Let *expr* be the result of evaluating *Expression*.
2. Let *oldValue* be `ToNumber(GetValue(expr))`.
3. Let *newValue* be the result of adding the value 1 to *oldValue*.
4. Call `PutValue(expr, newValue)`.
5. Return *newValue*.

ECMAScript 5.1 standard

rule `++ Expression =>`
`Let $expr = @GetReference(Expression);`
`Let $oldValue = ToNumber(GetValue($expr));`
`Let $newValue = @Addition($oldValue,1);`
`Call PutValue($expr,$newValue);`
`Return $newValue;`

KJS

KJS: complete formal semantics of JavaScript

KJS *faithfully* formalizes ECMAScript 5.1 standard.

each step of informal description → formal pseudo-code statement
systematic translation

The expression “`++ Expression`” is evaluated as follows:

1. Let `expr` be the result of evaluating `Expression`.
2. Let `oldValue` be `ToNumber(GetValue(expr))`.
3. Let `newValue` be the result of adding the value 1 to `oldValue`.
4. Call `PutValue(expr, newValue)`.
5. Return `newValue`.

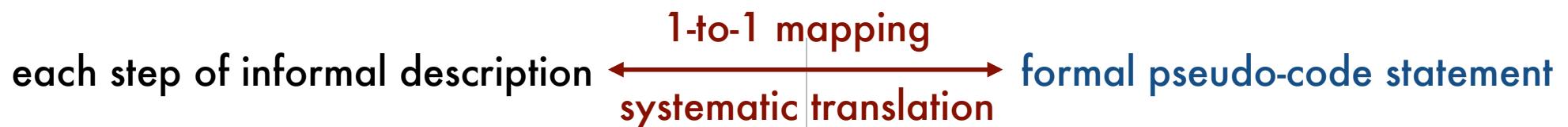
ECMAScript 5.1 standard

```
rule ++ Expression =>
  Let $expr = @GetReference(Expression);
  Let $oldValue = ToNumber(GetValue($expr));
  Let $newValue = @Addition($oldValue,1);
  Call PutValue($expr,$newValue);
  Return $newValue;
```

KJS

KJS: complete formal semantics of JavaScript

KJS *faithfully* formalizes ECMAScript 5.1 standard.



The expression “`++ Expression`” is evaluated as follows:

1. Let $expr$ be the result of evaluating $Expression$. $\xleftarrow{\text{rule } ++ \text{ Expression} \Rightarrow}$ Let $\$expr = @GetReference(Expression);$
2. Let $oldValue$ be $\text{ToNumber}(\text{GetValue}(expr))$. $\xleftarrow{\text{rule } ++ \text{ Expression} \Rightarrow}$ Let $\$oldValue = \text{ToNumber}(\text{GetValue}(\$expr));$
3. Let $newValue$ be the result of adding the value 1 to $oldValue$. $\xleftarrow{\text{rule } ++ \text{ Expression} \Rightarrow}$ Let $\$newValue = @Addition(\$oldValue, 1);$
4. Call $\text{PutValue}(expr, newValue)$. $\xleftarrow{\text{rule } ++ \text{ Expression} \Rightarrow}$ Call $\text{PutValue}(\$expr, \$newValue);$
5. Return $newValue$. $\xleftarrow{\text{rule } ++ \text{ Expression} \Rightarrow}$ Return $\$newValue;$

ECMAScript 5.1 standard

KJS

manual inspection

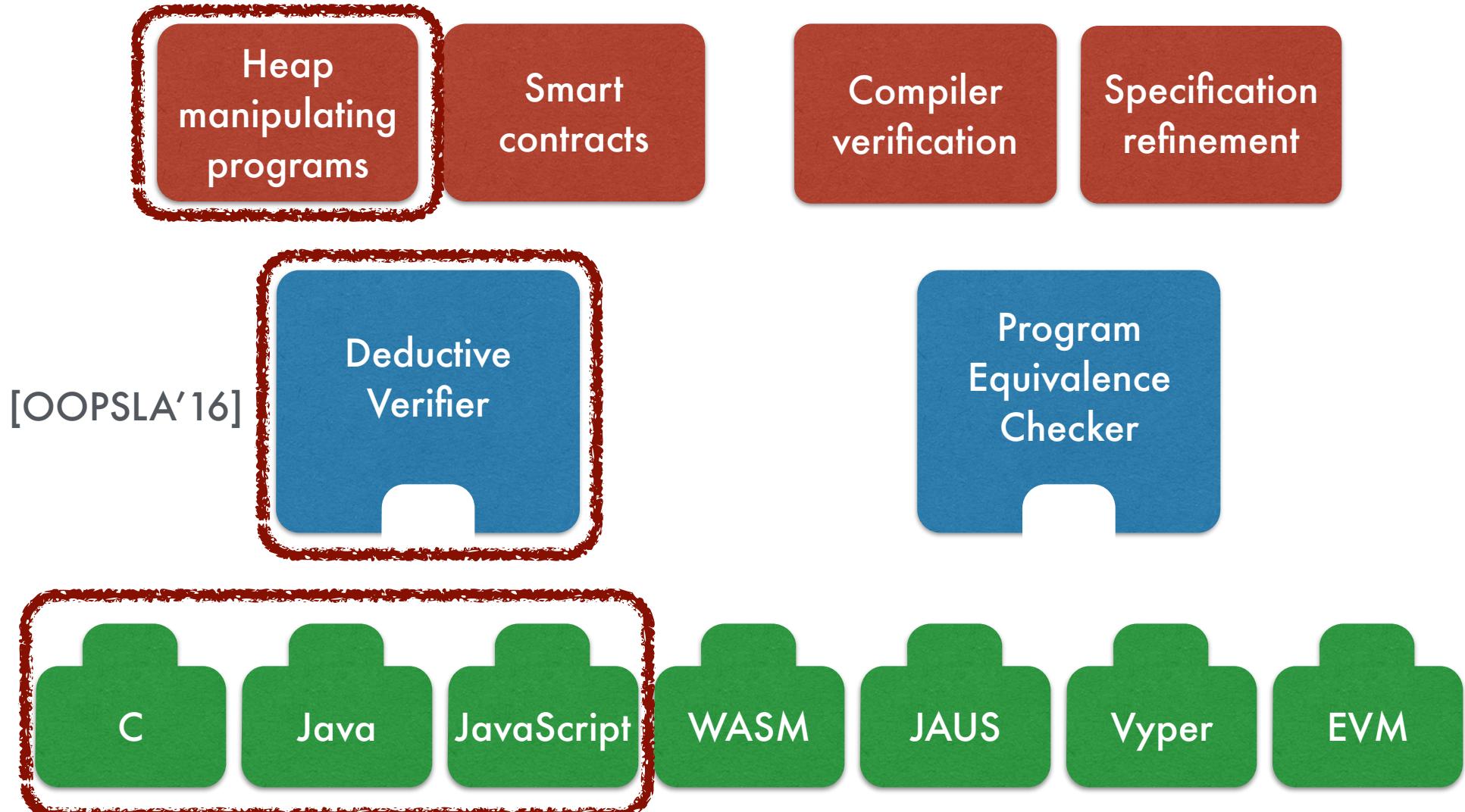
Completeness

Tested against ECMAScript conformance test suite.

| Formal Semantics | Passed | Failed | % passed | |
|-----------------------------------|--------|--------|----------|---|
| KJS | 2,782 | 0 | 100.0% | ✓ |
| [Politz et al. 2012] ⁵ | 2,470 | 345 | 87.7% | ✗ |
| [Bodin et al. 2014] | 1,796 | 986 | 64.6% | ✗ |

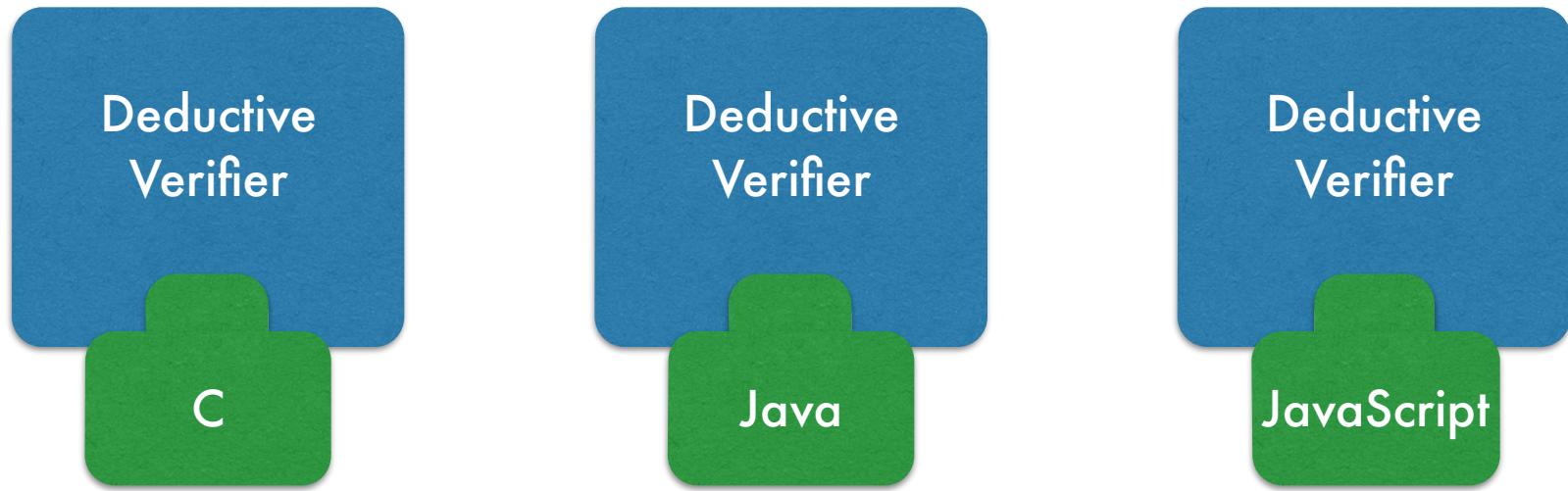
Took me only four months.

Instantiating with various languages



Deductive program verifiers

- Instantiated framework by plugging-in three language semantics.



- Verified challenging heap-manipulating programs implementing the same algorithms in all three languages.

Experiments [OOPSLA'16]

| | | | | Time (secs) | | | |
|------------|---------|-------|-------|----------------|---------|-------|-------|
| Programs | C | Java | JS | Programs | C | Java | JS |
| BST find | 14.0 | 4.7 | 6.3 | Treap find | 14.4 | 4.9 | 6.5 |
| BST insert | 30.2 | 8.6 | 8.2 | Treap insert | 67.7 | 23.1 | 18.9 |
| BST delete | 71.7 | 24.9 | 21.2 | Treap delete | 90.4 | 28.4 | 33.2 |
| AVL find | 13.0 | 4.9 | 6.4 | List reverse | 11.4 | 4.1 | 5.5 |
| AVL insert | 281.3 | 105.2 | 135.0 | List append | 14.8 | 7.3 | 5.3 |
| AVL delete | 633.7 | 271.6 | 239.6 | Bubble sort | 66.4 | 38.8 | 31.3 |
| RBT find | 14.5 | 5.0 | 6.8 | Insertion sort | 61.9 | 31.1 | 44.8 |
| RBT insert | 903.8 | 115.6 | 114.5 | Quick sort | 79.2 | 47.1 | 48.1 |
| RBT delete | 1,902.1 | 171.2 | 183.6 | Merge sort | 170.6 | 87.0 | 66.0 |
| | | | | Total | 4,441.1 | 983.5 | 981.2 |
| | | | | Average | 246.7 | 54.6 | 54.5 |

Experiments [OOPSLA'16]

| Programs | Time (secs) | | |
|------------|-------------|------|-----|
| | C | Java | JS |
| BST find | 14.0 | 4.7 | 6.3 |
| BST insert | 30.2 | 8.6 | 8.2 |

| Programs | Time (secs) | | |
|--------------|-------------|------|------|
| | C | Java | JS |
| Treap find | 14.4 | 4.9 | 6.5 |
| Treap insert | 67.7 | 23.1 | 18.9 |

Full functional correctness:

```
/*
 * @pre bst(t)
 * @post bst(t')
 * @post keys(t') == keys(t) \union { v }
 */
function insert(t, v) {
    ...
}
```

| | | | |
|---------|---------|-------|-------|
| Total | 4,441.1 | 983.5 | 981.2 |
| Average | 246.7 | 54.6 | 54.5 |

Experiments [OOPSLA'16]

| Programs | Time (secs) | | |
|--------------|-------------|-------|-------|
| | C | Java | JS |
| BST find | 14.0 | 4.7 | 6.3 |
| BST insert | 30.2 | 8.6 | 8.2 |
| BST delete | 71.7 | 24.9 | 21.2 |
| AVL find | 13.0 | 4.9 | 6.4 |
| AVL insert | 281.3 | 105.2 | 135.0 |
| Treap find | 14.4 | 4.9 | 6.5 |
| Treap insert | 67.7 | 23.1 | 18.9 |
| Treap delete | 90.4 | 28.4 | 33.2 |
| List reverse | 11.4 | 4.1 | 5.5 |
| List append | 14.8 | 7.3 | 5.3 |

Performance is comparable to a state-of-the-art verifier for C, VCDryad, based on a separation logic extension of VCC:
e.g., AVL insert : 260s vs 280s (ours)

| | | | |
|---------|---------|-------|-------|
| Total | 4,441.1 | 983.5 | 981.2 |
| Average | 246.7 | 54.6 | 54.5 |

Applying to real-world systems

[TR'17] [TR'18]

Heap
manipulating
programs

Smart
contracts

Compiler
verification

Specification
refinement

Deductive
Verifier

Program
Equivalence
Checker

C

Java

JavaScript

WASM

JAUS

Vyper

EVM

[CSF'18]

Smart contracts

- Programs that run on blockchain
- Usually written in a high-level language
 - Solidity (JavaScript-like), Vyper (Python-like), ...
- Compiled down to VM bytecode
 - EVM (Ethereum VM), IELE (LLVM-like VM), ...

our target
 - Runs on VM of blockchain nodes

Challenges for EVM bytecode verification

- Byte-twiddling operations
 - Non-linear integer arithmetic (e.g., modulo reduction)
- Arithmetic overflow detection
- Gas limit
 - Variable gas cost depending on contexts
- Hash collision

Byte-twiddling operations

Given:

$$x[n] \stackrel{\text{def}}{=} (x/256^n) \bmod 256$$

$$\text{merge}(x[i..j]) \stackrel{\text{def}}{=} \text{merge}(x[i..j+1]) * 256 \pm x[j] \quad \text{when } i > j$$

$$\text{merge}(x[i..i]) \stackrel{\text{def}}{=} x[i]$$

Prove:

$$\text{“}x = \text{merge}(x[31..0])\text{”}.$$

Abstractions

```
syntax Int ::= nthByte(Int, Int, Int) [function]
```

```
rule merge(nthByte(V, 0, N) ... nthByte(V, N-1, N))
  => V
  requires 0 <= V < 2 ^ (N * 8)
    and 1 <= N <= 32
```

Challenges for EVM bytecode verification

- Byte-twiddling operations
 - Non-linear integer arithmetic (e.g., modulo reduction)
- Arithmetic overflow detection
- Gas limit
 - Variable gas cost depending on contexts
- Hash collision

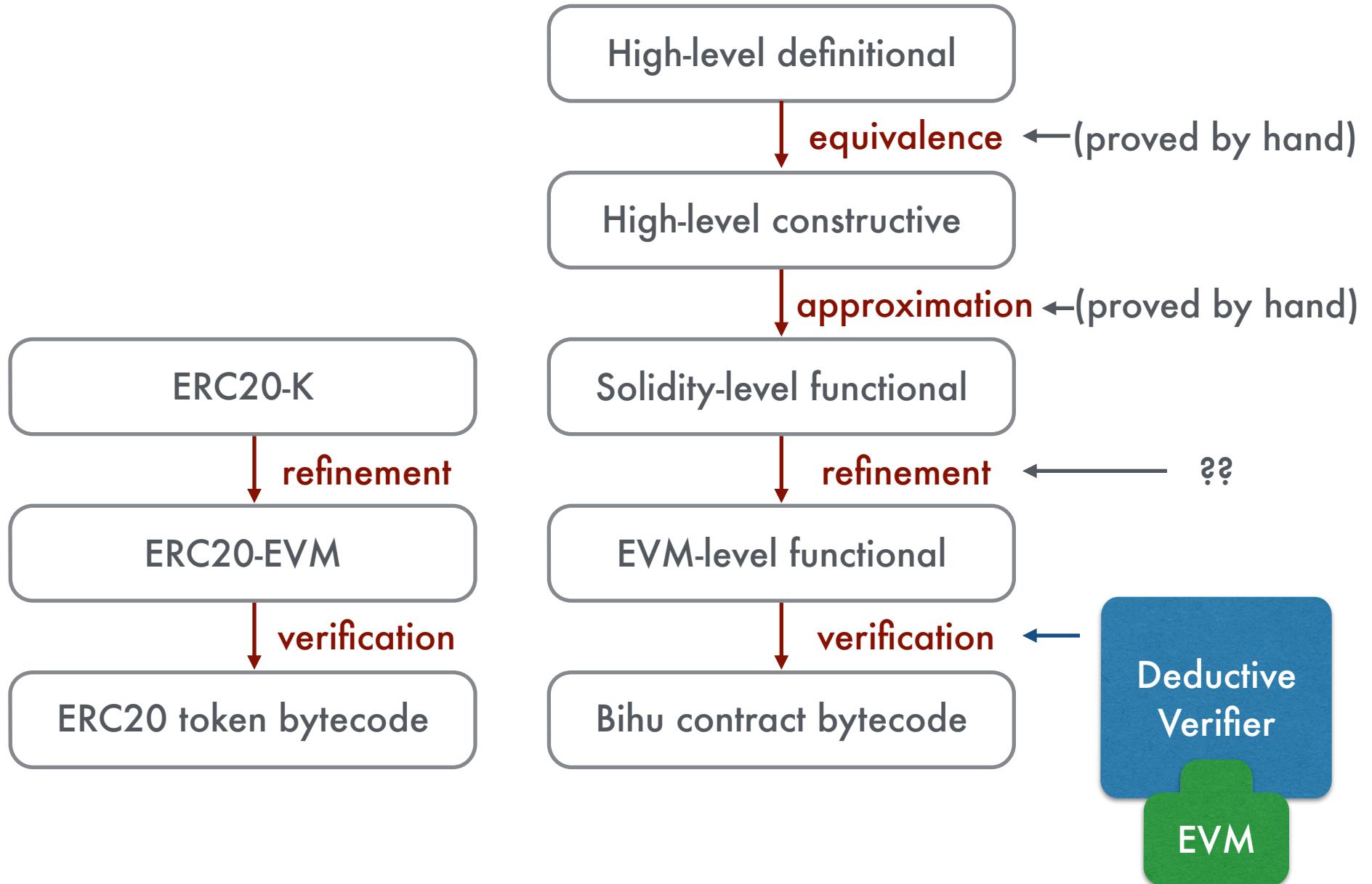
Verified smart contracts*

- High-profile ERC20 token contracts
 - OpenZeppelin's
 - ConsenSys's
 - Hacker Gold (HKG) token
 - Vyper ERC20 token
 - Commercial Bihu KEY operation contracts
 - KeyRewardPool
 - WarmWallet
- * <https://github.com/runtimeverification/verified-smart-contracts>
- 
- found divergent behaviors

End-to-end verification

- Formalize high-level business logic
 - Confirmed by contract developers
- Refine it to EVM level
 - Capturing EVM-specific details
- Verify EVM bytecode using derived EVM verifier
 - No need to trust Solidity/Vyper compilers

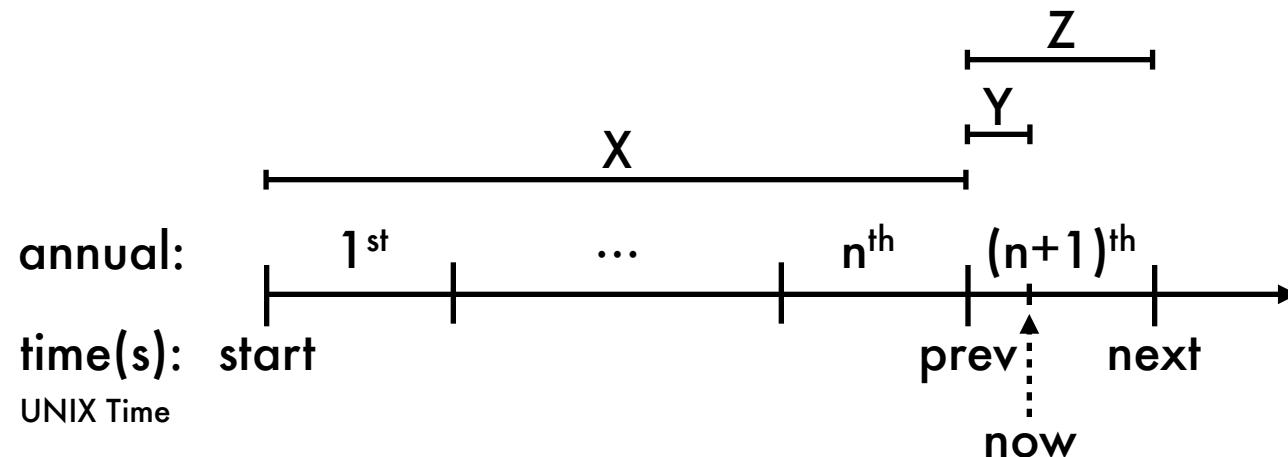
End-to-end verification



Informal specification

"KeyRewardPool contract is responsible for releasing key tokens in the reward pool (initially about 45 billion tokens). The reward pool has a start time. From the start time, every 365 days, we define it as an annual. In each year, a total of 10% of the remaining amount is released. In each year, the token is released linearly."

High-level definitional specification



$$Y = Z \times b$$

$$Z = T \times 0.9^n \times 0.1$$

$$X = T(1 - 0.9^n)$$

$$X + Z = T(1 - 0.9^{n+1})$$

$$b = \frac{\text{now} - \text{prev}}{\text{next} - \text{prev}}$$

High-level constructive specification

```
/* @input: balance                                // T - C
   *      collectedTokens                         // C
   *      rewardStartTime                         // start
   *      now                                    // now
   *
   * @output: balance'                            // T - C'
   *      collectedTokens'                      // C'
   *
   * @pre-condition: now > rewardStartTime
   */
procedure collectToken () {
    total := collectedTokens + balance           // T
    yearCount := floor (days(now - rewardStartTime) / 365) // n
    fractionOfThisYear := (days(now - rewardStartTime) % 365) / 365 // b

    remainingTokens := total * (0.9 ^ yearCount)
    totalRewardThisYear := remainingTokens * 0.1          // Z
    canExtractThisYear := totalRewardThisYear * fractionOfThisYear // Y
    canExtract := canExtractThisYear + (total - remainingTokens)
                  - collectedTokens                    // Y + X - C

    collectedTokens' := collectedTokens + canExtract      // C'
    balance' := balance - canExtract                     // T - C'
}
```

High-level constructive specification

```
/* @input: balance                                // T - C
   *      collectedTokens                         // C
   *      rewardStartTime                          // start
   *      now                                    // now
   *
   * @output: balance'                            // T - C'
   *      collectedTokens'                      // C'
```

Lemma 1. If the inputs of *collectToken* in Figure 2 satisfies the following equations:

$$\begin{aligned}balance &= T - C \\collectedTokens &= C \\rewardStartTime &= start \\now &= now\end{aligned}$$

then the following holds for the outputs of the function:

$$\begin{aligned}balance' &= T - C' = T - (X + Y) \\collectedTokens' &= C' = X + Y\end{aligned}$$

```
collectedTokens' := collectedTokens + canExtract          // C'
balance' := balance - canExtract                         // T - C'
}
```

Solidity-level functional specification

```
/* @input: uint balance
 *         unit collectedTokens
 *         unit rewardStartTime
 *         unit now
 *
 * @output: unit balance'
 *          unit collectedTokens'
 *
 * @pre-condition: now > rewardStartTime
 */
procedure collectToken() {
    unit total := collectedTokens + balance
    unit yearCount := days(now - rewardStartTime) / 365
    unit fractionOfThisYear365 := days(now - rewardStartTime) % 365

    unit remainingTokens := power(total, 90, 100, yearCount)
    unit totalRewardThisYear := remainingTokens * 10 / 100
    unit canExtractThisYear := totalRewardThisYear * fractionOfThisYear365 / 365
    unit canExtract := canExtractThisYear + (total - remainingTokens)
                           - collectedTokens

    collectedTokens' := collectedTokens + canExtract
    balance' := balance - canExtract
}

// return (conceptually): acc * ((base_n / base_d) ^ exp)
function power(acc, base_n, base_d, exp) {
    if exp == 0 {
        return acc
    } else {
        return power(acc * base_n / base_d, base_n, base_d, exp - 1)
    }
}
```

Solidity-level functional specification

```
/* @input: uint balance
 *         unit collectedTokens
 *         unit rewardStartTime
 *         unit now
```

Lemma 2. If the inputs of `collectToken` in Figure 3 satisfies the following equations:

$$\text{balance} = T - C$$

$$\text{collectedTokens} = C$$

$$\text{rewardStartTime} = \text{start}$$

$$\text{now} = \text{now}$$

then the following holds for the outputs of the function:

$$\begin{aligned} (T - C') - 10 &< \text{balance}' &< (T - C') + 3 \\ C' - 3 &< \text{collectedTokens}' &< C' + 10 \end{aligned}$$

Lemma 3. Suppose two `collectToken` function (as shown in Figure 3) calls are made at times t and t' , respectively, where $t < t'$. Let r and r' be the output values of `canExtractThisYear + (total - remainingTokens)` for each function call at t and t' , respectively. Assume that `total` does not decrease between t and t' . Then, $r \leq r'$.

Corollary 1. If `balance` does not decrease since the last `collectToken` function call, the following always hold:

$$\text{canExtractThisYear} + (\text{total} - \text{remainingTokens}) \geq \text{collectedTokens}$$

```
} else {
    return power(acc * base_n / base_d, base_n, base_d, exp - 1)
}
}
```

EVM-level functional specification

```
k: #execute => (RETURN RET_ADDR:Int 32 ~> _)
output: _
memoryUsed: 0 => _
callData: #abiCallData("collectToken", #uint256(NOW), #uint256(START))
wordStack: .WordStack => _
localMem: .Map => .Map[ RET_ADDR := #asByteStackInWidth(1, 32) ] _:Map
pc: 0 => _
gas: GASCAP => _
log: _
refund: _ => _
storage:
  #hashedLocation("Solidity", {_COLLECTEDTOKENS}, .IntList) |-> (COLLECTED => COLLECTED +Int VALUE)
  #hashedLocation("Solidity", {_BALANCE}, .IntList) |-> (BAL => BAL -Int VALUE)
  _:Map
requires:
  andBool 0 <=Int COLLECTED andBool COLLECTED <Int (2 ^Int 256)
  andBool 0 <=Int BAL andBool BAL <Int (2 ^Int 256)
  andBool 0 <=Int START andBool START <Int (2 ^Int 256)
  andBool 0 <=Int (COLLECTED +Int BAL) andBool (COLLECTED +Int BAL) *Int 3153600 <Int (2 ^Int 256)
  andBool 0 <Int (NOW -Int START) andBool (NOW -Int START) <Int (2 ^Int 256)
  andBool #accumulatedReleasedTokens(BAL, COLLECTED, START, NOW) >Int COLLECTED +Int 3
  andBool #accumulatedReleasedTokens(BAL, COLLECTED, START, NOW) <Int (BAL +Int COLLECTED) -Int 10
  andBool GASCAP >=Int (293 *Int ((NOW -Int START) /Int 3153600)) +Int 43000
ensures: VALUE ==Int @canExtractThisYear(COLLECTED +Int BAL, NOW, START)
          +Int BAL -Int @remainingTokens(COLLECTED +Int BAL, NOW, START)
```

Developing new formal method

Heap manipulating programs

Smart contracts

Compiler verification

Specification refinement

Deductive Verifier

Program Equivalence Checker

[TR'17]

C

Java

JavaScript

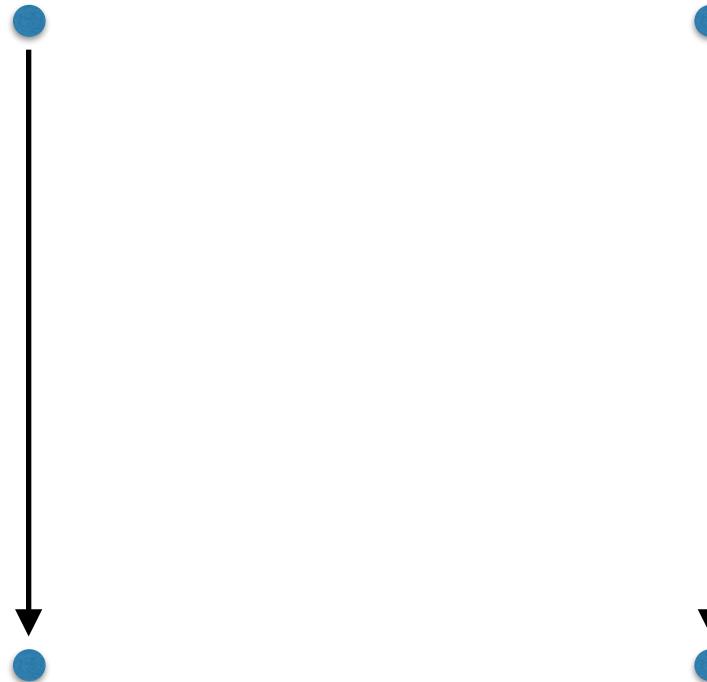
WASM

JAUS

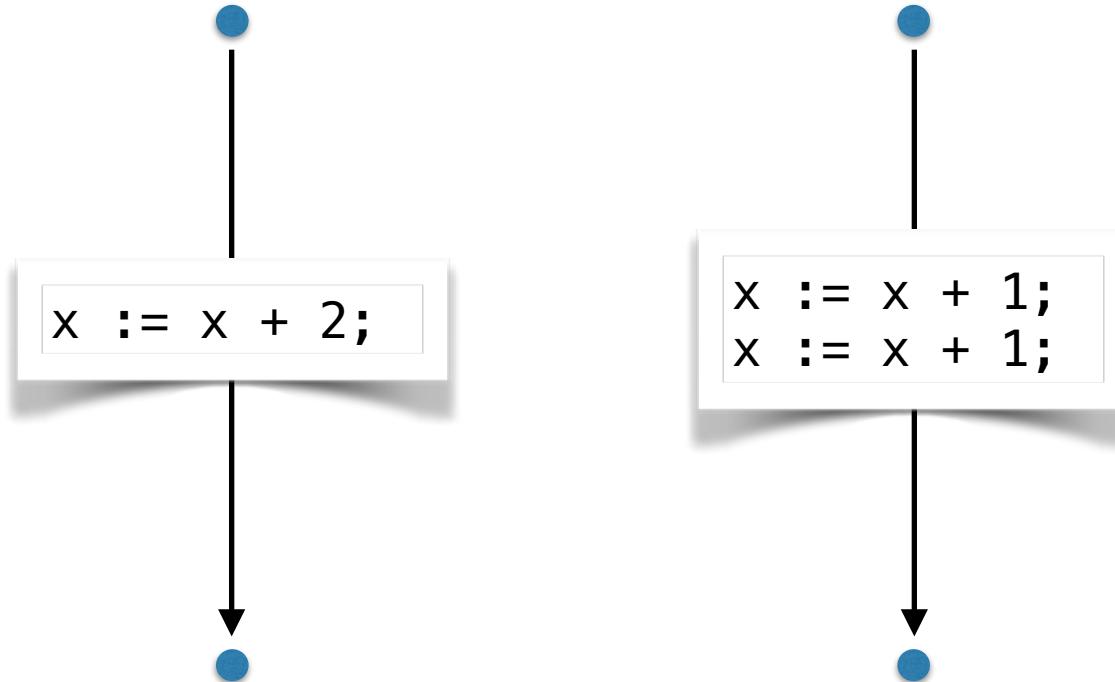
Vyper

EVM

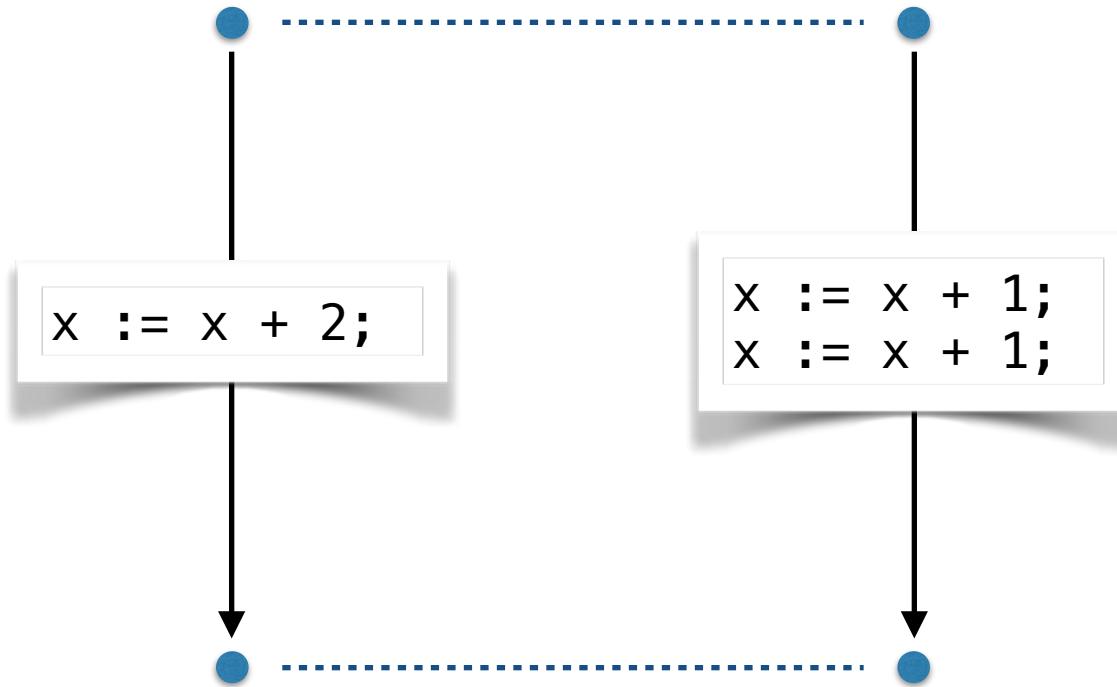
Deterministic, terminating programs



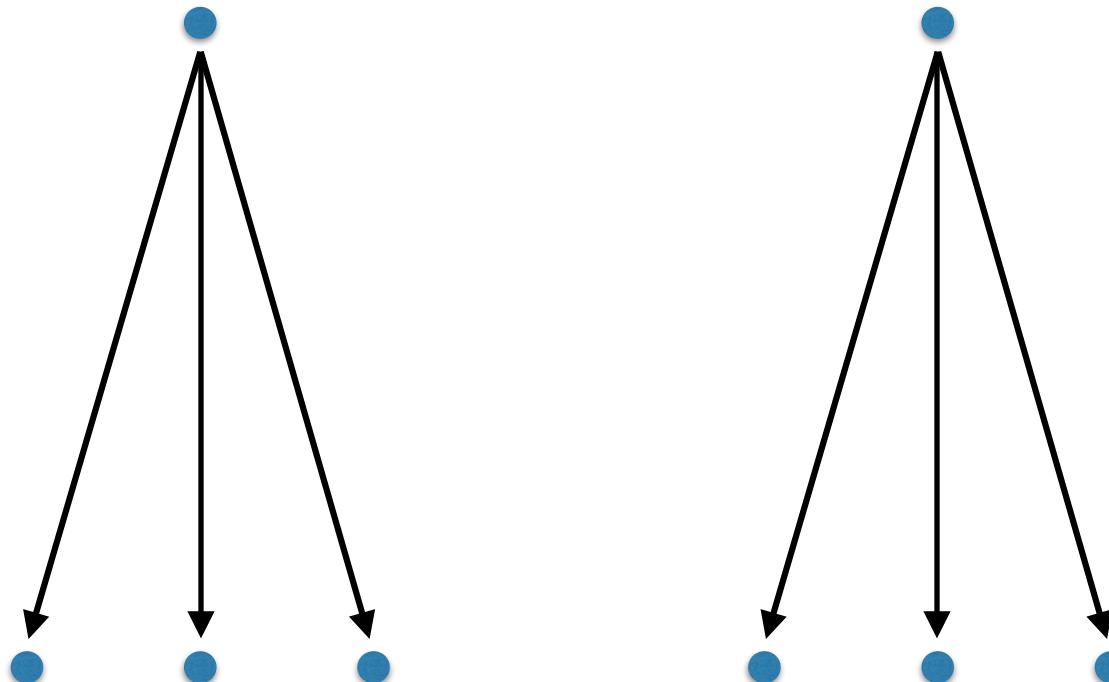
Deterministic, terminating programs



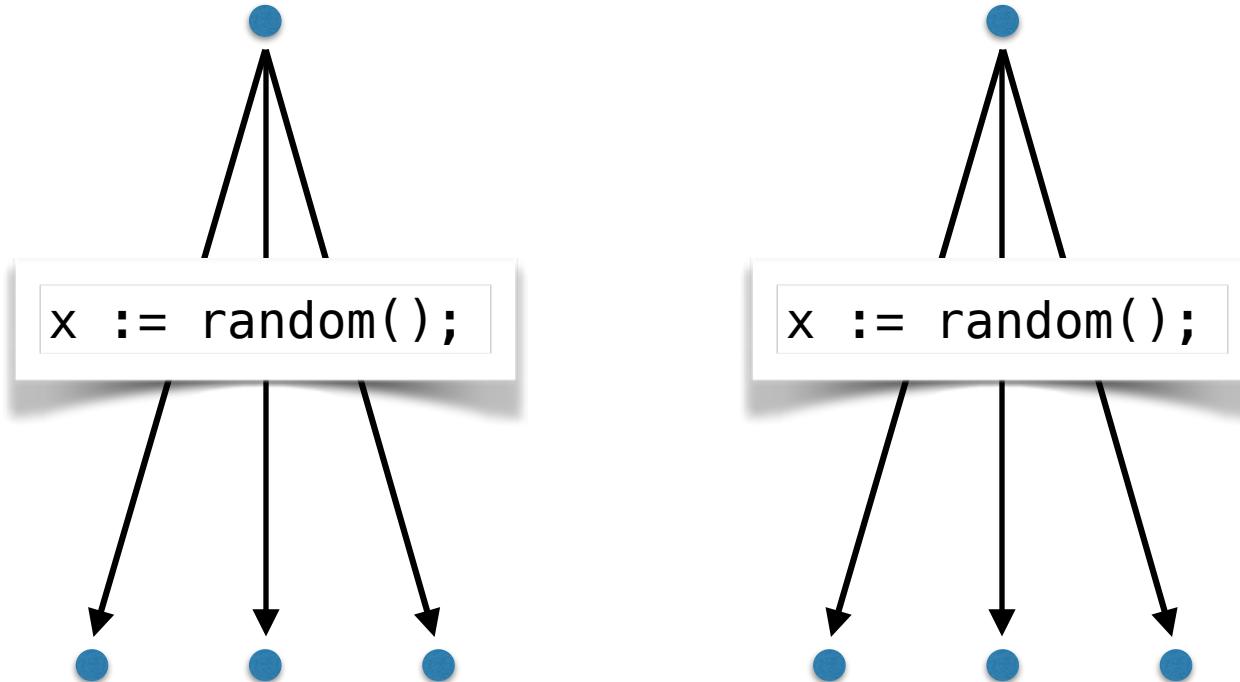
Deterministic, terminating programs



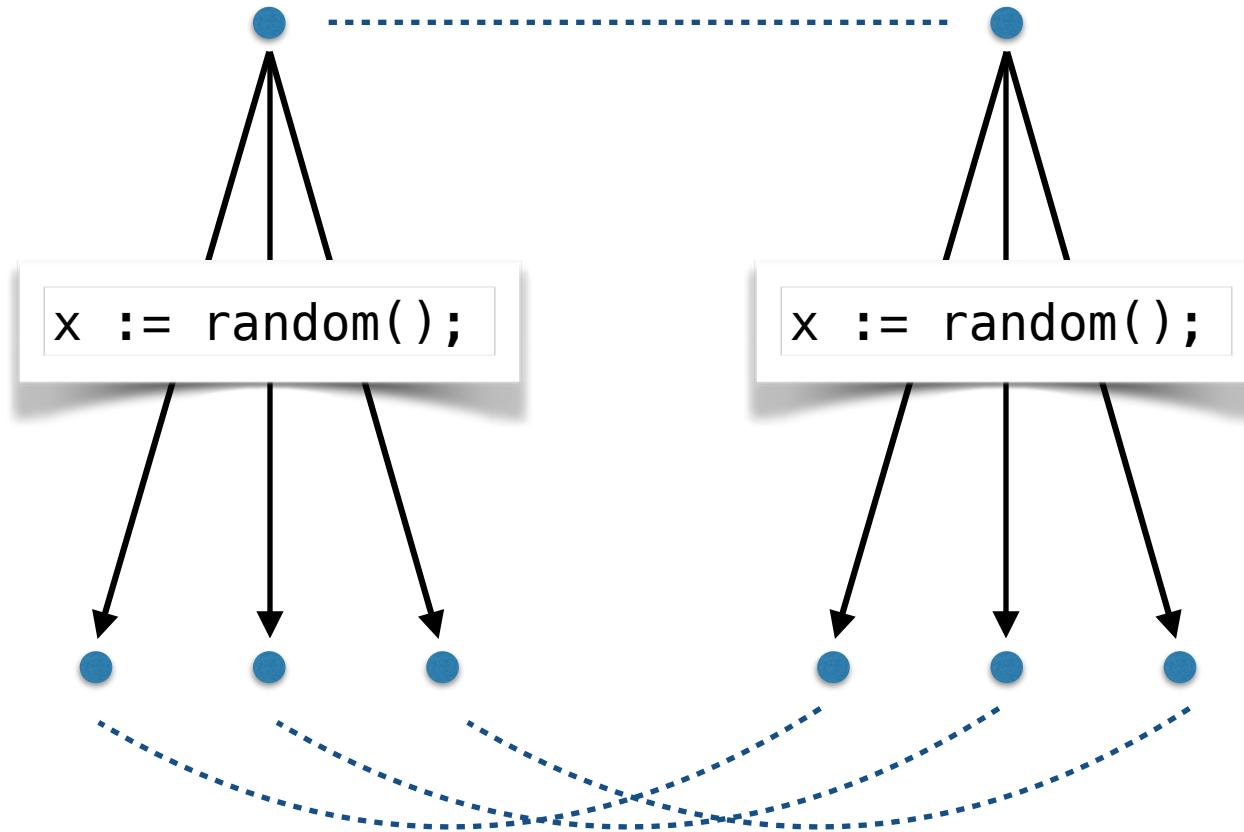
Nondeterministic programs



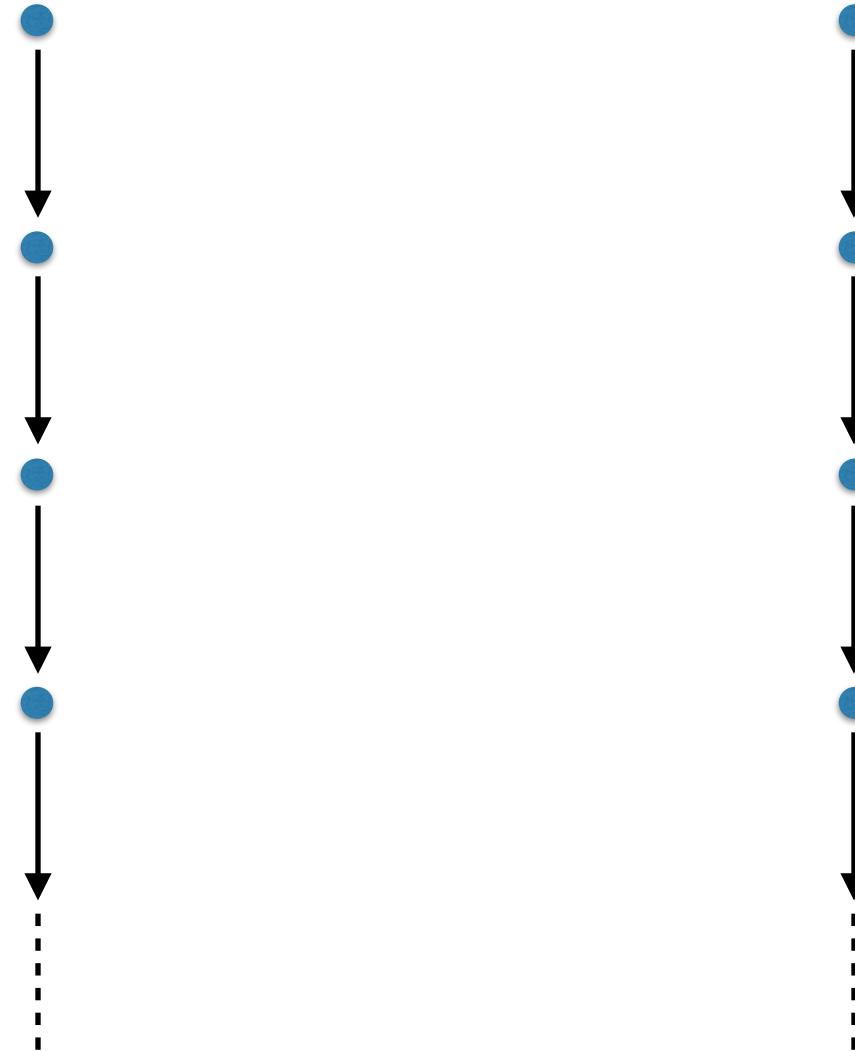
Nondeterministic programs



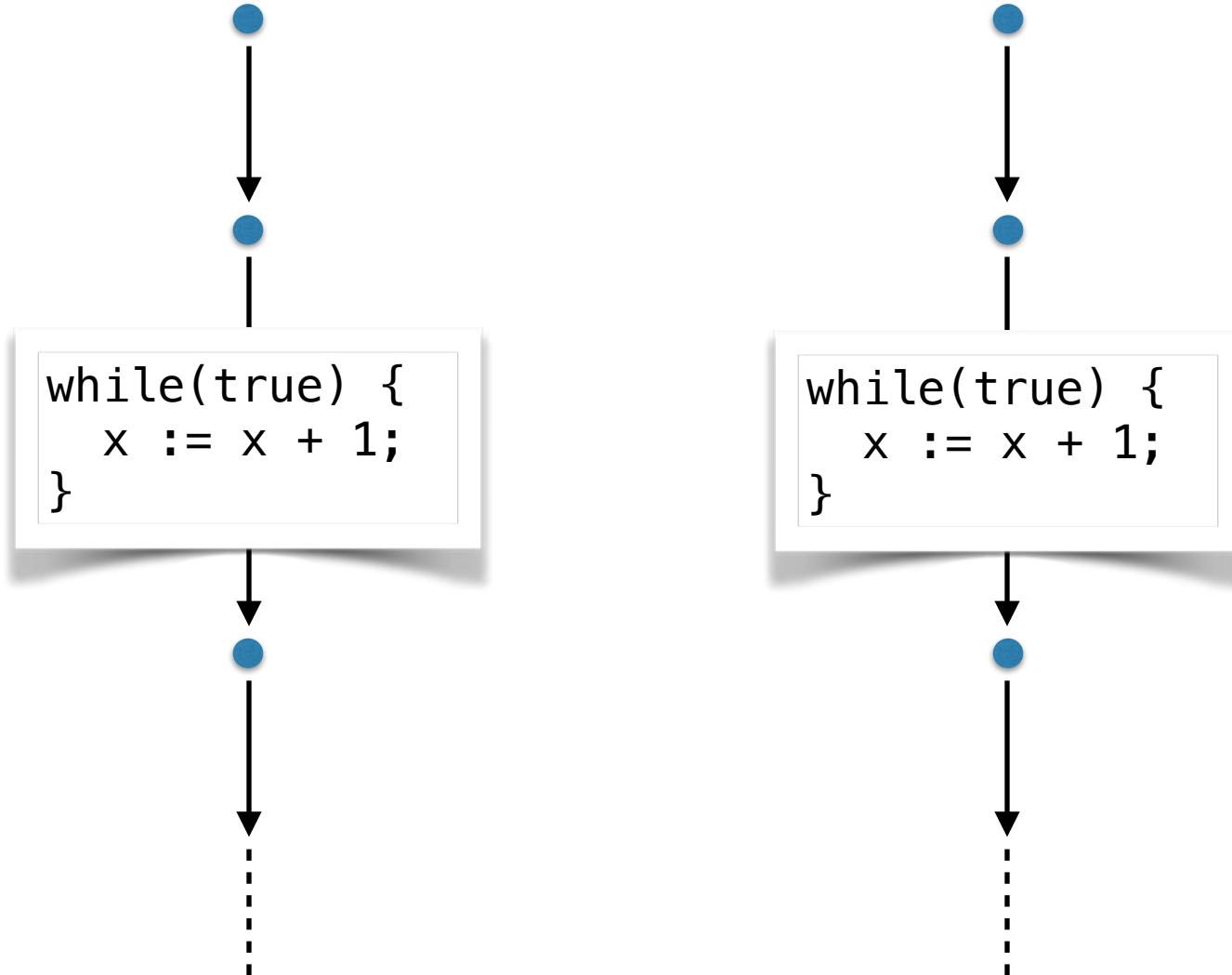
Nondeterministic programs



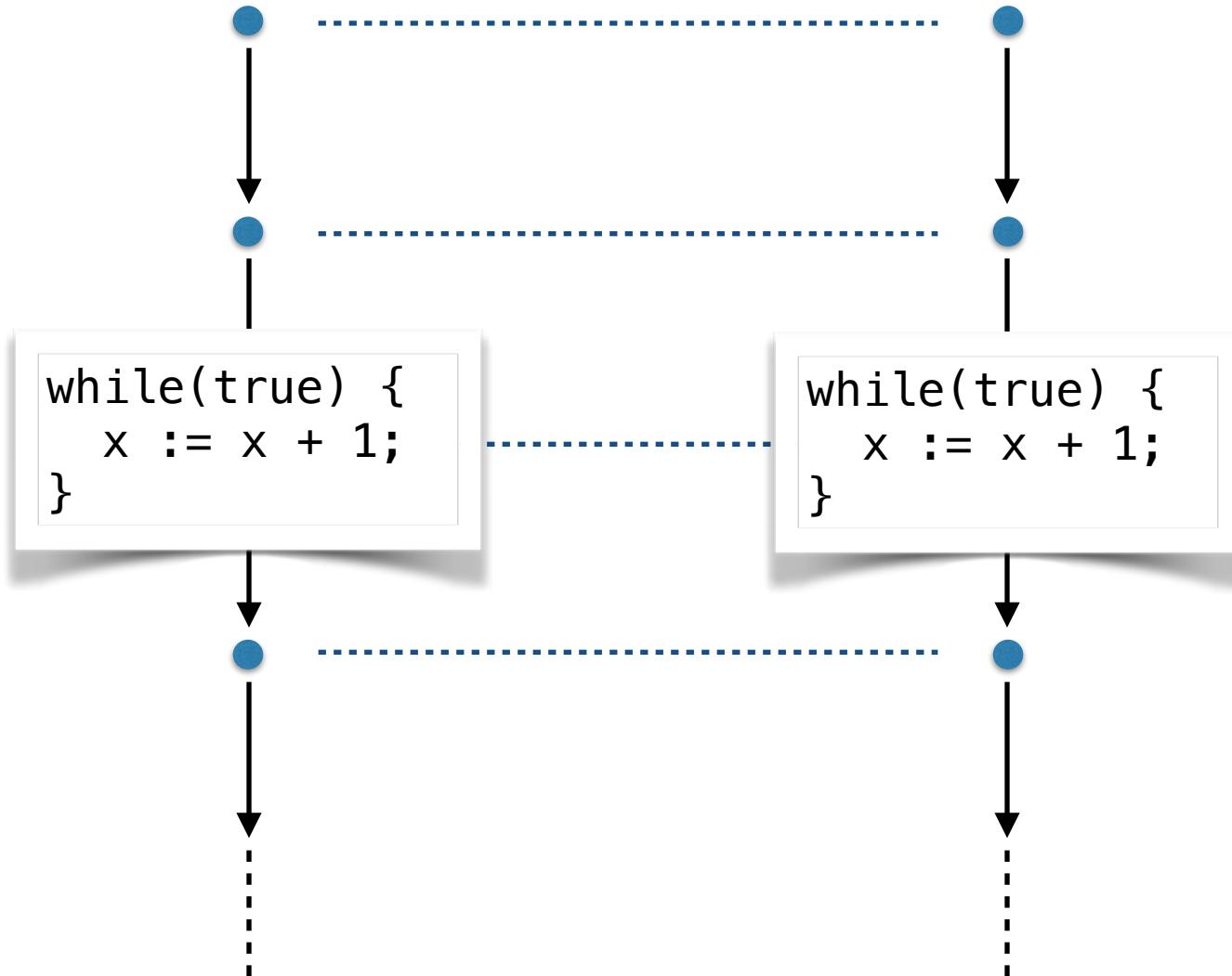
Non-terminating programs



Non-terminating programs



Non-terminating programs



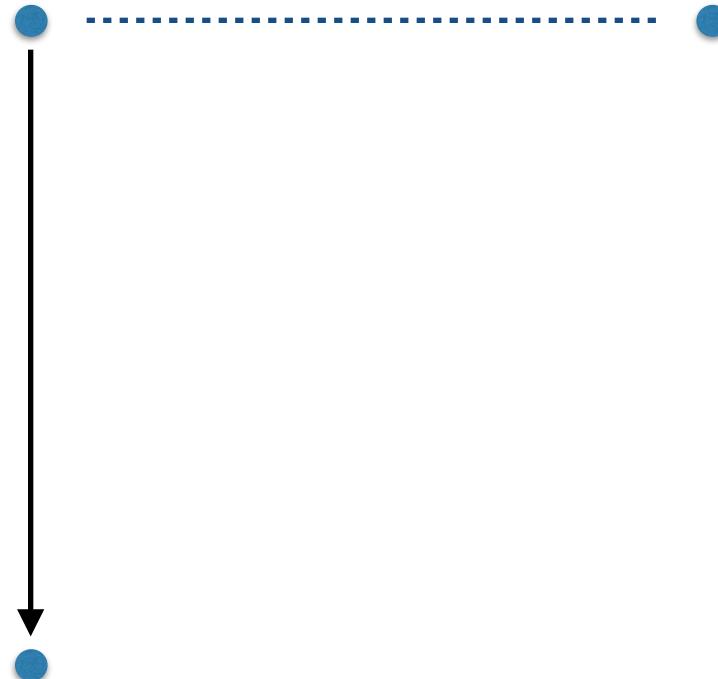
Bisimulation

for each pair of related states



Bisimulation

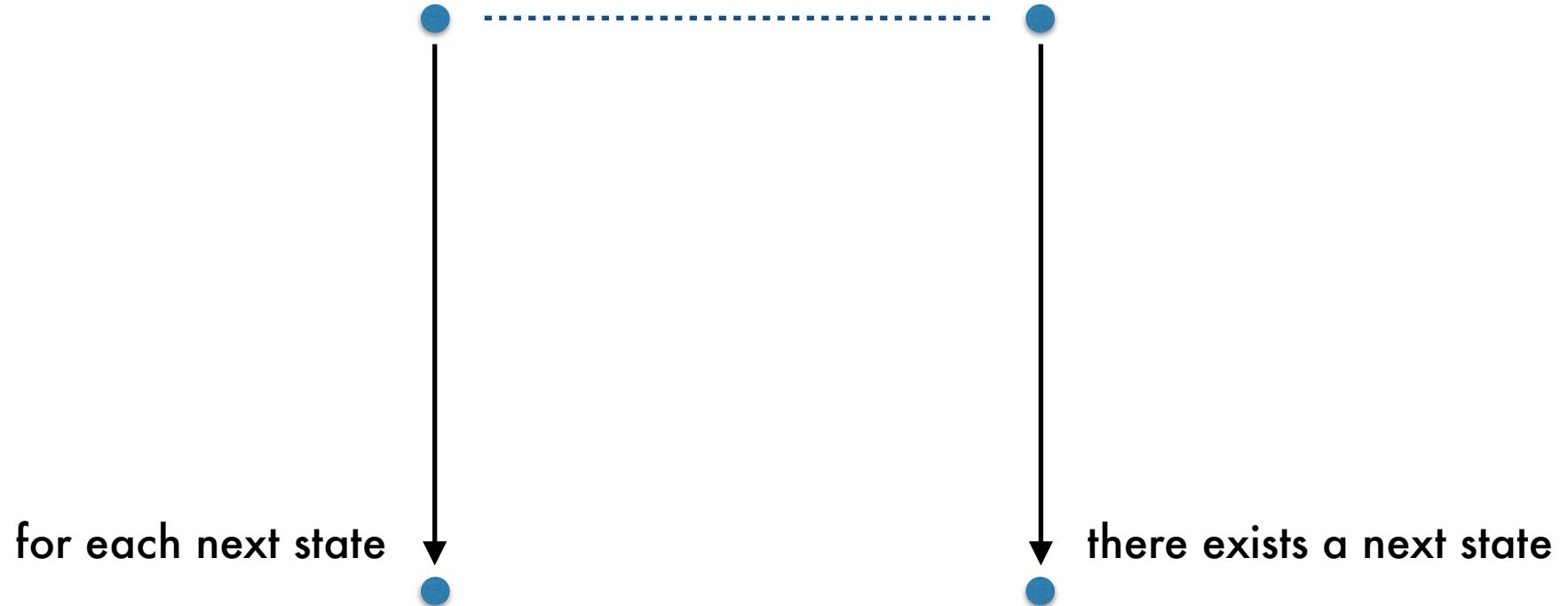
for each pair of related states



for each next state

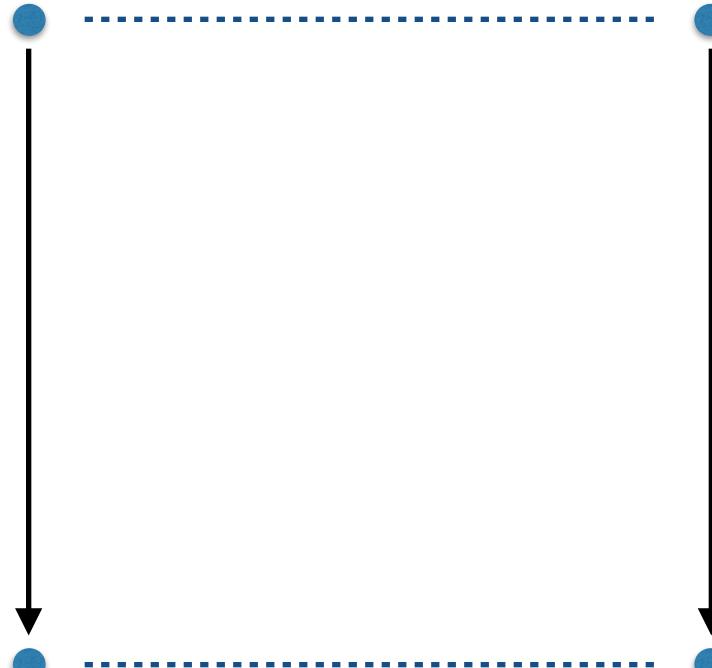
Bisimulation

for each pair of related states



Bisimulation

for each pair of related states



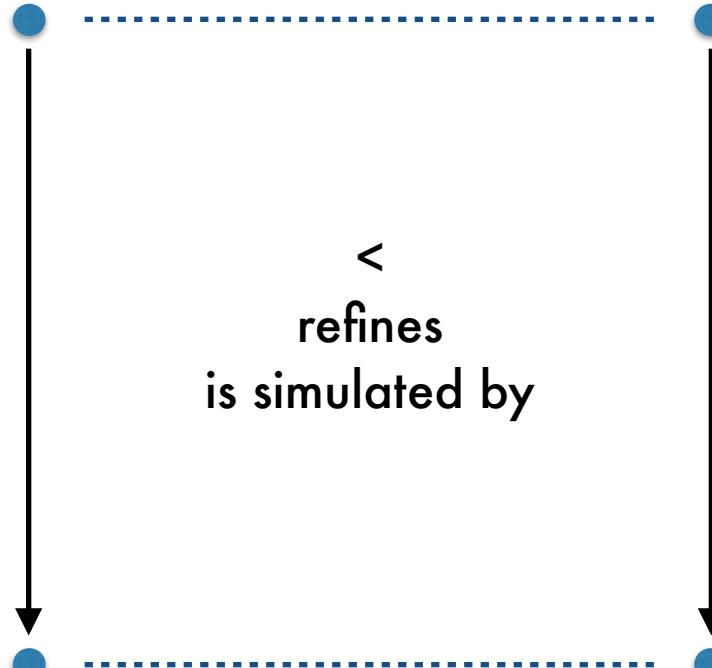
for each next state

there exists a next state

such that two states are related

Bisimulation

for each pair of related states

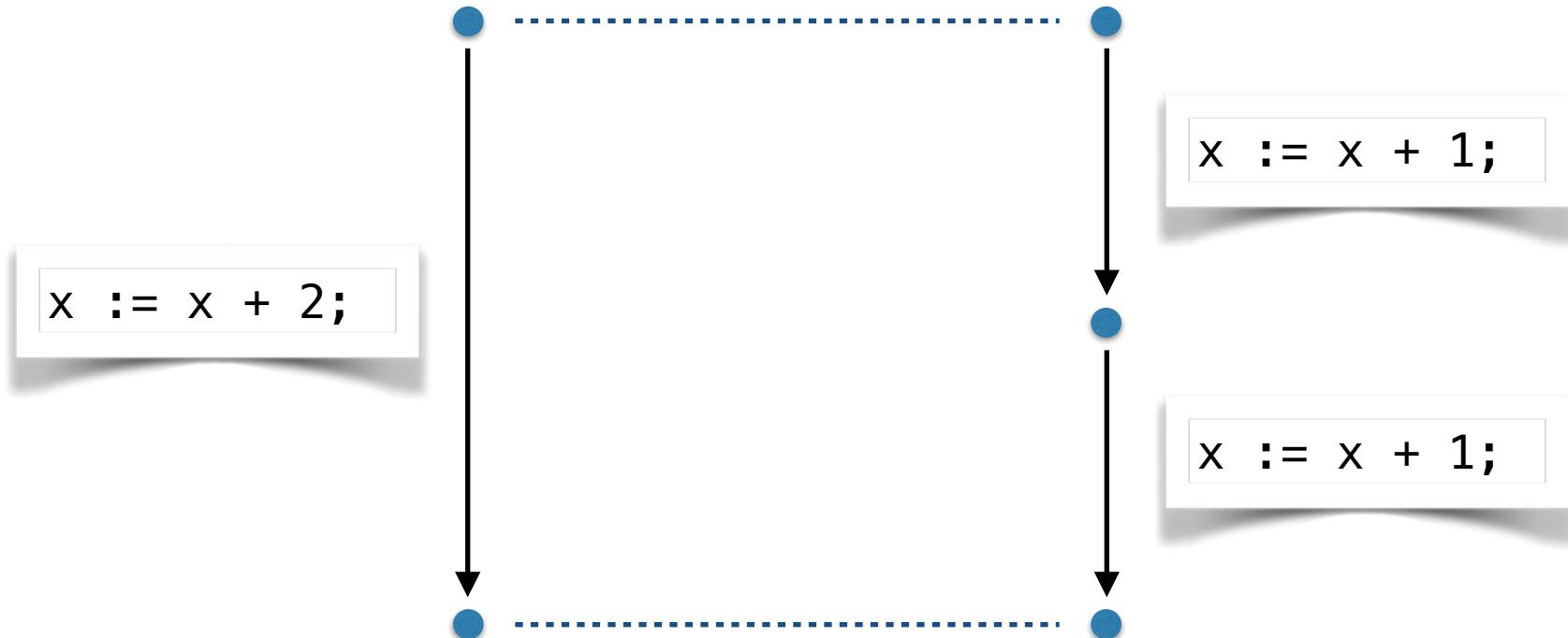


for each next state

such that two states are related

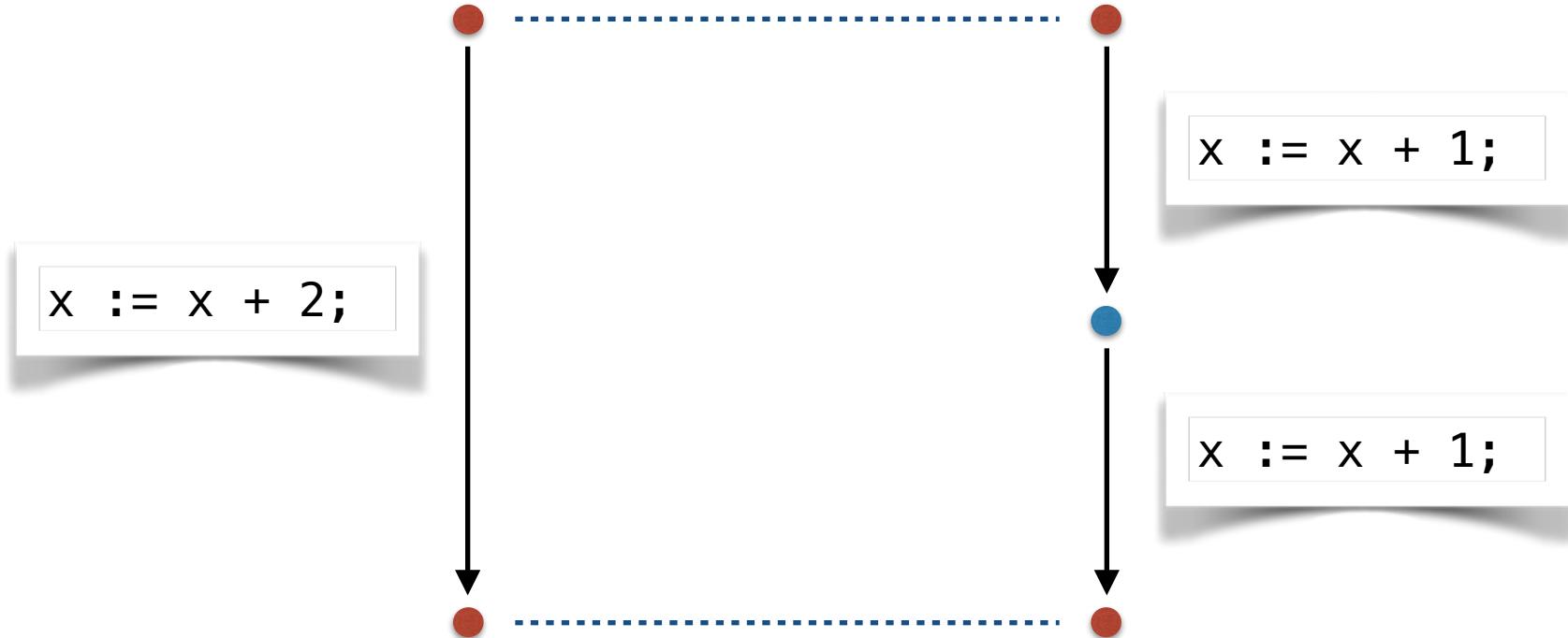
there exists a next state

Bisimulation?



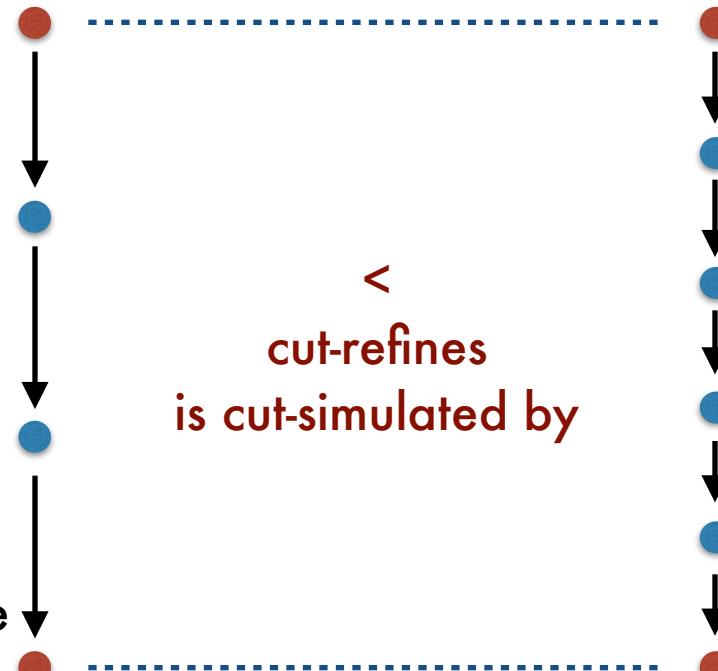
Cut-bisimulation

↑
considering only **relevant** states



Cut-bisimulation

for each pair of related **relevant** states

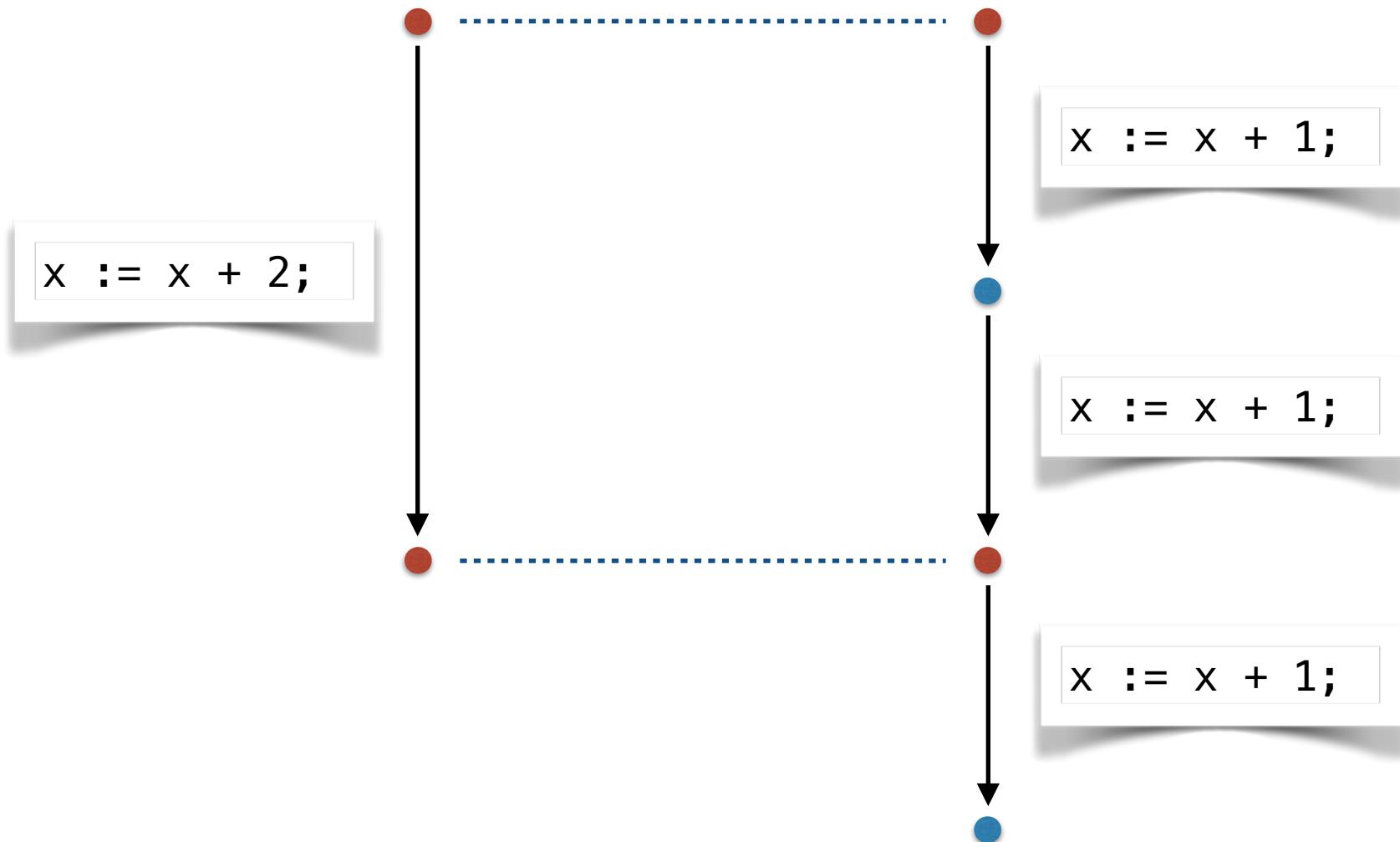


for each next **relevant** state

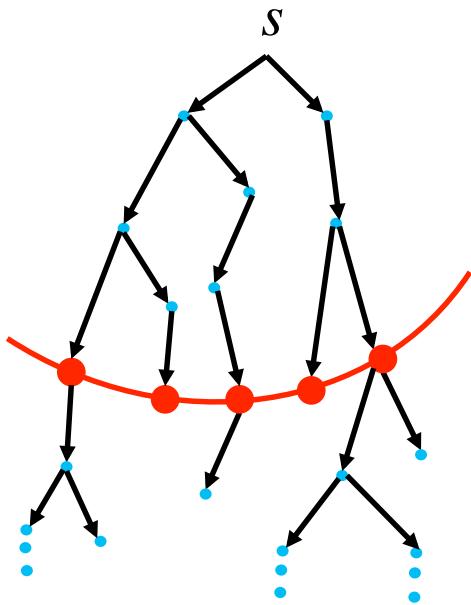
such that two **relevant** states are related

there exists a next **relevant** state

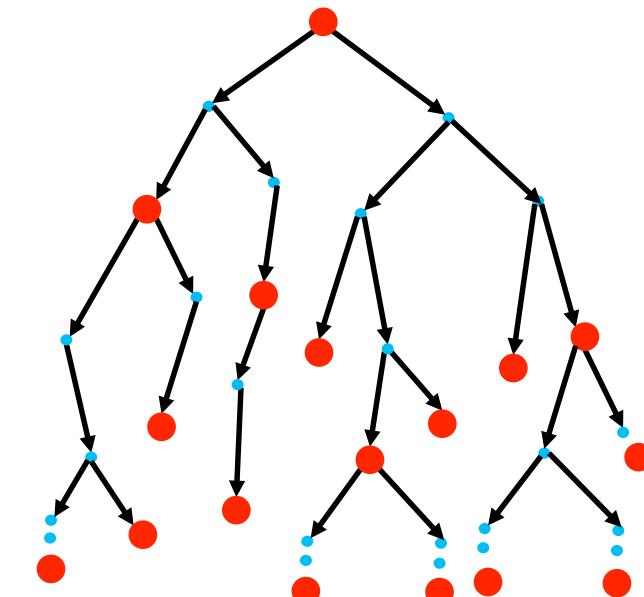
Well-formedness of relevant states



Cut



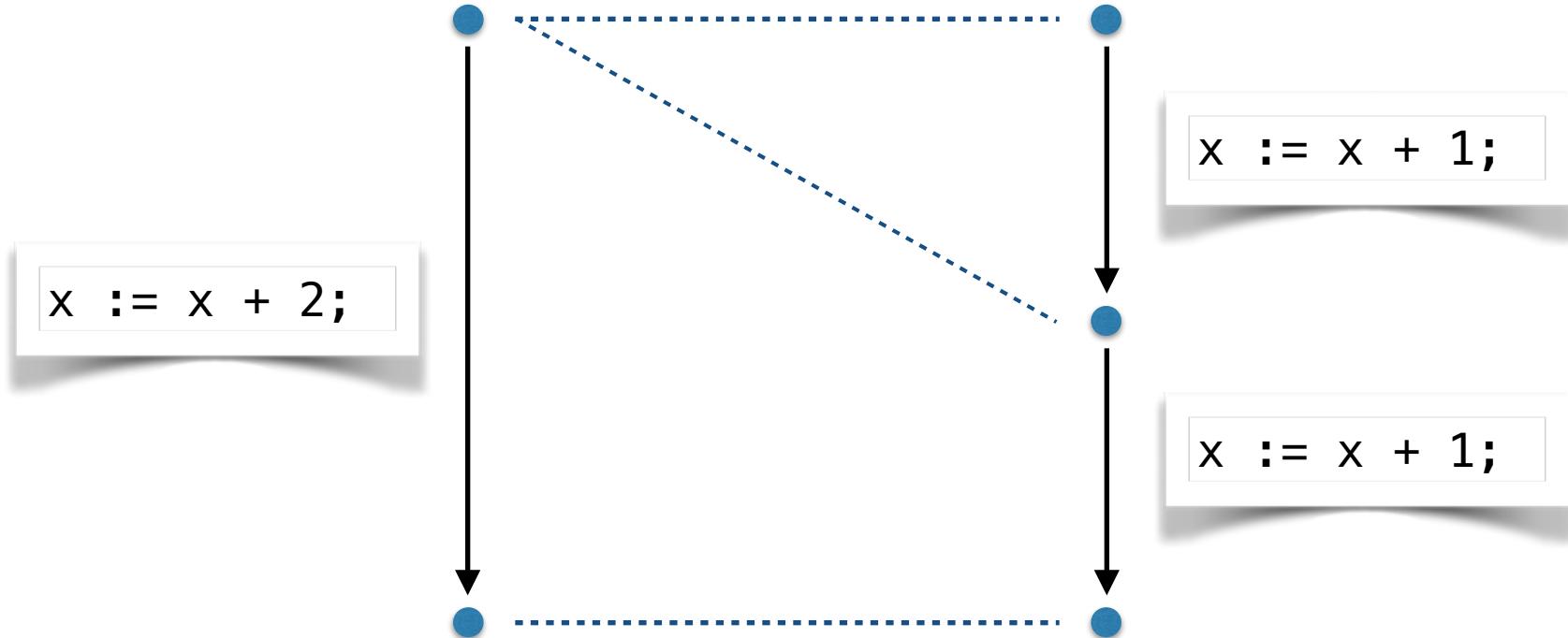
Cut for s



Cut for all

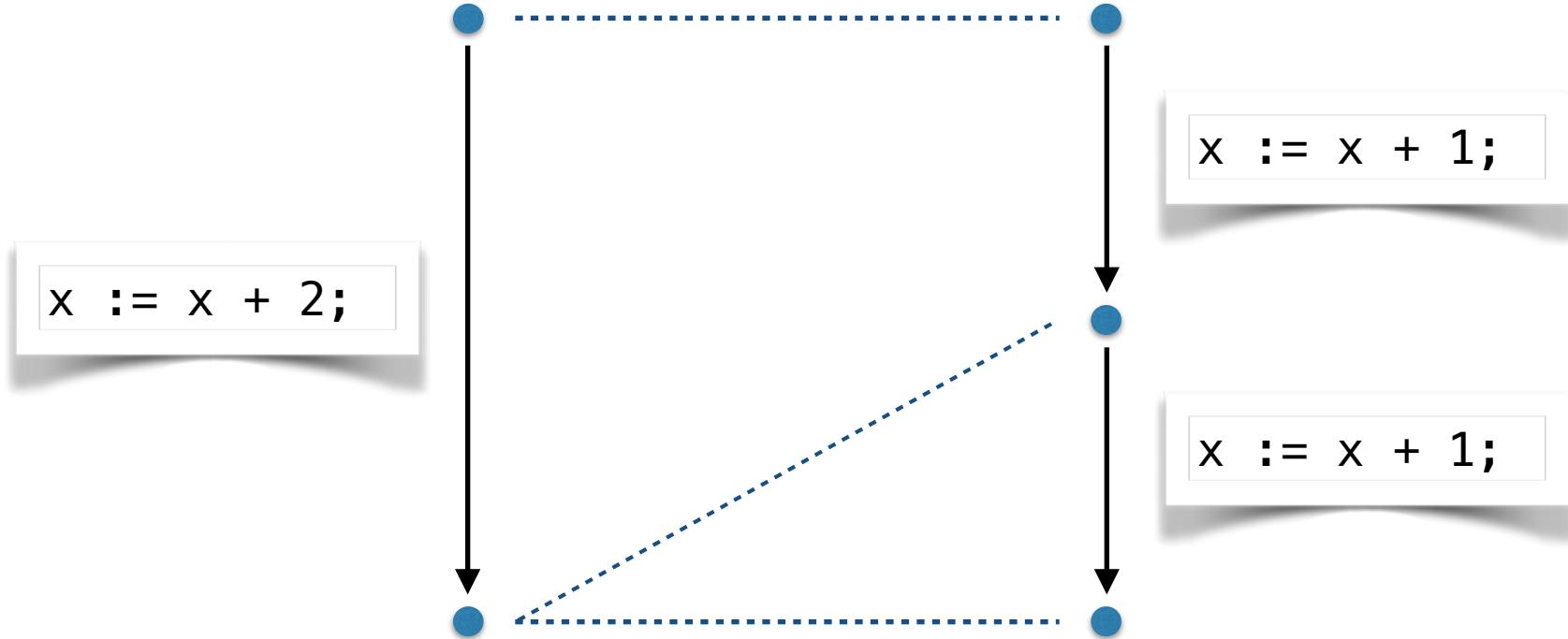
Stuttering bisimulation [BCG'88, dNV'90]

two possible stuttering bisimulations: 1 of 2

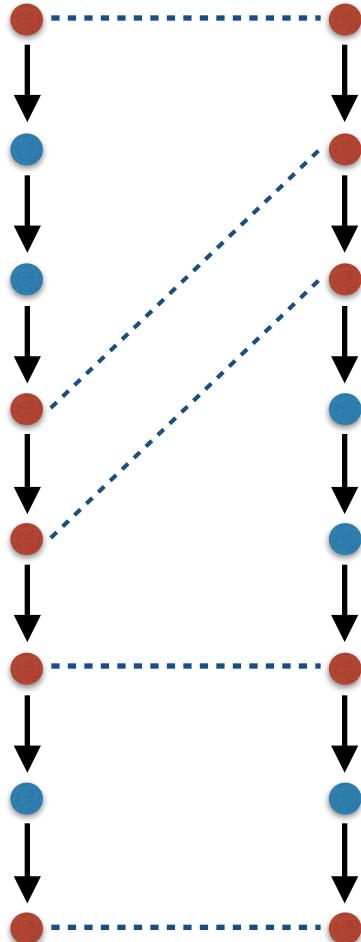


Stuttering bisimulation [BCG'88, dNV'90]

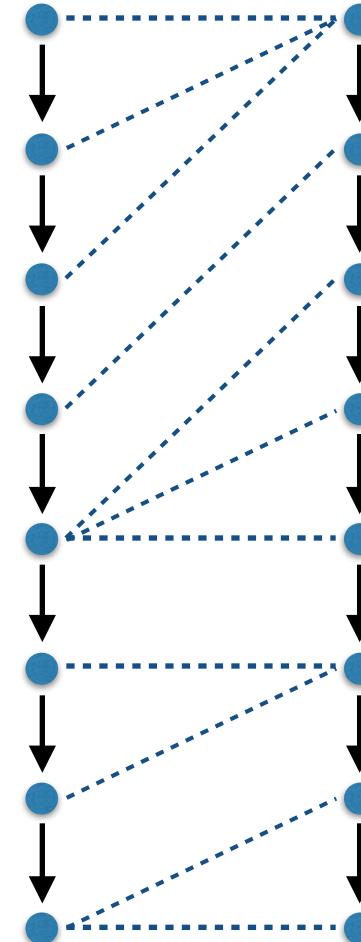
two possible stuttering bisimulations: 2 of 2



Cut-bisimulation vs stuttering bisimulation



vs



[Namjoshi'97]

KEQ: universal program equivalence checker

- Inputs
 - Two language semantics
 - Two programs
 - A relation (called synchronization points)
- Output
 - True if the relation is a cut-(bi)simulation
 - Can be used for translation validation for compiler

Translation validation [Pnueli et al.'98]

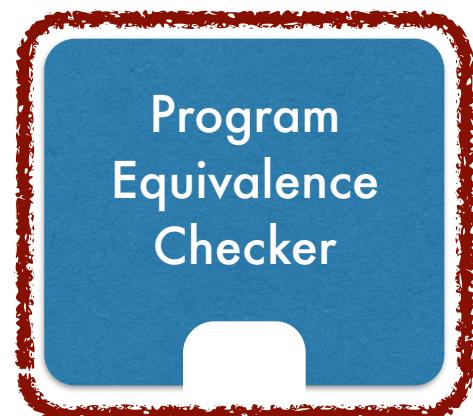
- Validate each instance of compilation
- Verify equivalence of input/output programs
 - KEQ can be used for checking equivalence
- Practical advantages for compiler verification

Current results

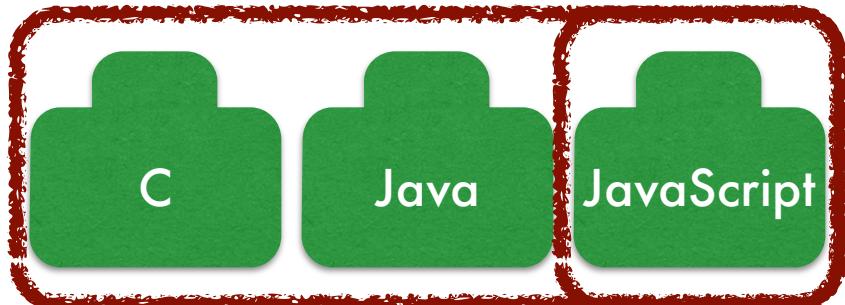
[TR'17] [TR'18]



[OOPSLA'16]



[TR'17]

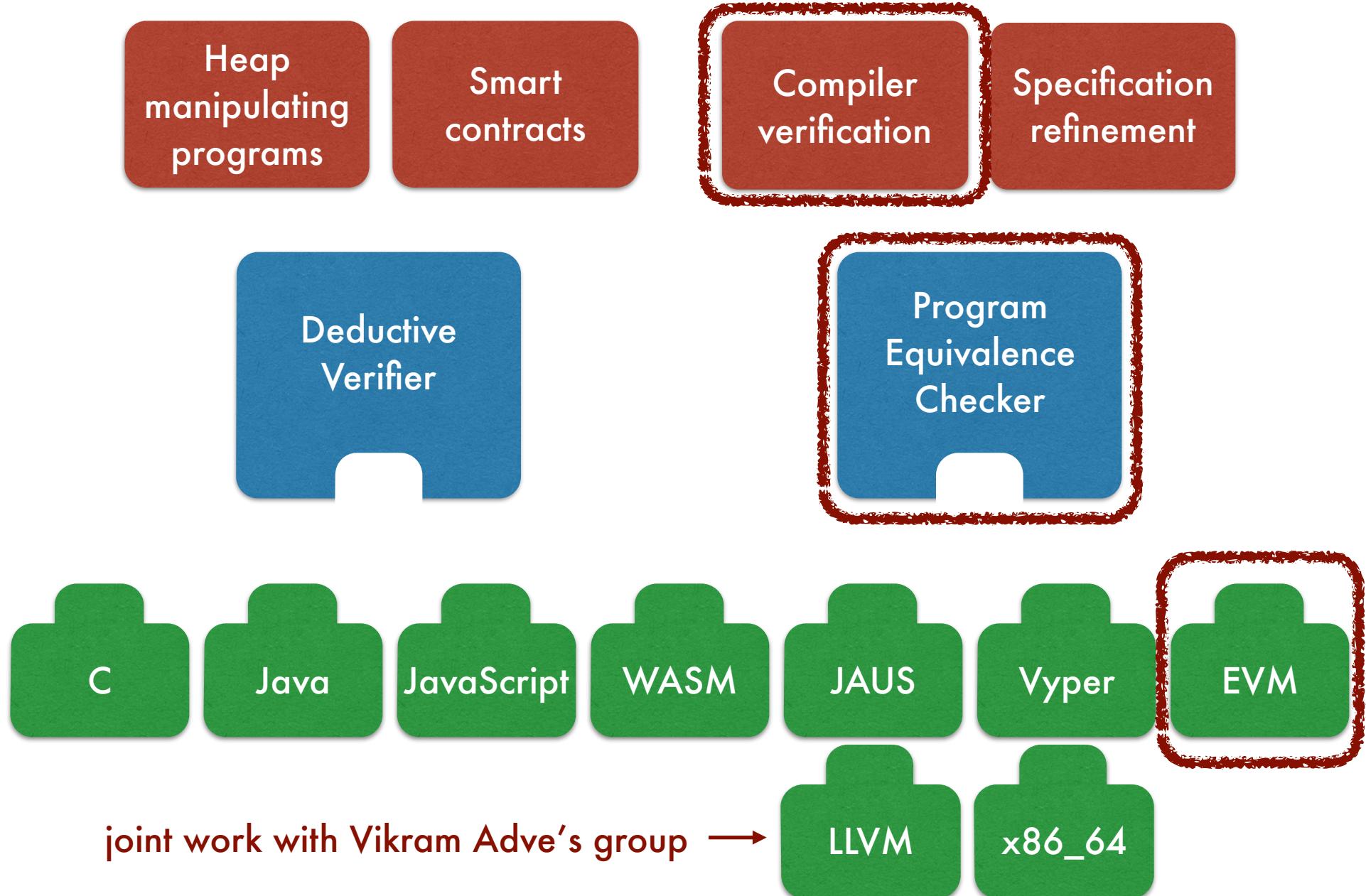


[PLDI'15]



[CSF'18]

Translation validation for compiler (ongoing)



Translation validation for compiler

- Vyper compiler verification
- Verify compatibility between
 - Vyper and Solidity compilers
 - Different versions of the same compiler

Conclusion

- Goal: demonstrate/improve the scalability of language-parametric formal methods
- Specified various real-world language semantics
- Improved existing universal deductive program verifier
 - Instantiated with various languages, and applied to high-profile real-world applications
- Newly developed a cross-language program equivalence checker with a novel notion of cut-bisimulation
 - Will evaluate in the domain of Ethereum smart contracts

Thank you