

Language-Parametric Formal Methods in the Field

Thesis Proposal

Daejun Park
April 20, 2018



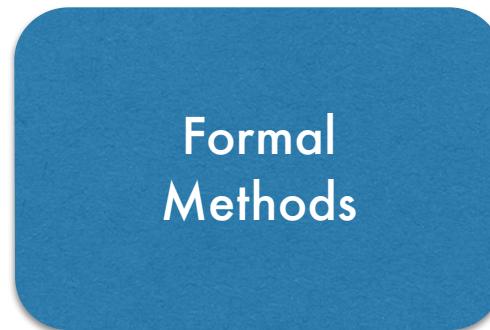
Reusability problem of formal tools

- 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

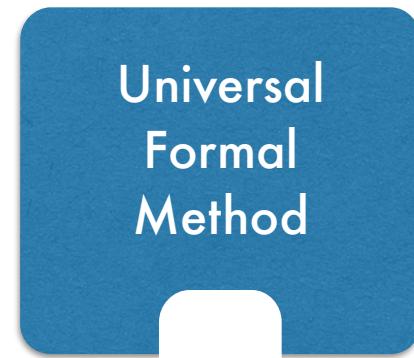


- C (c11, gcc, clang, ...)
 - Java (6, 7, 8, ...)
 - JavaScript (ES5, ES6, ...)
 - ...
-
- Four lines connect the four items above to the three items below. The first three items each have two lines connecting them to the first three items in the list below. The fourth item has one line connecting it to the last item in the list below.
- Deductive verification
 - Model checking
 - Program equivalence checking
 - ...

Defined/implemented once, and reused for all others

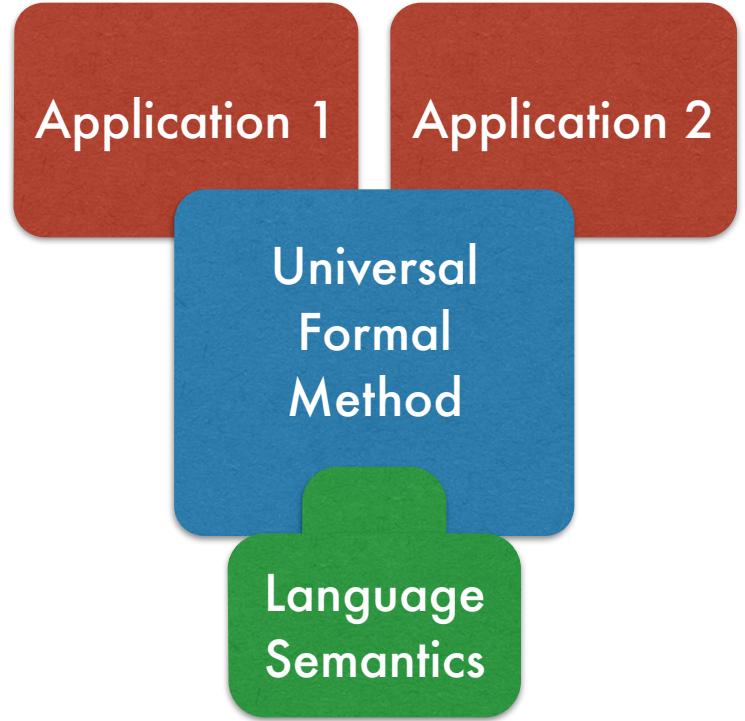
Thesis work

- 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



Thesis work

- 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



Thesis work

Heap
manipulating
programs

Smart
contracts

Compiler
verification

Specification
refinement

Deductive
Verifier

Program
Equivalence
Checker

C

Java

JavaScript

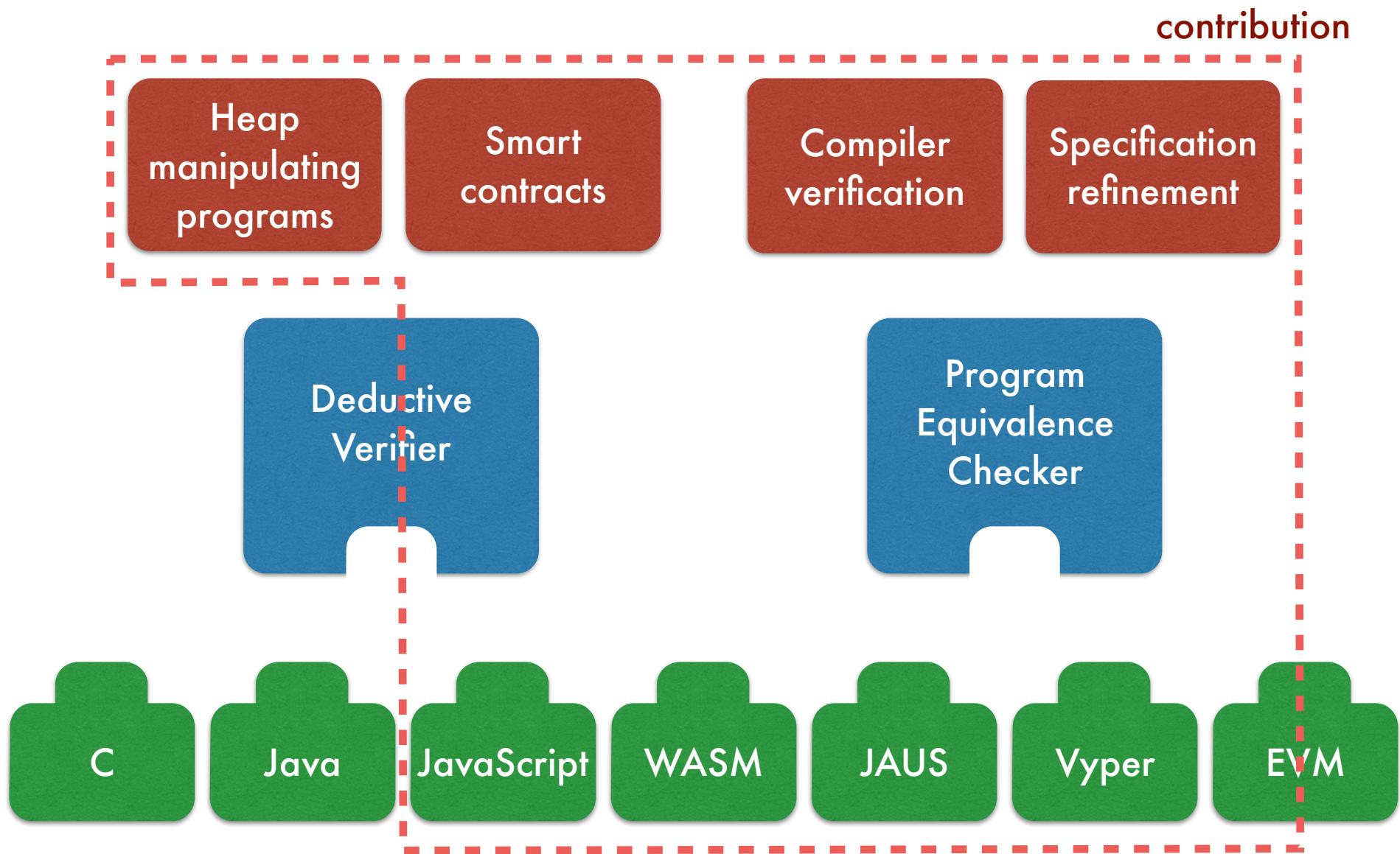
WASM

JAUS

Vyper

EVM

Thesis work



Current Results

Specifying language semantics

Heap manipulating programs

Smart contracts

Compiler verification

Specification refinement

Deductive Verifier

Program Equivalence Checker

C

Java

JavaScript

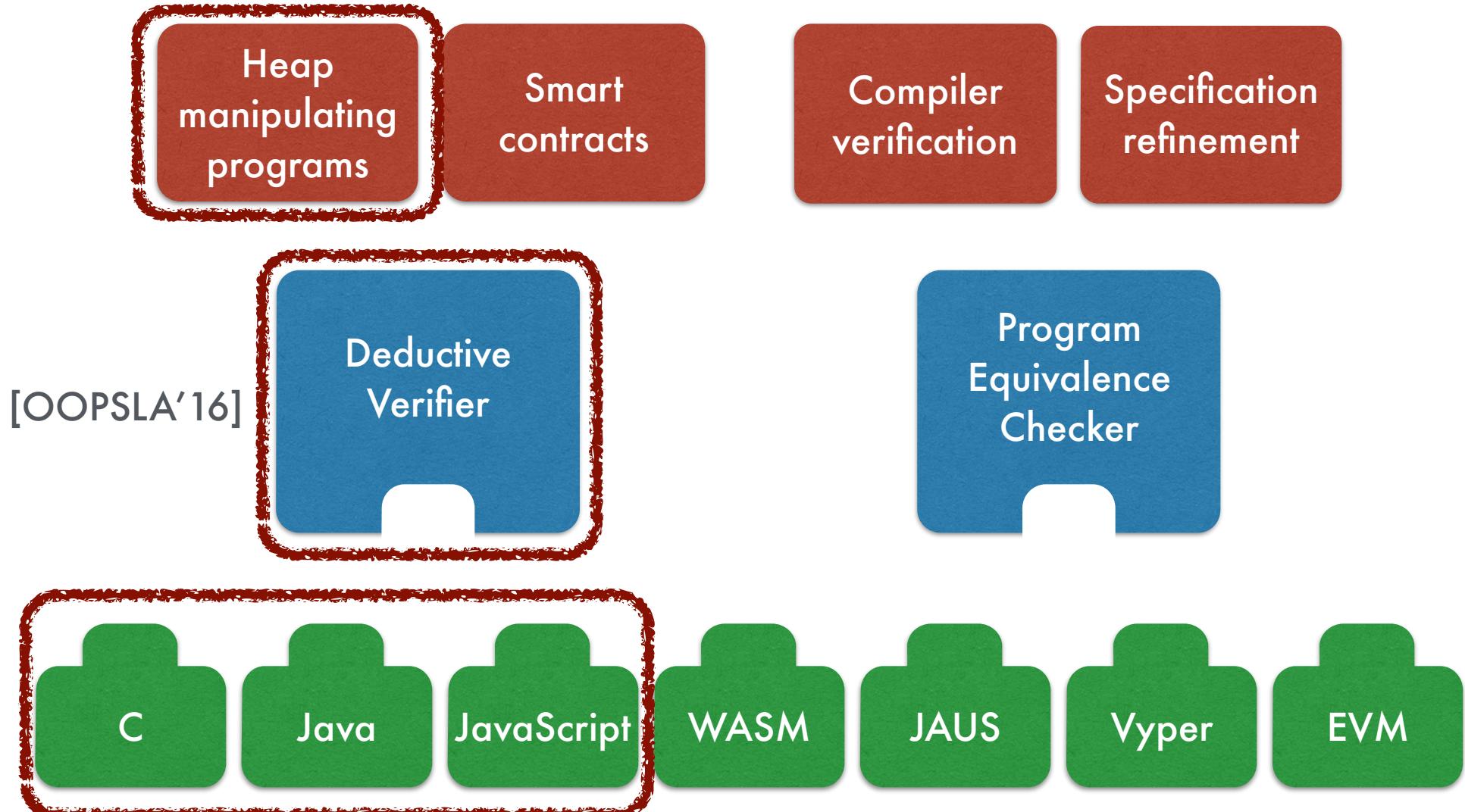
WASM

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

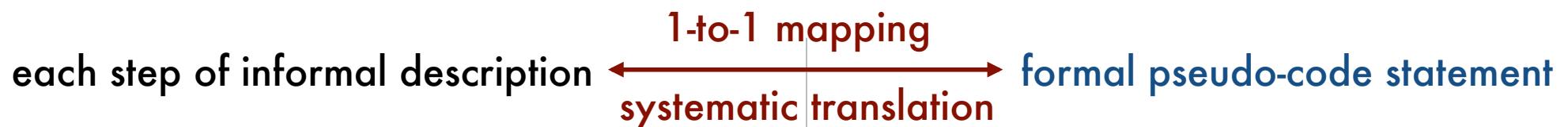
JAUS

Vyper

EVM

KJS: complete formal semantics of JavaScript

KJS *faithfully* formalizes ECMAScript 5.1 standard.



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

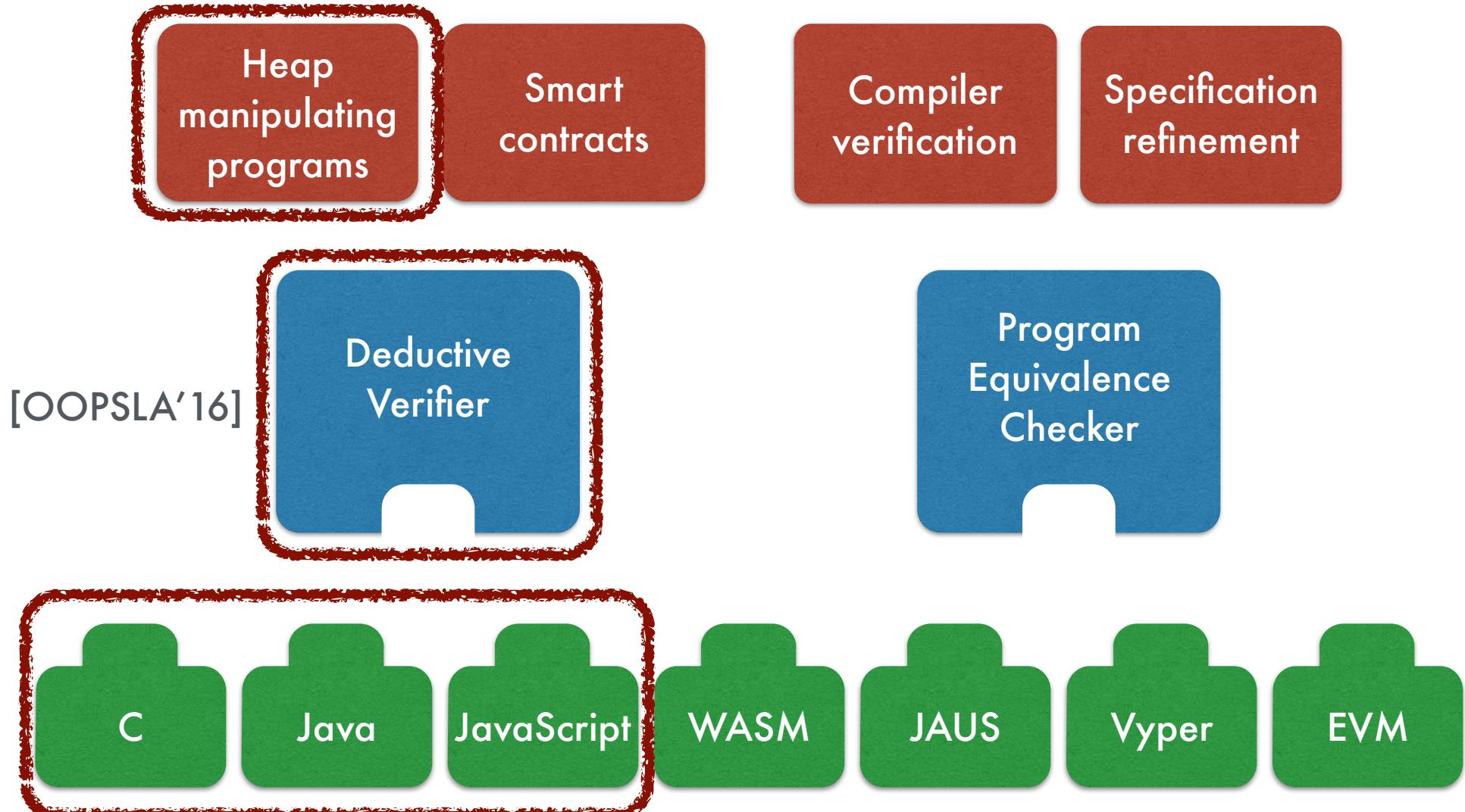
Completeness

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% | ✗ |

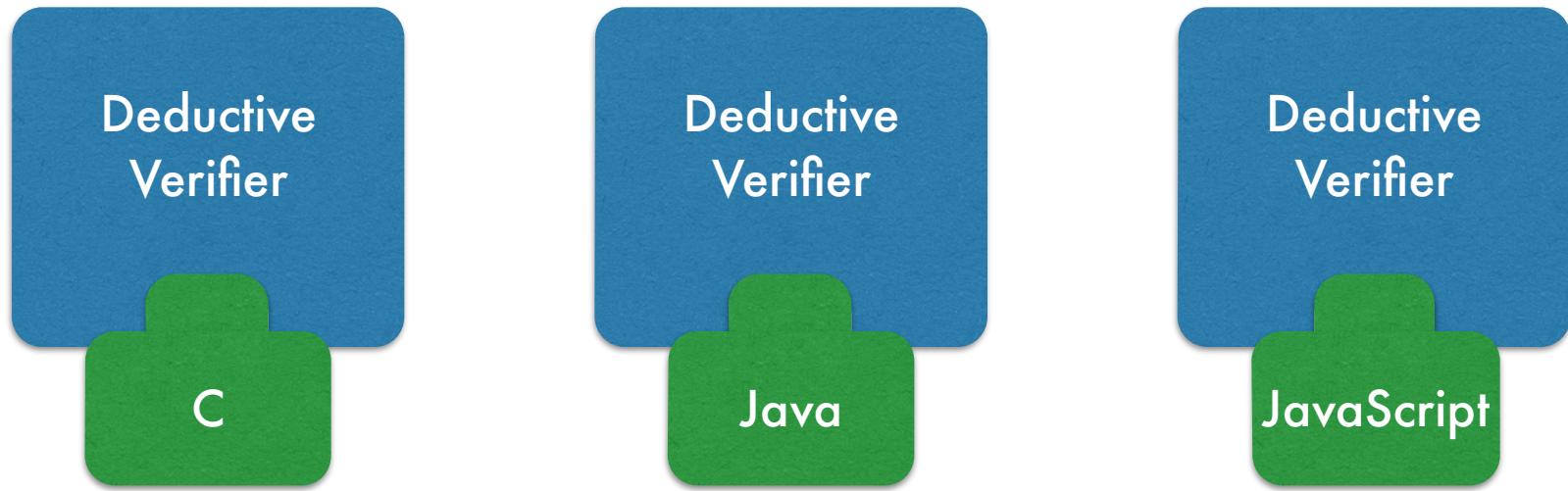
Took me only four months.

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

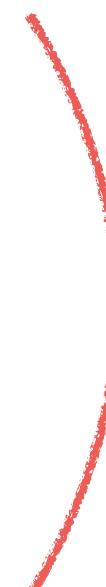
```
syntax Int ::= nthByte(Int, Int, Int) [function]
```

```
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

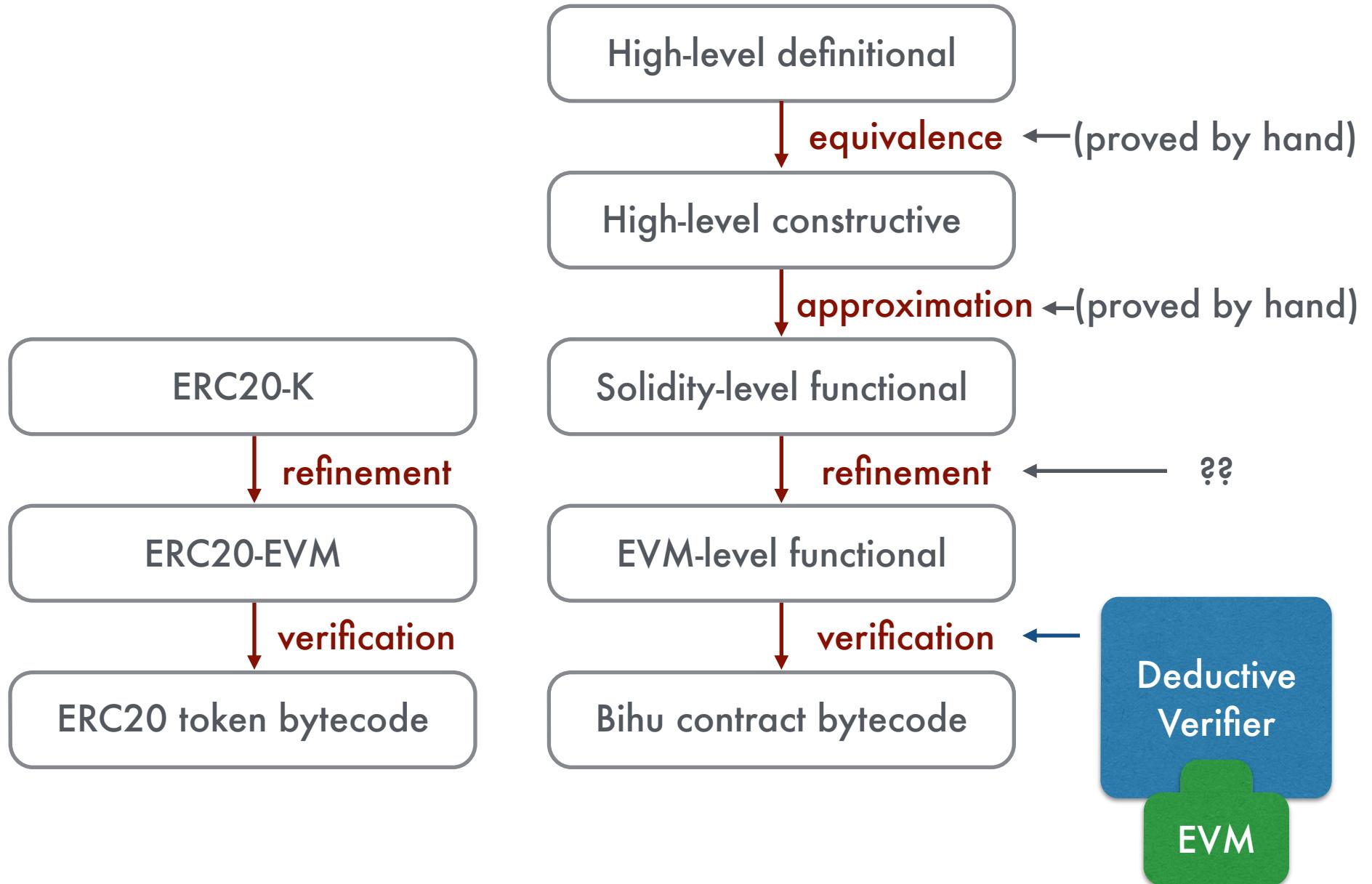
Verified smart contracts*

- 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



Developing new formal method

Heap manipulating programs

Smart contracts

Compiler verification

Specification refinement

Deductive Verifier

Program Equivalence Checker

[TR'17]

C

Java

JavaScript

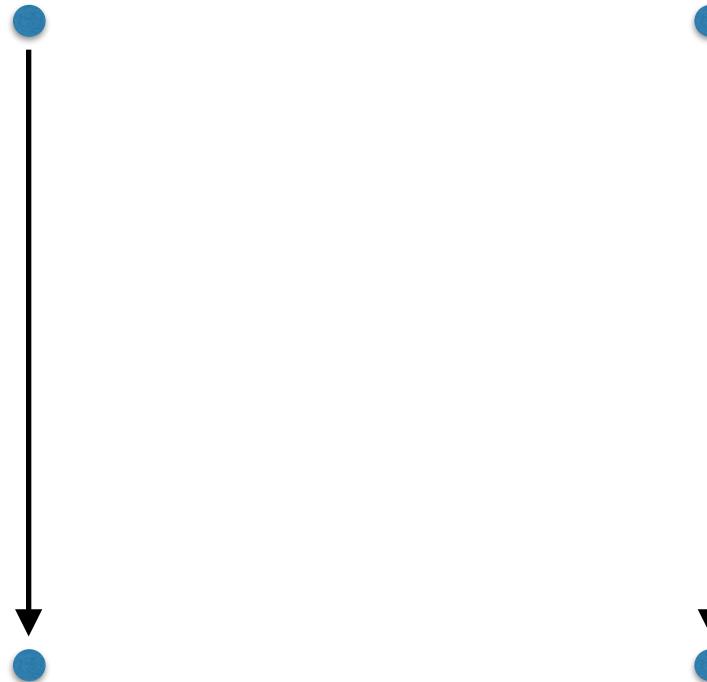
WASM

JAUS

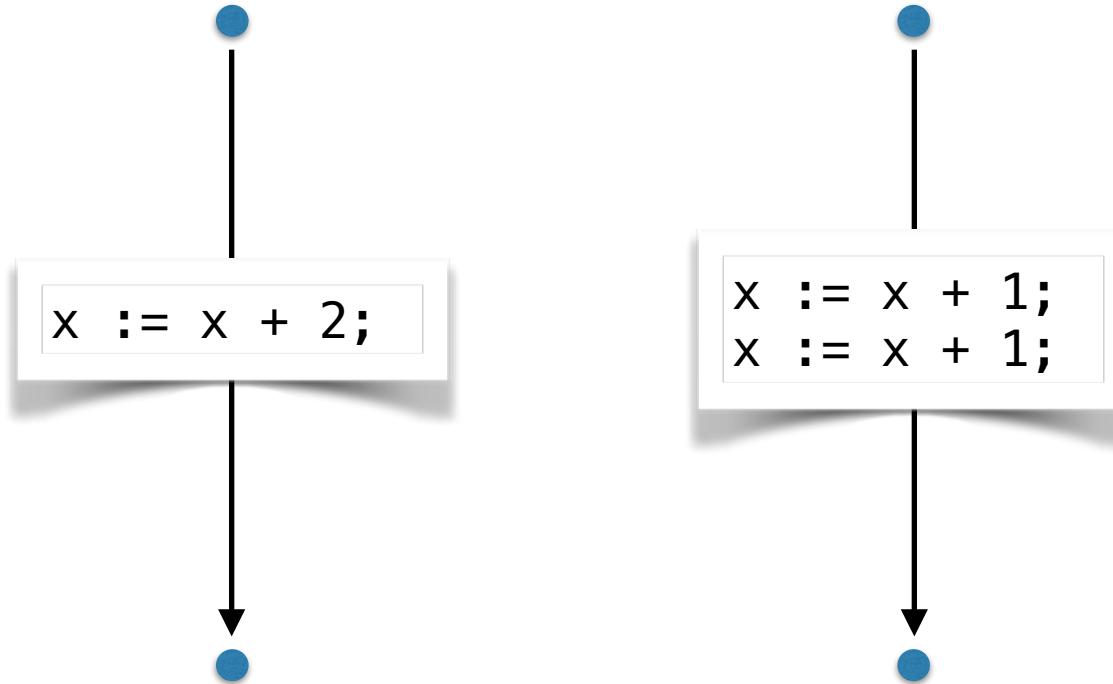
Vyper

EVM

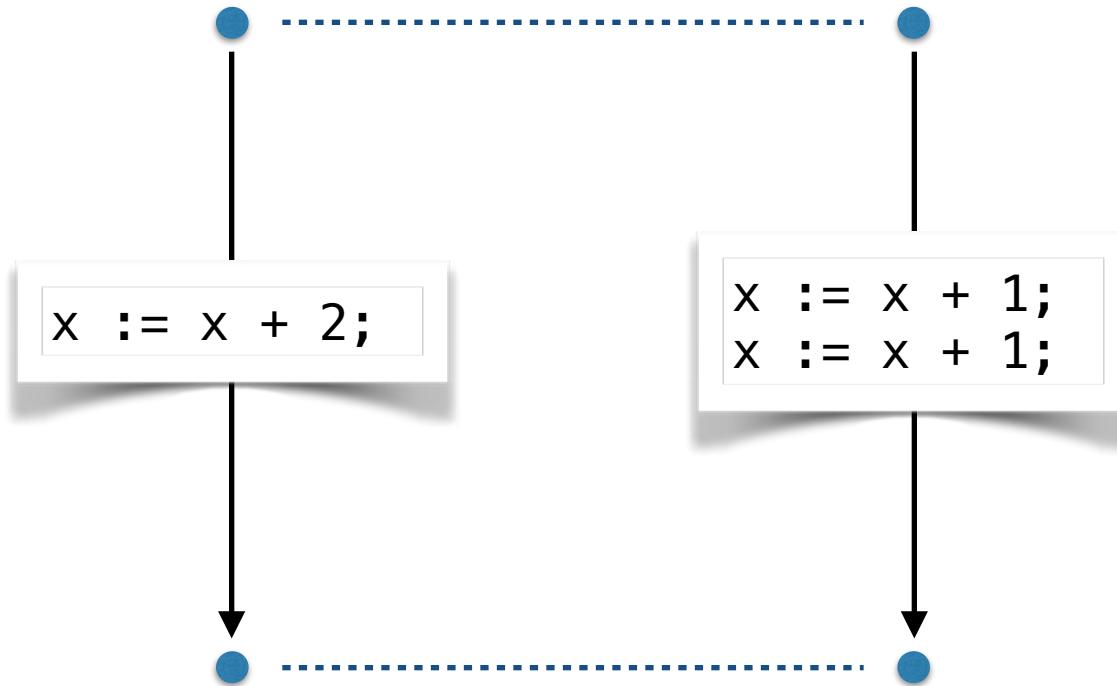
Deterministic, terminating programs



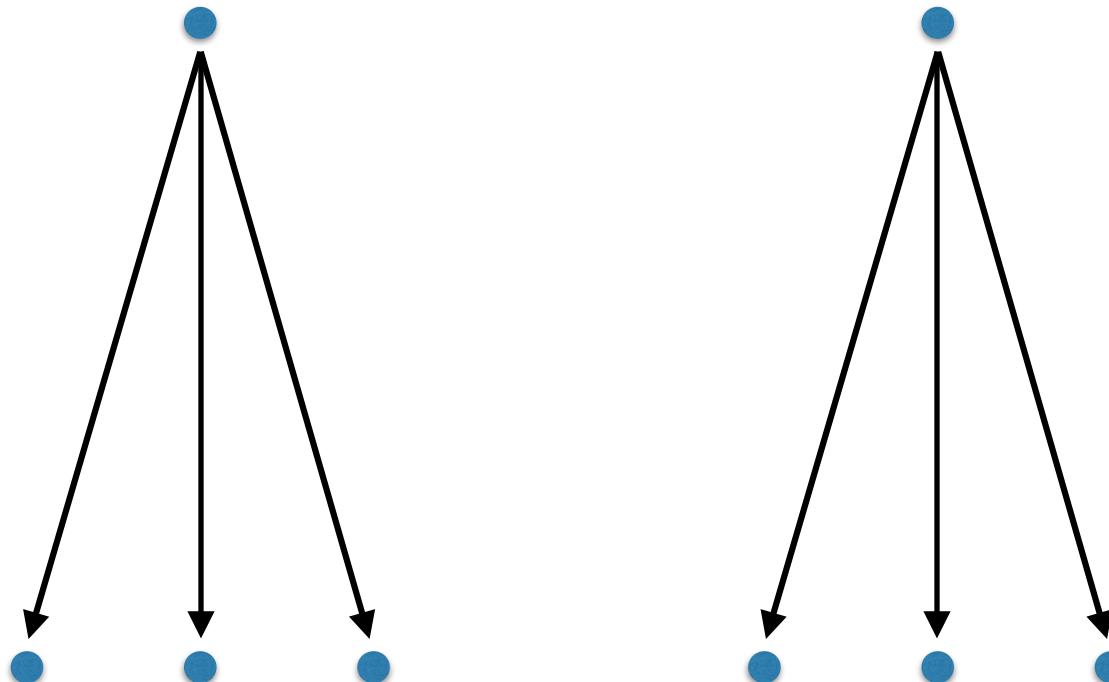
Deterministic, terminating programs



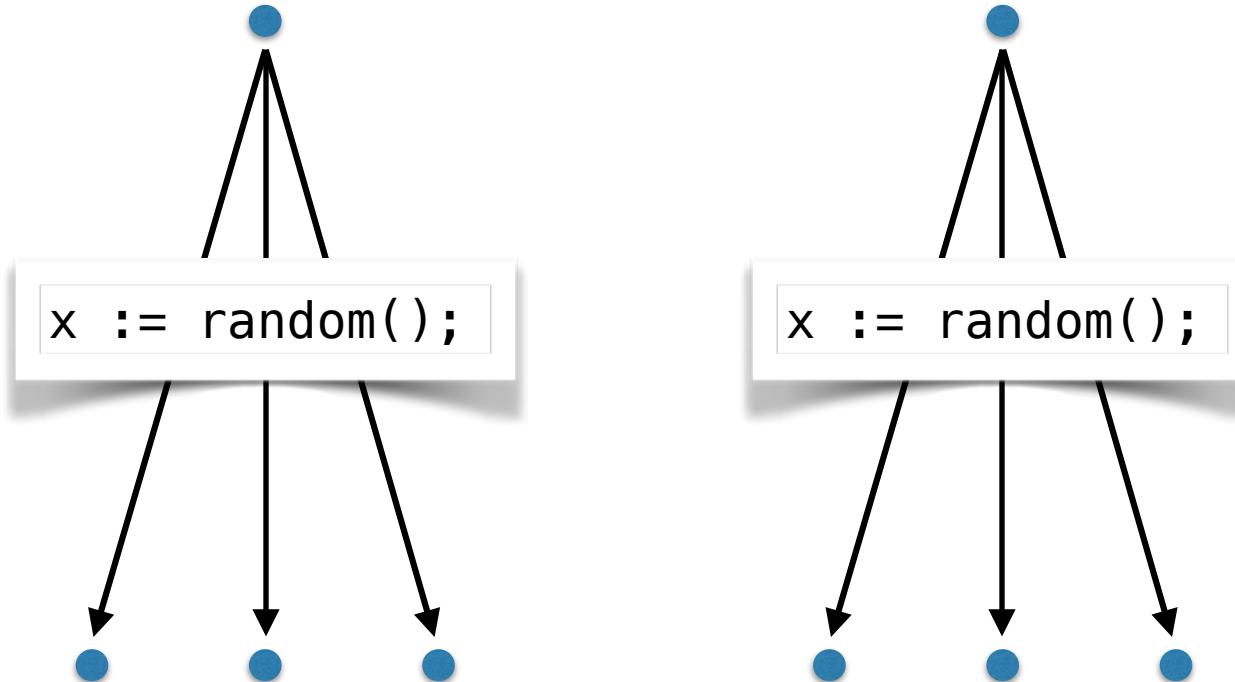
Deterministic, terminating programs



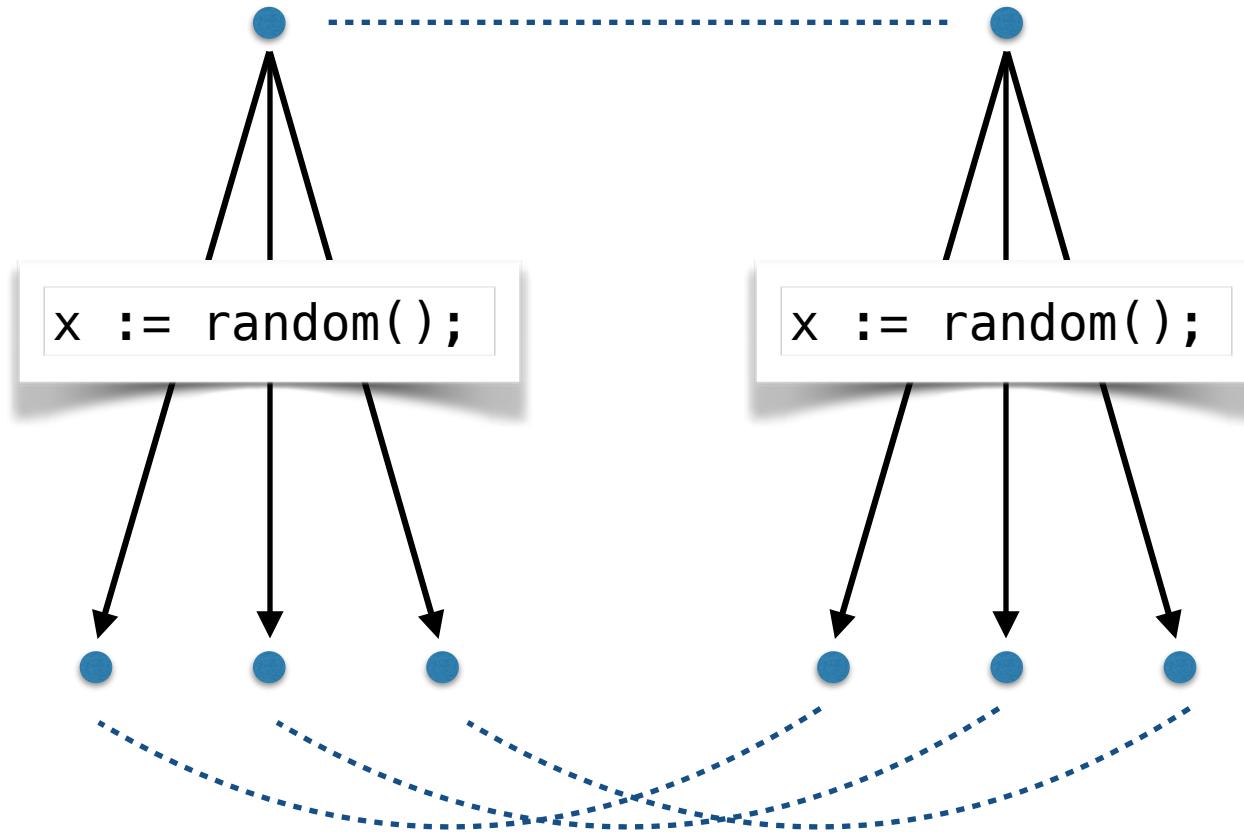
Nondeterministic programs



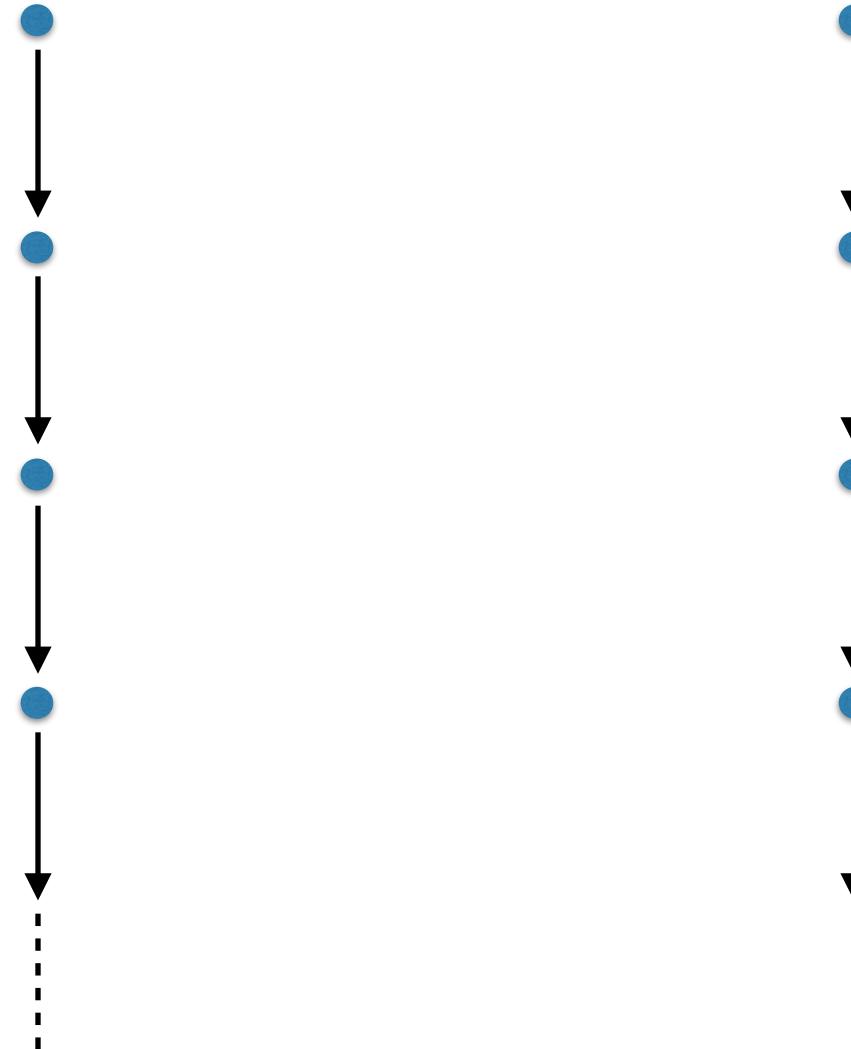
Nondeterministic programs



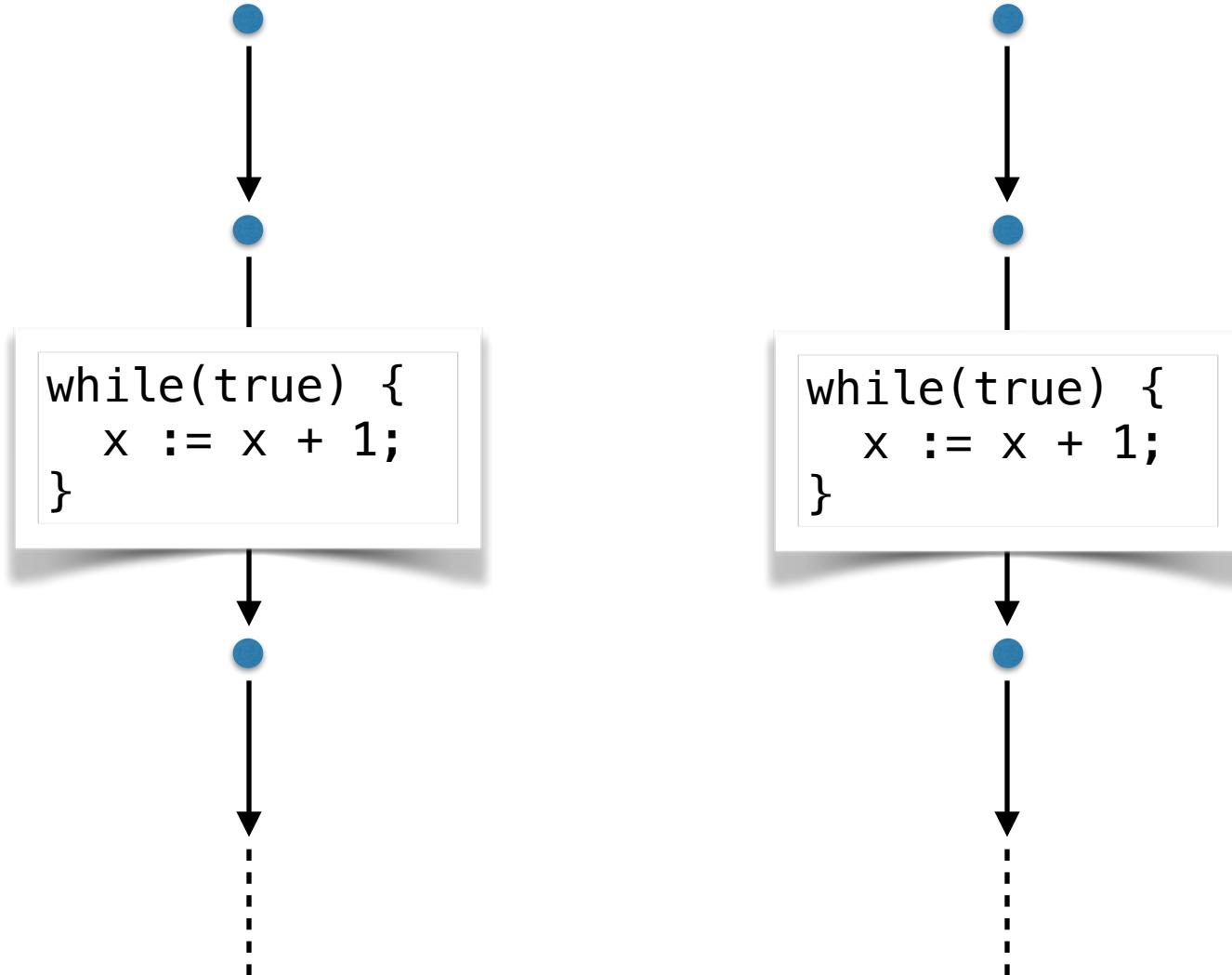
Nondeterministic programs



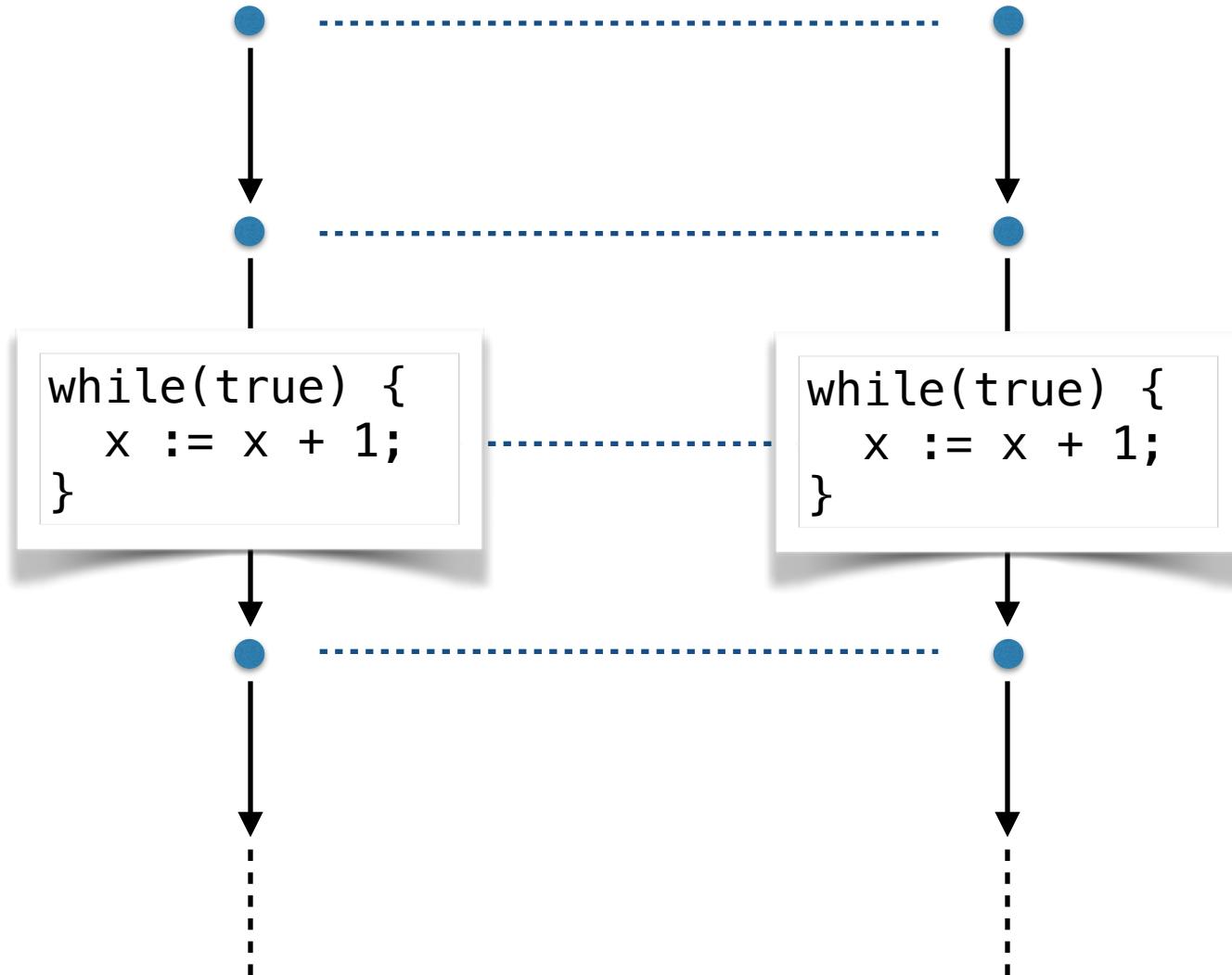
Non-terminating programs



Non-terminating programs



Non-terminating programs



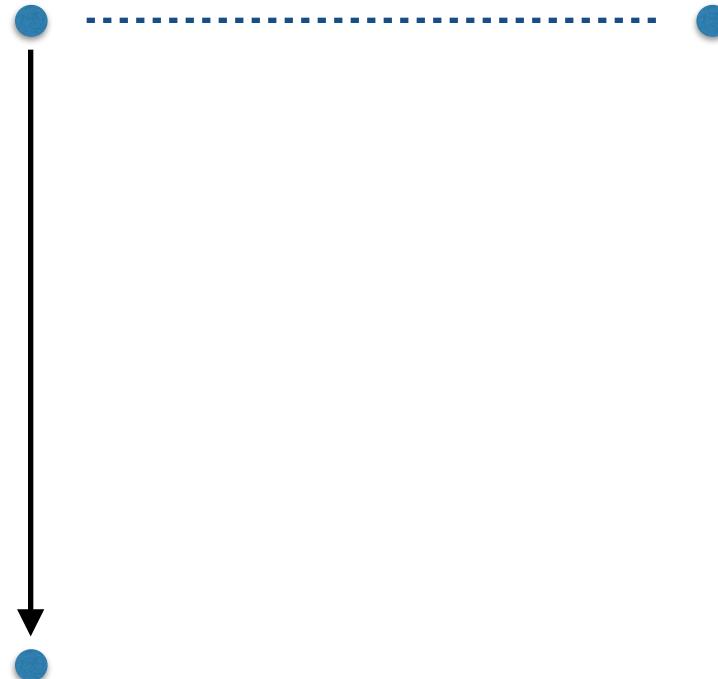
Bisimulation

for each pair of related states



Bisimulation

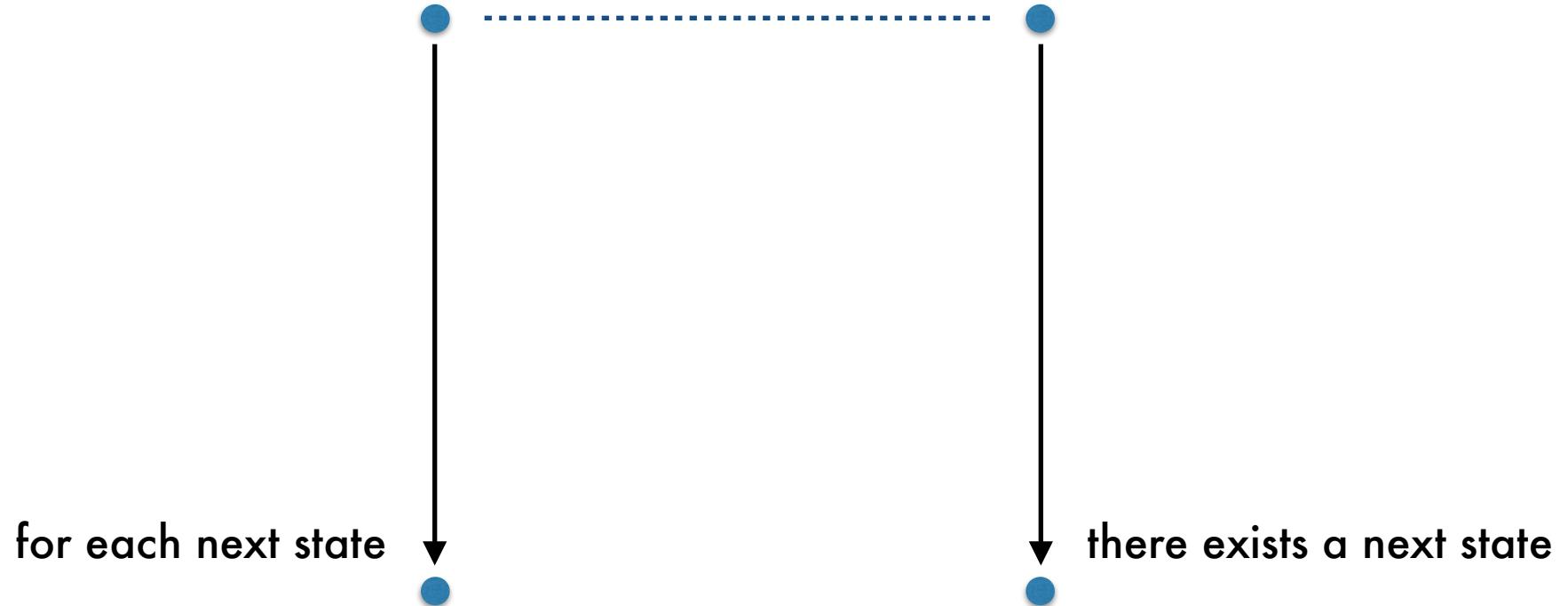
for each pair of related states



for each next state

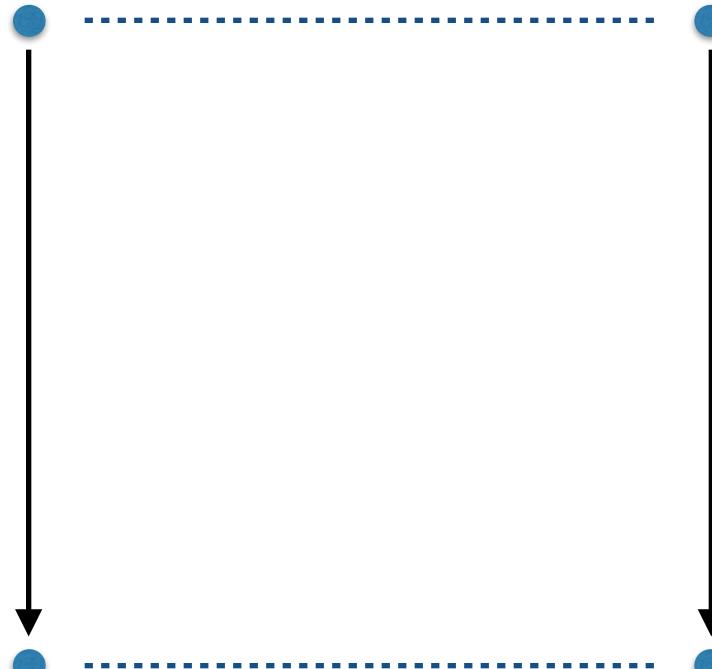
Bisimulation

for each pair of related states



Bisimulation

for each pair of related states



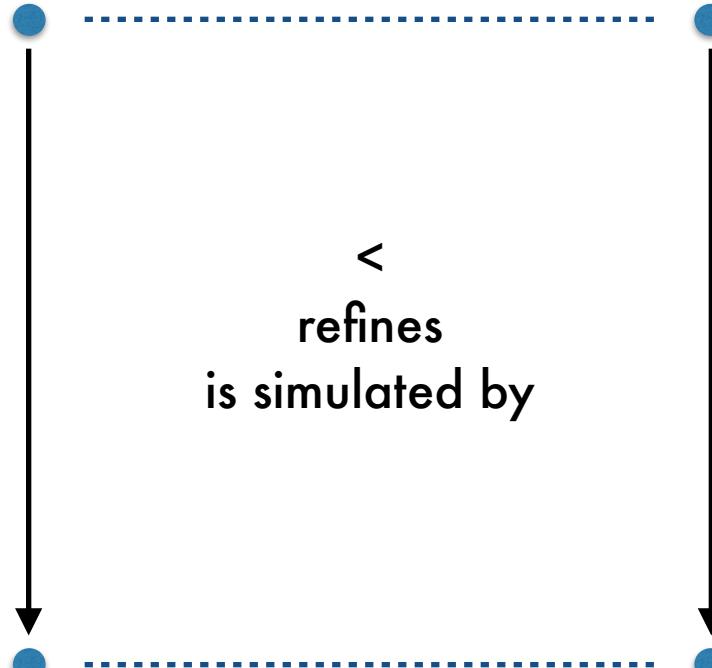
for each next state

there exists a next state

such that two states are related

Bisimulation

for each pair of related states

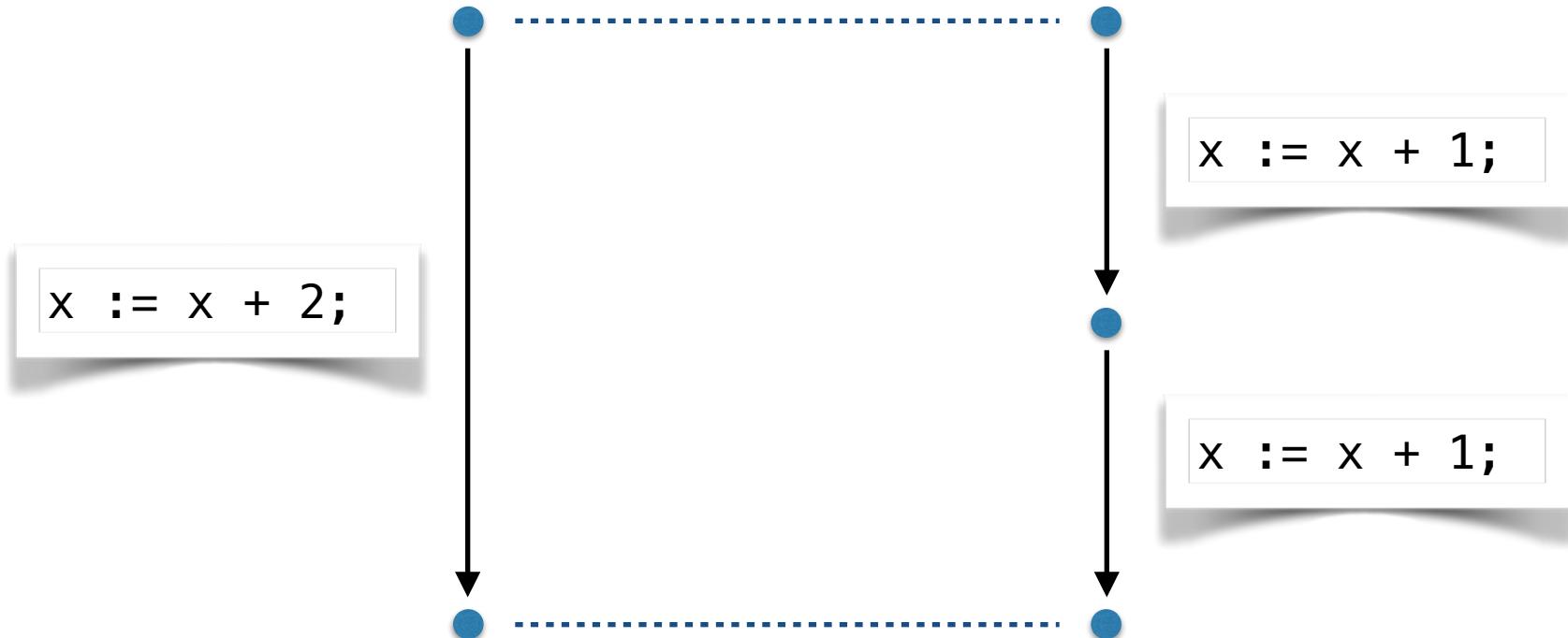


for each next state

such that two states are related

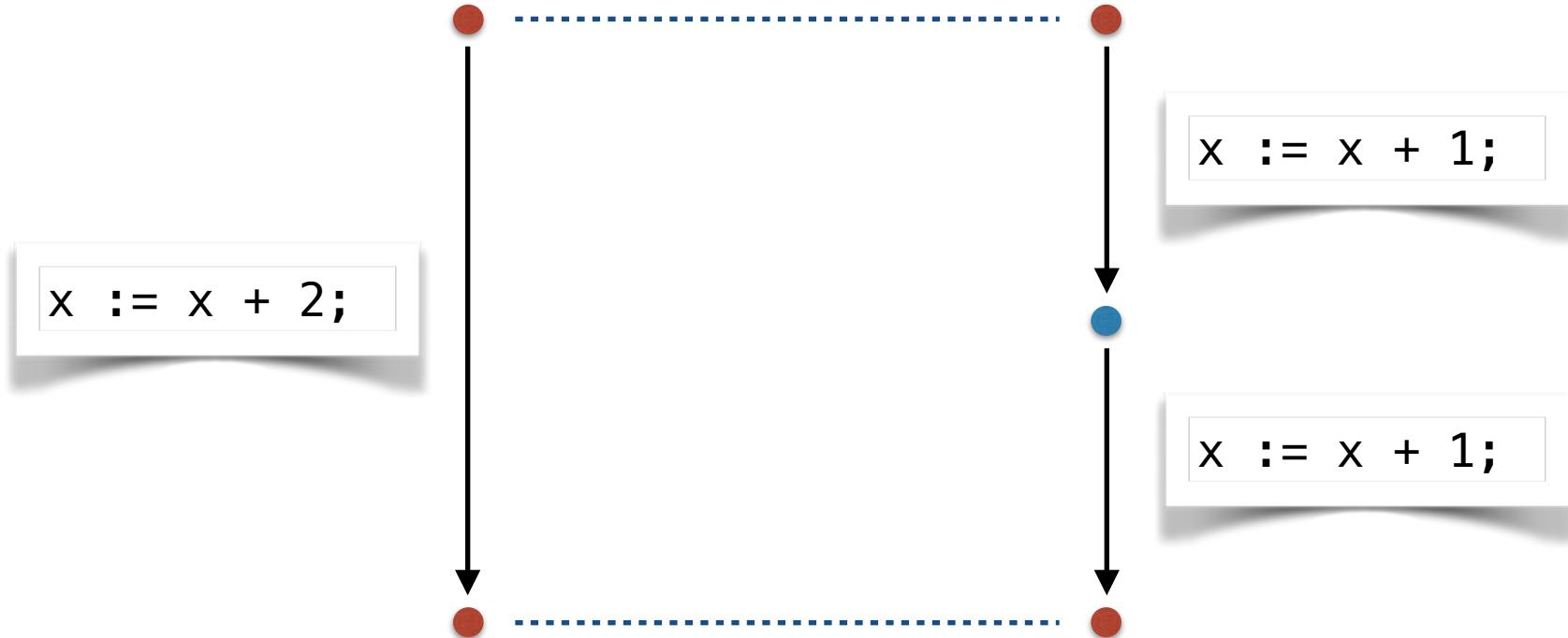
there exists a next state

Bisimulation?



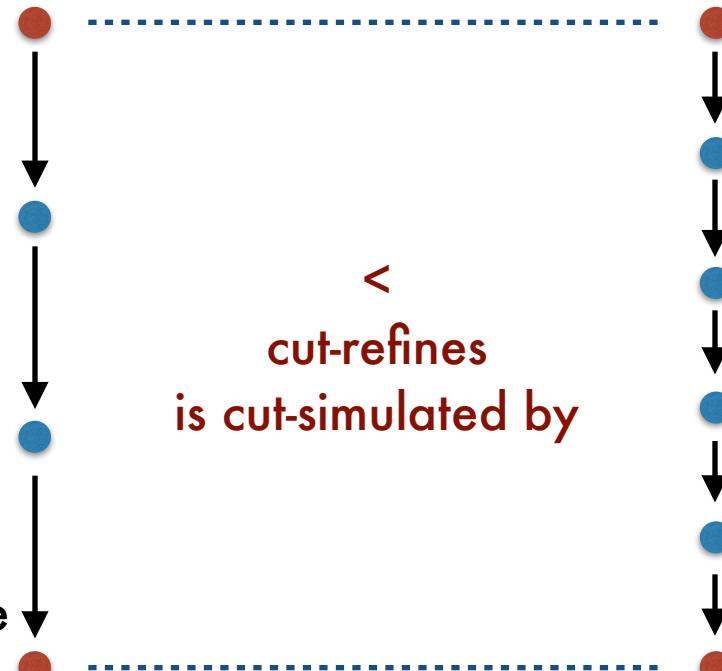
Cut-bisimulation

↑
considering only **relevant** states



Cut-bisimulation

for each pair of related **relevant** states

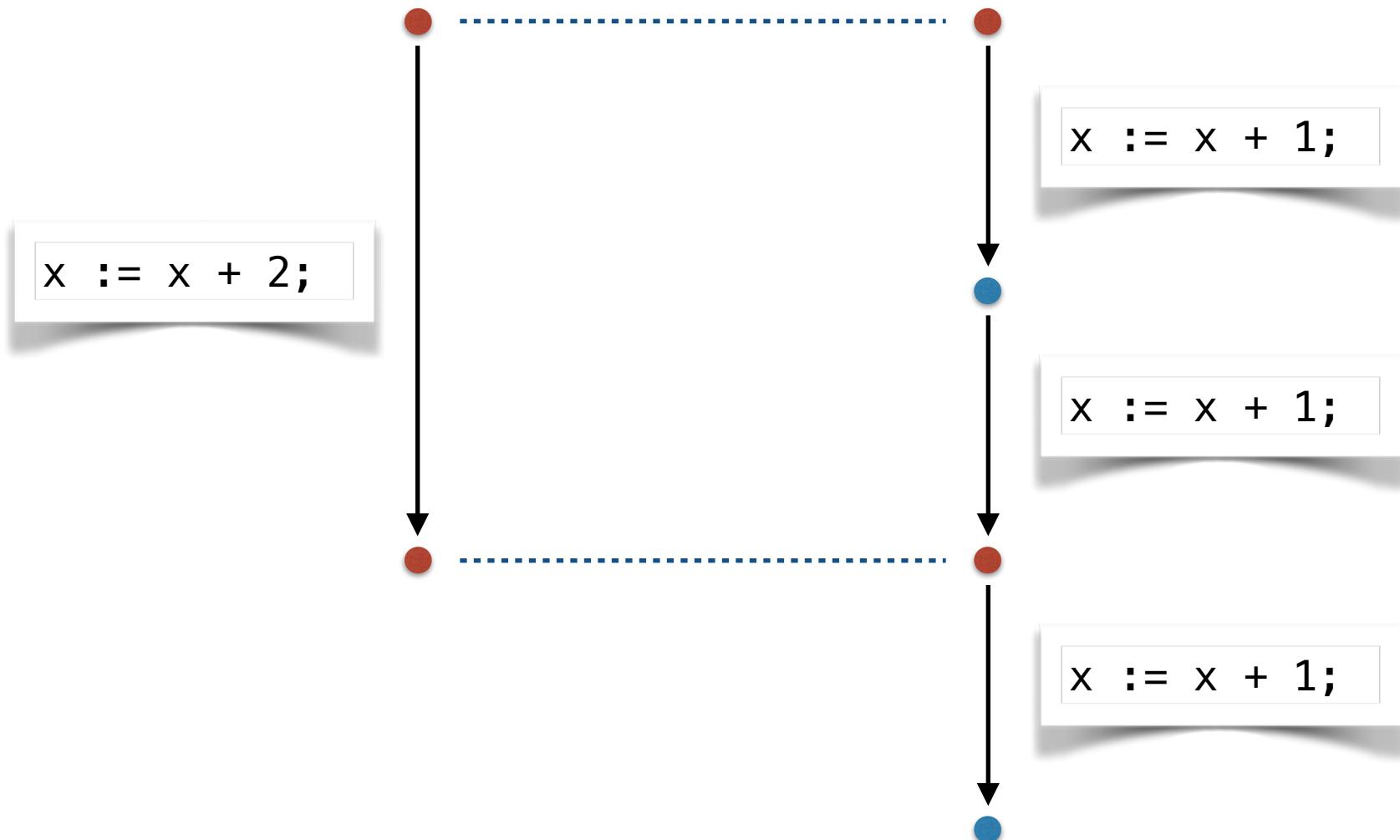


for each next **relevant** state

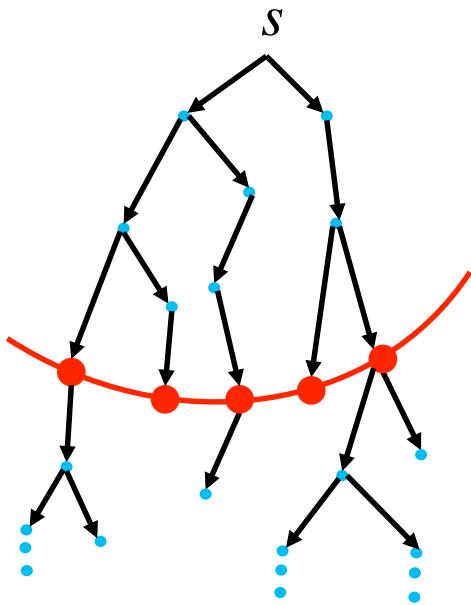
such that two **relevant** states are related

there exists a next **relevant** state

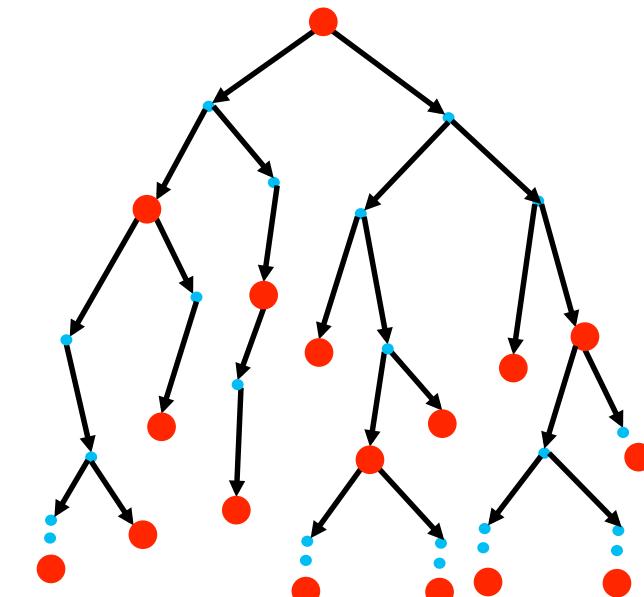
Well-formedness of relevant states



Cut



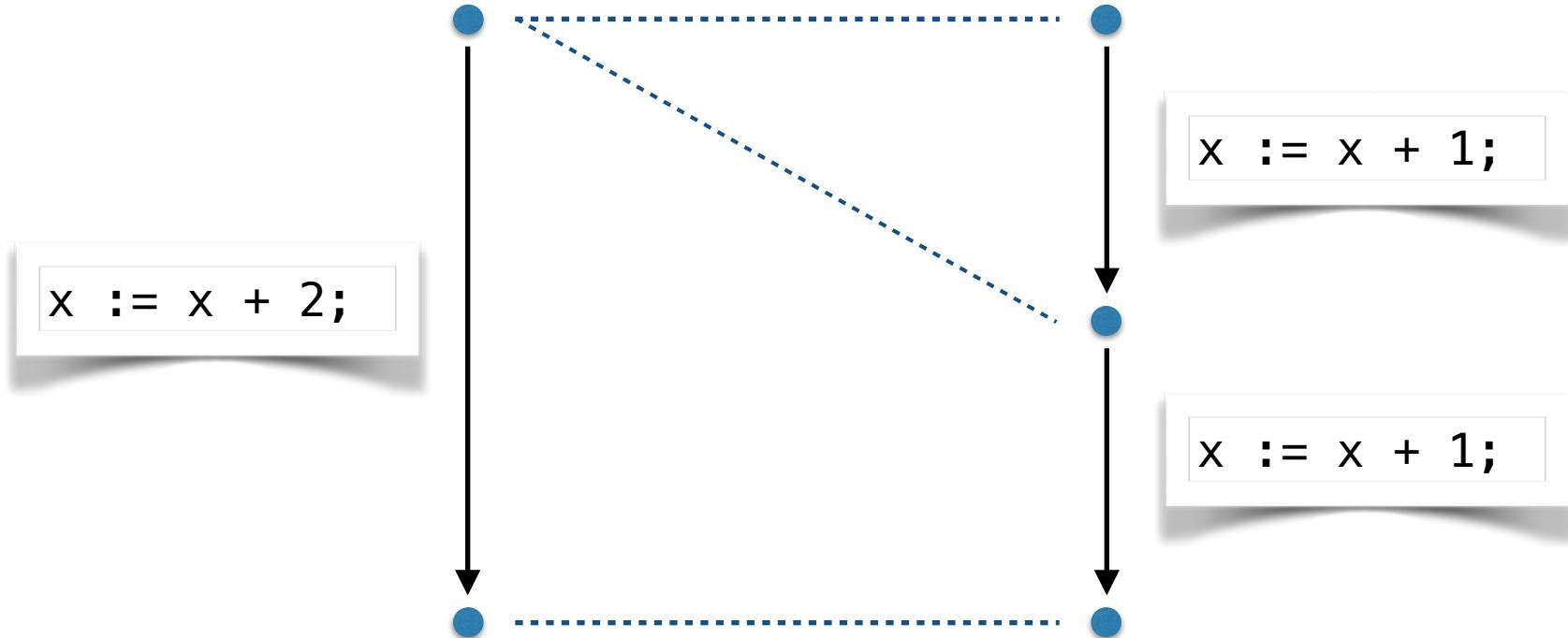
Cut for s



Cut for all

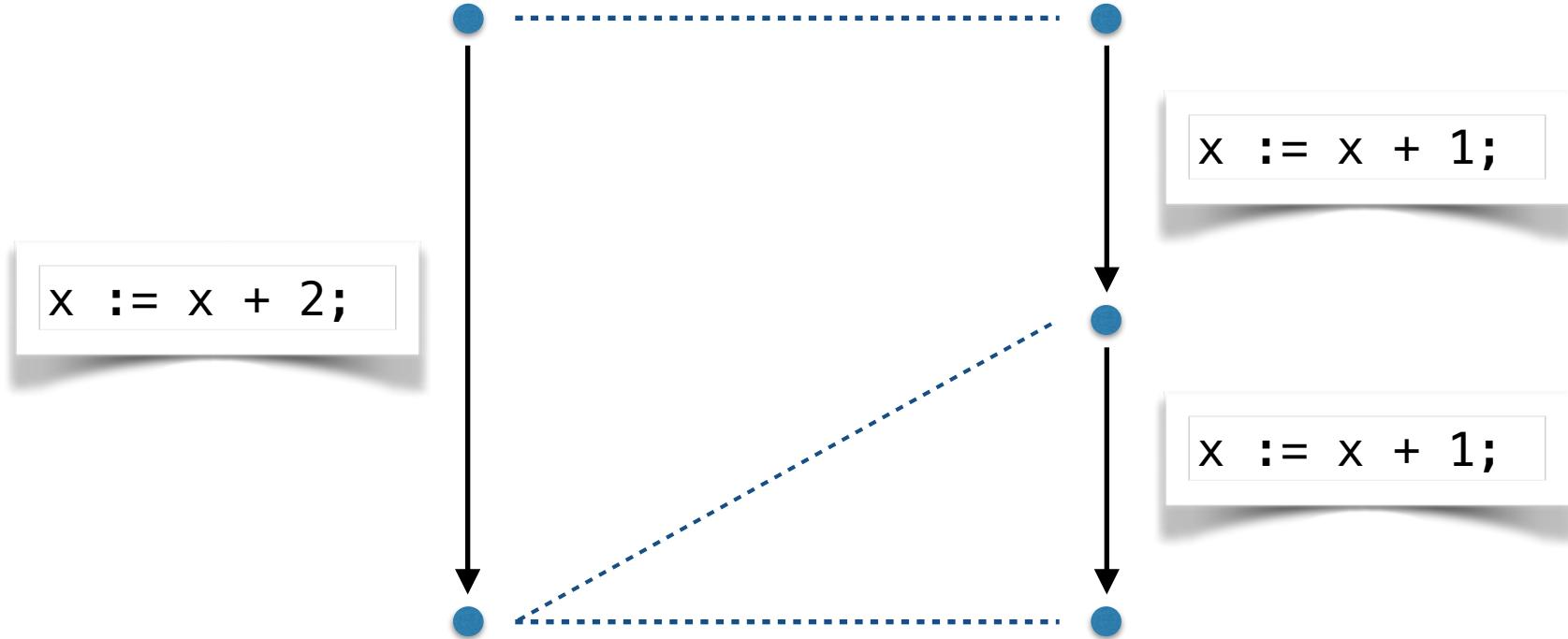
Stuttering bisimulation [BCG'88, dNV'90]

two possible stuttering bisimulations: 1 of 2

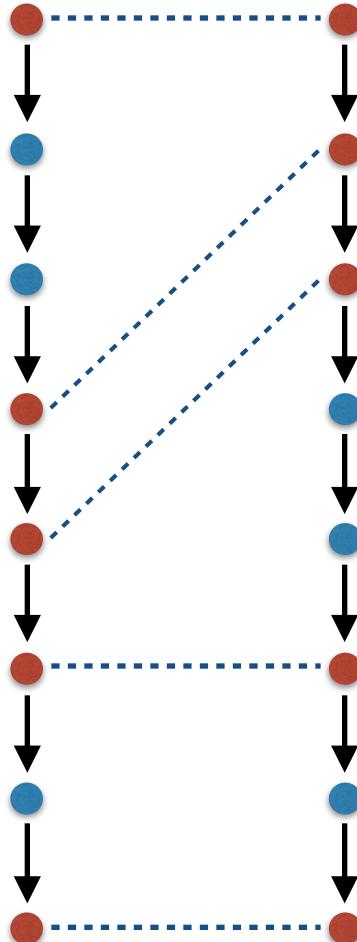


Stuttering bisimulation [BCG'88, dNV'90]

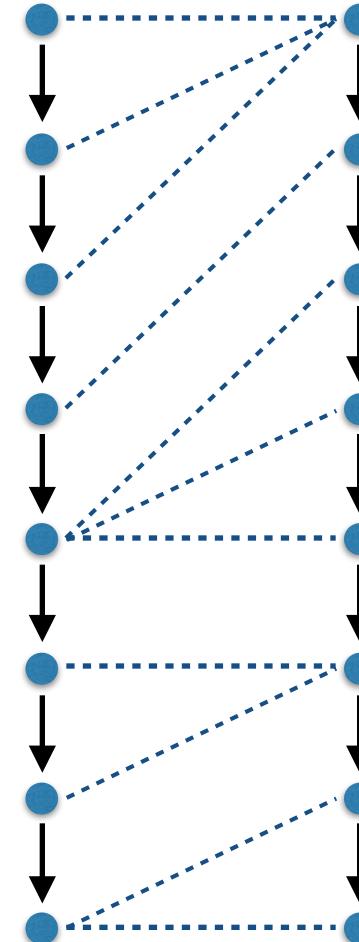
two possible stuttering bisimulations: 2 of 2



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]

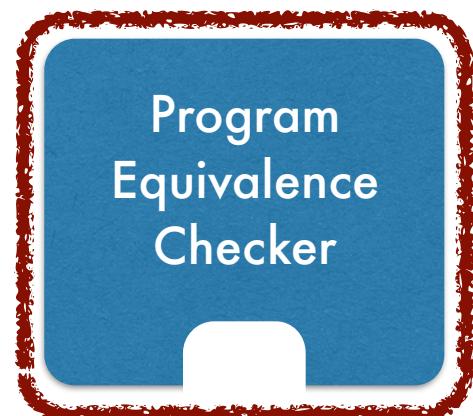
- Validate each instance of compilation
- Verify equivalence of input/output programs
 - KEQ can be used for checking equivalence
- Practical advantages for compiler verification

Current results

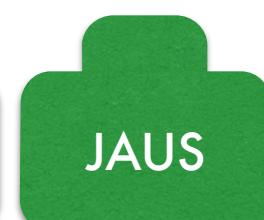
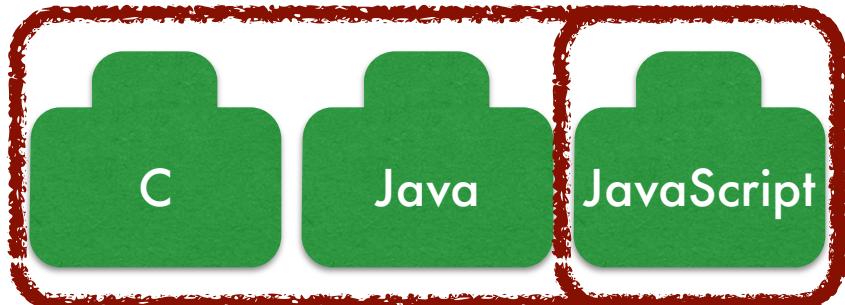
[TR'17] [TR'18]



[OOPSLA'16]



[TR'17]

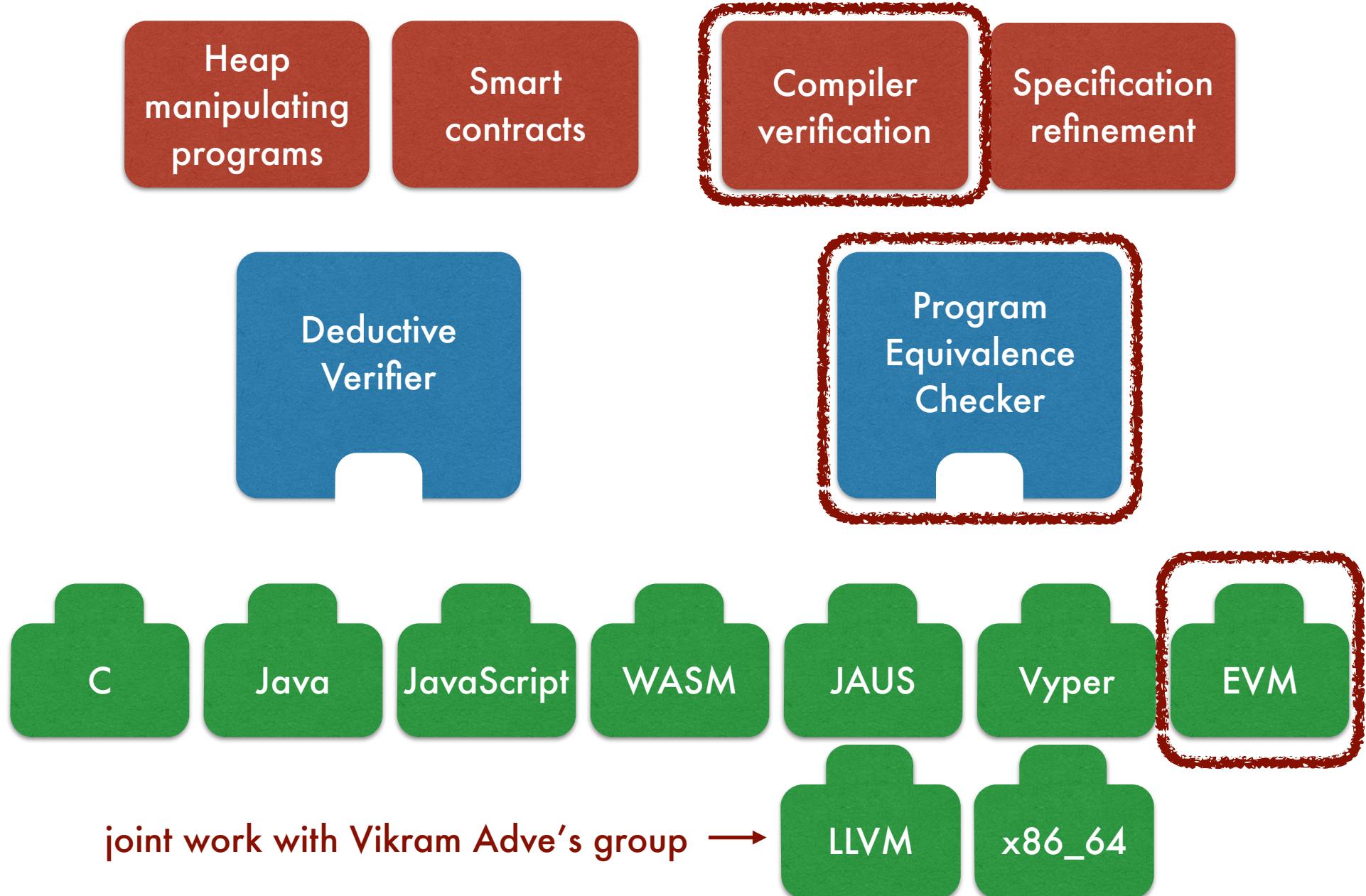


[PLDI'15]

[CSF'18]

Proposed Work

Proposal: translation validation for compiler



Translation validation for compiler

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

Proposal: verifying spec refinement

Heap manipulating programs

Smart contracts

Compiler verification

Specification refinement

Deductive Verifier

Program Equivalence Checker

C

Java

JavaScript

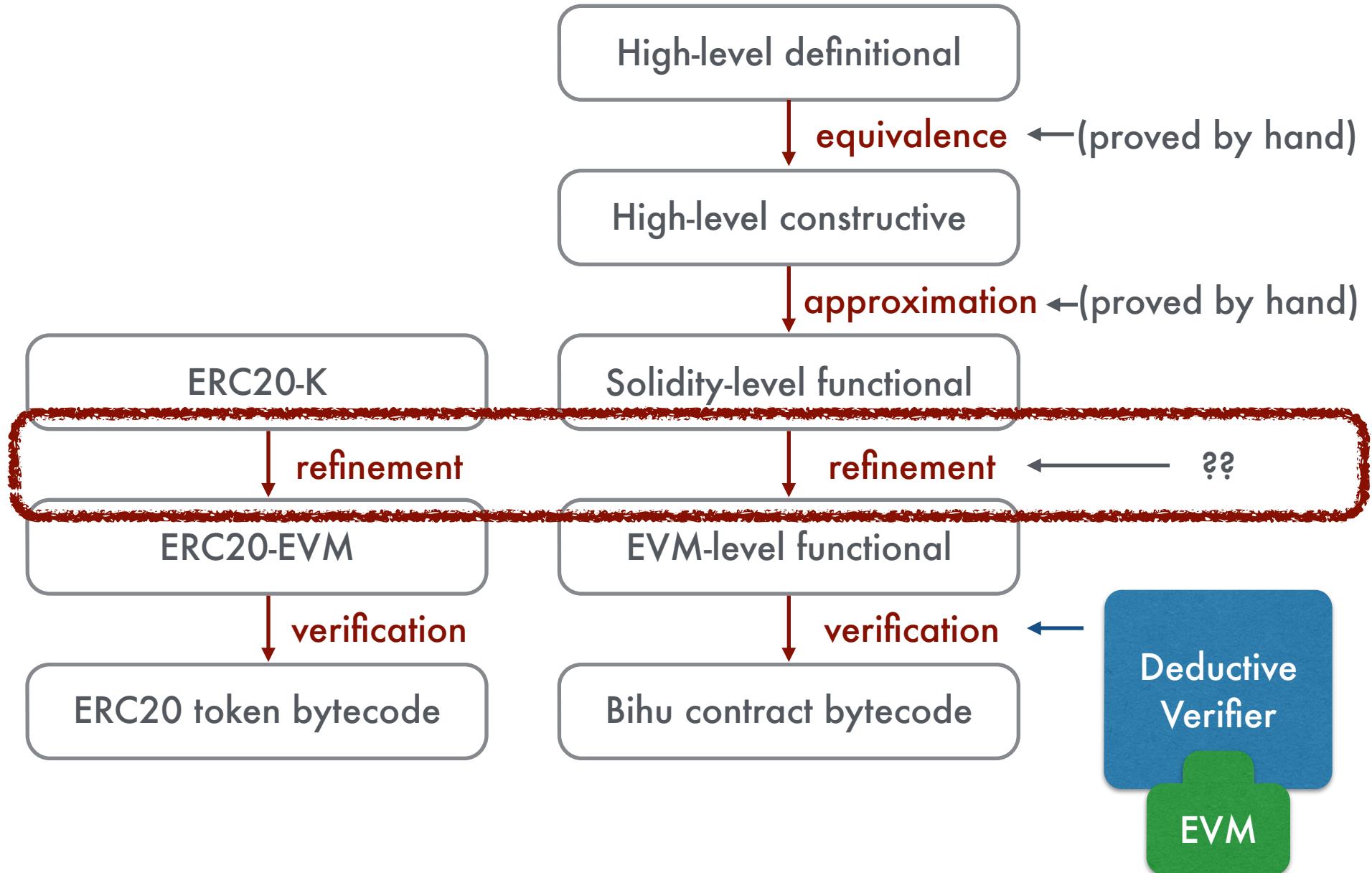
WASM

JAUS

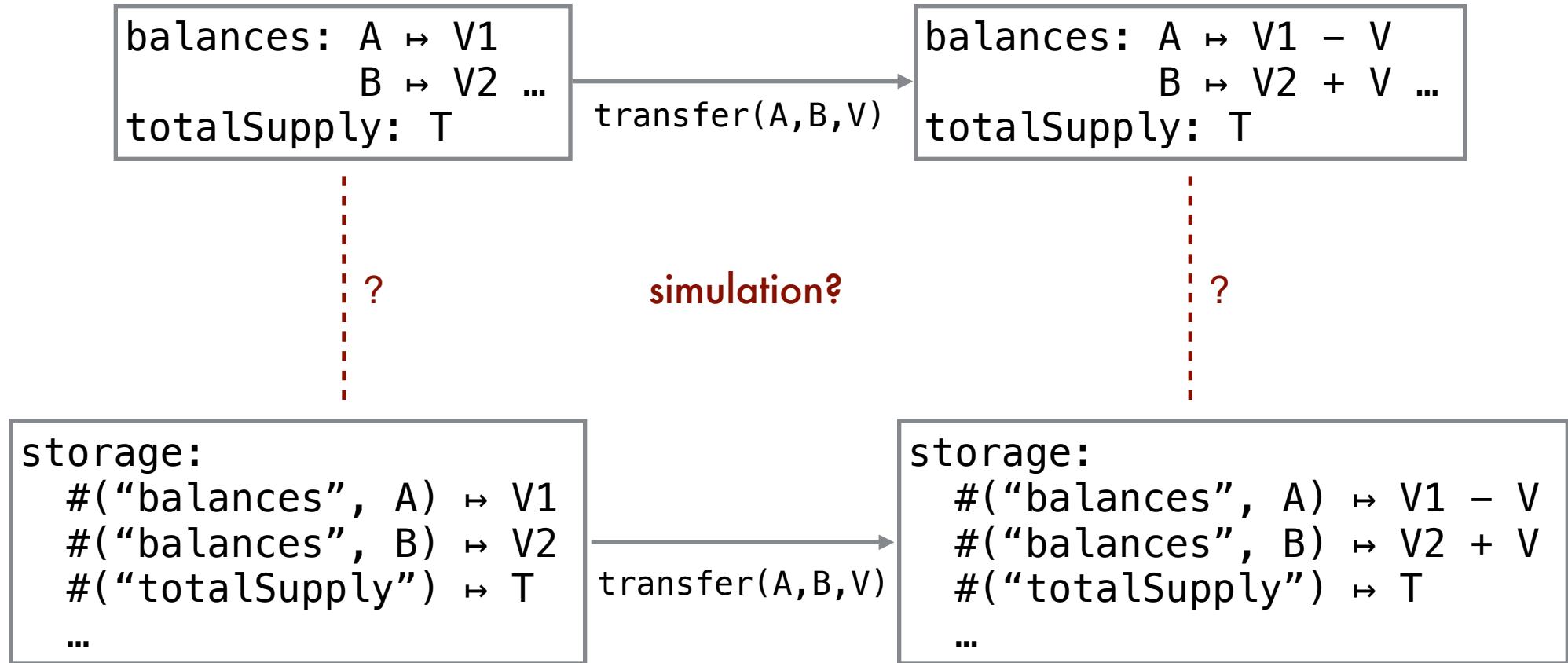
Vyper

EVM

Verifying specification refinement

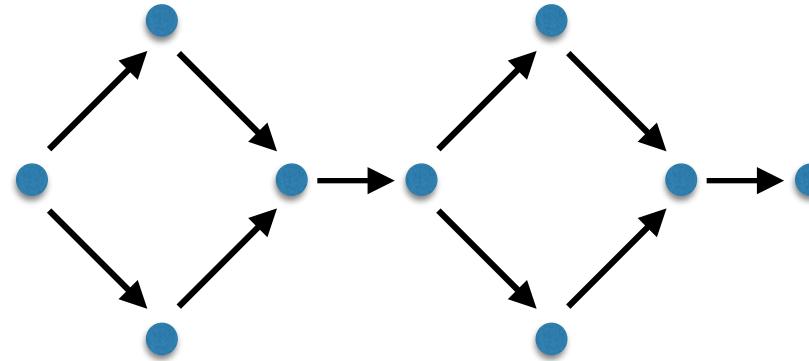


Soundness of specification refinement



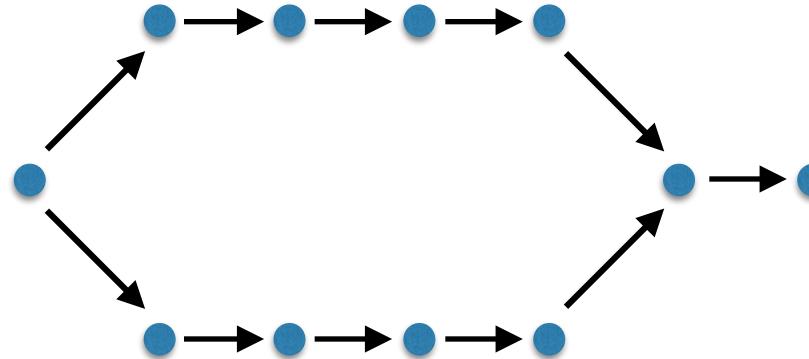
e.g., gas consumption is needed for the high-level spec?

Soundness of specification refinement

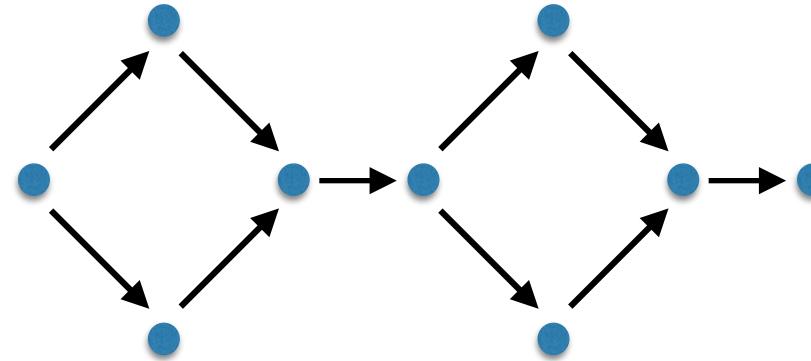


v

simulates

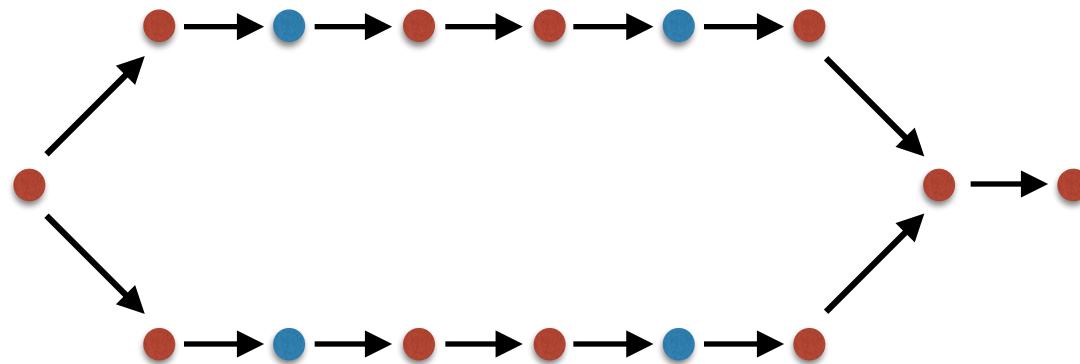


Soundness of specification refinement

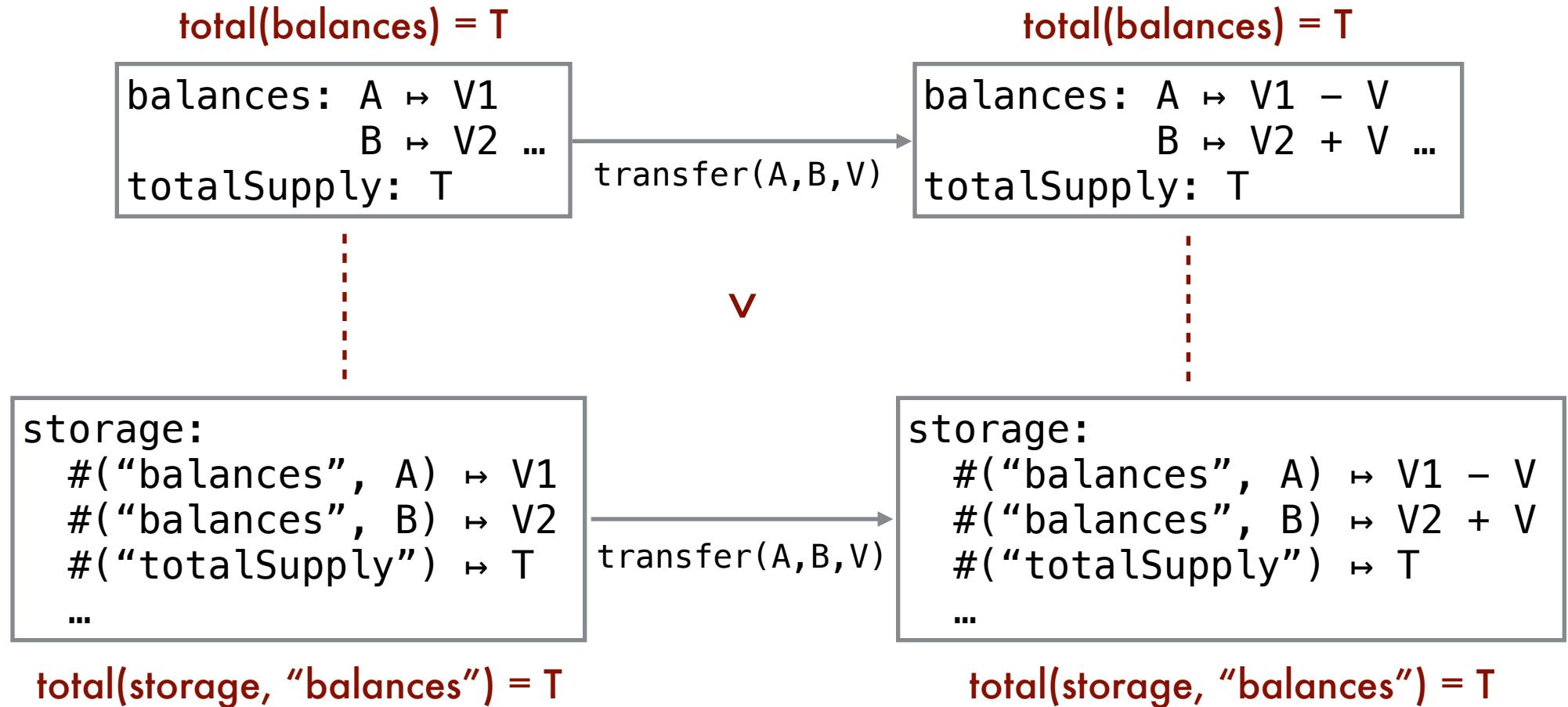


∨

cut-simulates



Proposal: property preservation



How to preserve inductive properties?

Property preservation

$P : D \rightarrow \text{bool}$

$f : S \rightarrow D$

$f' : S' \rightarrow D$

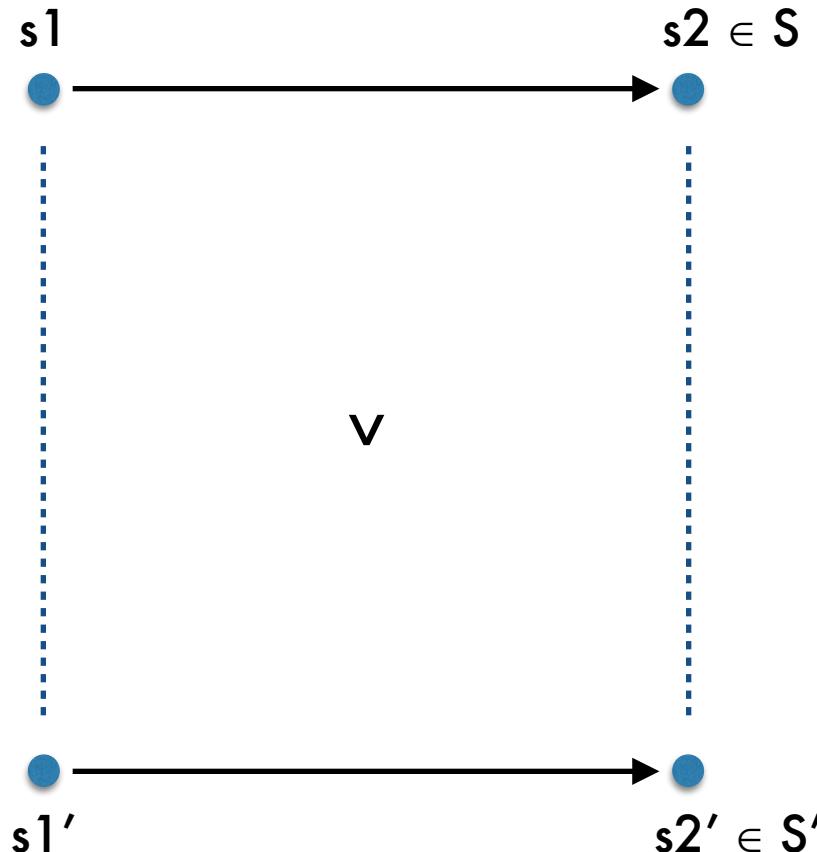
$P \circ f(s) = P(f(s))$

$P \circ f'(s') = P(f'(s'))$

$P \circ f$ is inductive

$f(s) = f'(s') \text{ if } s > s'$

$P \circ f'$ is inductive



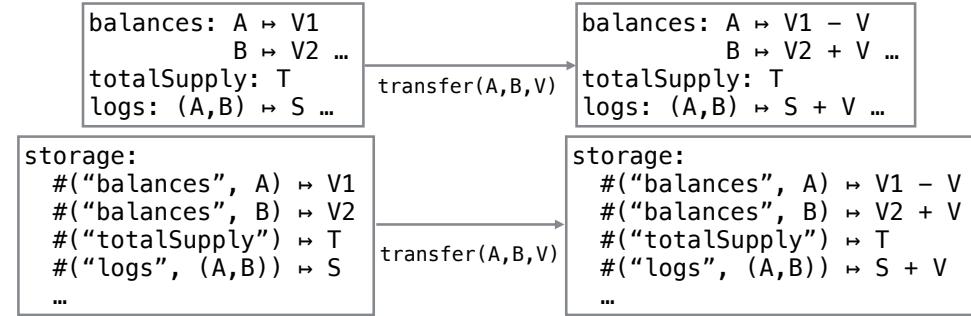
Property preservation

$$P_{\text{total}} \stackrel{\text{def}}{=} \text{total}(\text{balances}) = \text{totalSupply}$$

$$P_{\text{total}'} \stackrel{\text{def}}{=} \text{total}'(\text{storage}) = \text{totalSupply}$$

$$\text{total}(B) = \sum \{v \mid k \mapsto v \in B\}$$

$$\text{total}'(B') = \sum \{v' \mid k' \mapsto v' \in B' \wedge \exists k. \text{hash}(k) = k'\}$$



$$\text{total}(\text{balances}) = \text{total}'(\text{storage})$$

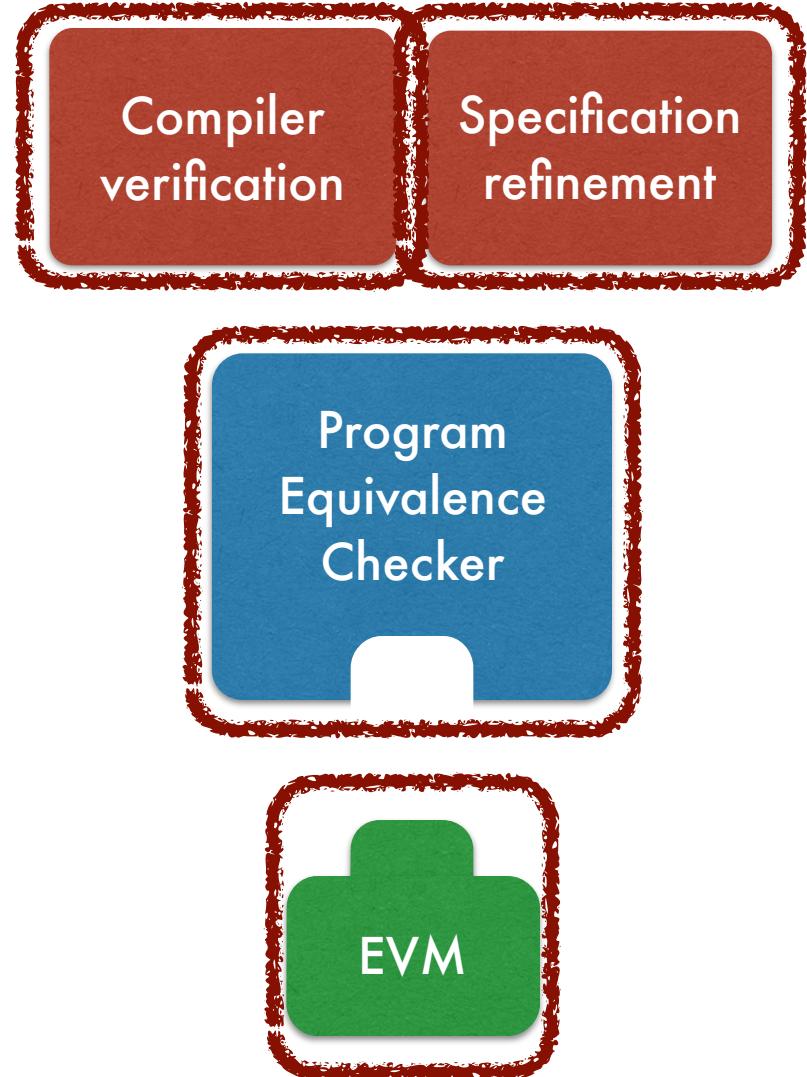
$$\text{balances} > \text{storage} \iff$$

$$\forall k \in \text{dom}(\text{balances}). \text{balances}(k) = \text{storage}(\text{hash}(k))$$

$$\wedge \text{dom}(\text{balances}) = \{k' \mid k' \in \text{dom}(\text{storage}) \wedge \exists k. \text{hash}(k) = k'\}$$

Proposed work

- Translation validation for compiler
- Verifying specification refinement
- Property preservation of cut-simulation



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