Language-Parametric Formal Methods in the Field

Daejun Park

May 23, 2018 @ Fasoo

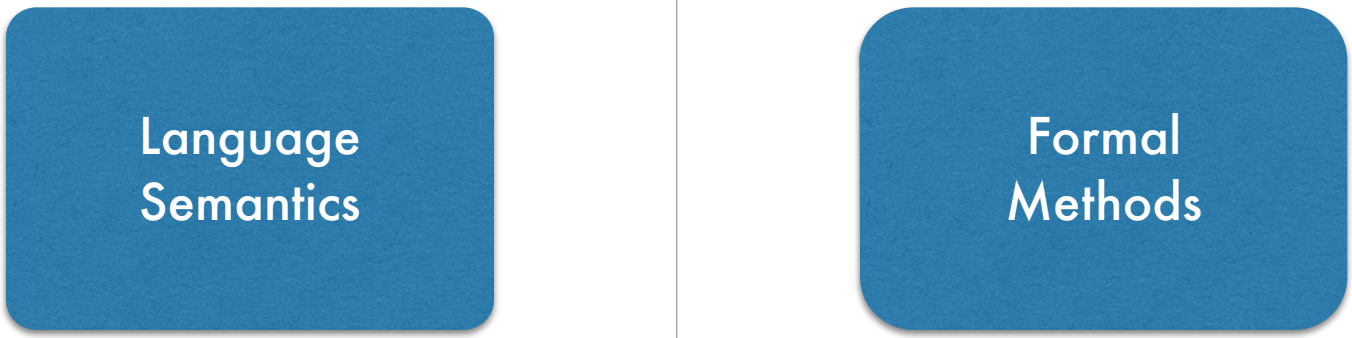
Reusability problem of formal tools

- 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, ...)
- ...

- Deductive verification
- Model checking
- Program equivalence checking
- ...

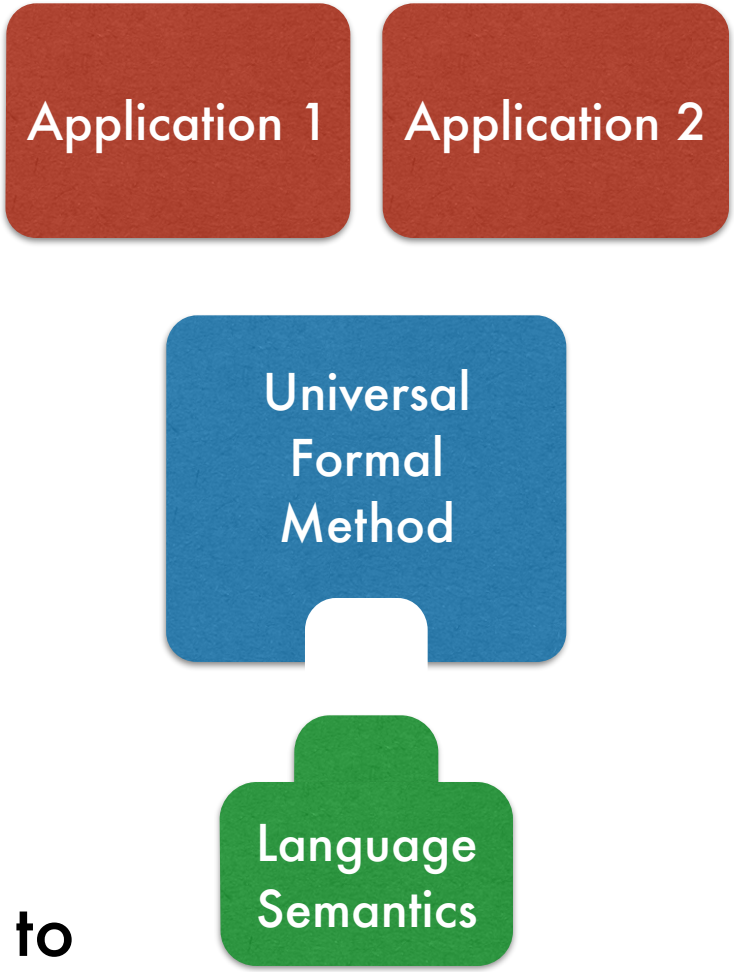
Defined/implemented once, and reused for all others

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, ...

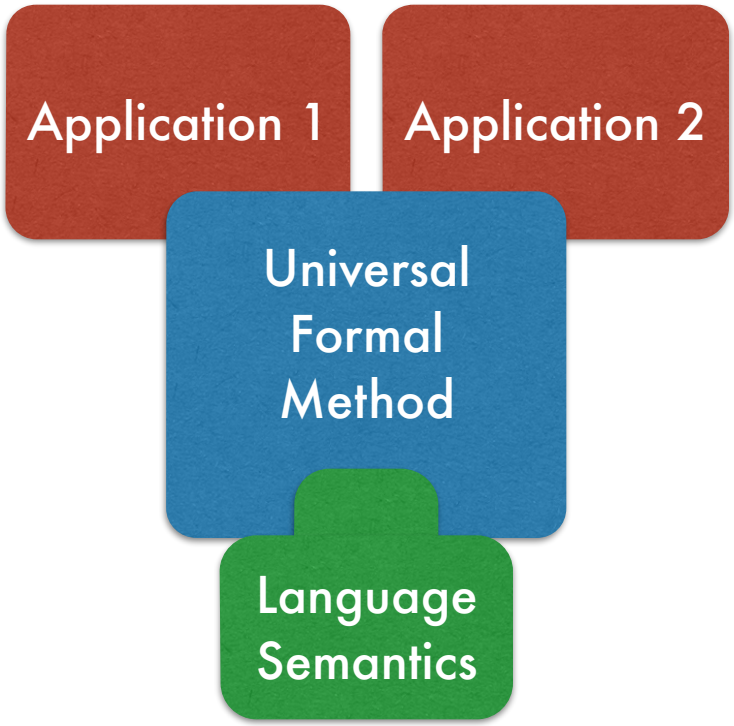
Overview

- 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

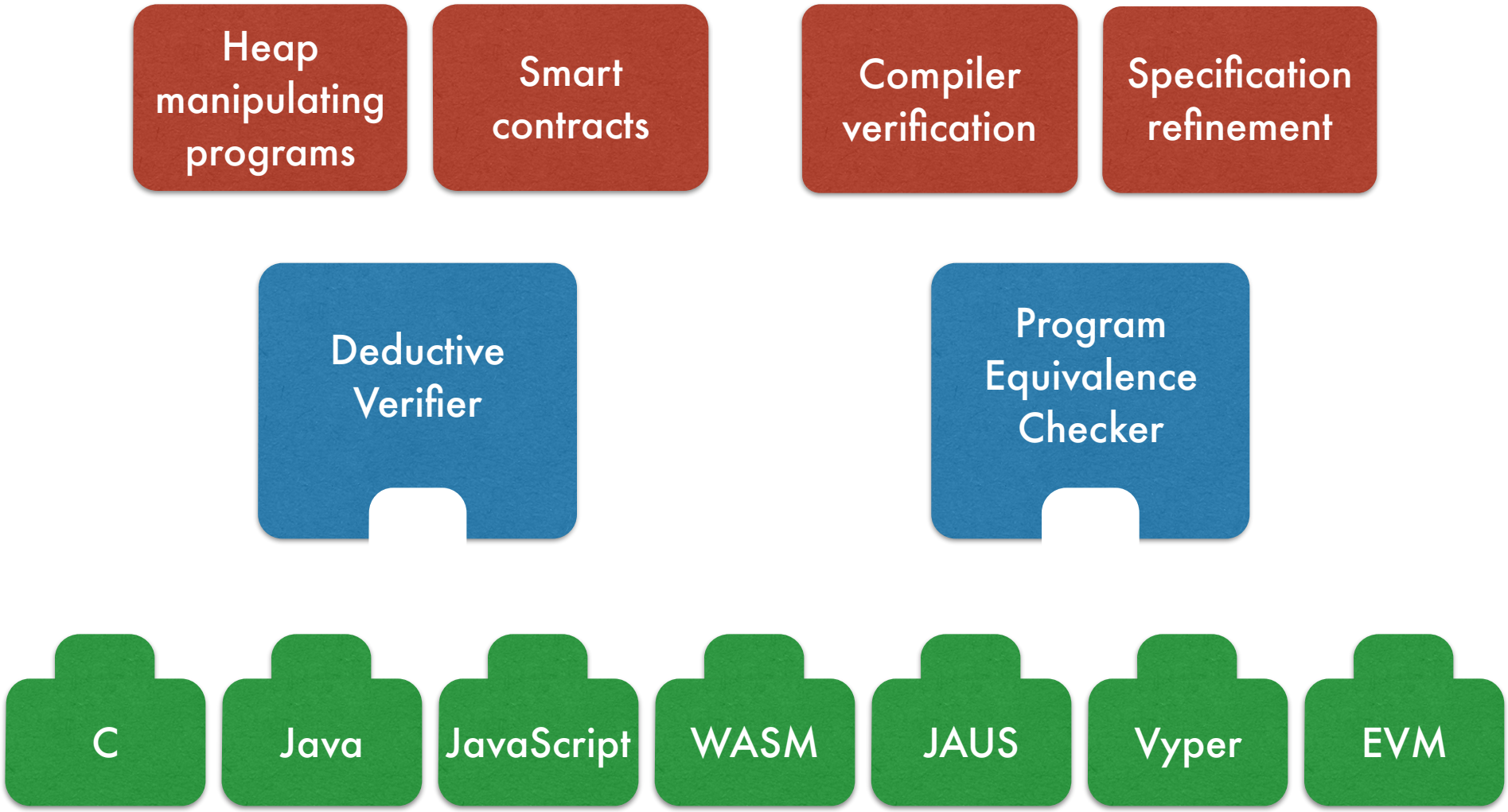


Overview

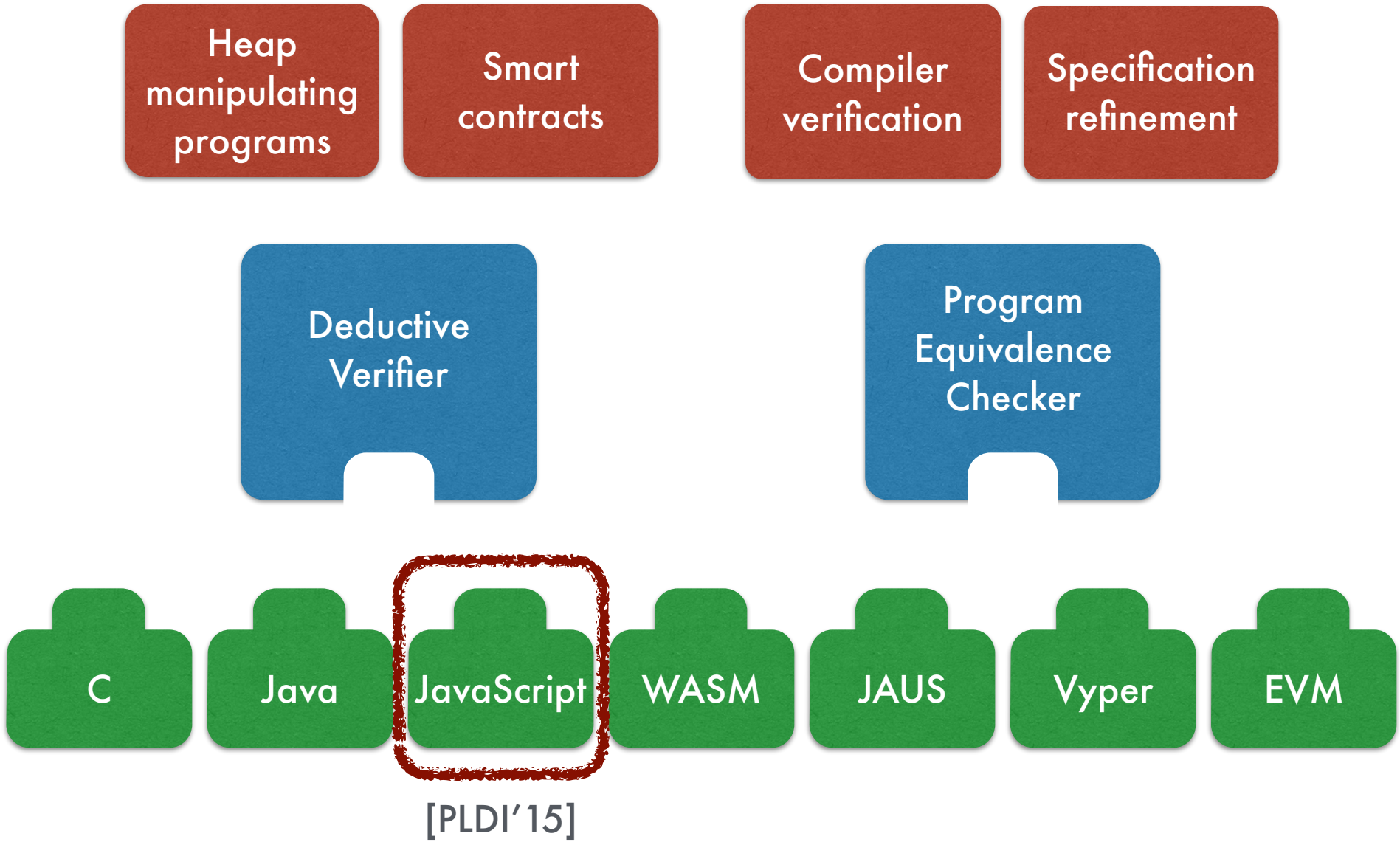
- Specifying real-world language semantics and measuring the specification effort
- Developing language-parametric formal methods
- Instantiating them by plugging-in various language semantics
- Applying the derived formal tools to real-world applications, demonstrating their practical feasibility



Overview

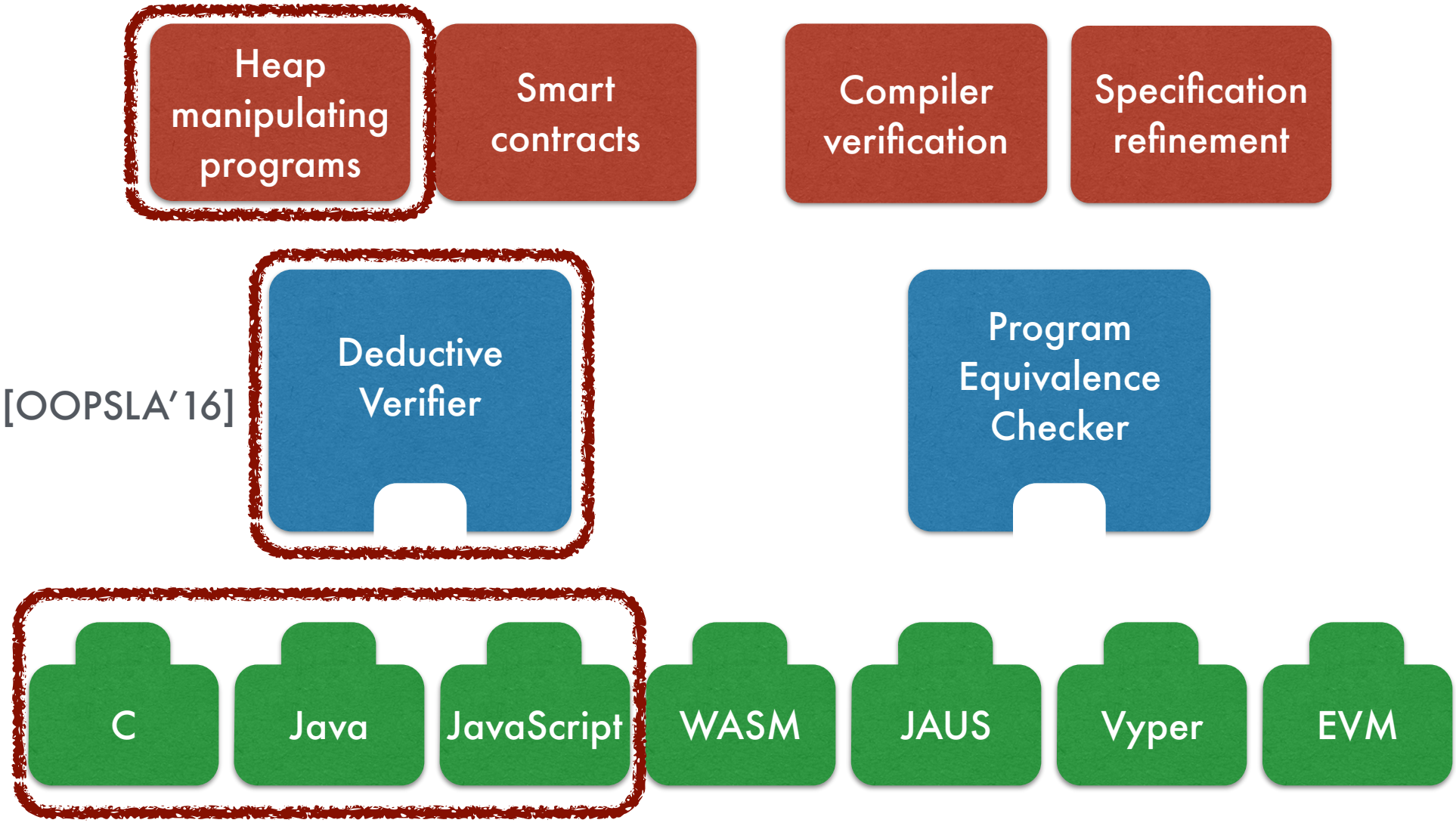

Heap manipulating programs
Smart contracts
Compiler verification
Specification refinement
Deductive Verifier
Program Equivalence Checker
C
Java
JavaScript
WASM
JAUS
Vyper
EVM

Specifying language semantics



[PLDI'15]

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

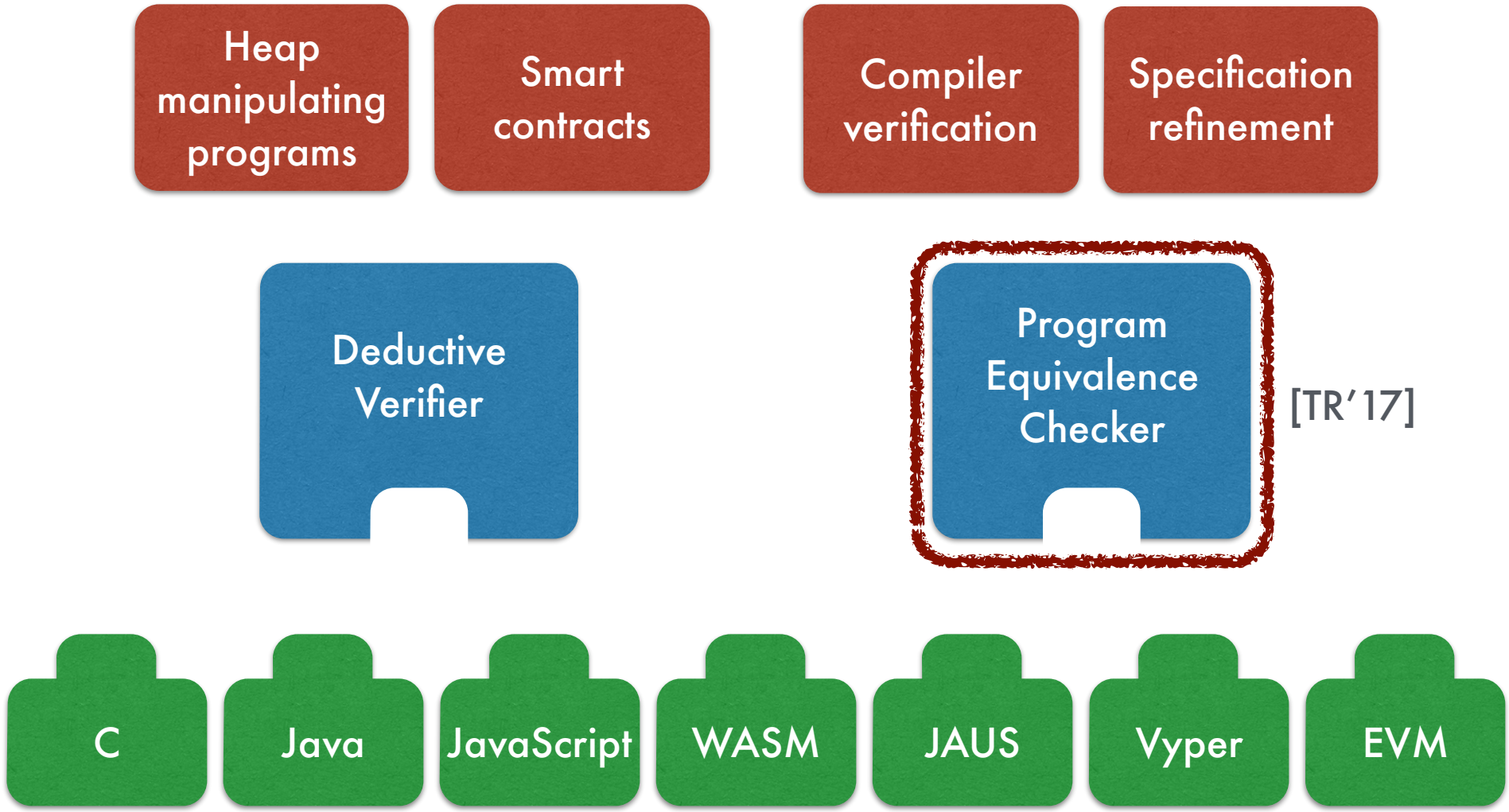
JAUS

Vyper

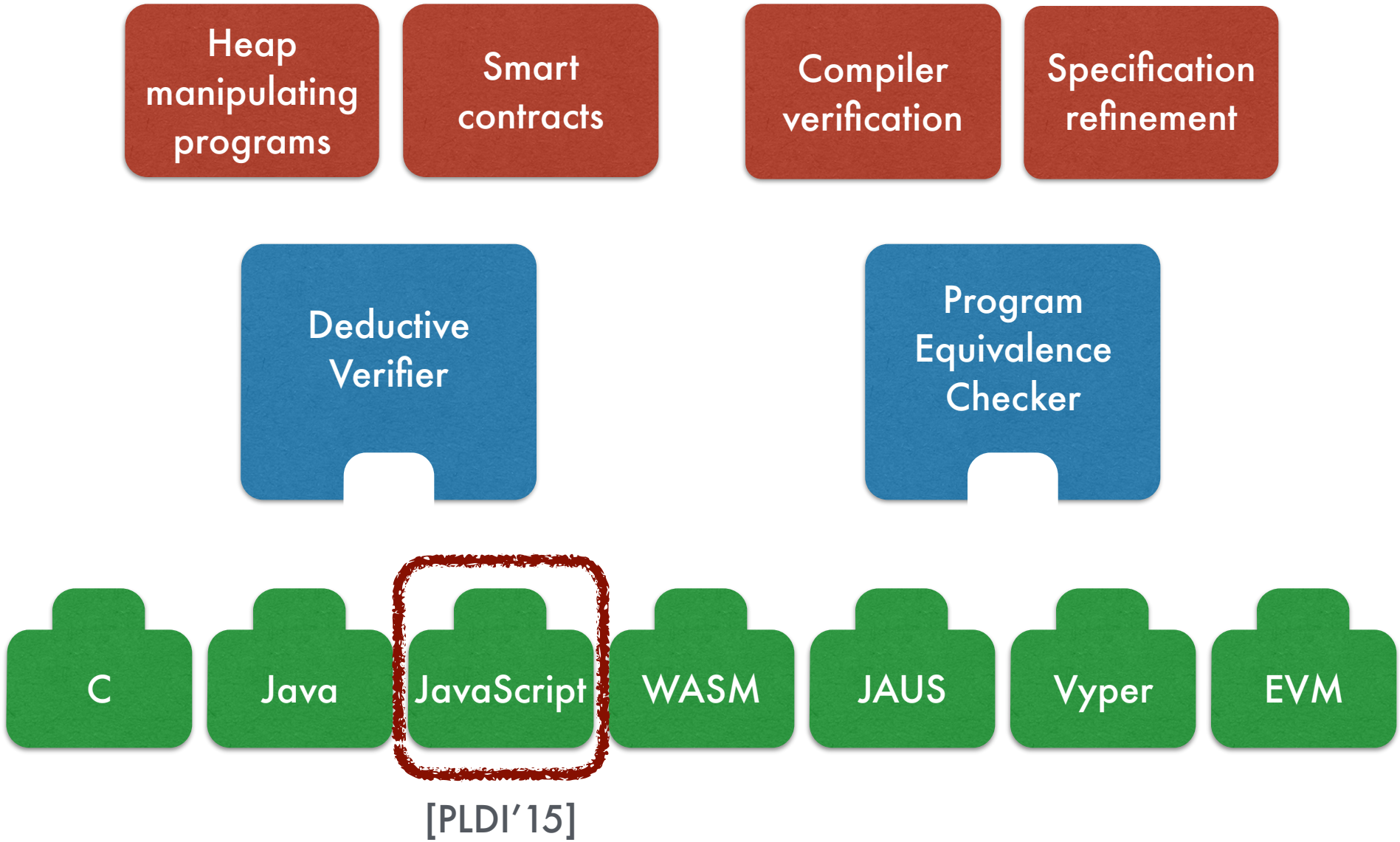
EVM

[CSF'18]

Developing new formal method



Specifying language semantics



[PLDI'15]

KJS: complete formal semantics of JavaScript

KJS *faithfully* formalizes ECMAScript 5.1 (informal) standard.

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 (informal) standard.

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 (informal) standard.

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

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 (informal) standard.

each step of informal description ⟷ formal pseudo-code statement

1-to-1 mapping

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*.

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

ECMAScript 5.1 standard

KJS

manual inspection

Completeness

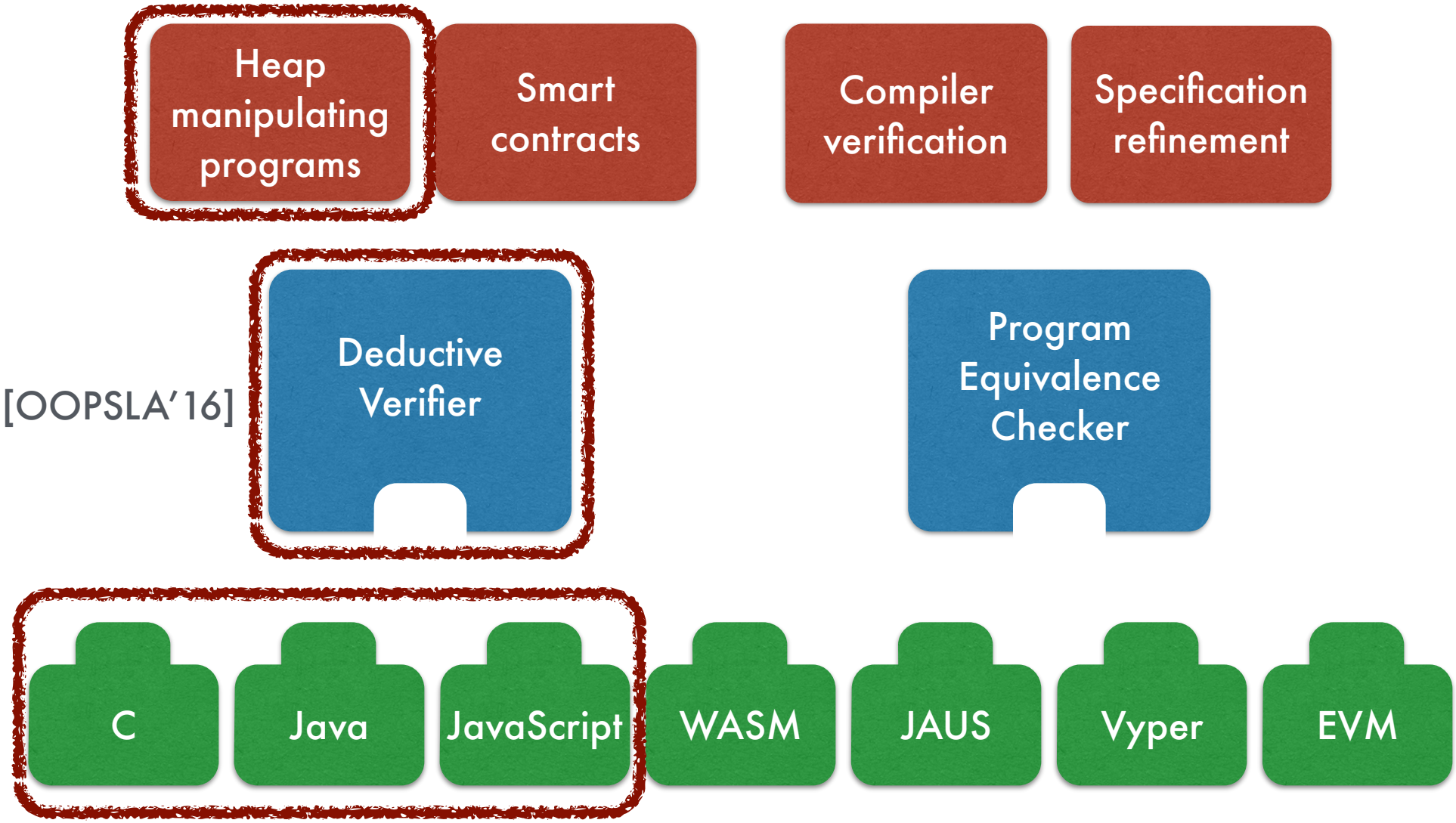
Tested against ECMAScript conformance test suite.

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%



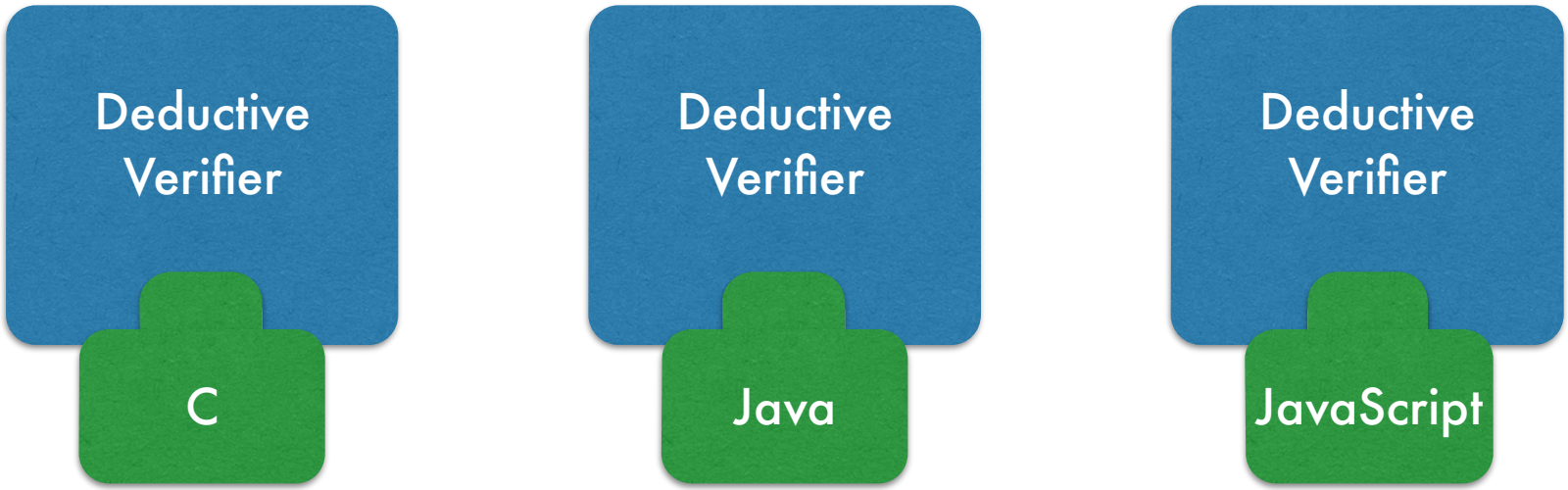
Took me only four months.

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
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
AVL delete	633.7	271.6	239.6
RBT find	14.5	5.0	6.8
RBT insert	903.8	115.6	114.5
RBT delete	1,902.1	171.2	183.6

Programs	C	Java	JS
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
Bubble sort	66.4	38.8	31.3
Insertion sort	61.9	31.1	44.8
Quick sort	79.2	47.1	48.1
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]

Time (secs)

Programs	C	Java	JS
BST find	14.0	4.7	6.3
BST insert	30.2	8.6	8.2

Programs	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]

Time (secs)

Programs	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

Programs	C	Java	JS
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), …

  - 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:

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

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

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

Prove:

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

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

found divergent behaviors

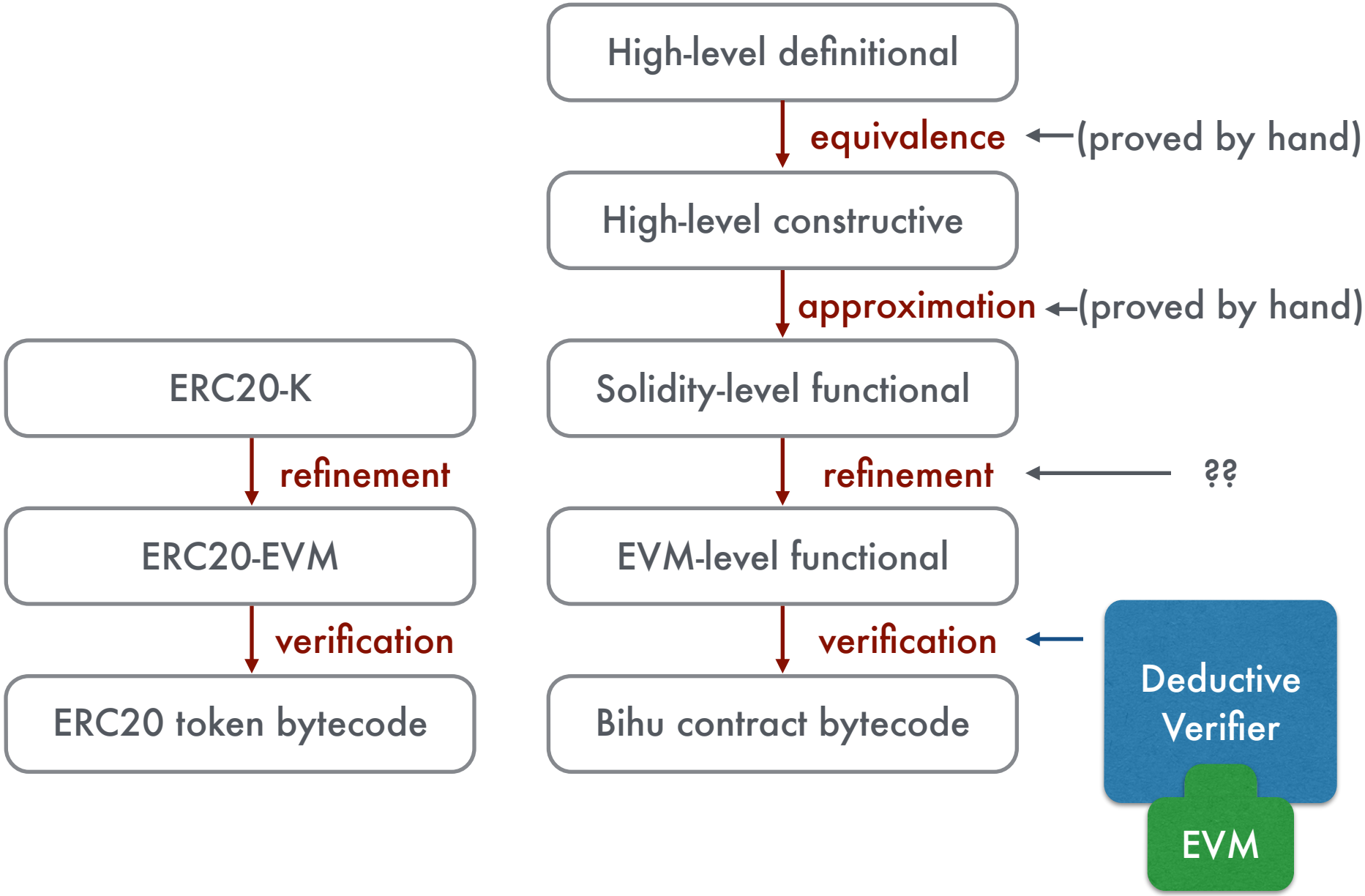
- Commercial Bihu KEY operation contracts
  - KeyRewardPool
  - WarmWallet

* https://github.com/runtimeverification/verified-smart-contracts

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

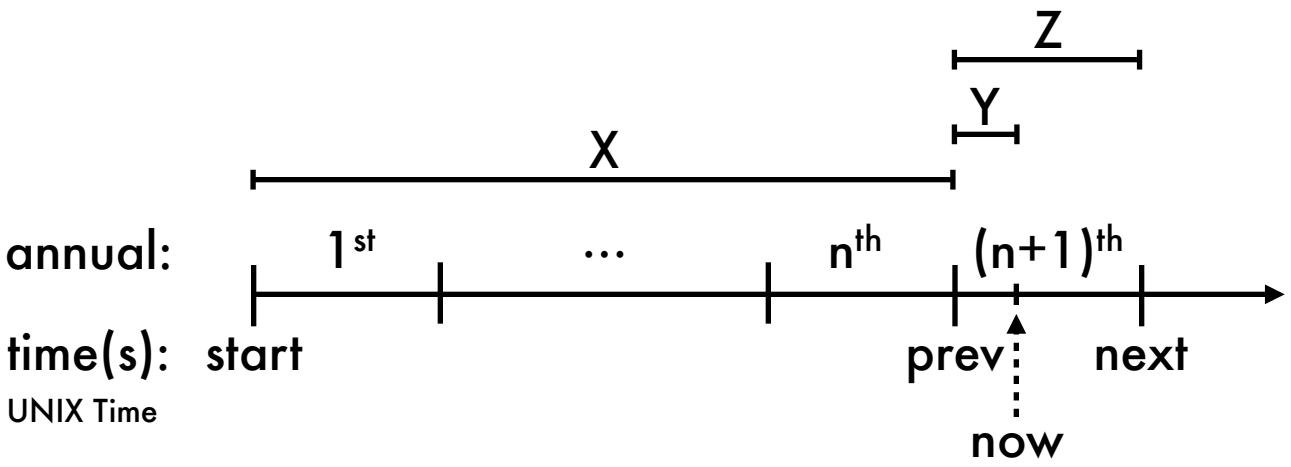
End-to-end verification



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:*

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

then the following holds for the outputs of the function:

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

```
    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:*

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

then the following holds for the outputs of the function:

$$\begin{array}{rcccl} (T - C') - 10 & < & \texttt{balance'} & < & (T - C') + 3 \\ C' - 3 & < & \texttt{collectedTokens'} & < & C' + 10 \end{array}$$

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* $\texttt{canExtractThisYear} + (\texttt{total} - \texttt{remainingTokens})$ *for each function call at* t *and* t'*, respectively. Assume that* `total` *does* not *descrese 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:*

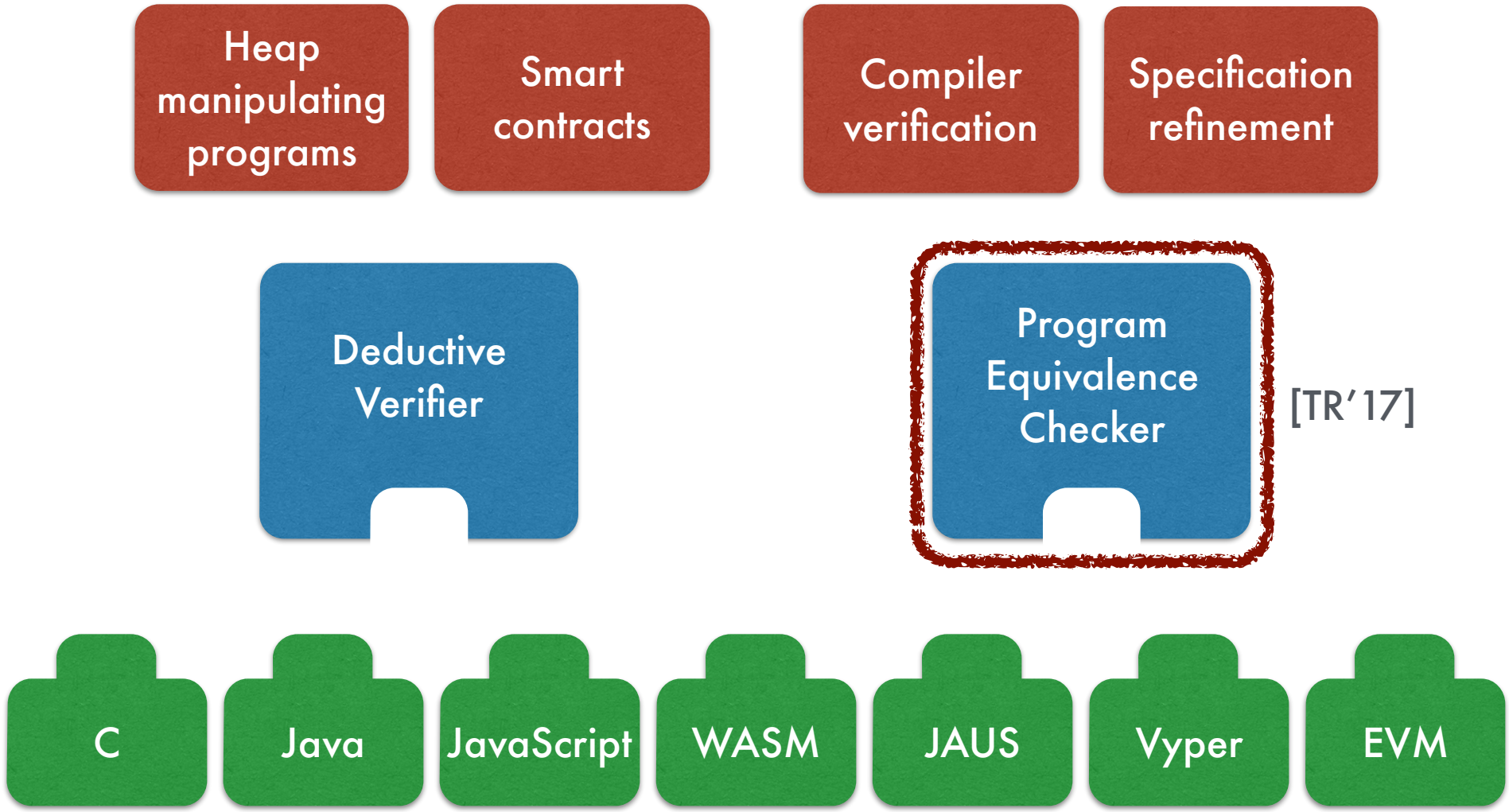
$$\texttt{canExtractThisYear} + (\texttt{total} - \texttt{remainingTokens}) \geq \texttt{collectedTokens}$$

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

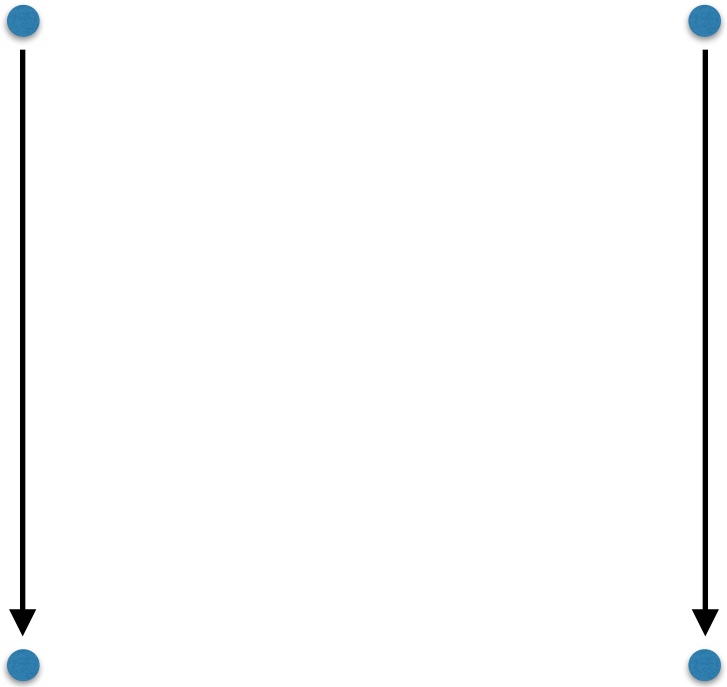
EVM-level functional specification

```
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 31536000)) +Int 43000
ensures: VALUE ==Int @canExtractThisYear(COLLECTED +Int BAL, NOW, START)
                     +Int BAL -Int @remainingTokens(COLLECTED +Int BAL, NOW, START)
```

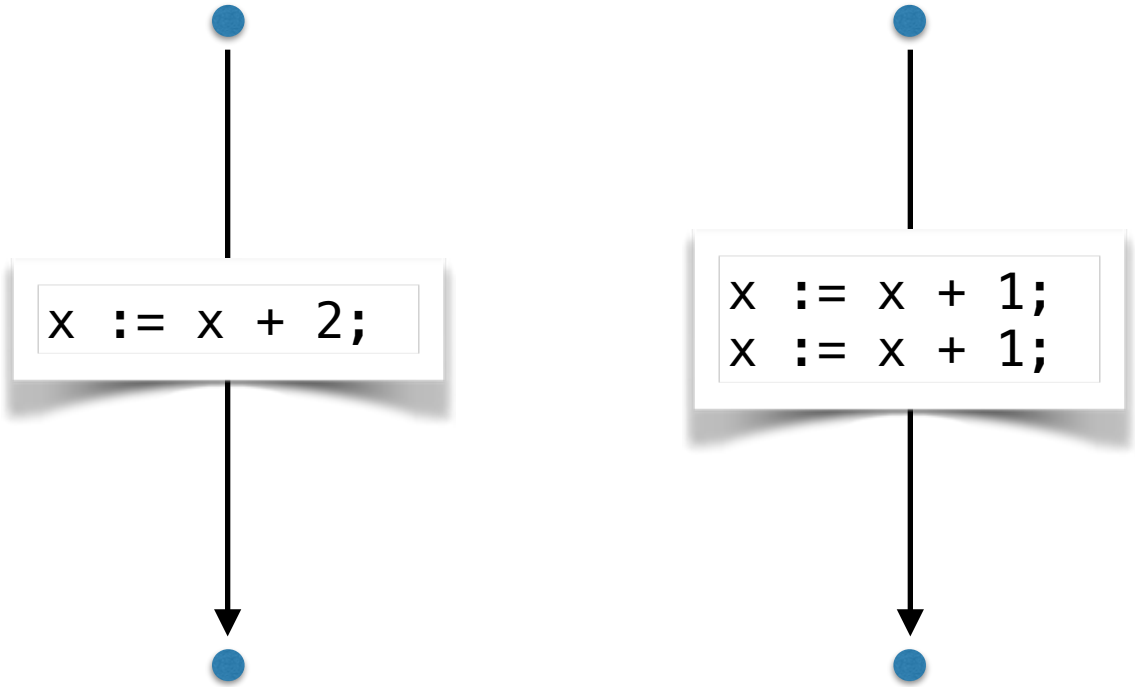
Developing new formal method



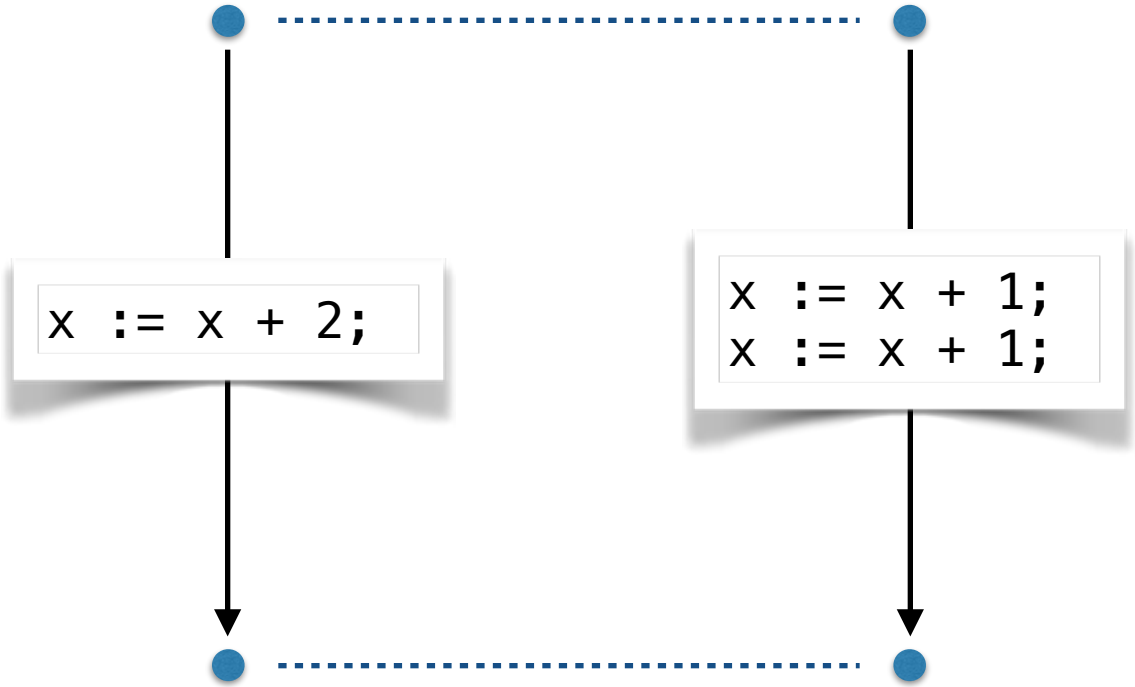
Deterministic, terminating programs



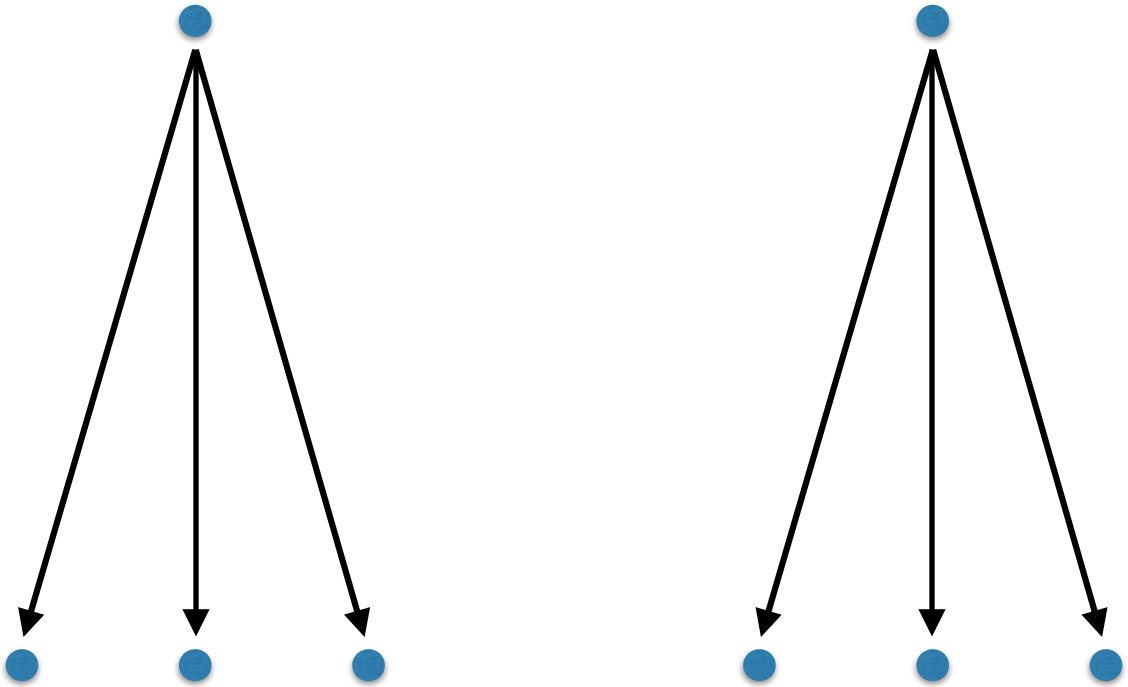
Deterministic, terminating programs



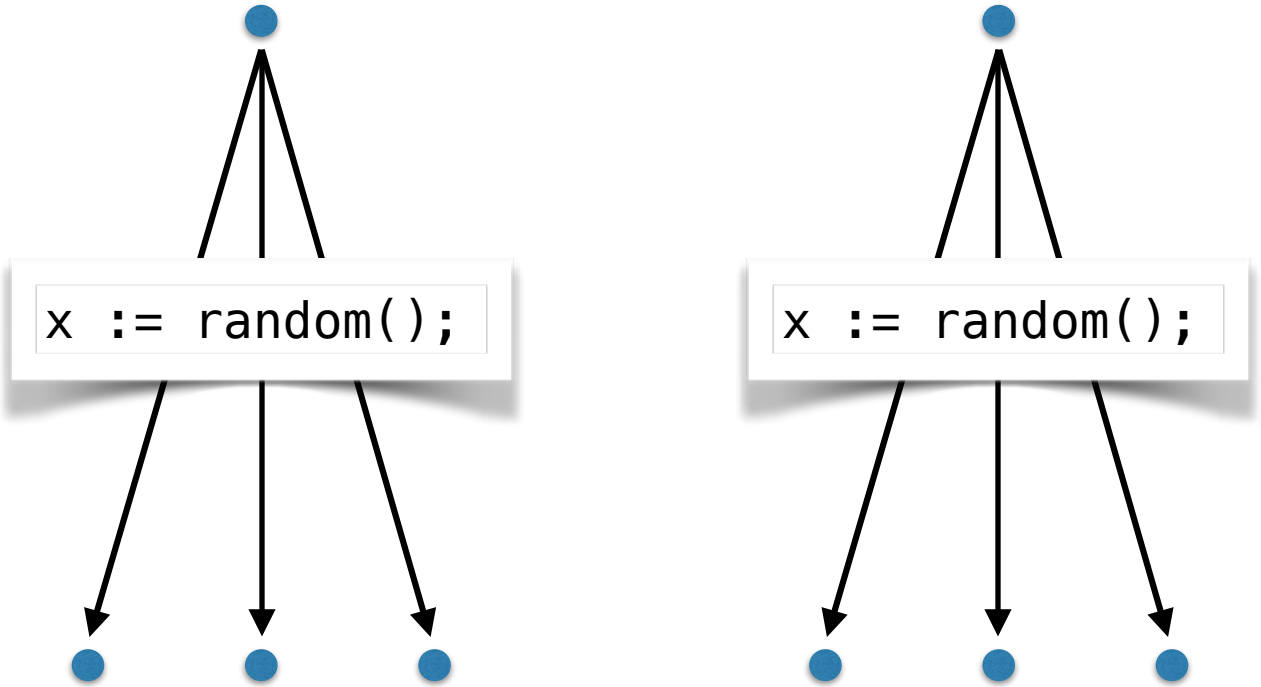
Deterministic, terminating programs



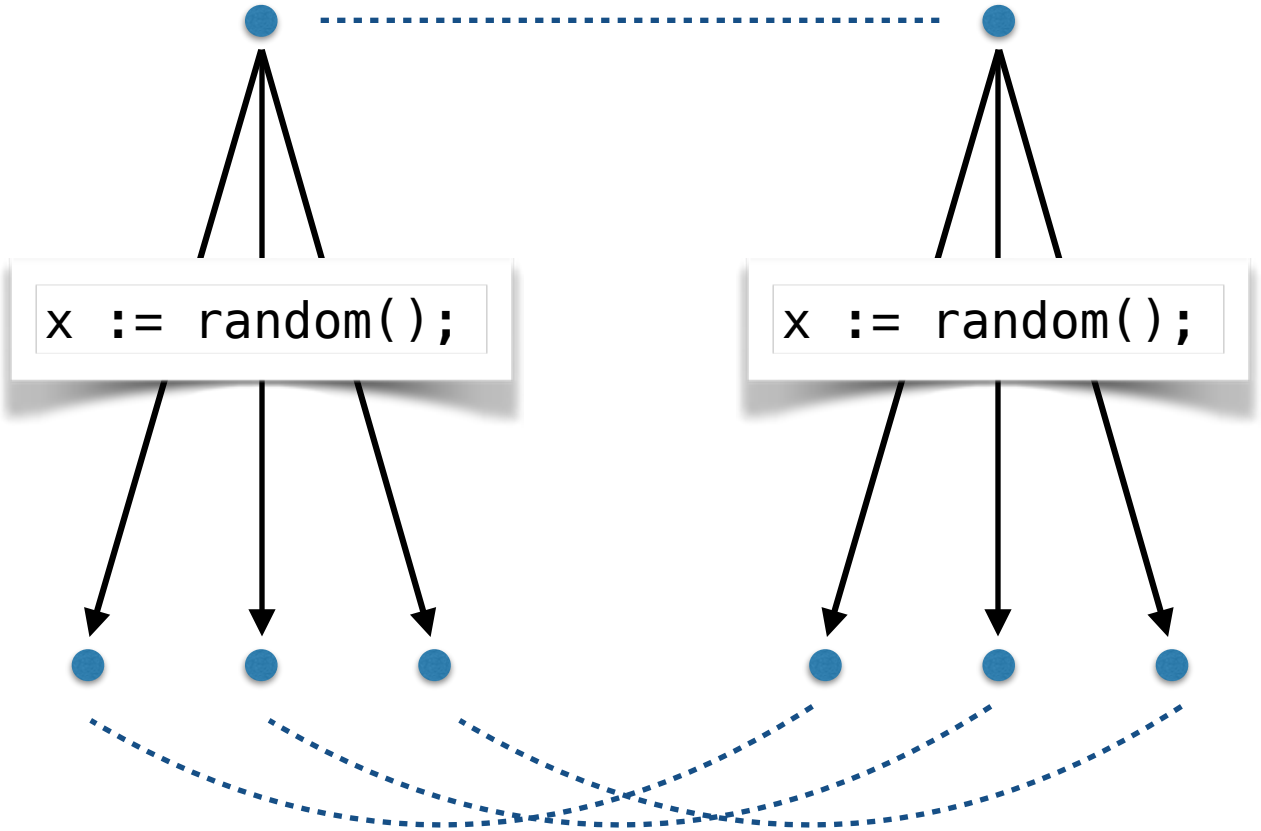
Nondeterministic programs



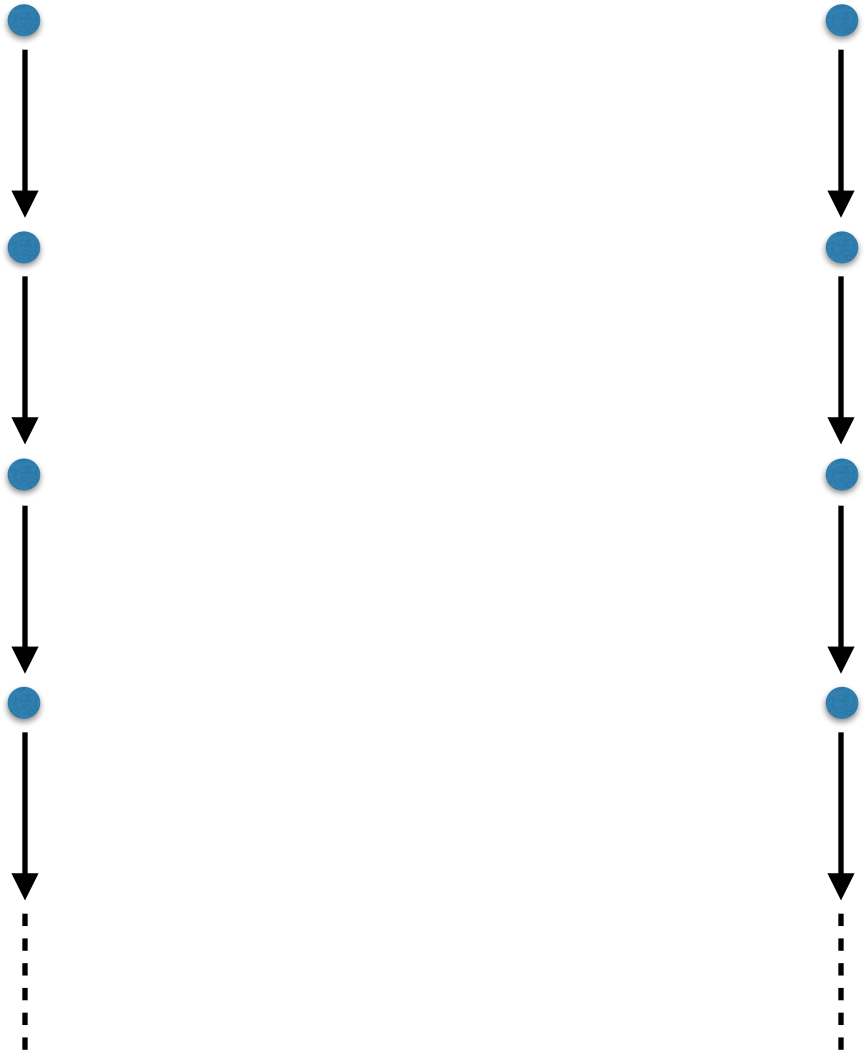
Nondeterministic programs



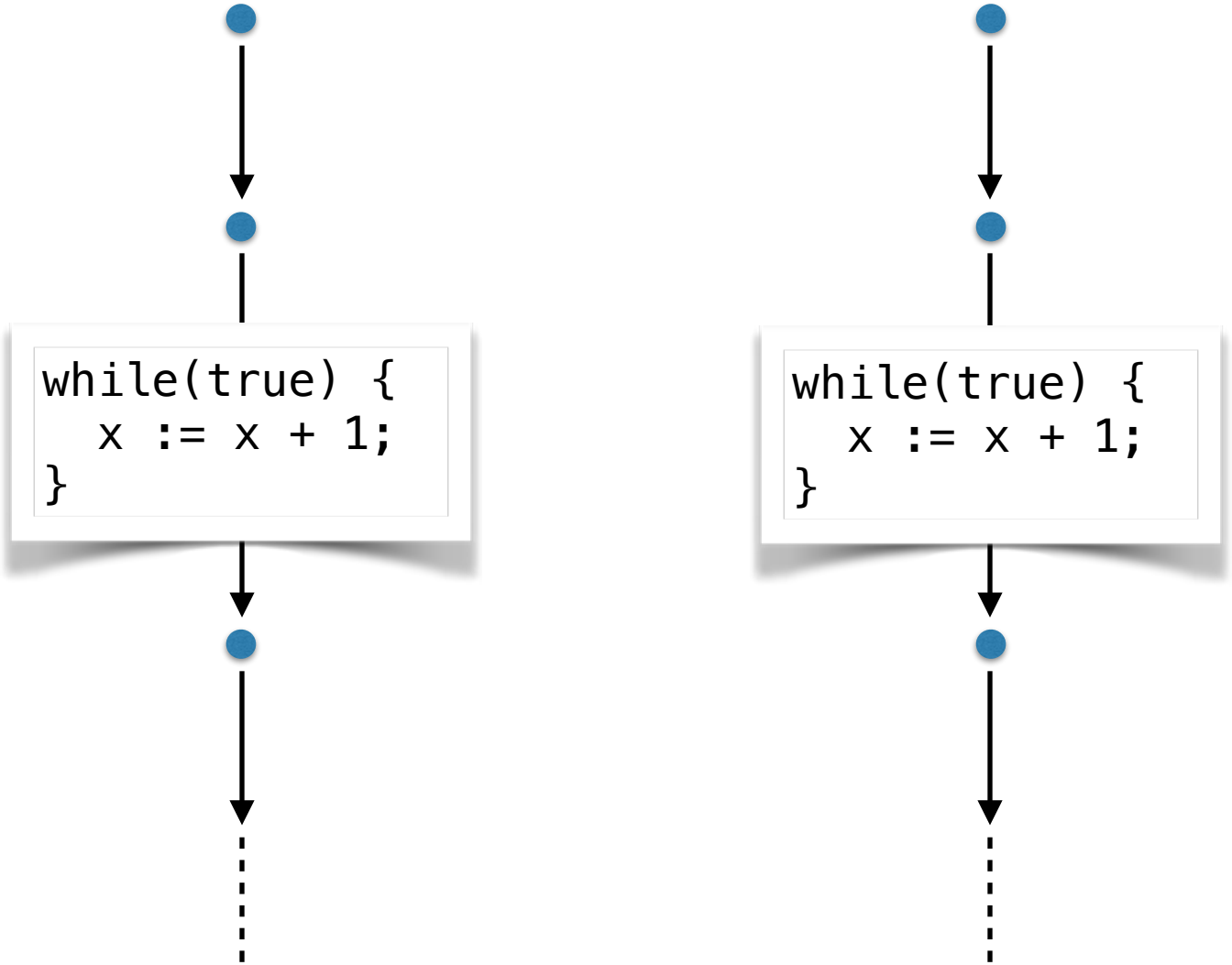
Nondeterministic programs



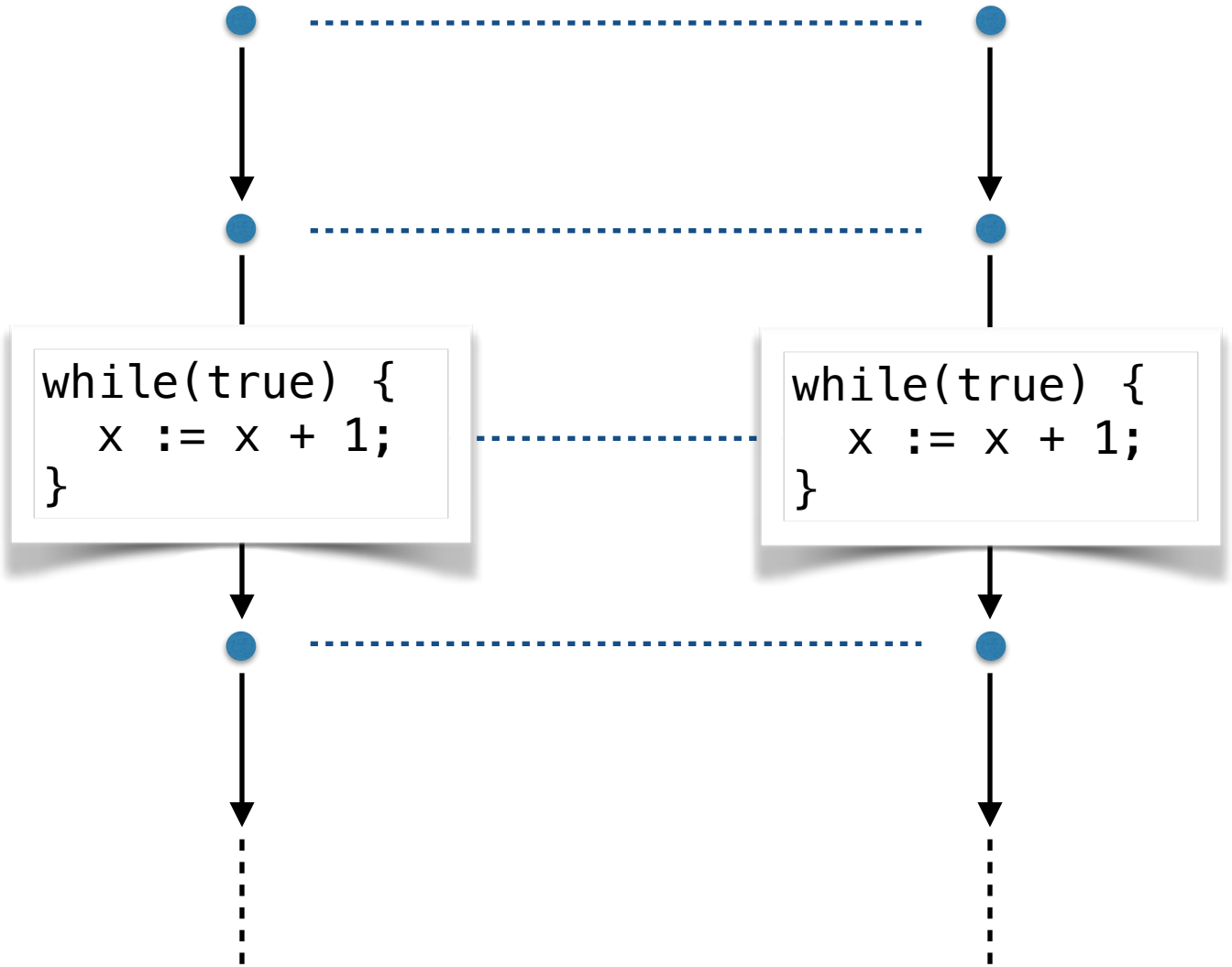
Non-terminating programs



Non-terminating programs



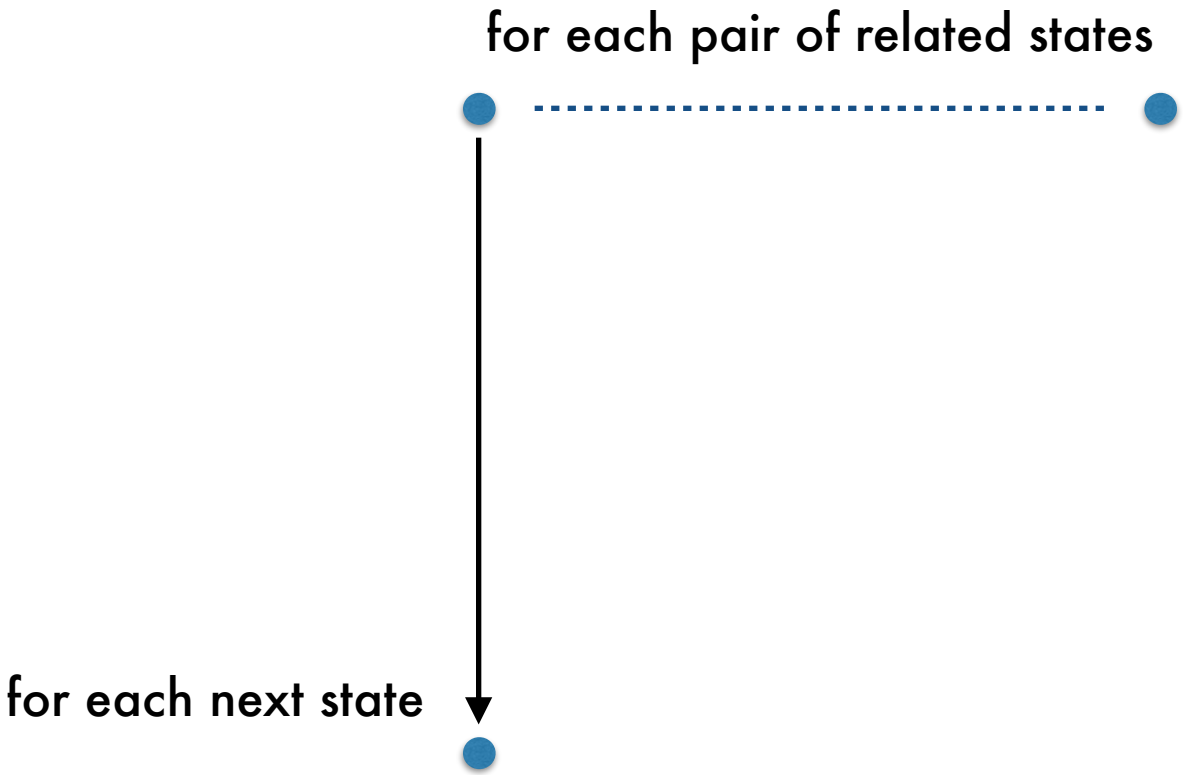
Non-terminating programs



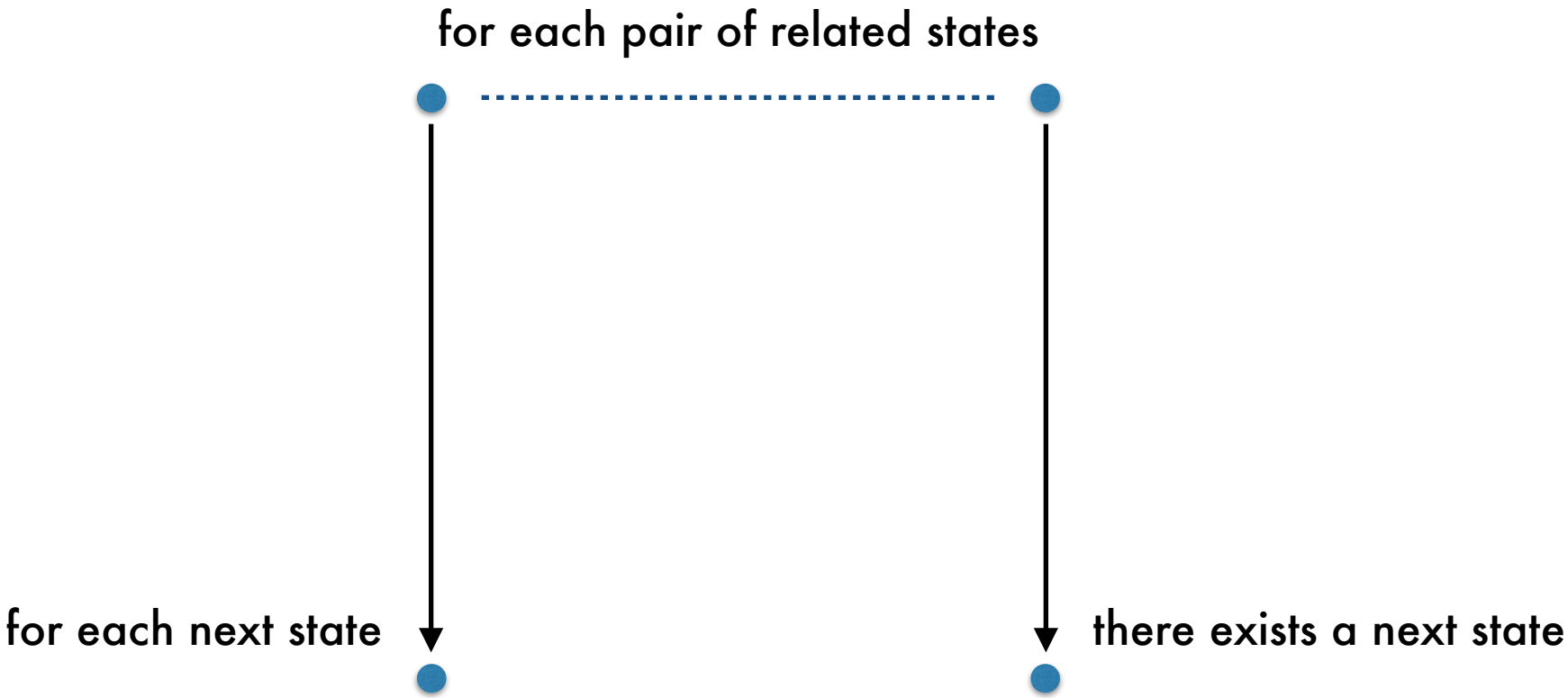
Bisimulation

for each pair of related states

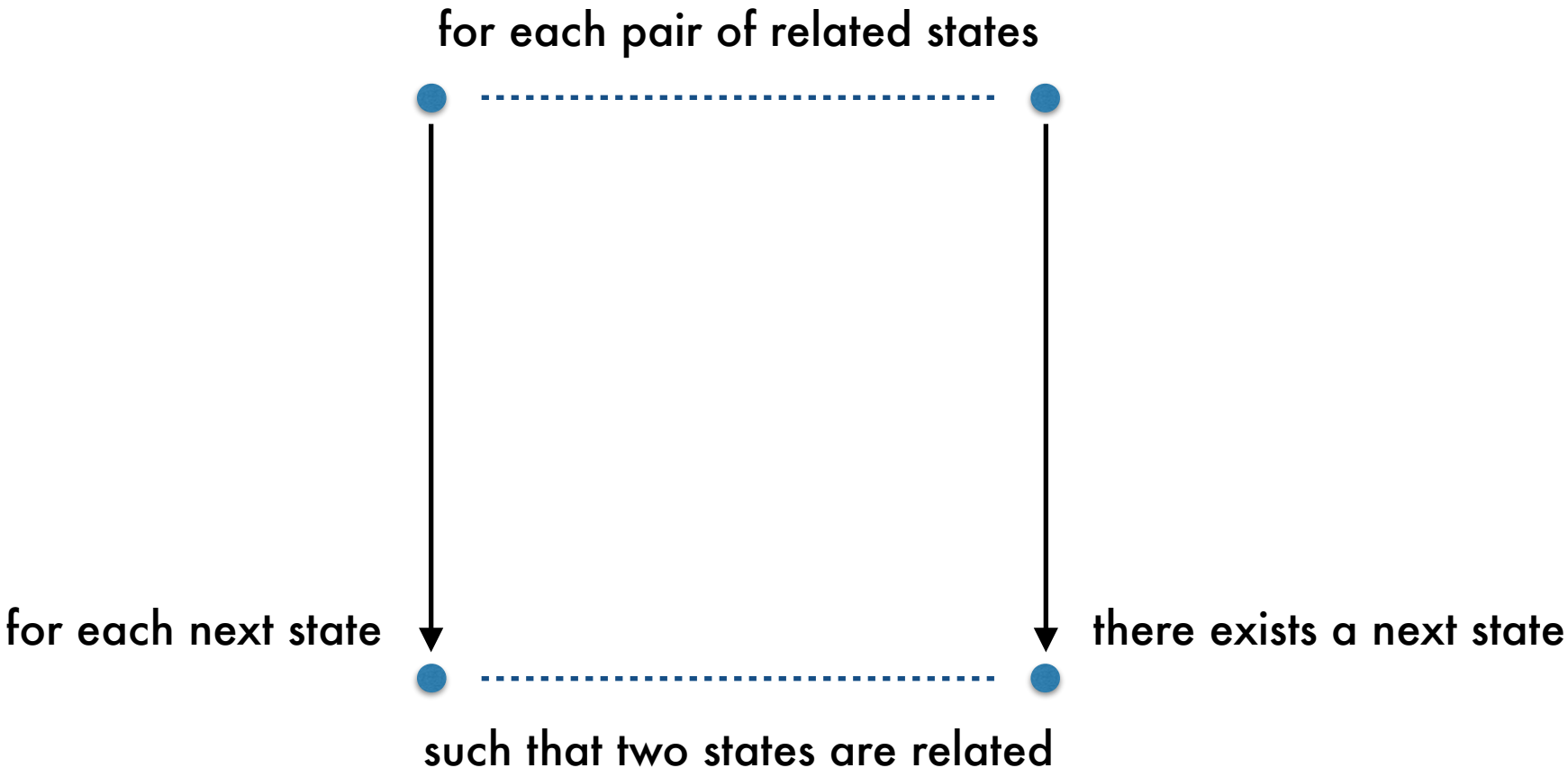
Bisimulation



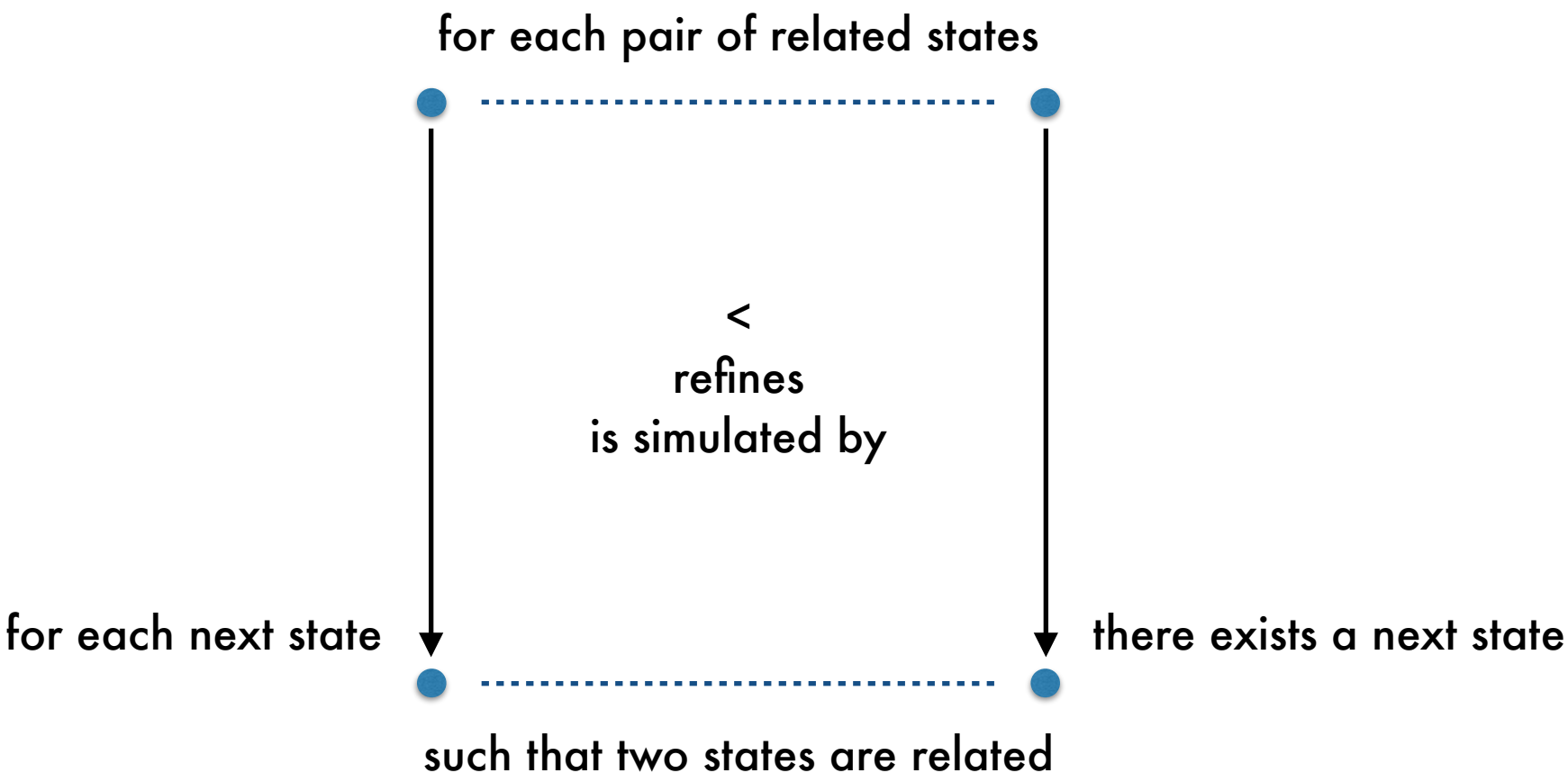
Bisimulation



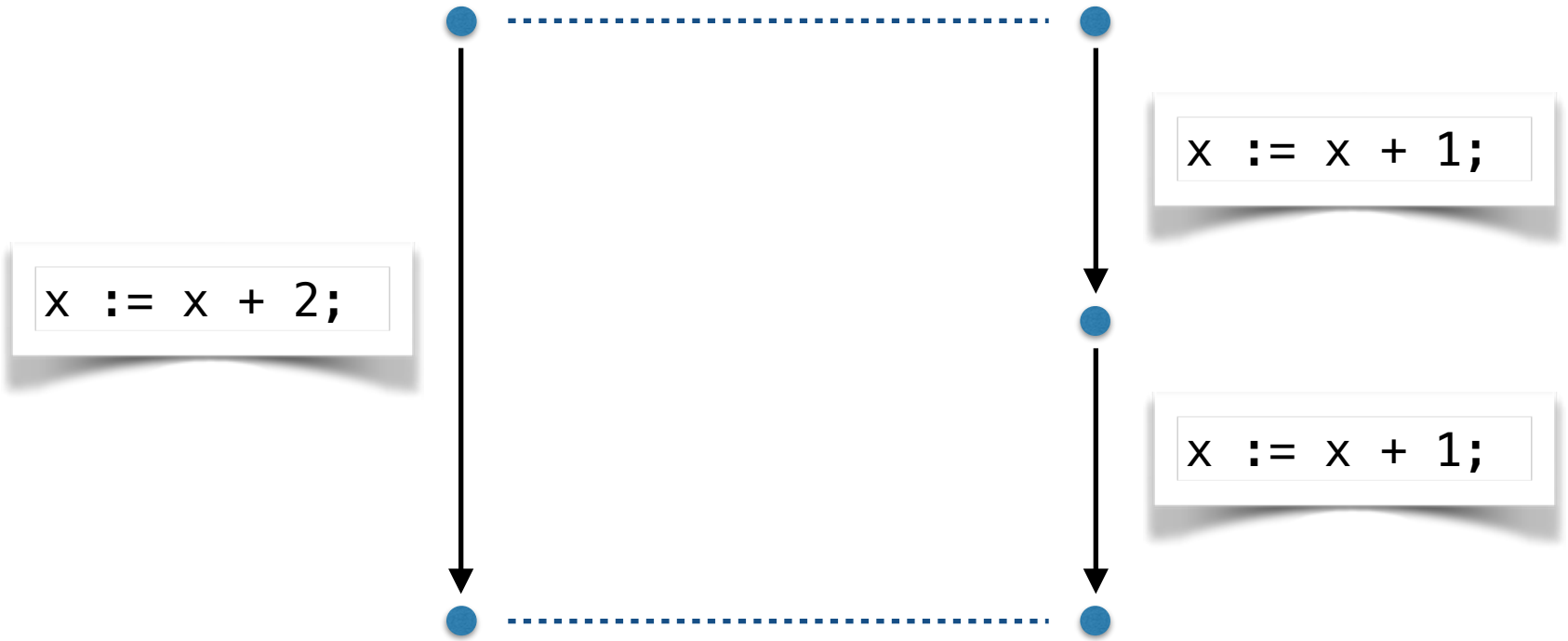
Bisimulation



Bisimulation

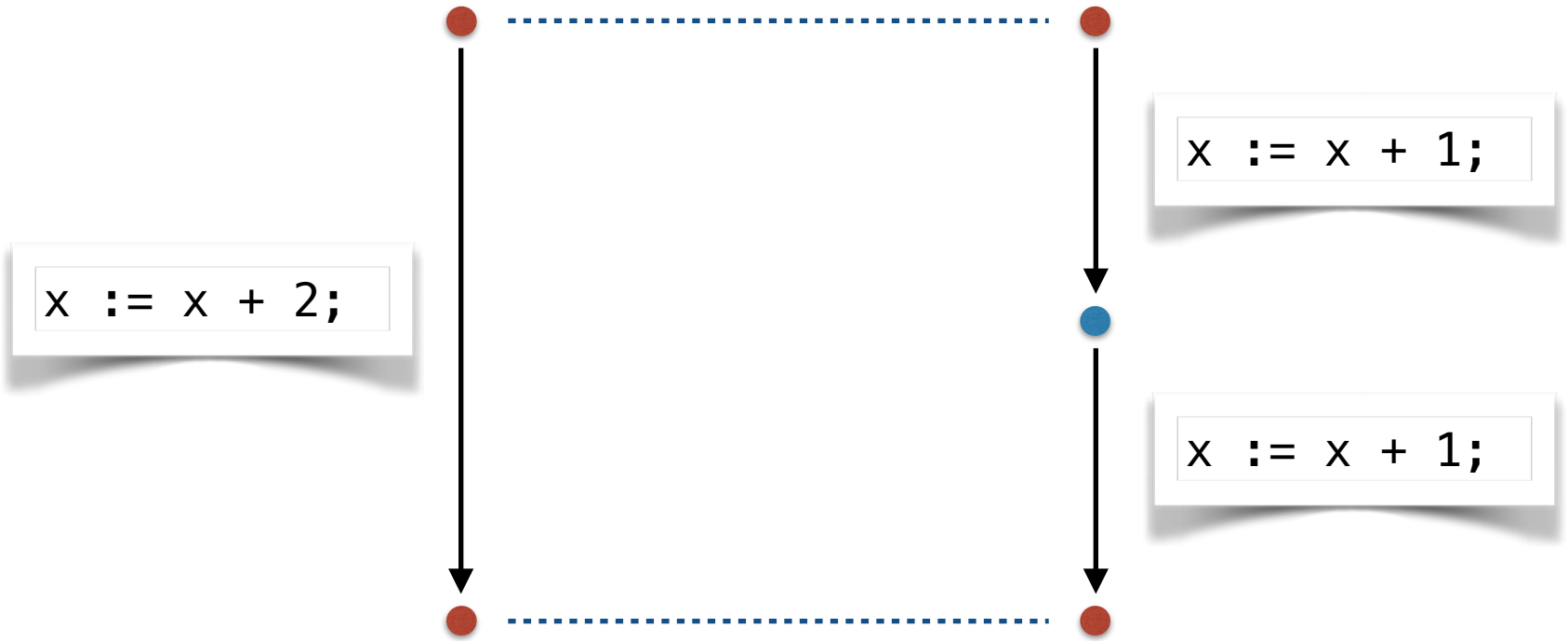


Bisimulation?

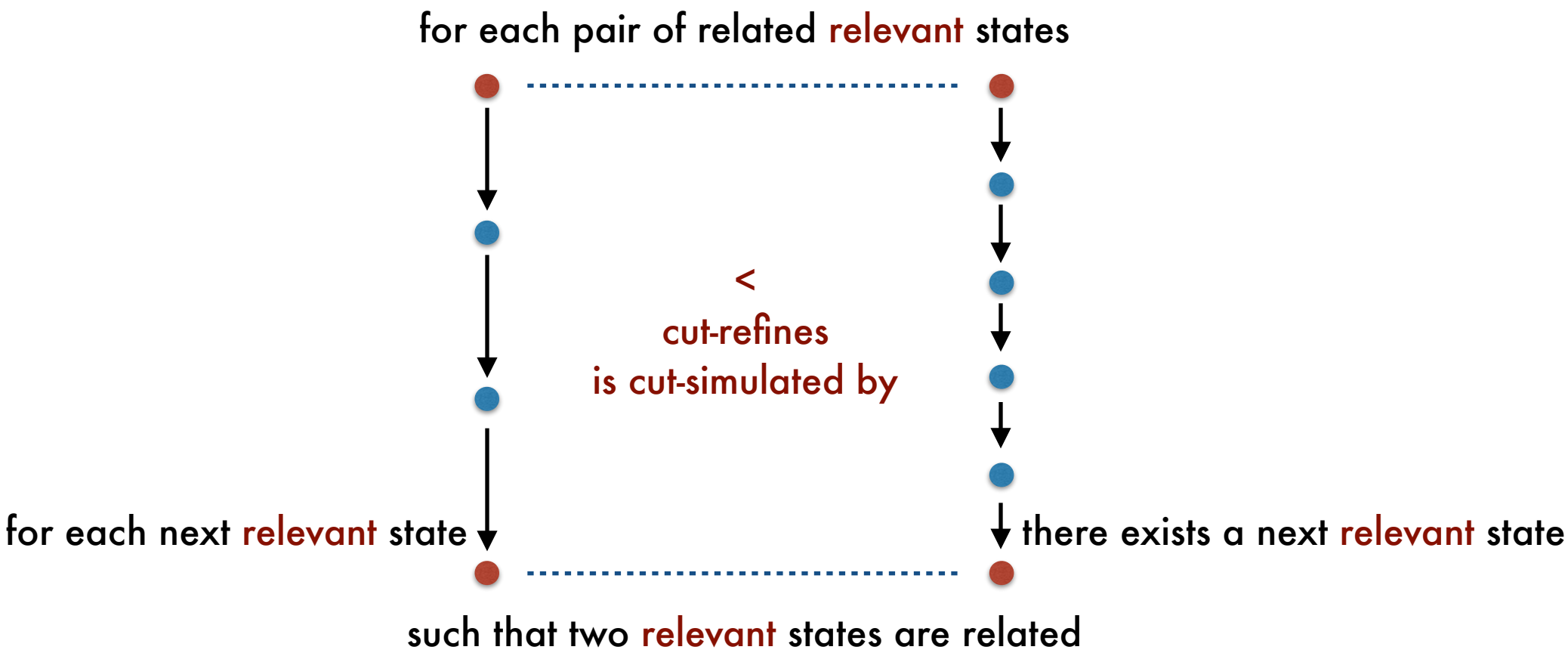


Cut-bisimulation

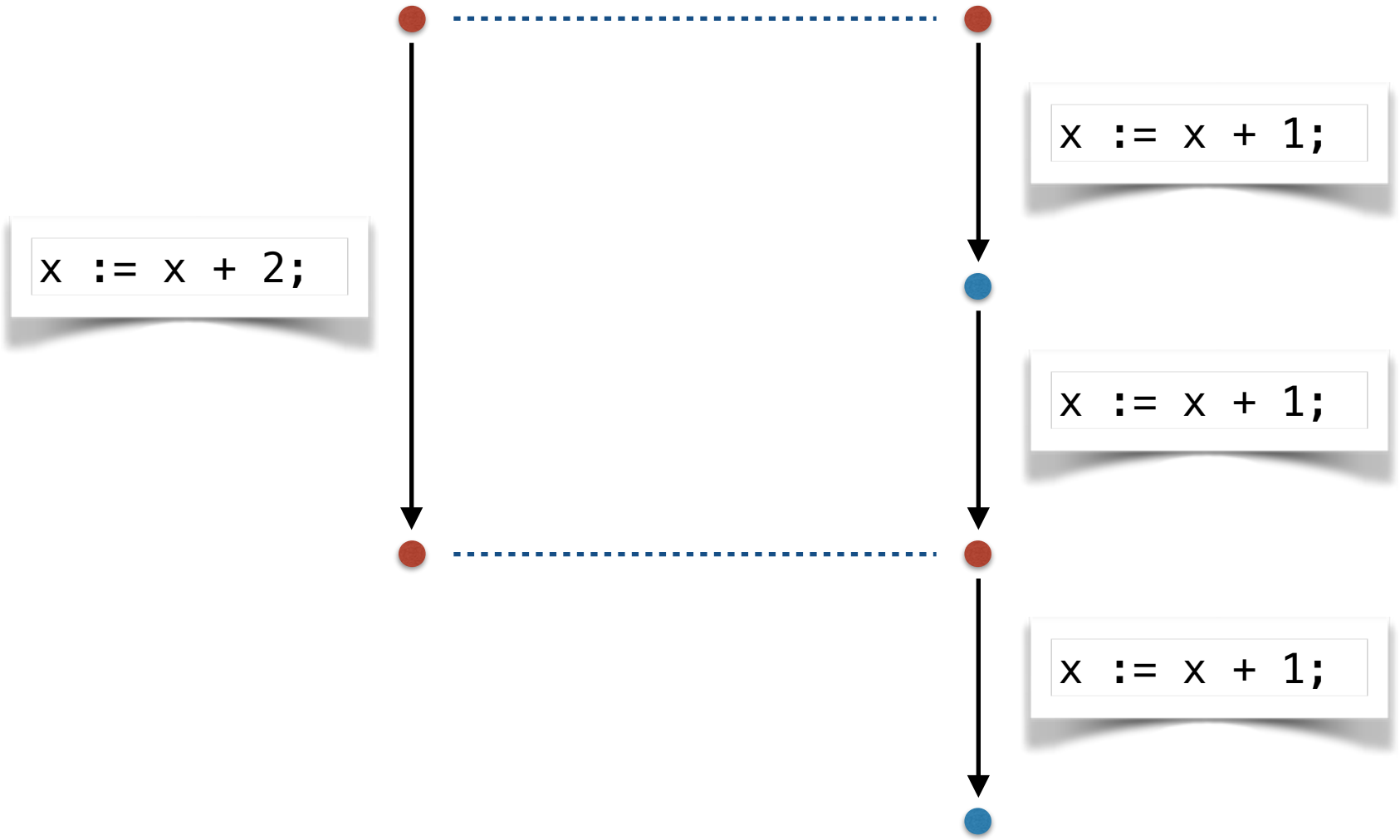
considering only relevant states



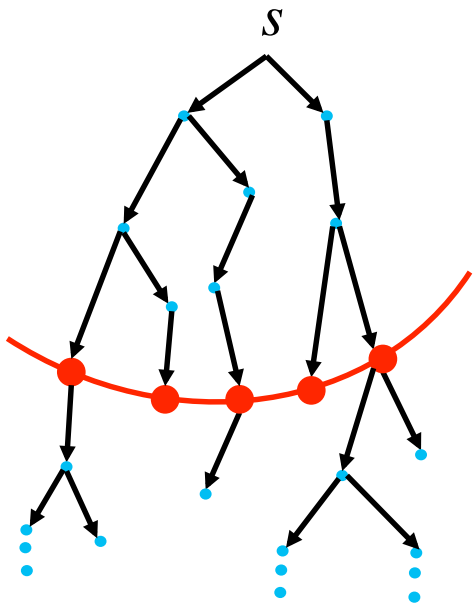
Cut-bisimulation



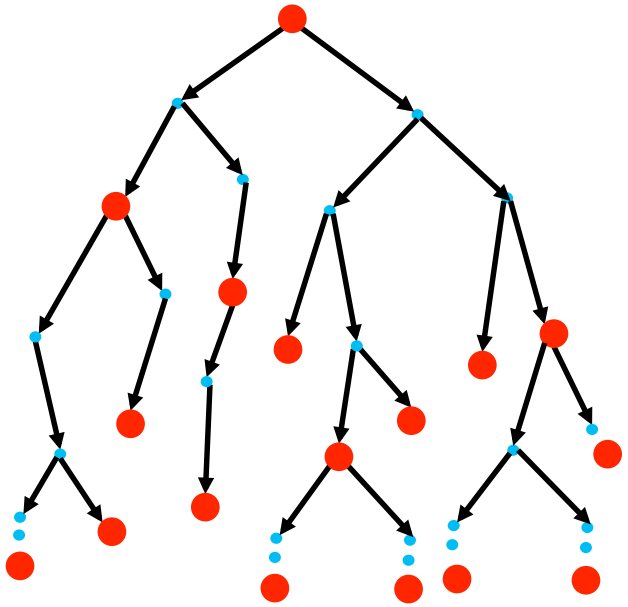
Well-formedness of relevant states



Cut



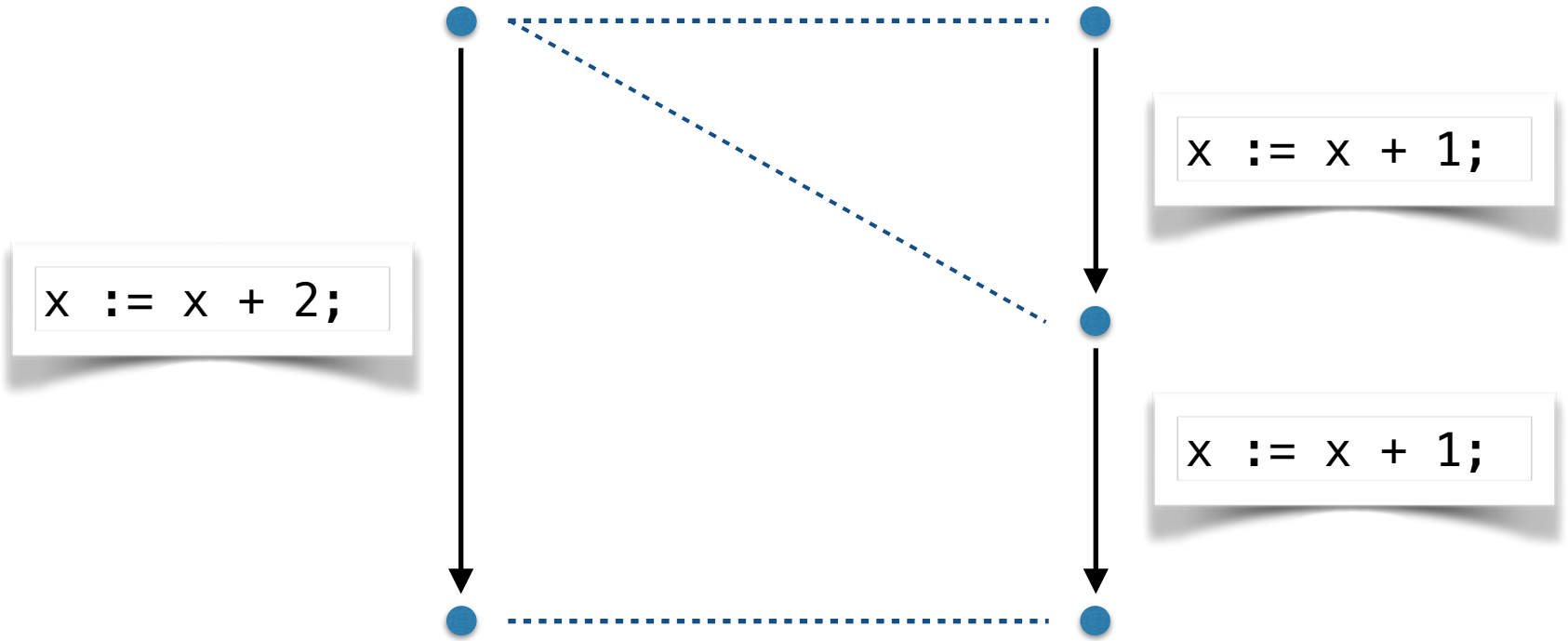
Cut for s



Cut for all

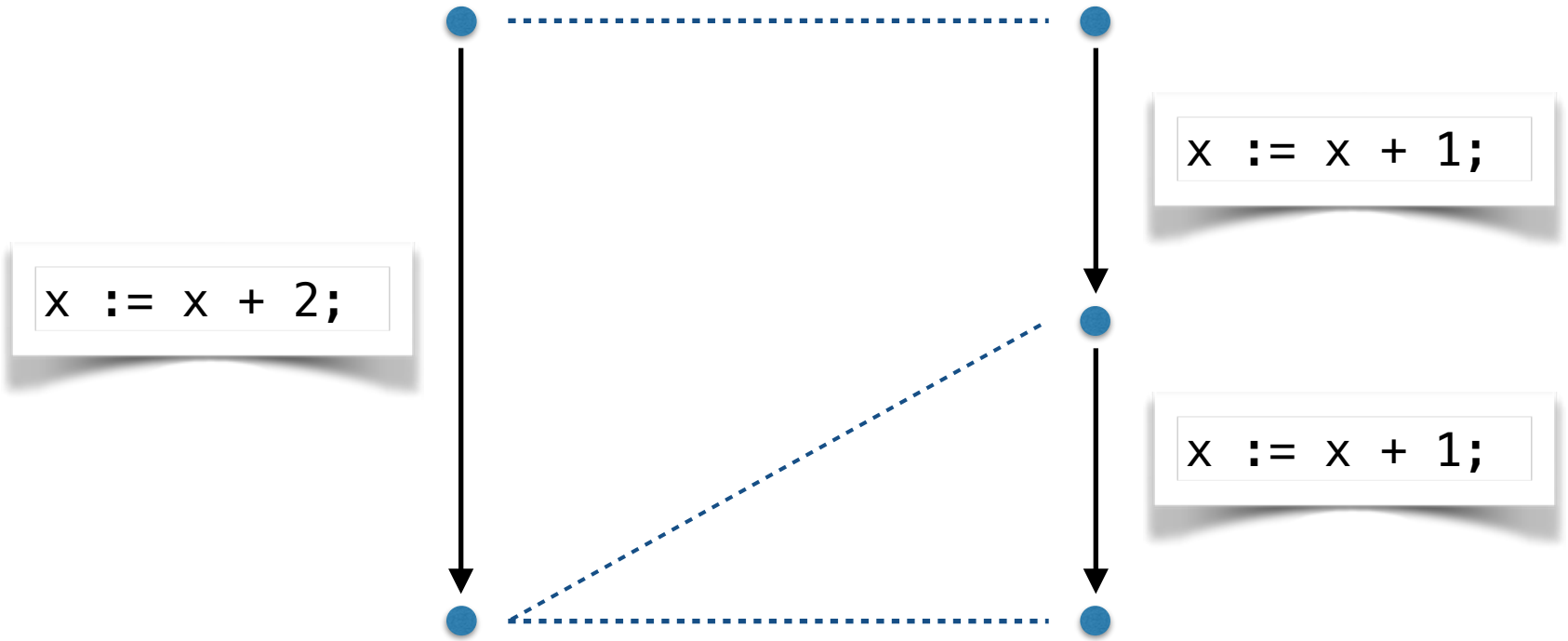
Stuttering bisimulation [BCG'88, dNV'90]

two possible stuttering bisimulations: 1 of 2

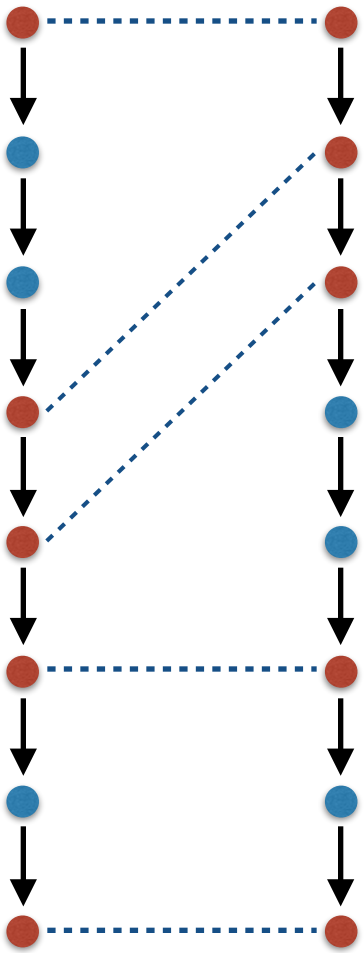


Stuttering bisimulation [BCG'88, dNV'90]

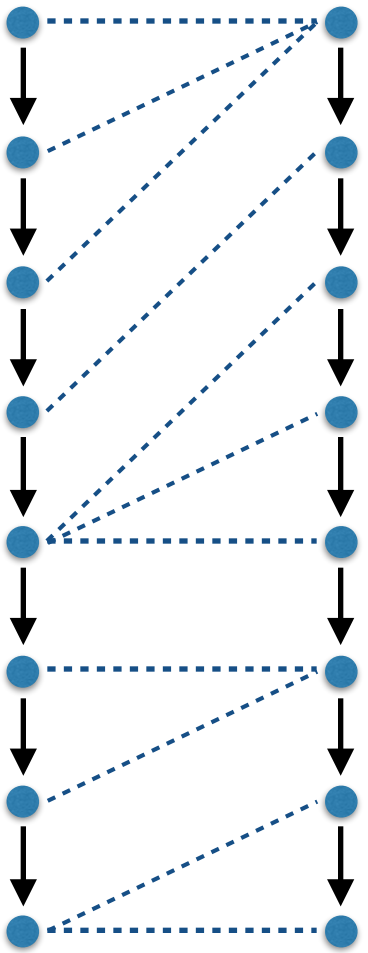
two possible stuttering bisimulations: 2 of 2



Cut-bisimulation vs stuttering bisimulation



vs



[Namjoshi'97]

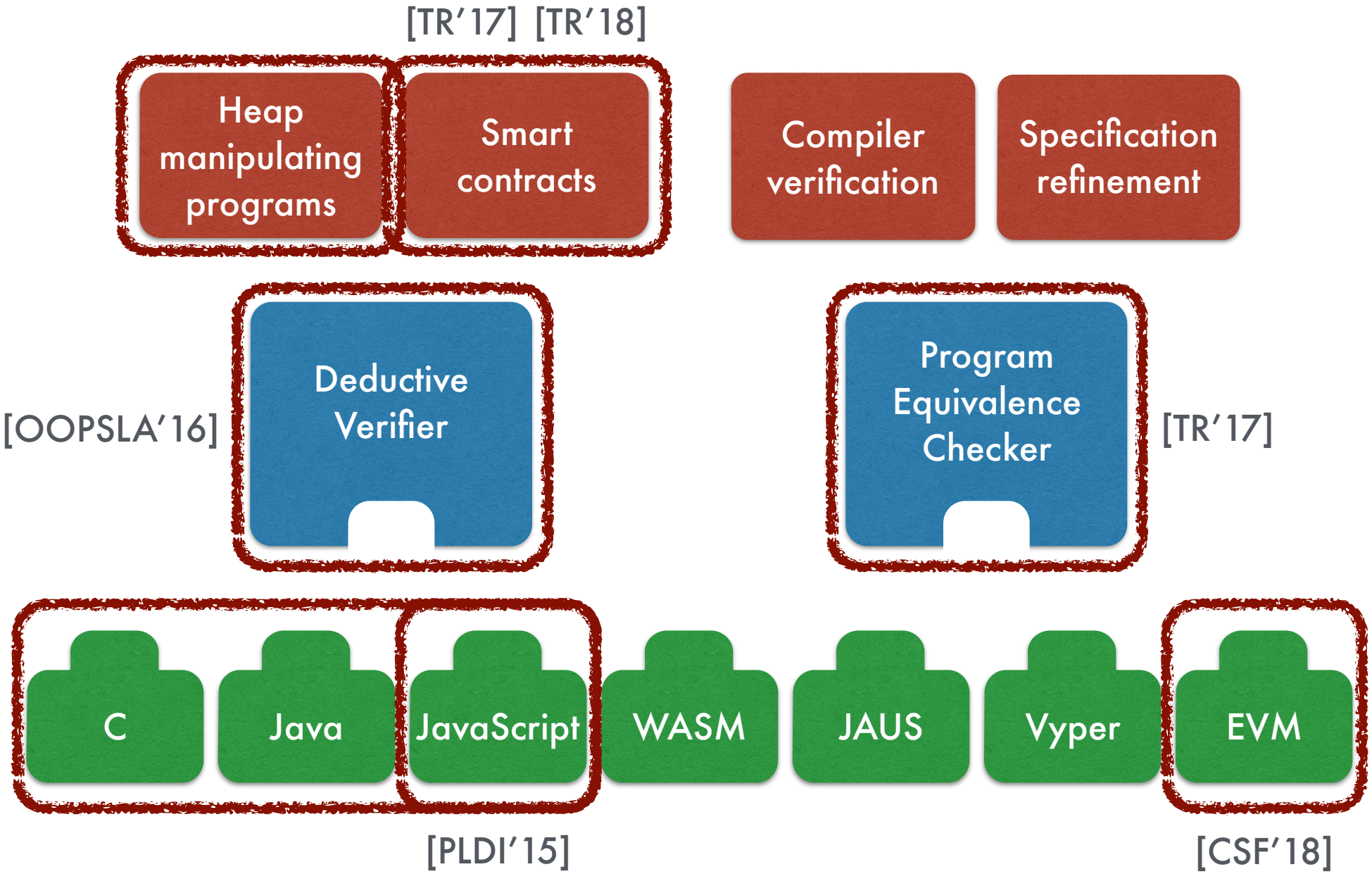
KEQ: universal program equivalence checker

- 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

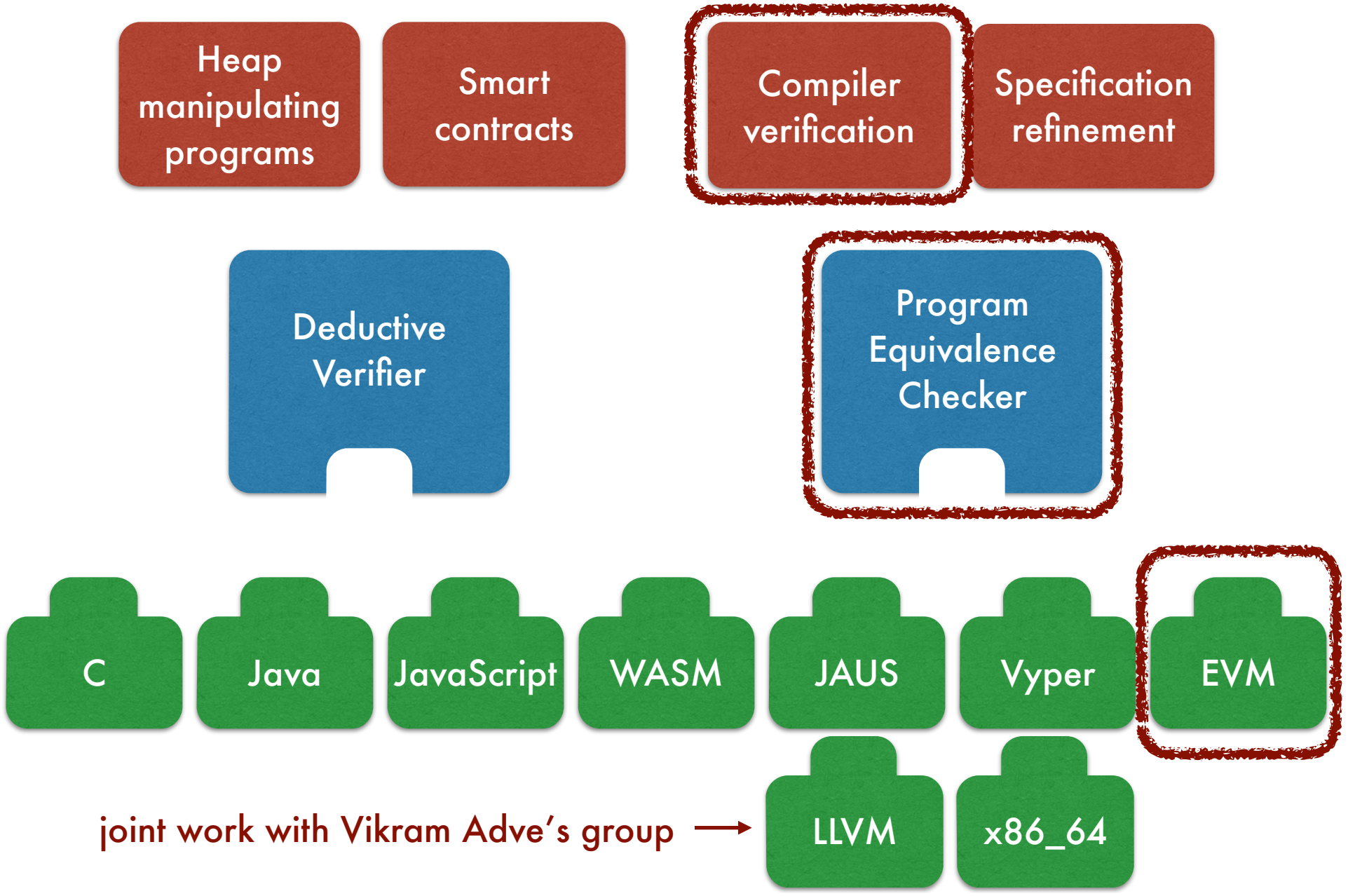
Translation validation [Pnueli *et al.*'98]

- 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



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